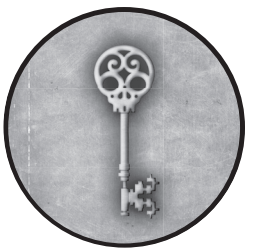
3

INTELLIGENCE GATHERING



Intelligence gathering is the second step of a penetration test, following the preengagement activities. Your goals during this phase are to gain accurate information about your targets without revealing your presence, learn how the organization operates, and determine the best way in. If you don't perform these tasks thoroughly, you may miss vulnerable systems and viable attack vectors. It takes time and patience to sort through web pages, perform Google hacking, and map systems to fully understand the infrastructure of a particular target. You'll also need careful planning, research, and, most importantly, the ability to think like an attacker.

Before you begin intelligence gathering, consider how you will record your actions and the results you achieve. Most security professionals quickly learn that detailed notes can mean the difference between success and failure. Just as a scientist must achieve reproducible results, other experienced penetration testers should be able to reproduce your work using your documentation alone.

WARNING *If you follow the procedures in this book, you can damage your system and your target, so be sure to operate in a test environment. (For help, see Appendix A.) Many of the examples in these chapters can be destructive and make a target system unusable. Some of these activities could even be considered illegal if undertaken by someone with bad intentions, so follow the rules and don't be stupid.*

Most people find themselves eager to exploit systems and get root privileges, but you need to learn to walk before you can run.

Passive Information Gathering

By using *passive* or *indirect information gathering techniques*, you can discover details about targets without touching their systems. For example, you can use these techniques to locate network boundaries, identify network maintainers, and even learn what operating system and web server software is on the target network.

Open source intelligence (OSINT) is a form of intelligence collection that uses open or readily available information to find, select, and acquire details about a target. Several tools make passive information gathering almost painless, including complex software such as Yeti and Whois. In this section, we'll explore the process of passive information gathering and the tools that you might use for this step.

Imagine, for example, an attack against *https://www.trustedsec.com*. Our goal is to determine, as part of a penetration test, what systems the company owns and what systems we can attack. Some systems may not be owned by the company and could be considered out of scope and unavailable for attack.

Whois Lookups

Whois is a tool that allows you to search for information about domains and internet infrastructure. Let's begin by using Kali Linux's Whois lookup to find the names of *trustedsec.com*'s domain servers:

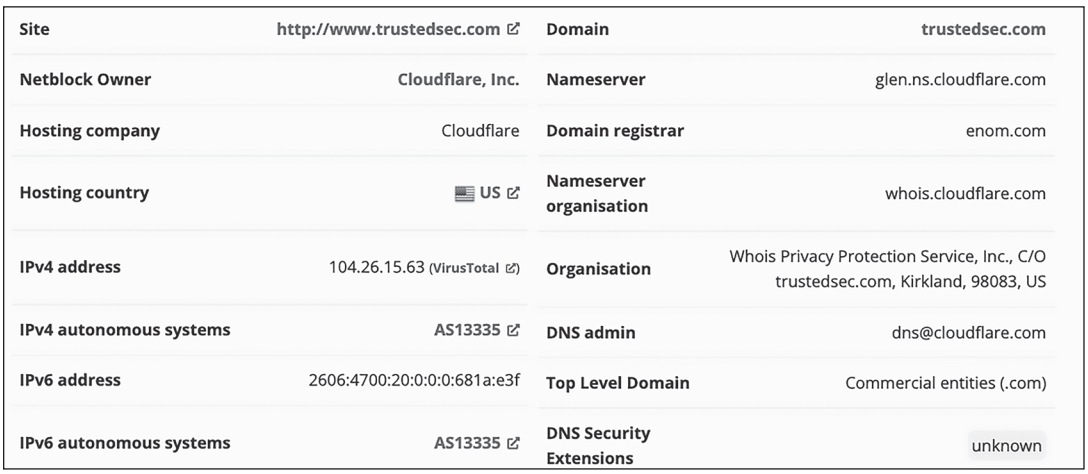
```
msf > whois trustedsec.com
[*] exec: whois trustedsec.com
--snip--
   Domain Name: trustedsec.COM

   Domain servers in listed order:
      GLEN.NS.CLOUDFLARE.COM
      LEIA.NS.CLOUDFLARE.COM
```

We learn that the Domain Name System (DNS) servers are hosted by Cloudflare, a third party, so we should not include these systems in our penetration test because we have no authority to attack them. In most large organizations, however, the DNS servers are housed within the company and are viable attack vectors. Zone transfers and similar DNS attacks can often be used to learn more about a network from both the inside and outside. But in this scenario, we should instead move on to a different attack vector.

Netcraft

Netcraft (*https://searchdns.netcraft.com*) is a web-based tool that we can use to find the IP address of a server hosting a particular website, as shown in Figure 3-1.



Site	http://www.trustedsec.com	Domain	trustedsec.com
Netblock Owner	Cloudflare, Inc.	Nameserver	glen.ns.cloudflare.com
Hosting company	Cloudflare	Domain registrar	enom.com
Hosting country	US	Nameserver organisation	whois.cloudflare.com
IPv4 address	104.26.15.63 (VirusTotal)	Organisation	Whois Privacy Protection Service, Inc., C/O trustedsec.com, Kirkland, 98083, US
IPv4 autonomous systems	AS13335	DNS admin	dns@cloudflare.com
IPv6 address	2606:4700:20:0:0:0:681a:e3f	Top Level Domain	Commercial entities (.com)
IPv6 autonomous systems	AS13335	DNS Security Extensions	unknown

Figure 3-1: Using Netcraft to find the IP address of the server hosting a particular website

Once we've identified *trustedsec.com*'s IP address as 104.26.15.63, we can do another Whois lookup on that IP address to discover additional information about the target:

```
msf > whois 104.26.15.63
[*] exec: whois 104.26.15.63
NetRange:       104.16.0.0 - 104.31.255.255
CIDR:           104.16.0.0/12
NetName:        CLOUDFLARENET
NetHandle:      NET-104-16-0-0-1
Parent:         NET104 (NET-104-0-0-0-0)
NetType:        Direct Allocation
OriginAS:       AS13335
Organization:   Cloudflare, Inc. (CLOUD14)
```

We see from the Whois lookup and a quick internet search that this IP address, belonging to Cloudflare, appears to be that of a legitimate service provider. Cloudflare helps improve internet security by serving as

a reverse proxy between our request and *trustedsec.com*'s servers. As our requests pass through Cloudflare, it inspects the traffic and applies security rules. Other services, such as Amazon CloudFront, Envoy Proxy, and Microsoft Azure CDN, also provide reverse proxy services.

Reverse proxies attempt to hide the original IP addresses. However, an attacker may still be able to recover an IP address using other strategies. An article detailing some of these techniques is available at *https://citadelo.com/en/blog/cloudflare-how-to-do-it-right-and-do-not-reveal-your-real-ip*. Many of these strategies have been incorporated into Metasploit's *cloud look and bypass* module.

DNS Analysis

DNS servers contain information about domains. To get additional domain information, we'll use `dig`, a tool built into most Unix operating systems, to query DNS servers about *trustedsec.com*. Some other great tools for DNS analysis are `fierce` and `dnsrecon`.

In the following example, we use `dig` to look for the domain's mail exchange (MX) record. The MX record contains information about the server used to process email for that domain:

```
kali@kali:~$ sudo dig mx trustedsec.com

;; QUESTION SECTION:
;trustedsec.com.         IN    MX

;; ANSWER SECTION:
trustedsec.com.  5       IN    MX    20 mx2-us1.ppe-hosted.com.
trustedsec.com.  5       IN    MX    10 mx1-us1.ppe-hosted.com.
```

We see that the mail servers are pointing to *mx2-us1.ppe-hosted.com* and *mx1-us1.ppe-hosted.com*. Some quick research tells us that these websites are hosted by a third party, which removes them from the scope of our penetration test.

At this point, we have gathered some valuable information that we might be able to use against the target. Ultimately, however, we may have to resort to active information-gathering techniques to get more details.

NOTE *The art of passive information gathering isn't easily mastered in just a few pages of discussion. See the PTES (*http://www.pentest-standard.org*) and Cyber Detective's OSINT tools collection (*https://github.com/cipher387/osint_stuff_tool_collection*) for a list of potential ways to perform additional passive intelligence gathering.*

Active Information Gathering

In *active information gathering*, we interact directly with a system to learn more about it. We might, for example, conduct scans to find open ports on the target or to determine what services are running. Each system or running service that we discover gives us another opportunity for exploitation.

But beware: if you get careless during active information gathering, you might be nabbed by an intrusion detection system (IDS) or intrusion prevention system (IPS)—not a good outcome for the covert penetration tester.

Port Scanning with Nmap

Having identified the target IP range and *trustedsec.com*'s IP address with passive information gathering, we can begin to scan for open ports on the target by *port scanning*, a process whereby we meticulously connect to ports on the remote host to identify those that are active. (In a larger enterprise, we would have multiple IP ranges to attack instead of only one IP address.)

Nmap is, by far, the most popular port-scanning tool. It integrates with Metasploit quite elegantly, storing scan output in a database backend for later use. Nmap lets you scan hosts to identify the services running on each, any of which might offer a way in.

For this example, let's leave *trustedsec.com* behind and instead use *scanme.nmap.org* (45.33.32.156), a server maintained by the team at Nmap. If you would rather scan your own machine, use one of the virtual machines described in Appendix A. Before we get started, take a quick look at the basic Nmap syntax by entering **`nmap`** from the command line on your Kali machine. You'll see immediately that it has several options, but you'll likely use only a few of them.

One of the most useful Nmap options is `-sS`, which runs a stealth TCP scan that determines whether a specific TCP-based port is open. Another preferred option is `-Pn`, which tells Nmap not to use `ping` to determine whether a system is running; instead, it considers all hosts to be "alive." If you're performing internet-based penetration tests, you should use this flag because most networks don't allow Internet Control Message Protocol (ICMP), which is the protocol that `ping` uses. If you're performing this scan internally, you can probably ignore this flag.

Let's run a quick Nmap scan against the *scanme.nmap.org* (45.33.32.156) machine using both the `-sS` and `-Pn` flags:

```
kali@kali:~$ sudo nmap -sS -Pn scanme.nmap.org
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.088s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 989 closed tcp ports (reset)
PORT      STATE    SERVICE
21/tcp    open     ftp
22/tcp    open     ssh
25/tcp    filtered smtp
80/tcp    open     http
135/tcp   filtered msrpc
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
554/tcp   open     rtsp
7070/tcp  open     realserver
9929/tcp  open     nping-echo
31337/tcp open     Elite
```

Nmap reports a list of open ports, along with a description of the associated service for each.

For more detail, try using the `-A` flag. This option will attempt advanced service enumeration and banner grabbing, which may give you even more details about the target system. For example, here's what we'd see if we were to call Nmap with the `-sS` and `-A` flags, using our same target system:

```
kali@kali:~$ sudo nmap -Pn -sS -A scanme.nmap.org
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.075s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 989 closed tcp ports (reset)
PORT      STATE    SERVICE     VERSION
21/tcp    open     tcpwrapped
22/tcp    open     ssh         OpenSSH Ubuntu 2ubuntu (Ubuntu Linux; protocol 2) ❶
| ssh-hostkey:
|   1024 ac:00:a0:1a:82:ff:cc:55:99:dc:67:2b:34:97:6b:75 (DSA) ❷
|   2048 20:3d:2d:44:62:2a:b0:5a:9d:b5:b3:05:14:c2:a6:b2 (RSA)
|   256 96:02:bb:5e:57:54:1c:4e:45:2f:56:4c:4a:24:b2:57 (ECDSA)
|_  256 33:fa:91:0f:e0:e1:7b:1f:6d:05:a2:b0:f1:54:41:56 (ED25519)
25/tcp    filtered smtp
80/tcp    open     http         Apache httpd ((Ubuntu)),
|_http-favicon: Nmap Project
|_http-title: Go ahead and ScanMe!
|_http-server-header: Apache/2.4.7 (Ubuntu)
135/tcp   filtered msrpc
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
554/tcp   open     tcpwrapped
7070/tcp  open     tcpwrapped
9929/tcp  open     nping-echo   Nping echo
31337/tcp open     tcpwrapped
Aggressive OS guesses: Linux 4.4 (95%), Linux 3.2 (93%), DD-WRT v24-sp2 (Linux 2.4.37) (92%) ❸

--snip--

No exact OS matches for host (test conditions non-ideal).
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 443/tcp) ❹
HOP RTT      ADDRESS
1   0.24 ms  192.168.40.2
2   85.56 ms scanme.nmap.org (45.33.32.156)
```

This advanced service-enumeration scan gives us even more information, including the application versions ❶, the SSH host keys used to authenticate the server ❷, a guess about the target's operating system ❸, and a list of the hops made in the network from your machine to the target's machine ❹.

Importing Nmap Results into Metasploit

When you're working with other team members who might be scanning at different times and from different locations, it helps to know how to run

Nmap on its own and then import its results into the Framework. Metasploit lets you easily import a basic Nmap-generated XML export file (created with Nmap's `-oX` option).

Metasploit comes with built-in support for the PostgreSQL database system, which is installed by default in both Kali and the official Metasploit installer. Before you can import files from Nmap into Metasploit, you'll need to start and initialize this database by running the following commands:

```
kali@kali:~$ sudo systemctl start postgresql
kali@kali:~$ sudo msfdb init
```

To verify that PostgreSQL is running, run the following:

```
kali@kali:~$ sudo netstat -antp|grep postgres
tcp   0  0 127.0.0.1:5432          0.0.0.0:*           LISTEN      2091/postgres
tcp6  0  0 ::1:5432                :::*                LISTEN      2091/postgres
```

Using Metasploit with database support requires no additional configuration, as it connects to PostgreSQL once you launch MSFconsole. The very first time you launch MSFconsole, you should see a great deal of output as Metasploit initially creates the necessary database tables.

Metasploit provides several commands that we can use to interact with the database, as you'll see throughout this book. (For a complete list, use the `help` command.) For now, we'll use `db_status` to make sure we're connected correctly:

```
msf > db_status
[*] Connected to msf. Connection type: postgresql.
```

Everything seems to be set up just fine.

Here is an example of how you might use Nmap to scan all the machines in the subnet 192.168.1.0/24 with the `-oX` option, which saves the results to a file called *Results-Subnet1.xml*:

```
kali@kali:~$ sudo nmap -Pn -sS -A -oX Results-Subnet1.xml 192.168.1.0/24
```

After generating the XML file, we use the `db_import` command to import it into our database. We can then verify that the import worked by using the `hosts` command, which lists the system entries that have been created, as shown here:

```
msf > db_import Results-Subnet1.xml
msf > hosts -c address

Hosts
=====

address

-------
```

```
192.168.1.1
192.168.1.10
192.168.1.101
192.168.1.102
192.168.1.109
192.168.1.116
192.168.1.142
192.168.1.152
192.168.1.154
192.168.1.171
192.168.1.155
192.168.1.174
192.168.1.180
192.168.1.181
192.168.1.2
192.168.1.99

msf >
```

This tells us we've successfully imported the output of our Nmap scans into Metasploit, as evidenced by the IP addresses populated when we run the `hosts` commands.

Performing TCP Idle Scans

A more advanced Nmap scan method, the *TCP idle scan*, allows us to scan a target stealthily by spoofing the IP address of another host on the network. For this type of scan to work, we first need to locate an idle host on the network that uses incremental IP IDs (which are used to track packet order). When an idle system uses incremental IP IDs, these IDs become predictable, allowing us to calculate the next one. Whenever a break in the predictability of the IP ID sequence occurs, we know that we have discovered an open port. To learn more about IP ID sequences and this module, visit *https://nmap.org/book/idlescan.html* and *https://www.metasploit.com/modules/auxiliary/scanner/ip/ipidseq*.

However, many operating systems protect against this type of attack by randomizing the IP IDs. Use the Framework's *scanner/ip/ipidseq* module to scan for a host that fits the TCP idle scan requirements:

```
msf > use auxiliary/scanner/ip/ipidseq
msf auxiliary(ipidseq) > show options

Module options:

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   INTERFACE                   no        The name of the interface
❶  RHOSTS                      yes       The target address range or CIDR...
   RPORT      80               yes       The target port
   SNAPLEN    65535            yes       The number of bytes to capture
❷  THREADS    1                yes       The number of concurrent threads
   TIMEOUT    500              yes       The reply read timeout in milliseconds
```

This listing displays the required options for the *ipidseq* scan. One notable option, `RHOSTS` ❶, can take IP ranges (such as 192.168.1.20 to 192.168.1.30); Classless Inter-Domain Routing (CIDR) ranges (such as 192.168.1.0/24); multiple ranges separated by commas (such as 192.168.1.0/24, 192.168.3.0/24); or a text file with one host per line (such as *file:/tmp/hostlist.txt*). All these options give us flexibility in specifying our targets.

The `THREADS` value ❷ sets the number of concurrent threads to use while scanning. By default, all scanner modules have their `THREADS` value initially set to 1. We can raise this value to speed up our scans or lower it to reduce network traffic.

Let's set our values and run the module. In this example, we'll set the value for `RHOSTS` to 192.168.1.0/24, set `THREADS` to 50, and then run the scan:

```
msf auxiliary(ipidseq) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(ipidseq) > set THREADS 50
THREADS => 50
msf auxiliary(ipidseq) > run

[*] 192.168.1.1's IPID sequence class: All zeros
[*] 192.168.1.10's IPID sequence class: Incremental!
[*] Scanned 030 of 256 hosts (011% complete)
[*] 192.168.1.116's IPID sequence class: All zeros
❶ [*] 192.168.1.109's IPID sequence class: Incremental!
[*] Scanned 128 of 256 hosts (050% complete)
[*] 192.168.1.154's IPID sequence class: Incremental!
[*] 192.168.1.155's IPID sequence class: Incremental!
[*] Scanned 155 of 256 hosts (060% complete)
[*] 192.168.1.180's IPID sequence class: All zeros
[*] 192.168.1.181's IPID sequence class: Incremental!
[*] 192.168.1.185's IPID sequence class: All zeros
[*] 192.168.1.184's IPID sequence class: Randomized
[*] Scanned 232 of 256 hosts (090% complete)
[*] Scanned 256 of 256 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(ipidseq) >
```

Judging by the results of our scan, we see several potential idle hosts that we can use to perform idle scanning. We'll try scanning a host using the system at 192.168.1.109 ❶ by using the `-sI` command line flag to specify it:

```
msf auxiliary(ipidseq) > nmap -PN -sI 192.168.1.109 192.168.1.155
[*] exec: nmap -PN -sI 192.168.1.109 192.168.1.155

Idle scan using zombie 192.168.1.109 (192.168.1.109:80); Class: Incremental
Interesting ports on 192.168.1.155:
Not shown: 996 closed|filtered ports
PORT    STATE SERVICE
135/tcp open  msrpc
139/tcp open  netbios-ssn
445/tcp open  microsoft-ds
MAC Address: 00:0C:29:E4:59:7C (VMware)
```

```
Nmap done: 1 IP address (1 host up) scanned in 7.12 seconds
msf auxiliary(ipidseq) >
```

By scanning the idle host, we were able to discover a few open ports on our target system without sending a single packet to the system for our IP address.

Running Nmap from MSFconsole

Now that we've performed advanced reconnaissance on our target, let's connect Nmap with Metasploit. To do this, we just make sure our database is connected:

```
msf > db_status
```

We should be able to enter the `db_nmap` command from within MSFconsole to run Nmap and have its results automatically stored in our new database:

```
msf > db_nmap -sS -A 10.10.11.129

[*] Nmap: Starting Nmap( https://nmap.org )
[*] Nmap: Nmap scan report for 10.10.11.129
[*] Nmap: Host is up (0.023s latency).
[*] Nmap: Not shown: 987 filtered tcp ports (no-response)
[*] Nmap: PORT      STATE SERVICE     VERSION
[*] Nmap: 53/tcp ❶  open  domain      Simple DNS Plus
[*] Nmap: 80/tcp    open  http        Microsoft IIS httpd 10.0 ❷
[*] Nmap: |_http-server-header: Microsoft-IIS/10.0
[*] Nmap: | http-methods:
[*] Nmap: |_  Potentially risky methods: TRACE
[*] Nmap: |_http-title: Search &mdash; Just Testing IIS
[*] Nmap: 88/tcp   open  kerberos-sec  Microsoft Windows Kerberos
[*] Nmap: 135/tcp  open  msrpc         Microsoft Windows RPC
[*] Nmap: 139/tcp  open  netbios-ssn   Microsoft Windows netbios-ssn
[*] Nmap: 389/tcp  open  ldap          Microsoft Windows Active Directory LDAP
[*] Nmap: | ssl-cert: Subject: commonName=research
[*] Nmap: 443/tcp  open  ssl/http      Microsoft IIS httpd 10.0
[*] Nmap: | ssl-cert: Subject: commonName=research
[*] Nmap: |_http-server-header: Microsoft-IIS/10.0
[*] Nmap: | tls-alpn:
[*] Nmap: |_  http/1.1
[*] Nmap: | http-methods:
[*] Nmap: |_  Potentially risky methods: TRACE
[*] Nmap: |_http-title: Search &mdash; Just Testing IIS
[*] Nmap: 445/tcp  open  microsoft-ds?
[*] Nmap: 464/tcp  open  kpasswd5?
[*] Nmap: 593/tcp  open  ncacn_http    Microsoft Windows RPC over HTTP 1.0
[*] Nmap: 636/tcp  open  ssl/ldap      Microsoft Windows Active Directory LDAP
[*] Nmap: No OS matches for host
[*] Nmap: Network Distance: 2 hops
[*] Nmap: Service Info: Host: RESEARCH; OS: Windows; CPE: cpe:/o:microsoft:windows
[*] Nmap: Host script results:
[*] Nmap: | smb2-security-mode:
```

```
[*] Nmap: |   3.1.1:
[*] Nmap: |_    Message signing enabled and required
[*] Nmap: TRACEROUTE (using port 135/tcp)
[*] Nmap: HOP RTT      ADDRESS
[*] Nmap: 1   22.96 ms 10.10.14.1
[*] Nmap: 2   22.95 ms 10.10.11.129
[*] Nmap: OS and Service detection performed. Please report any incorrect results... ❸
[*] Nmap: Nmap done: 1 IP address (1 host up) scanned in 108.13 seconds
```

We scanned only one system in this example, but you can specify multiple IPs using CIDR notation or ranges (for example, 192.168.1.1/24 or 192.168.1.1–254). If you would like to try this yourself, you can scan *scanme.nmap.org* (45.33.32.156) or one of the machines you set up in Appendix A.

Notice a series of open ports ❶, software versions ❷, and even a prediction about the target's operating system. In this scan, Nmap was not able to determine the operating system ❸, but sometimes you'll get lucky.

To check that the results from the scan are stored in the database, we run the `services` command:

```
msf > services
Services
========

host          port   proto  name          state     info
----          ----   -----  ----          -----     ----
10.0.1.10     62078  tcp    tcpwrapped    open
10.10.11.129  53     tcp    domain        open      Simple DNS Plus
10.10.11.129  80     tcp    http          open      Microsoft IIS httpd 10.0
10.10.11.129  88     tcp    kerberos-sec  open      Microsoft Windows Kerberos
10.10.11.129  135    tcp    msrpc         open      Microsoft Windows RPC
10.10.11.129  139    tcp    netbios-ssn   open      Microsoft Windows netbios...
10.10.11.129  389    tcp    ldap          open      Microsoft Windows Active...
10.10.11.129  443    tcp    ssl/http      open      Microsoft IIS httpd 10.0
10.10.11.129  445    tcp    microsoft-ds  open
10.10.11.129  464    tcp    kpasswd5      open
10.10.11.129  593    tcp    ncacn_http    open      Microsoft Windows RPC over HTTP 1.0
10.10.11.129  636    tcp    ssl/ldap      open      Microsoft Windows Active...
```

We're beginning to develop a picture of our target and exposed ports for use as potential attack vectors.

Port Scanning with Metasploit

In addition to its ability to use third-party scanners, Metasploit has several port scanners built into its auxiliary modules that directly integrate with most aspects of the Framework. In later chapters, we'll leverage compromised systems to scan and attack other systems; this process, often called *pivoting*, allows us to use internally connected systems to route traffic to a network that would otherwise be inaccessible.

For example, suppose you compromise a system behind a firewall that is using Network Address Translation (NAT). The system behind the

NAT-based firewall uses private IP addresses, which you cannot contact directly from the internet. If you use Metasploit to compromise a system behind a NAT firewall, you might be able to use that compromised internal system to pass traffic (or pivot) to internally hosted and private IP-based systems and penetrate the network farther behind the firewall.

To see the list of port-scanning tools the Framework offers, enter the following:

```
msf > search portscan
```

Let's perform an example scan of a single host using Metasploit's SYN port scanner. In the following listing, we set `RHOSTS` to 192.168.1.155, set `THREADS` to 50, and then run the scan:

```
msf > use auxiliary/scanner/portscan/syn
msf auxiliary(syn) > set RHOSTS 192.168.1.155
RHOSTS => 192.168.1.155
msf auxiliary(syn) > set THREADS 50
THREADS => 50
msf auxiliary(syn) > run
[*] TCP OPEN 192.168.1.155:135
[*] TCP OPEN 192.168.1.155:139
[*] TCP OPEN 192.168.1.155:445
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(syn) >
```

From the results, you can see that ports 135, 139, and 445 are open on IP address 192.168.1.155.

Targeted Scanning

When you are conducting a penetration test, there is no shame in looking for an easy win. A *targeted scan* looks for specific operating systems, services, program versions, or configurations that are known to be exploitable and that provide an easy door into a target network. Rapid7 maintains a repository of verified scanner and exploit modules (*https://www.rapid7.com/db/?q=&type=metasploit*). It's a good idea to start with the newest scanners.

Scanning for Server Message Block

Metasploit can scour a network and attempt to identify versions of Microsoft Windows using its *smb_version* module. This scanner relies on detecting Server Message Block (SMB), a common file-sharing protocol.

NOTE *If you're not familiar with SMB, study up a bit before you continue. Here is a great resource from the team at Microsoft on some of the fundamentals of SMB:* https://docs.microsoft.com/en-us/windows/win32/fileio/microsoft-smb-protocol-and-cifs-protocol-overview.

We run the module, list our options, set RHOSTS, and begin scanning:

```
msf > use auxiliary/scanner/smb/smb_version
msf auxiliary(smb_version) > show options

Module options (auxiliary/scanner/smb/smb_version):

   Name       Current Setting  Required  Description
   ----       ---------------  --------  -----------
   RHOSTS                      yes       The target address range or CIDR...
   THREADS    1                yes       The number of concurrent threads

msf auxiliary(smb_version) > set RHOSTS 10.10.11.129
RHOSTS => 10.10.11.129
msf auxiliary(smb_version) > run

[*] 10.10.11.129:445      - SMB Detected (compression capabilities:) (encryption capabilities:
AES-128-CCM) (signatures:optional) (guid:{e76d4bf1-3d3c-45e7-aec6-08f7be28070c})
(authentication domain:SEARCH)
[*] 10.10.11.129:         - Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

The *smb_version* scanner has detected the preferred dialect, encryption capabilities, and other properties of the SMB service running on this machine. Because we were scanning only one system, we left THREADS set to 1. If we had been scanning several systems, such as a class C subnet range, we might have considered upping THREADS using the `set THREADS` *number* option.

The results of this scan are stored in the Metasploit database for use at a later time and can be accessed with the `hosts` command:

```
msf auxiliary(smb_version) > hosts -c address,os_flavor,vulns,svcs,workspace

Hosts
=====

address       os_flavor  vulns  svcs  workspace
-------       ---------  -----  ----  ---------
10.10.11.129              1      13    default
msf auxiliary(smb_version) >
```

This is a great way to quickly and quietly target hosts that are likely to be more vulnerable when our goal is to avoid being noticed. We have discovered the system has a vulnerability. We can use the `vulns` command to find more information about that vulnerability:

```
msf auxiliary(scanner/smb/smb_version) > vulns

Vulnerabilities
===============
```

```
Host          Name                        References
----          ----                        ----------
10.10.11.129  SMB Signing Is Not Required  URL...
```

We'll discuss exploiting such vulnerabilities in later chapters.

Hunting for Poorly Configured Microsoft SQL Servers

Poorly configured Microsoft SQL Server (MS SQL) installations may provide an initial way into a target network. In fact, some system administrators don't even realize that they have MS SQL servers installed on their workstations at all, because the service is installed as a prerequisite for some common software, such as Microsoft Visual Studio. These installations may be unused, unpatched, or never even configured.

When MS SQL is installed, it listens by default either on TCP port 1433 or on a random dynamic TCP port. If MS SQL is listening on a dynamic port, simply query UDP port 1434 to discover which one. Of course, Metasploit has a module that can make use of this feature: *mssql_ping*.

Because *mssql_ping* uses UDP, it can be quite slow to run across several subnets due to timeouts. But on a local LAN, setting `THREADS` to 255 will greatly speed up the scan. As Metasploit finds MS SQL servers, it should display all the details it can extract from them, including (and perhaps most importantly) the TCP port on which the server is listening.

Here's how you might run an *mssql_ping* scan, which includes starting the scan, listing and setting options, and viewing the results:

```
msf > use auxiliary/scanner/mssql/mssql_ping
msf auxiliary(mssql_ping) > show options

Module options (auxiliary/scanner/mssql/mssql_ping):

   Name                Current Setting  Required  Description
   ----                ---------------  --------  -----------
   PASSWORD                             no        The password for the specified username
   RHOSTS                               yes       The target address range or CIDR identifier
   TDSENCRYPTION       false            yes       Use TLS/SSL for TDS data "Force Encryption"
   THREADS             1                yes       The number of concurrent threads
   USERNAME            sa               no        The username to authenticate as
   USE_WINDOWS_AUTHENT false            yes       Use windows authentication

msf auxiliary(mssql_ping) > set RHOSTS 10.10.1.0/24
RHOSTS => 10.10.1.0/24
msf auxiliary(mssql_ping) > set THREADS 255
THREADS => 255
msf auxiliary(mssql_ping) > run
  [*] 128.143.124.123:      - SQL Server information for 10.10.1.123:
  [+] 128.143.124.123:      -    ServerName      = REALESTATEFILE
  [+] 128.143.124.123:      -    InstanceName    = SQLEXPRESS
  [+] 128.143.124.123:      -    IsClustered     = No
  [+] 128.143.124.123:      -    Version         = 15.0.2000.5
  [+] 128.143.124.123:      -    tcp             = 49741
```

Not only does the scanner locate an MS SQL server but it also identifies the instance name, the SQL server version, and the TCP port number on which it is listening. Just think of how much time this targeted scan for SQL servers would save over running Nmap against all ports on all machines in a target subnet in search of the elusive TCP port.

Scanning for S3 Buckets

If you are evaluating a cloud environment, you might also want to scan for Amazon Simple Storage Service (S3) buckets, a form of cloud storage. If an S3 bucket has been configured incorrectly, it might leak information to an attacker. S3Scanner (*https://github.com/sa7mon/S3Scanner*) is a great tool for scanning S3 buckets. You can install S3Scanner on your Kali machine using `pip3`:

```
kali@kali:~$ sudo pip3 install s3scanner
```

We'll scan the *http://flaws.cloud* site created by Scott Piper. This intentionally vulnerable site and its sibling, *http://flaws2.cloud*, are great resources for practicing your cloud pentesting skills. Once you've installed the scanner, scan *http://flaws2.cloud* by running the following command:

```
kali@kali:~$ s3scanner scan --bucket flaws2.cloud
http.cloud | bucket_exists | AuthUsers: [], AllUsers: []
```

The scanner has discovered an S3 bucket that is readable by all users, including the public.

Scanning for SSH Server Version

If, during your scanning, you encounter machines running Secure Shell (SSH), you should determine which version is running on the target. SSH is a secure protocol, but researchers have identified vulnerabilities in various implementations of it. You never know when you might get lucky and come across an old machine that hasn't been updated.

You can use the Framework's *ssh_version* module to determine the SSH version running on the target server:

```
msf > use auxiliary/scanner/ssh/ssh_version
msf auxiliary(ssh_version) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(ssh_version) > set THREADS 50
THREADS => 50
msf auxiliary(ssh_version) > run

[*] 192.168.1.1:22, SSH server version: SSH-2.0-OpenSSH_7.4...
[*] Scanned 044 of 256 hosts (017% complete)
[*] 192.168.1.101:22, SSH server version: SSH-2.0-OpenSSH_5.1p1 Debian-3ubuntu1
[*] Scanned 100 of 256 hosts (039% complete)
[*] 192.168.1.153:22, SSH server version: SSH-2.0-OpenSSH_4.3p2 Debian-8ubuntu1
[*] 192.168.1.185:22, SSH server version: SSH-2.0-OpenSSH_4.3
```

This output tells us that a few different servers are running with various patch levels. This information could prove useful if, for example, we wanted to attack a specific version of OpenSSH found with the *ssh_version* scan.

Scanning for FTP Servers

FTP is a complicated and insecure protocol. FTP servers are often the easiest way into a target network, and you should always scan for, identify, and fingerprint any FTP servers running on your target.

Let's look at an example scan for FTP services using the Framework's *ftp_version* module:

```
msf > use auxiliary/scanner/ftp/ftp_version
msf auxiliary(ftp_version) > show options

Module options (auxiliary/scanner/ftp/ftp_version):

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

msf auxiliary(ftp_version) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(ftp_version) > set THREADS 255
THREADS => 255
msf auxiliary(ftp_version) > run

[*] 192.168.1.155:21 FTP Banner: Minftpd ready
```

The scanner successfully identified an FTP server. Now let's see if this FTP server allows anonymous logins using the Framework's *anonymous* module:

```
msf > use auxiliary/scanner/ftp/anonymous
msf auxiliary(anonymous) > set RHOSTS 192.168.1.155
RHOSTS => 192.168.1.155
msf auxiliary(anonymous) > set THREADS 50
THREADS => 50
msf auxiliary(anonymous) > run

[*] Scanned 045 of 256 hosts (017% complete)
[*] 192.168.1.155:21 Anonymous READ/WRITE (220 Minftpd ready)
```

The scanner reports that anonymous access is allowed and that anonymous users have both read and write access to the server; in other words, we have full access to the remote system and the ability to upload or download any file that can be accessed by the FTP server software.

Sweeping for Simple Network Management Protocol

Simple Network Management Protocol (SNMP) is typically used in network devices to report information such as bandwidth utilization and collision rates. However, some operating systems also have SNMP servers that can provide information such as CPU utilization, free memory, and other system-specific details.

Convenience for the system administrator can be a gold mine for the penetration tester, and accessible SNMP servers can offer considerable information about a specific system or even make it possible to compromise a remote device. If, for instance, you can get the read/write SNMP community string for a Cisco router, you can download the router's entire configuration, modify it, and upload it back to the router. (*Community strings* are essentially passwords used to query a device for information or to write configuration information to the device.)

The Metasploit Framework includes a built-in auxiliary module called *snmp_enum* that is designed specifically for SNMP sweeps. Before you start the scan, keep in mind that the read-only (RO) and read/write (RW) community strings will play an important role in the type of information you will be able to extract from a given device. On Windows-based devices configured with SNMP, you can often use the RO or RW community strings to extract patch levels, running services, usernames, uptime, routes, and other information that can make things much easier for you during a pentest.

To gain access to a switch, you'll first need to attempt to find its community strings. After you guess the community strings, some versions of SNMP will allow anything from excessive information disclosure to full system compromise. SNMPv1 and v2 are inherently flawed protocols. SNMPv3, which incorporates encryption and better check mechanisms, is significantly more secure.

The Framework's *snmp_login* module will attempt to guess community strings by sending entries in a wordlist to one IP address or a range of IP addresses:

```
msf > use auxiliary/scanner/snmp/snmp_login
msf auxiliary(snmp_login) > set RHOSTS 192.168.1.0/24
RHOSTS => 192.168.1.0/24
msf auxiliary(snmp_login) > set THREADS 50
THREADS => 50
msf auxiliary(snmp_login) > run

[*] >> progress (192.168.1.0-192.168.1.255) 0/30208...
[*] 192.168.1.2 'public' 'GSM7224 L2 Managed Gigabit Switch'
[*] 192.168.1.2 'private' 'GSM7224 L2 Managed Gigabit Switch'
[*] Auxiliary module execution completed
msf auxiliary(snmp_login) >
```

A quick Google search for GSM7224, listed in the output, tells us that the scanner has found both the public and private community strings for a NETGEAR switch. This result, believe it or not, has not been staged for this book. These are the default factory settings for this switch.

You will encounter many jaw-dropping situations like these throughout your pentesting career because many administrators simply attach devices to a network with all their defaults still in place. The situation is even scarier when you find these devices accessible from the internet within a large corporation.

Writing a Custom Scanner

It can be useful to write your own scanner during security assessments because many applications and services lack scanner modules in Metasploit. Thankfully, the Framework has many features to help you build a custom scanner, including support for proxies, the Secure Sockets Layer (SSL) protocol, and threading.

Metasploit Framework scanner modules often include features using various *mixins*, which are portions of code with predefined functions. The `Auxiliary::Scanner` mixin overloads the auxiliary `run` method; calls the `run_host(ip)`, `run_range(range)`, or `run_batch(batch)` methods; and then processes the IP addresses you specified for scanning. While we'll cover auxiliary modules in more detail in Chapter 11, we'll demonstrate here how to leverage `Auxiliary::Scanner` to call additional built-in Metasploit functionality. Let's write some code.

The following is a Ruby script for a simple TCP scanner that connects to a remote host on a default port of 12345 and, upon connecting, sends the message "HELLO SERVER," receives the server response, and prints it out along with the server's IP address:

```
#Metasploit
require 'msf/core'
class Metasploit3 < Msf::Auxiliary
  ❶ include Msf::Exploit::Remote::Tcp
  ❷ include Msf::Auxiliary::Scanner
    def initialize
        super(
            'Name'           => 'My custom TCP scan',
            'Version'        => '$Revision: 1 $',
            'Description'    => 'My quick scanner',
            'Author'         => 'Your name here',
            'License'        => MSF_LICENSE
        )
        register_options(
            [
              ❸ Opt::RPORT(12345)
            ], self.class)
    end

    def run_host(ip)
        connect()
      ❹ sock.puts('HELLO SERVER')
        data = sock.recv(1024)
      ❺ print_status("Received: #{data} from #{ip}")
```

```
        disconnect()
    end
end
```

If you aren't familiar with Ruby, you may want to take some time to familiarize yourself with the language and revisit this section later.

This simple scanner uses the `Msf::Exploit::Remote::Tcp` mixin ❶ to handle the TCP networking. The `Msf::Auxiliary::Scanner` mixin exposes the various settings that are required for scanners within the Framework ❷. This scanner is configured to use the default port of 12345 ❸, and upon connecting to the server, it sends a message ❹, receives the reply from the server, and then prints it out to the screen along with the server IP address ❺.

We have saved this custom script under *modules/auxiliary/scanner/* as *simple_tcp.rb*. The saved location is important in Metasploit. For example, if the module were saved under *modules/auxiliary/scanner/http/*, it would show up in the modules list as *scanner/http/simple_tcp*.

To test this rudimentary scanner, we set up a Netcat listener on port 12345 and pipe in a text file to act as the server response:

```
kali@kali:/$ echo "Hello Metasploit" > banner.txt
kali@kali:/$ nc -lvnp 12345 < banner.txt
listening on [any] 12345...
```

Next, we load MSFconsole, select our scanner module, set its parameters, and run it to see if it works:

```
msf > use auxiliary/scanner/simple_tcp
msf auxiliary(simple_tcp) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   RHOSTS                    yes       The target address range or CIDR identifier
   RPORT    12345            yes       The target port
   THREADS  1                yes       The number of concurrent threads

msf auxiliary(simple_tcp) > set RHOSTS 192.168.1.101
RHOSTS => 192.168.1.101
msf auxiliary(simple_tcp) > run

[*] Received: Hello Metasploit from 192.168.1.101
[*] Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
msf auxiliary(simple_tcp) >
```

Although this is only a simple example, the level of versatility afforded by the Metasploit Framework can be of great assistance when you need to get some custom code up and running quickly in the middle of a pentest. Hopefully, this example demonstrates the power of the Framework and modular code.

Wrapping Up

In this chapter, you learned how to leverage the Metasploit Framework for intelligence gathering, as outlined in the PTES. Intelligence gathering takes practice and requires a deep understanding of how an organization operates and how to identify the best potential attack vectors. As with anything, you should adapt and improve your own methodologies throughout your penetration-testing career. Just remember that your main focus for this phase is to learn about the organization you're attacking and its overall footprint. Regardless of whether your work occurs over the internet, on an internal network, wirelessly, or via social engineering, the goals of intelligence gathering will always be the same.

In the next chapter, we'll move on to an important step of the vulnerability analysis phase: automated vulnerability scanning. In later chapters, we will explore more in-depth examples of how to create your own modules, exploits, and scripts.