https://resources.infosecinstitute.com/

Nmap Cheat Sheet: From Discovery to Exploits – Part 1: Introduction to Nmap

As always during reconnaissance, scanning is the initial stage for information gathering.

What is Reconnaissance?
Reconnaissance is to collect as much as information about a target network as possible. From a hacker's perspective, the information gathered is very helpful to make an attack, so to block that type of malicious attempt, generally a penetration tester tries to find the information and to patch the vulnerabilities, if found. This is also called Footprinting. Usually by information gathering, someone can find the below information:

- E-mail Address
- Port no/Protocols
- OS details
- Services Running
- Traceroute information/DNS information
- Firewall Identification and evasion
- And many more…

So for information gathering, scanning is the first part. For scanning, Nmap is a great tool for discovering Open ports, protocol numbers, OS details, firewall details, etc.

Introduction To Nmap
Nmap (Network Mapper) is an open-source tool that specializes in network exploration and security auditing, originally published by Gordon "Fyodor" Lyon. The official website is (http://nmap.org). Nmap is a free and open source (license) utility for network discovery and security auditing. Many systems and network administrators also find it useful for tasks such as network inventory, managing service upgrade schedules, and monitoring host or service uptime.
Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics. It was designed to rapidly scan large networks, but works fine against single hosts. Nmap runs on all major computer operating systems, and official binary packages are available for Linux, Windows, and Mac OS X.

Installation Of Nmap
Nmap has great support for different environments.
Windows: Install from the official site http://nmap.org For Windows, both GUI and command line options are available. The GUI option for Nmap is Zenmap.
Linux (Ubuntu and Debian): Fire the command in the Linux terminal: apt-get install nmap

In the below image, I have already installed Nmap.

```
root@ubuntu:~# apt-get install nmap
Reading package lists... Done
Building dependency tree
Reading state information... Done
nmap is already the newest version.
The following packages were automatically installed and are no longer required:
  gir1.2-timezonemap-1.0 openjdk-7-jre-lib
Use 'apt-get autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

For Red Hat and Fedora based systems: yum install nmap

For Gentoo Linux based systems: emerge nmap

Here, I will show everything in the Linux terminal.

Nmap Scripting Engine
The Nmap Scripting Engine (NSE) is one of Nmap's most powerful and flexible features. It allows users to write (and share) simple scripts to automate a wide variety of networking tasks. Basically these scripts are written in Lua programming language. Generally Nmap's script engine does lots of things, some of them are below:

Network discovery

This is Nmap's bread and butter. Examples include looking up WhoIs data based on the target domain, querying ARIN, RIPE, or APNIC for the target IP to determine ownership, performing identd lookups on open ports, SNMP queries, and listing available NFS/SMB/RPC shares and services.

Vulnerability detection

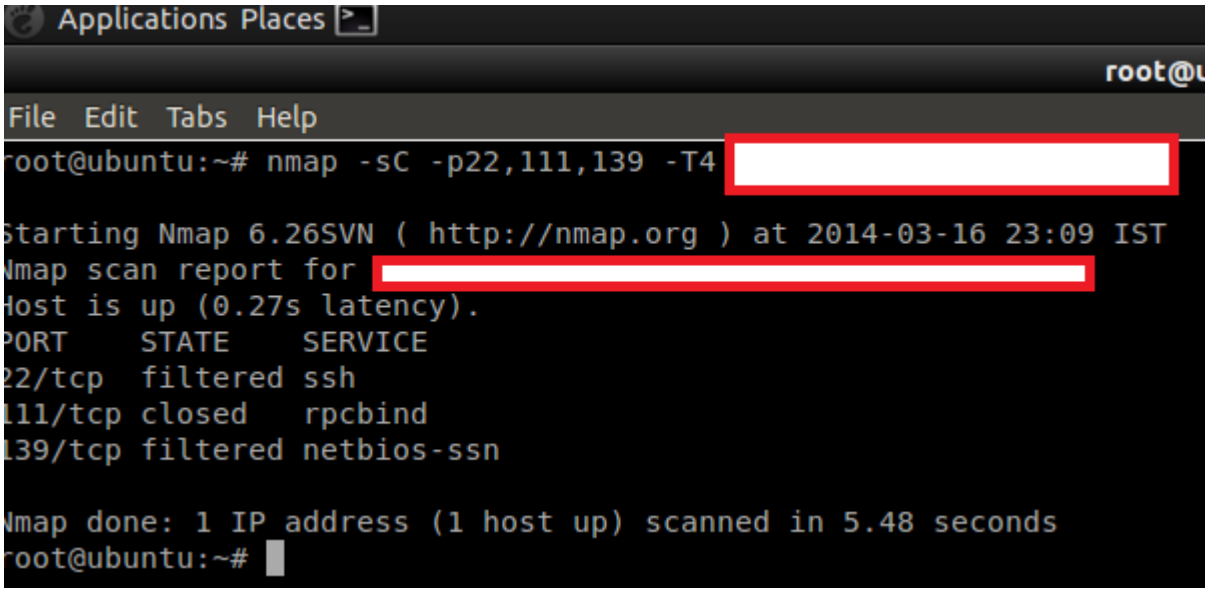
When a new vulnerability is discovered, you often want to scan your networks quickly to identify vulnerable systems before the bad guys do. While Nmap isn't a comprehensive vulnerability scanner, NSE is powerful enough to handle even demanding vulnerability checks. Many vulnerability detection scripts are already available, and they plan to distribute more as they are written.

Backdoor detection

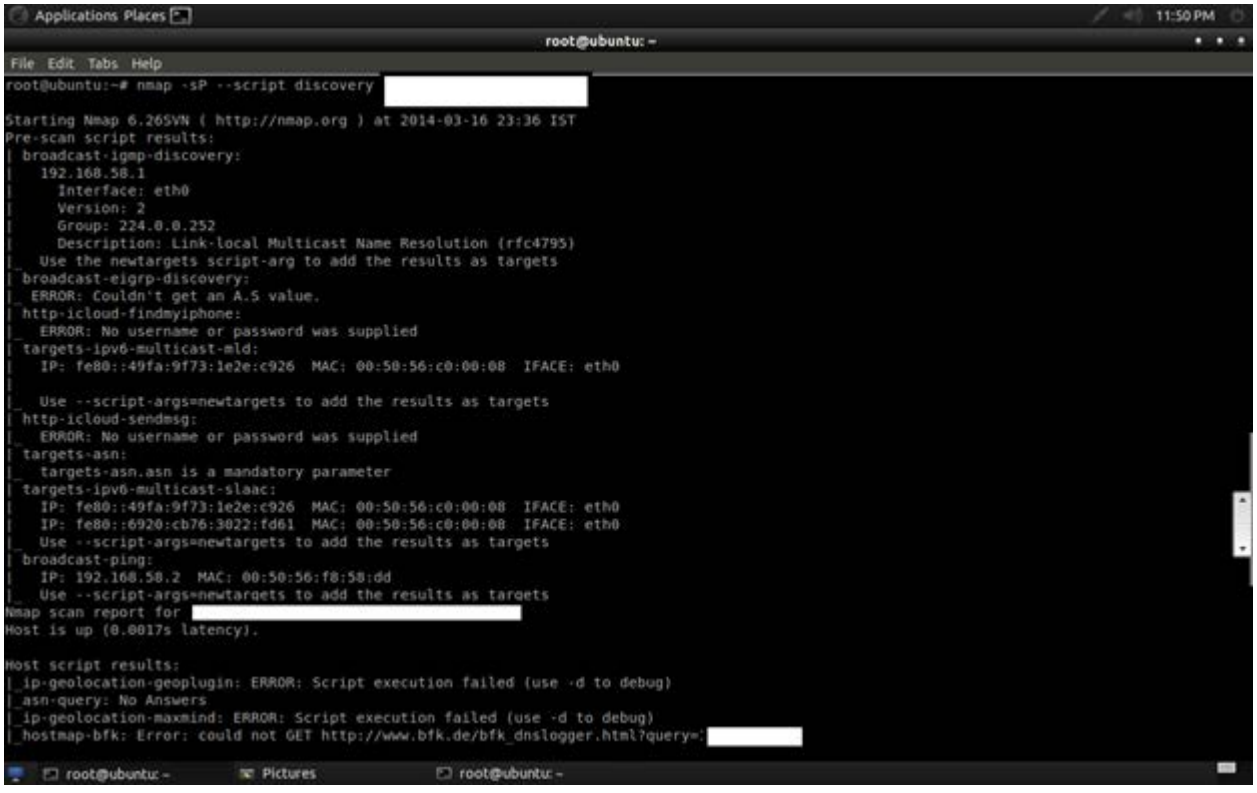
Many attackers and some automated worms leave backdoors to enable later reentry. Some of these can be detected by Nmap's regular expression-based version detection.

Vulnerability exploitation

As a general scripting language, NSE can even be used to exploit vulnerabilities rather than just find them. The capability to add custom exploit scripts may be valuable for some people (particularly penetration testers), though they aren't planning to turn Nmap into an exploitation framework such as Metasploit.
As you can see below, I have used (-sc) options (or –script), which is a default script scan for the target network. You can see we got ssh, rpcbind, netbios-sn but the ports are either filtered or closed, so we can say that may be there are some firewall which is blocking our request. **Later we will discuss how to identify firewalls and try to evade them.**



Now I m going to run a ping scan with discovery mode on (script) so that it will try all possible methods for scanning, that way I will get more juicy information.



As you can see in the image, it is trying all possible methods as per script rules. See the next image for more information.

Can you see the interesting ports and protocols? You can see dns-bruteforce found that host contains some blog, cms, sql, log, mail, and many more. So here we can perform SQL injection, the blog may be WordPress, Joomla, etc., so we can attack for a known CMS vulnerability, and obviously the method will be black-box pentesting.

In the upcoming chapter I will describe how to write your own Nmap script engine, and how to exploit them using Nmap.

Basic Scanning Techniques

So here I will show the basic techniques for scanning network/host. But before that, you should know some basic stuff regarding Nmap status after scanning.

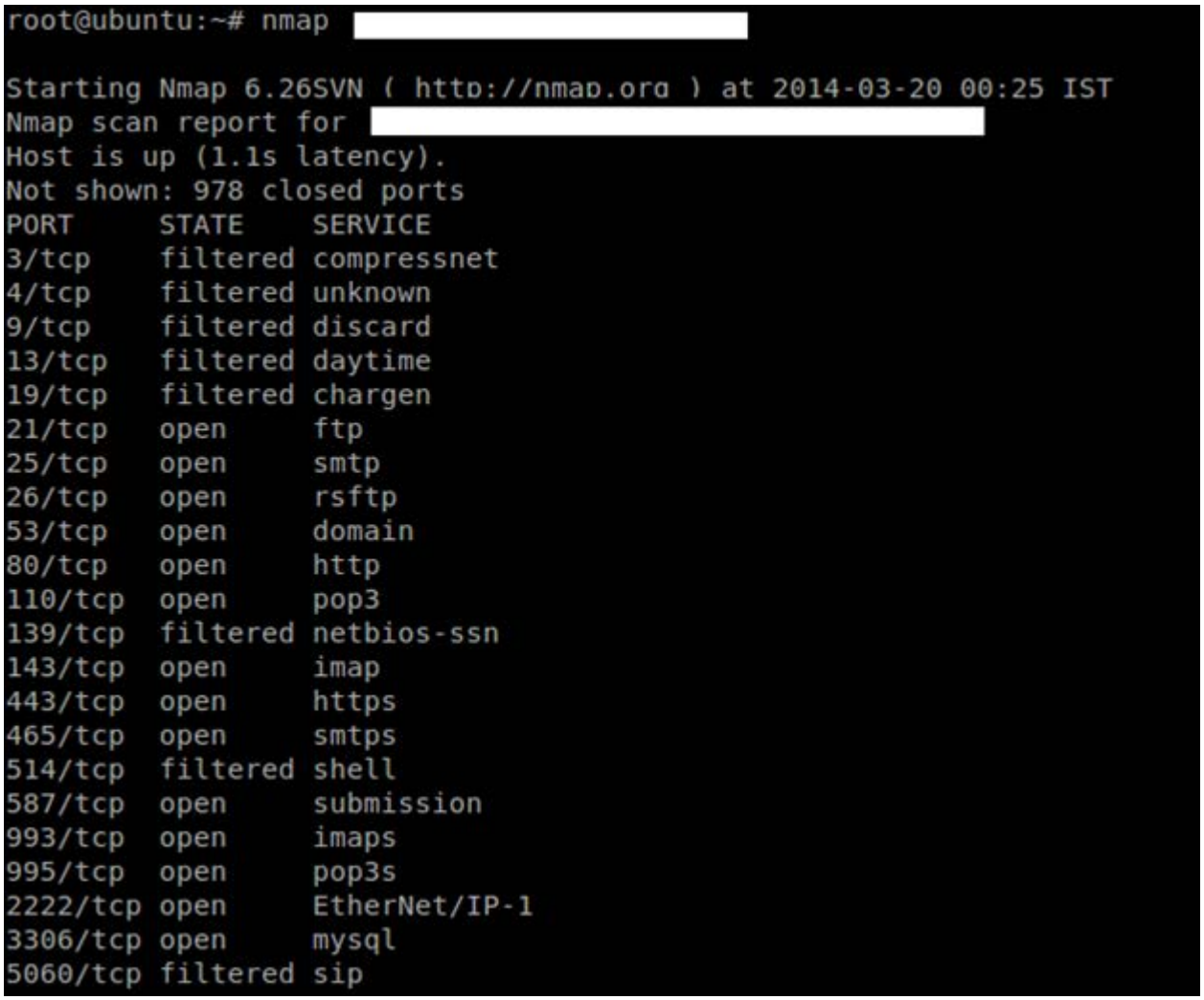
Port Status: After scanning, you may see some results with a port status like filtered, open, closed, etc. Let me explain this.

- Open: This indicates that an application is listening for connections on this port.
- Closed: This indicates that the probes were received but there is no application listening on this port.
- Filtered: This indicates that the probes were not received and the state could not be established. It also indicates that the probes are being dropped by some kind of filtering.
- Unfiltered: This indicates that the probes were received but a state could not be established.
- Open/Filtered: This indicates that the port was filtered or open but Nmap couldn't establish the state.
- Closed/Filtered: This indicates that the port was filtered or closed but Nmap couldn't establish the state.

Let's Scan Hosts

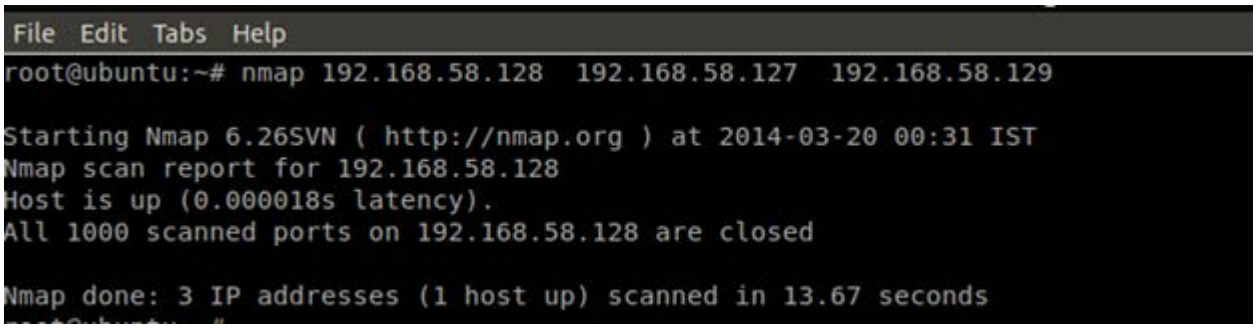
Scan A Single Network

Go to your Nmap (either Windows/Linux) and fire the command: nmap 192.168.1.1(or) host name.

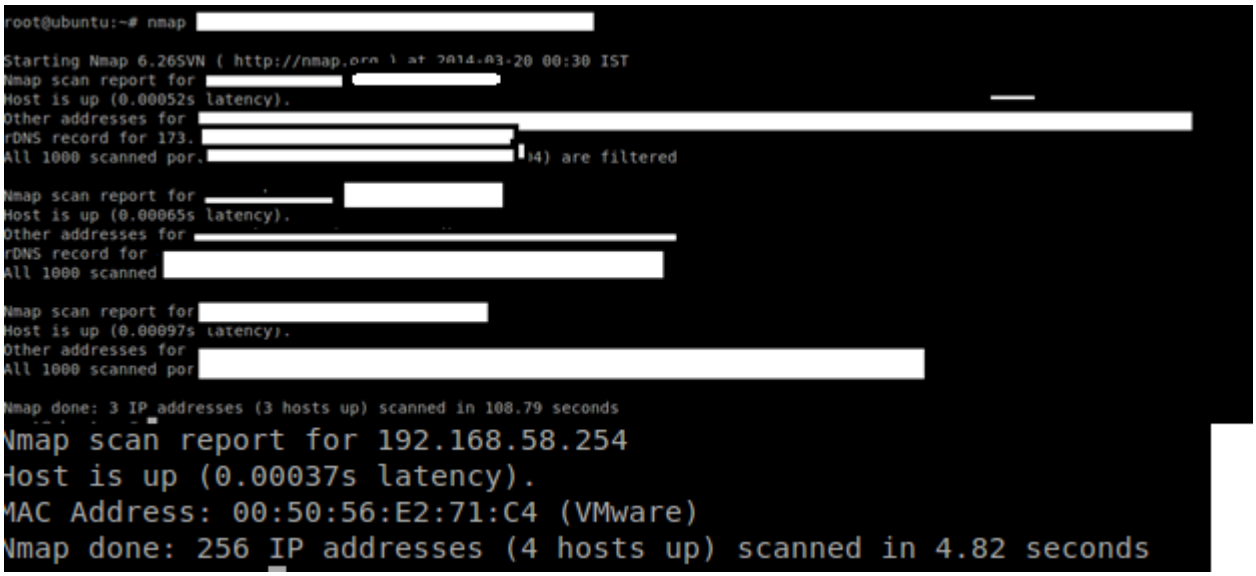


Scan Multiple Network/Targets
In Nmap you can even scan multiple targets for host discovery/information gathering.

Command: map host1 host2 host3 etc….It will work for the entire subnet as well as different IP addresses.



You can also scan multiple website/domain names at a time with the same command. See the below picture. It will convert the domain name to its equivalent IP address and scan the targets.



Scan a Range Of IP address
Command:nmap 192.168.2.1-192.168.2.100

Nmap can also be used to scan an entire subnet using **CIDR (Classless Inter-Domain Routing) notation.**
Usage syntax: nmap [Network/CIDR]
Ex:nmap 192.168.2.1/24

Scan a list of targets
If you have a large number of systems to scan, you can enter the IP address (or host names) in a text file and use that file as input for Nmap on the command line.

syntax: nmap -iL [list.txt]
Scan Random Targets
The **-iR** parameter can be used to select random Internet hosts to scan. Nmap will randomly generate the specified number of targets and attempt to scan them.
syntax: nmap -iR [number of host]
It is not a good habit to do a random scan unless you have been given some project.

The –exclude option is used with Nmap to exclude hosts from a scan.
syntax: nmap [targets] –exclude [host(s)]
ex:nmap 192.168.2.1/24 –exclude 192.168.2.10

Aggressive Scan
The aggressive scan selects most commonly used options within Nmap to try to give a simple alternative to writing long strings. It will also work for traceroute, etc.

Command:nmap –A host

Discovery With Nmap
Discovery with Nmap is very interesting and very helpful for penetration testers. During discovery one can learn about services, port numbers, firewall presence, protocol, operating system, etc. We will discuss one by one.

Don’t Ping

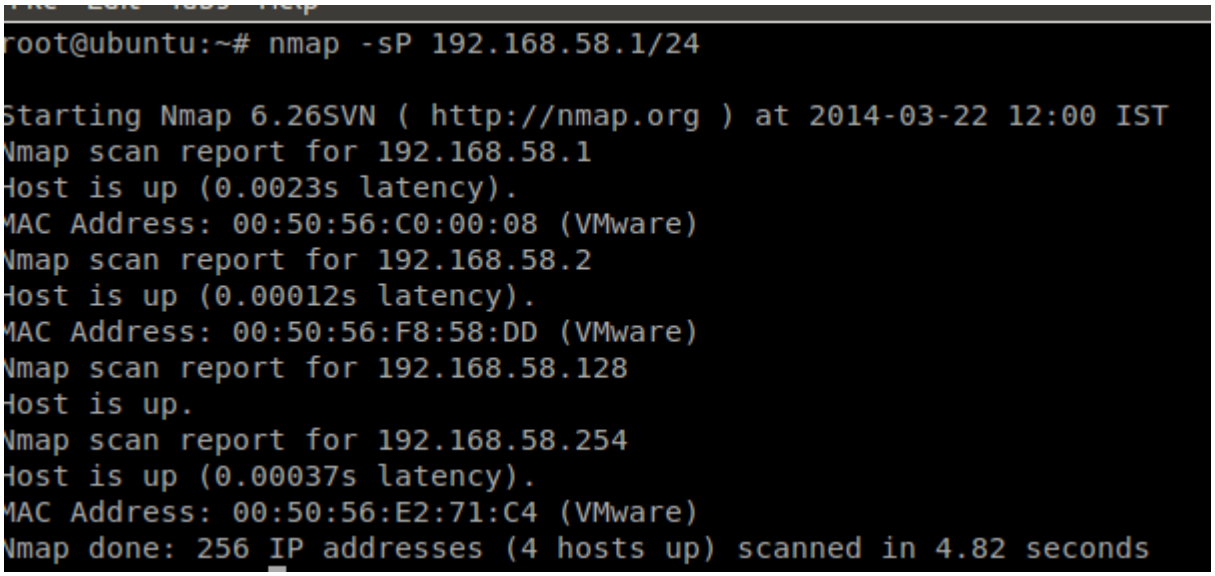
The **-PN** option instructs Nmap to skip the default discovery check and perform a complete port scan on the target. This is useful when scanning hosts that are protected by a firewall that blocks ping probes.
Syntax:nmap –PN Target

```
root@ubuntu:~# nmap -PN [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 11:49 IST
Nmap scan report for [redacted]
Host is up (1.1s latency).
Not shown: 975 closed ports
PORT     STATE    SERVICE
1/tcp    filtered tcpmux
3/tcp    filtered compressnet
4/tcp    filtered unknown
6/tcp    filtered unknown
7/tcp    filtered echo
9/tcp    filtered discard
13/tcp   filtered daytime
17/tcp   filtered qotd
19/tcp   filtered chargen
21/tcp   open     ftp
22/tcp   filtered ssh
25/tcp   open     smtp
26/tcp   open     rsftp
53/tcp   open     domain
80/tcp   open     http
110/tcp  open     pop3
143/tcp  open     imap
443/tcp  open     https
465/tcp  open     smtps
514/tcp  filtered shell
587/tcp  open     submission
993/tcp  open     imaps
995/tcp  open     pop3s
2222/tcp open     EtherNet/IP-1
3306/tcp open     mysql
```

By specifying these options, Nmap will discover the open ports without ping, which is the unpingable system.

Ping Only Scan
The -Sp option is responsible for a ping only scan. It will be more useful when you have a group of IP addresses and you don’t know which one is reachable. By specifying a particular target, you can get even more information, like MAC address.

Syntax:nmap –Sp target

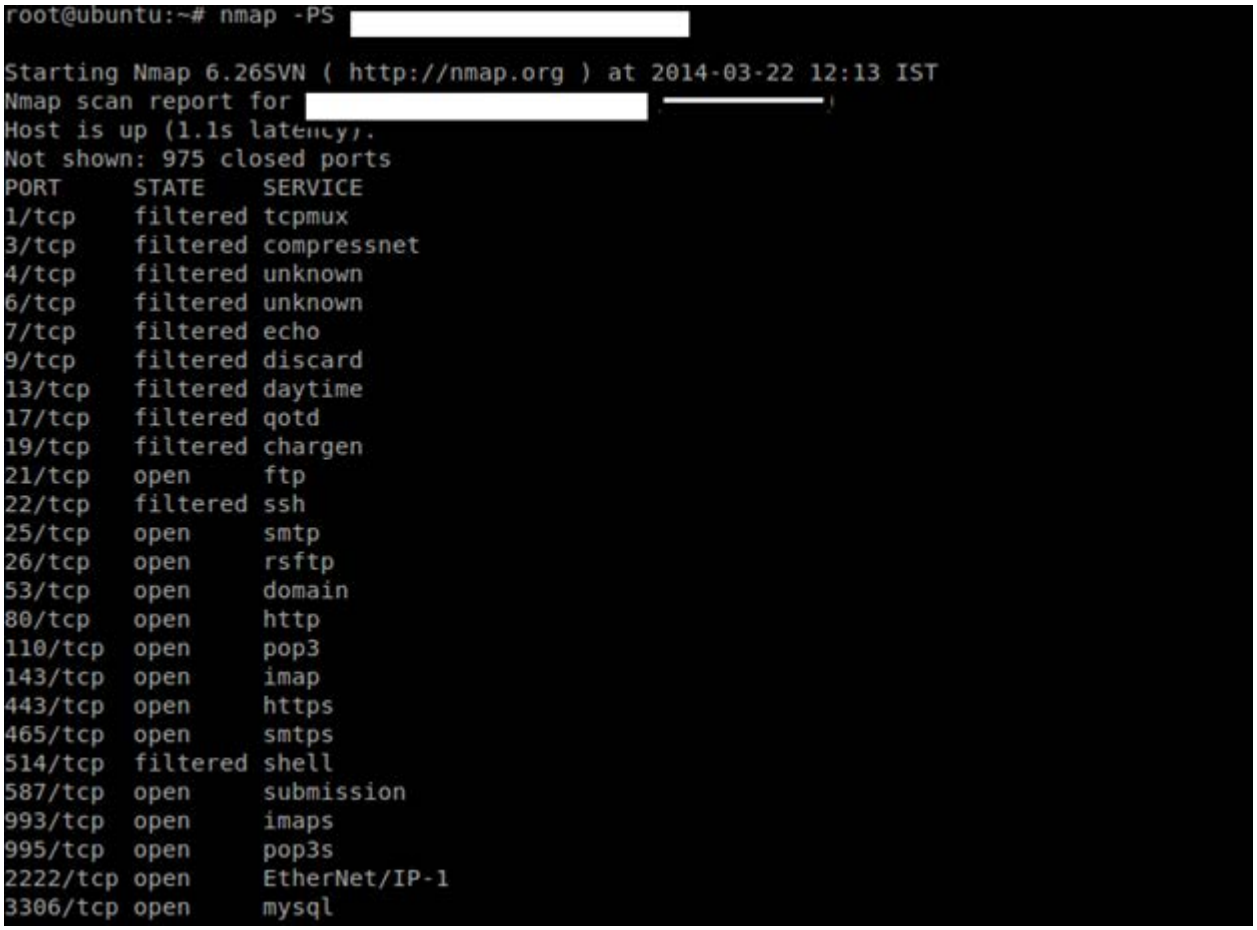


TCP Syn Scan
Before we start, we must know the syn packet.

Basically a syn packet is used to initiate the connection between the two hosts.

The TCP SYN ping sends a SYN packet to the target system and listens for a response. This alternative discovery method is useful for systems that are configured to block standard ICMP pings.

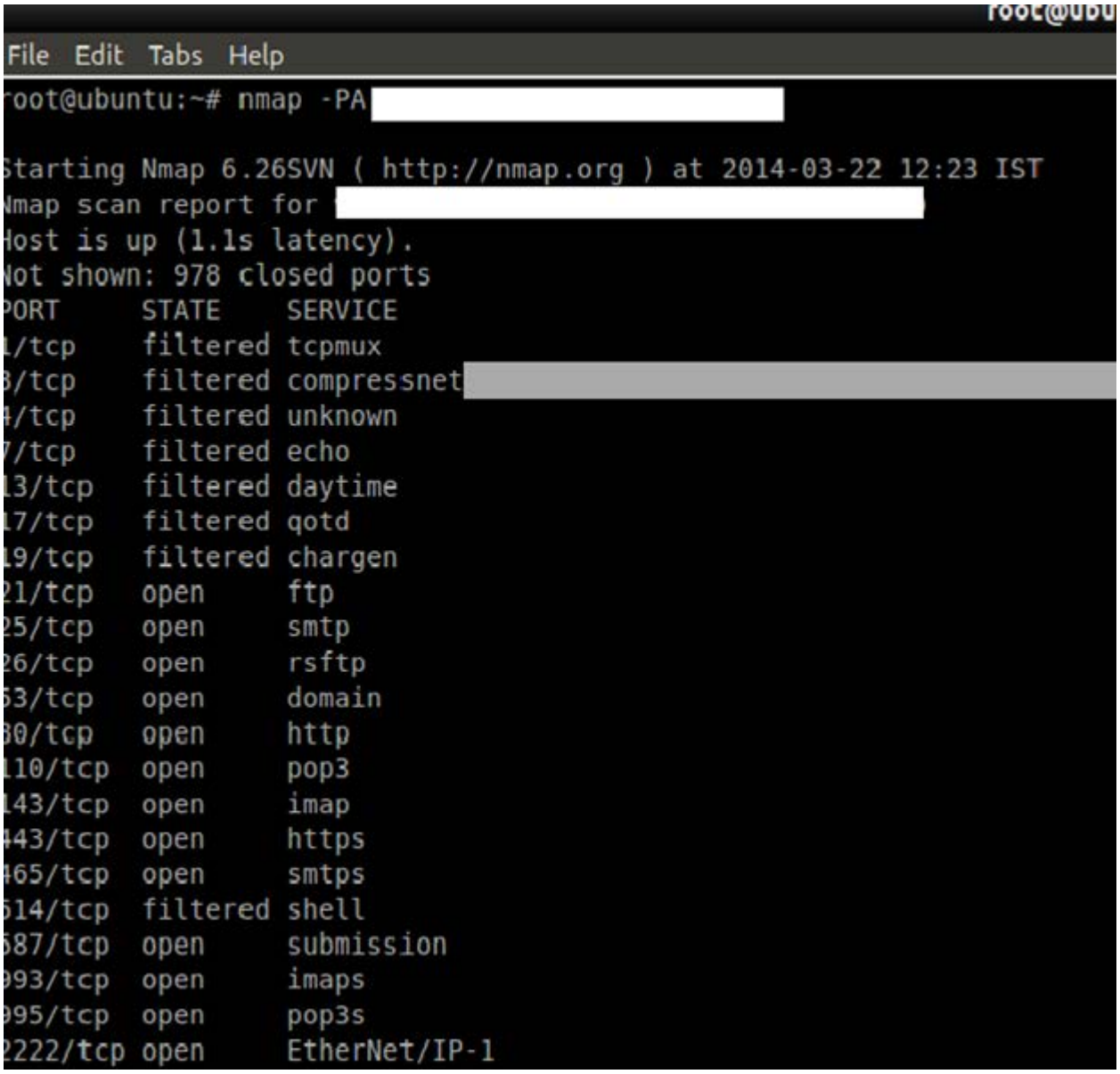
The -**PS** option performs a TCP SYN ping.
Syntax:nmap –PS targets



The default port is port80. You can also specify other ports like –PS22, 23, 25, 443.

TCP Ack Ping Scan
This type of scan will only scan of Acknowledgement(ACK) packet.

The **-PA** performs a TCP ACK ping on the specified target.
The **-PA** option causes Nmap to send TCP ACK packets to the specified hosts.
Syntax:nmap –PA target



This method attempts to discover hosts by responding to TCP connections that are nonexistent in an attempt to solicit a response from the target. Like other ping options, it is useful in situations where standard ICMP pings are blocked.

UDP Ping scan

The –PU scan only on udp ping scans on the target. This type of scan sends udp packets to get a response.

Syntax:nmap –PU target

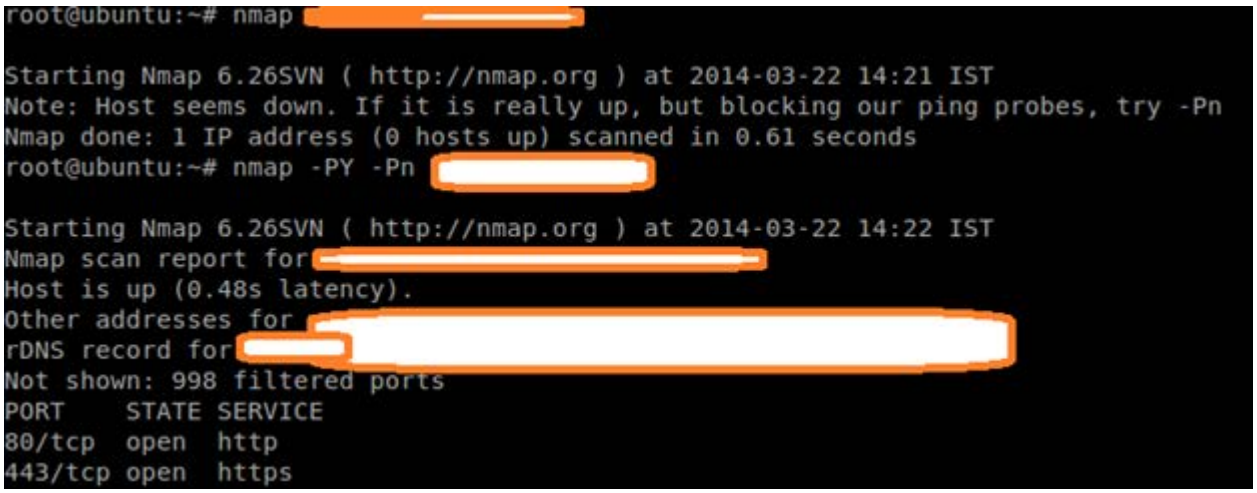
```
root@ubuntu:~# nmap -PU 192.168.58.128

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 12:36 IST
Nmap scan report for 192.168.58.128
Host is up (0.000016s latency).
All 1000 scanned ports on 192.168.58.128 are closed

Nmap done: 1 IP address (1 host up) scanned in 0.43 seconds
root@ubuntu:~# nmap -PU22,80,25,443 192.168.58.128

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 12:36 IST
Nmap scan report for 192.168.58.128
Host is up (0.000015s latency).
All 1000 scanned ports on 192.168.58.128 are closed

Nmap done: 1 IP address (1 host up) scanned in 0.40 seconds
```

You can also specify the port number for scanning, like –PU 22, 80, 25, etc. In the above picture, the target is my LAN's IP, which doesn't have any UDP services.

Sctp init ping

The **-PY** parameter instructs Nmap to perform an SCTP INIT ping. This option sends an SCTP packet containing a minimal INIT chunk. This discovery method attempts to locate hosts using the Stream Control Transmission Protocol (SCTP). SCTP is typically used on systems for IP based telephony.

Syntax:nmap –PY target

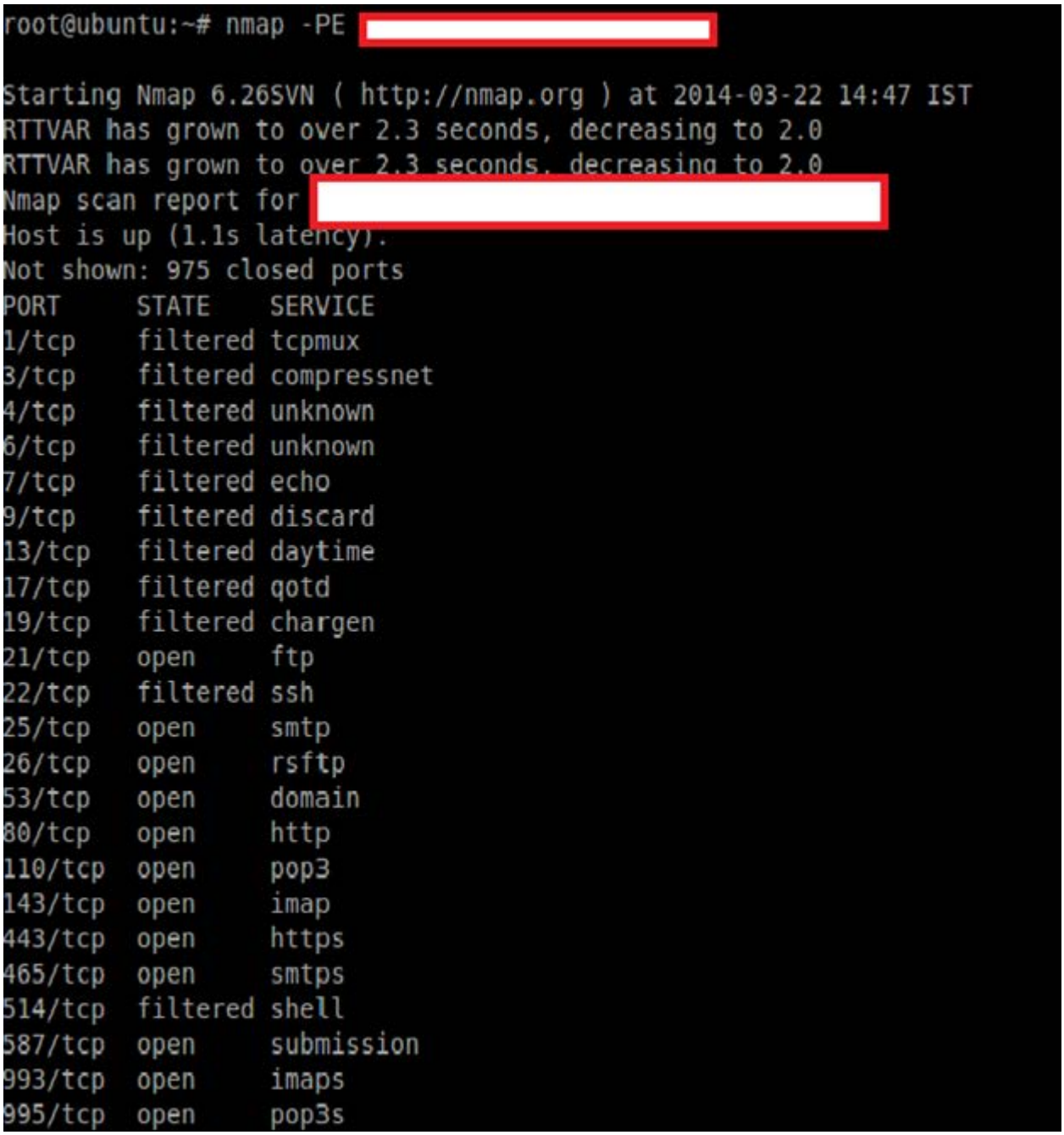


In the picture, though there is no sctp services on the machine, we have to use the –pn option for discovery.

ICMP Echo ping

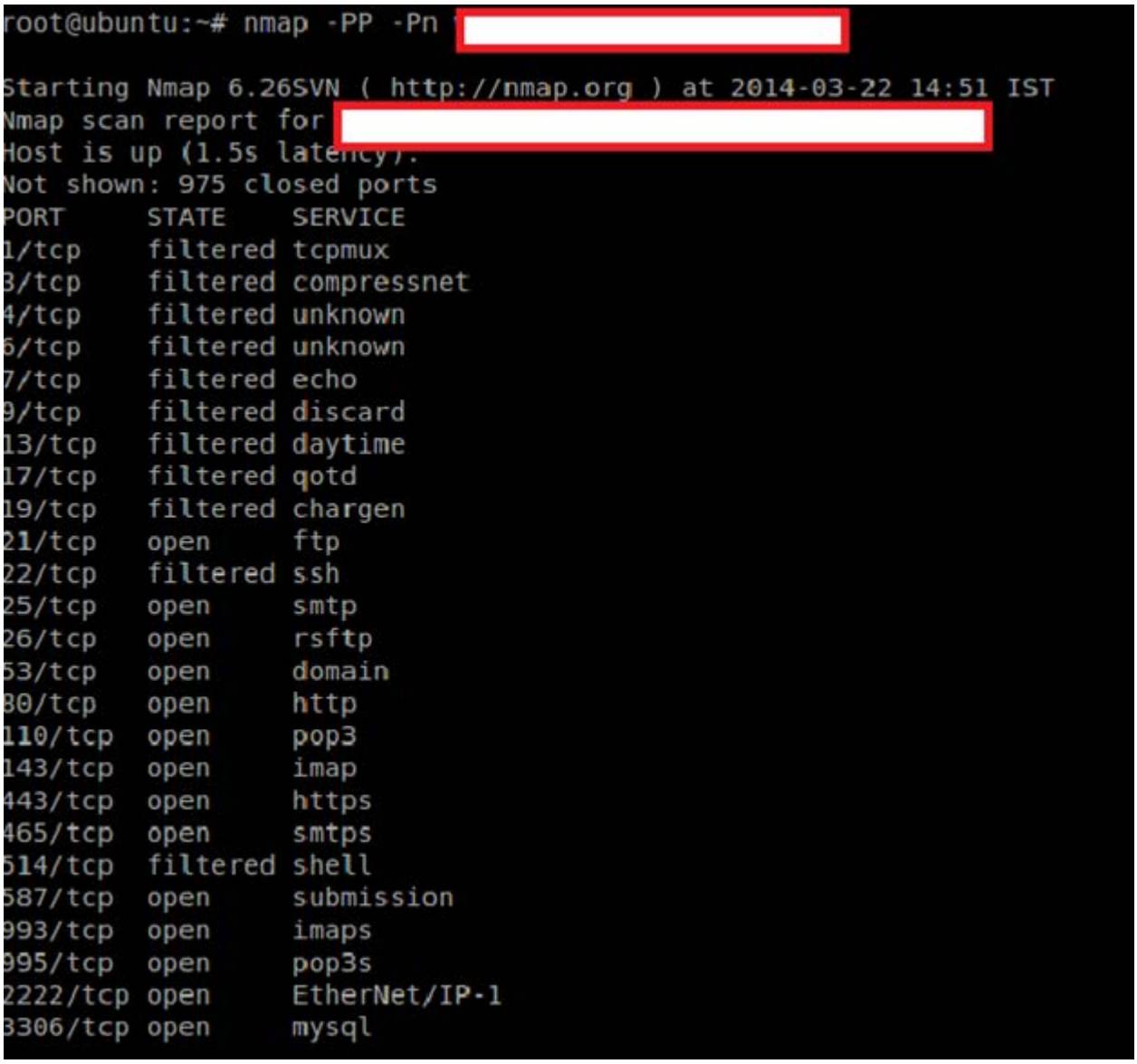
The **-PE** option performs an ICMP (Internet Control Message Protocol) echo ping on the specified system.

Syntax:nmap –PE target



This type of discovery works best on local networks where ICMP packets can be transmitted with few restrictions.

ICMP Timestamp ping
The **-PP** option performs an ICMP timestamp ping.



ICMP Address mask ping
The **-PM** option performs an ICMP address mask ping.
Syntax:nmap –PM target

```
root@ubuntu:~# nmap -PM [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:48 IST
Note: Host seems down. If it is really up, but blocking our ping probes, try
Nmap done: 1 IP address (0 hosts up) scanned in 2.66 seconds
root@ubuntu:~# nmap -PP [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:48 IST
Note: Host seems down. If it is really up, but blocking our ping probes, try
Nmap done: 1 IP address (0 hosts up) scanned in 2.36 seconds
root@ubuntu:~# nmap -PM [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:50 IST
Nmap scan report for 192.168.58.128
Host is up (0.000017s latency).
All 1000 scanned ports on 192.168.58.128 are closed

Nmap done: 1 IP address (1 host up) scanned in 0.45 seconds
```

This unconventional ICMP query (similar to the **-PP** option) attempts to ping the specified host using alternative ICMP registers. This type of ping can occasionally sneak past a firewall that is configured to block standard echo requests.

IP Protocol Ping

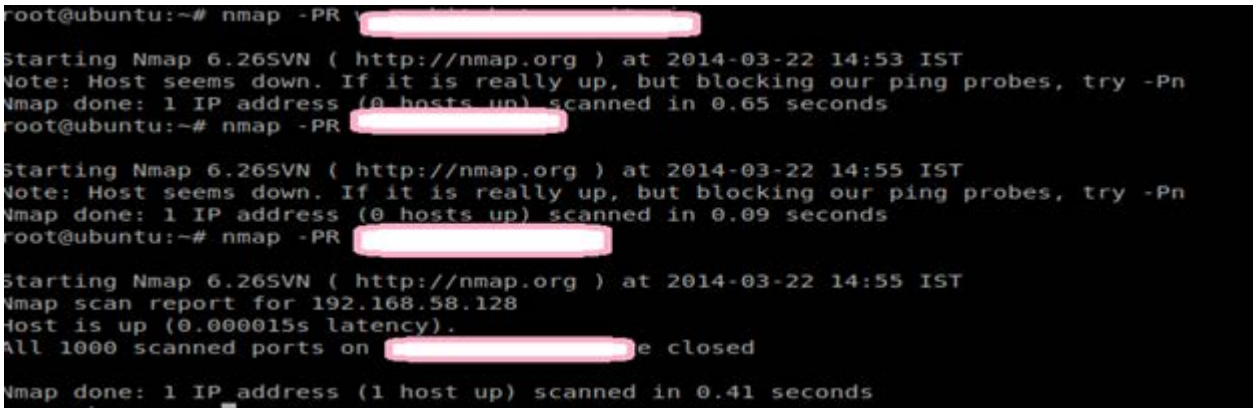
The **-PO** option performs an IP protocol ping.

Syntax:nmap –PO protocol target

```
root@ubuntu:~# nmap -PO [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:52 IST
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
Nmap scan report for [redacted]
Host is up (1.3s latency).
Not shown: 975 closed ports
PORT     STATE    SERVICE
1/tcp    filtered tcpmux
3/tcp    filtered compressnet
4/tcp    filtered unknown
6/tcp    filtered unknown
7/tcp    filtered echo
9/tcp    filtered discard
13/tcp   filtered daytime
17/tcp   filtered qotd
19/tcp   filtered chargen
21/tcp   open     ftp
22/tcp   filtered ssh
25/tcp   open     smtp
26/tcp   open     rsftp
53/tcp   open     domain
80/tcp   open     http
110/tcp  open     pop3
143/tcp  open     imap
443/tcp  open     https
465/tcp  open     smtps
514/tcp  filtered shell
587/tcp  open     submission
993/tcp  open     imaps
995/tcp  open     pop3s
2222/tcp open     EtherNet/IP-1
```

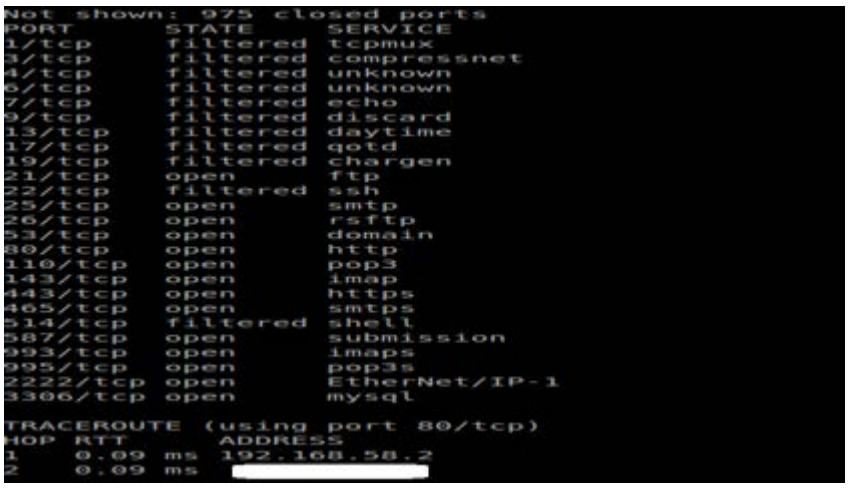
An IP protocol ping sends packets with the specified protocol to the target. If no protocols are specified, the default protocols 1 (ICMP), 2 (IGMP), and 4 (IP-in-IP) are used.

ARP ping
The –PR option is used to perform an arp ping scan. The **-PR** option instructs Nmap to perform an ARP (Address Resolution Protocol) ping on the specified target.
SYTAX: nmap –PR target



The -**PR** option is automatically implied when scanning the local network. This type of discovery is much faster than the other ping methods.
Traceroute

The **–traceroute** parameter can be use to trace the network path to the specified host.
Syntax: nmap –traceroute target



Force Reverse DNS Resolution
The **-R** parameter instructs Nmap to always perform a reverse DNS resolution on the target IP address.
Syntax: nmap –R target

```
root@ubuntu:~# nmap -r [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:59 IST
Nmap scan report for [redacted]
Host is up (1.3s latency).
Not shown: 975 closed ports
PORT      STATE     SERVICE
1/tcp     filtered  tcpmux
3/tcp     filtered  compressnet
4/tcp     filtered  unknown
5/tcp     filtered  unknown
7/tcp     filtered  echo
9/tcp     filtered  discard
13/tcp    filtered  daytime
17/tcp    filtered  qotd
19/tcp    filtered  chargen
21/tcp    open      ftp
22/tcp    filtered  ssh
25/tcp    open      smtp
26/tcp    open      rsftp
53/tcp    open      domain
80/tcp    open      http
110/tcp   open      pop3
143/tcp   open      imap
443/tcp   open      https
465/tcp   open      smtps
514/tcp   filtered  shell
587/tcp   open      submission
993/tcp   open      imaps
995/tcp   open      pop3s
2222/tcp  open      EtherNet/IP-1
3306/tcp  open      mysql
```

The **-R** option is useful when performing reconnaissance on a block of IP addresses, as Nmap will try to resolve the reverse DNS information of every IP address.

Disable Reverse DNS Resolution

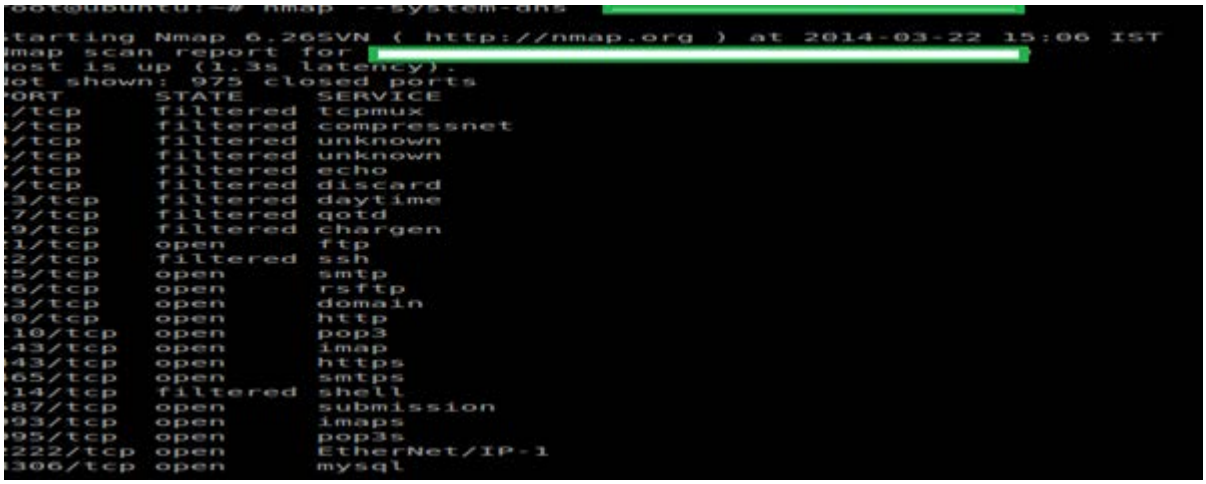
The **-n** parameter is used to disable reverse DNS lookups.

Syntax:nmap –n target

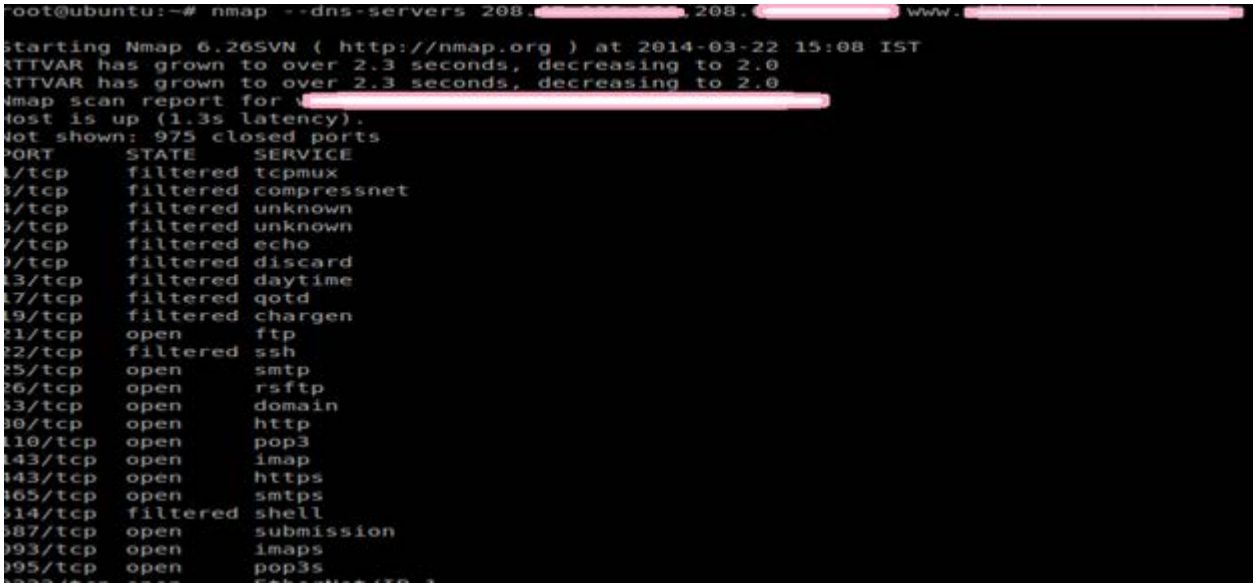
```
root@ubuntu:~# nmap -n [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-03-22 14:59 IST
Nmap scan report for [redacted]
Host is up (1.1s latency).
Not shown: 975 closed ports
PORT      STATE     SERVICE
1/tcp     filtered  tcpmux
3/tcp     filtered  compressnet
4/tcp     filtered  unknown
5/tcp     filtered  unknown
7/tcp     filtered  echo
9/tcp     filtered  discard
13/tcp    filtered  daytime
17/tcp    filtered  qotd
19/tcp    filtered  chargen
21/tcp    open      ftp
22/tcp    filtered  ssh
25/tcp    open      smtp
26/tcp    open      rsftp
53/tcp    open      domain
80/tcp    open      http
110/tcp   open      pop3
143/tcp   open      imap
443/tcp   open      https
465/tcp   open      smtps
514/tcp   filtered  shell
587/tcp   open      submission
993/tcp   open      imaps
995/tcp   open      pop3s
2222/tcp  open      EtherNet/IP-1
3306/tcp  open      mysql
```

Reverse DNS can significantly slow an Nmap scan. Using the **-n** option greatly reduces scanning times – especially when scanning a large number of hosts. This option is useful if you don't care about the DNS information for the target system and prefer to perform a scan which produces faster results.

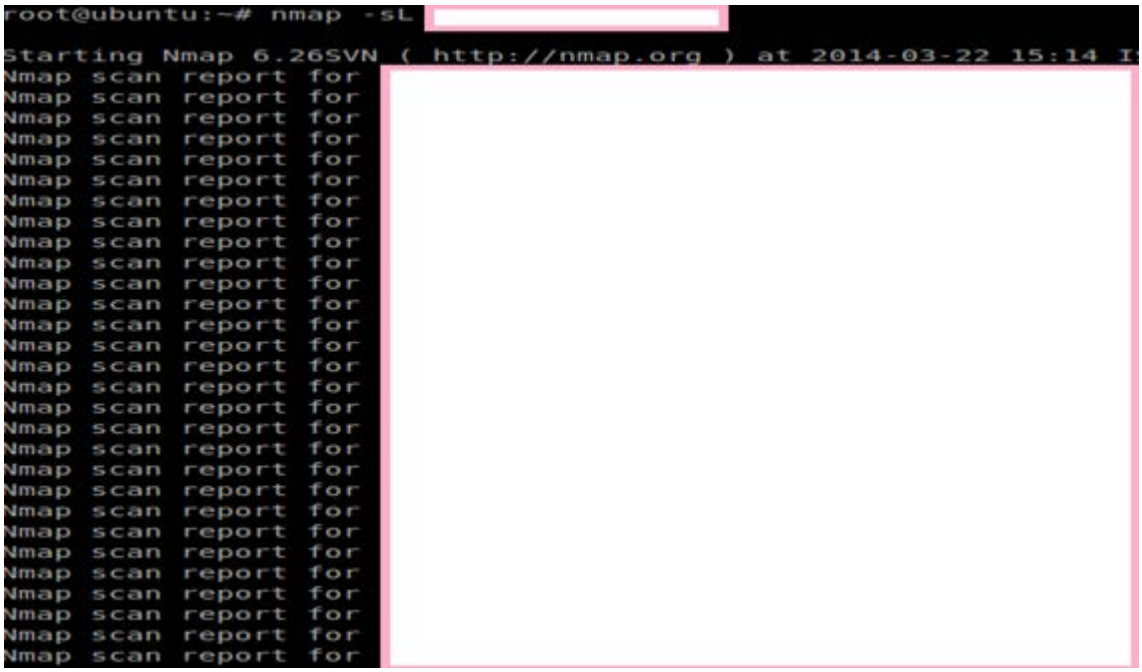
Alternative DNS lookup method
The **–system-dns** option instructs Nmap to use the host system's DNS resolver instead of its own internal method.
Syntax:nmap –system-dns target



Manually Specify DNS server
The **–dns-servers** option is used to manually specify DNS servers to be queried when scanning.
Syntax: nmap –dns-servers server1 server2 target



The **–dns-servers** option allows you to specify one or more alternative servers for Nmap to query. This can be useful for systems that do not have DNS configured or if you want to prevent your scan lookups from appearing in your locally configured DNS server's log file.
List Scan
The **-sL** option will display a list and performs a reverse DNS lookup of the specified IP addresses.
Syntax:nmap –sL target



In the next installment, I will discuss how to discover services, host, and banners using different methods, and will also discuss how to find firewalls and how to evade them using NSE by Nmap, and how to write your own Nmap script engine. The most important part of Nmap is knowing how to find vulnerability and try to exploit them. Stay tuned.

Reference

http://nmap.org/

Nmap Cheat Sheet: From Discovery to Exploits, Part 2: Advance Port Scanning with Nmap And Custom Idle Scan

This is our second installment of Nmap cheat sheet. Basically, we will discuss some advanced techniques for Nmap scanning and we will conduct a Man In The Middle Attack (MITM). Let's start our game now.

ETHICAL HACKING TRAINING – RESOURCES (INFOSEC)

TCP SYN Scan

SYN scan is the default and most popular scan option, for good reasons. It can be performed quickly, scanning thousands of ports per second on a fast network not hampered by restrictive firewalls. It is also relatively unobtrusive and stealthy, since it never completes TCP connections.

Command: nmap –sS target

TCP Connect Scan

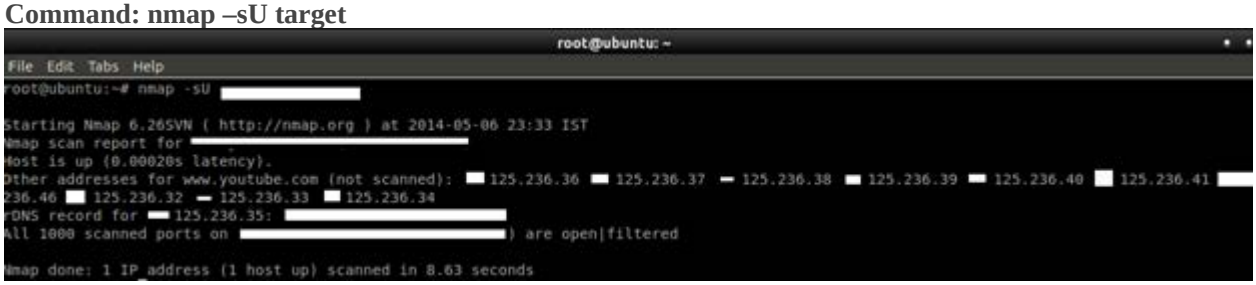
TCP connect scan is the default TCP scan type when SYN scan is not an option. This is the case when a user does not have raw packet privileges. Instead of writing raw packets as most other scan types do, Nmap asks the underlying operating system to establish a connection with the target machine and port by issuing the connect system call.

Command: nmap –sT target

UDP SCANS

While most popular services on the Internet run over the TCP protocol, UDP services are widely deployed. DNS, SNMP, and DHCP (registered ports 53, 161/162, and 67/68) are three of the most common. Because UDP scanning is generally slower and more difficult than TCP, some security auditors ignore these ports. This is a mistake, as exploitable UDP services are quite common and attackers certainly don't ignore the whole protocol.

Command: nmap –sU target



The –data-length option can be used to send a fixed-length random payload to every port or (if you specify a value of 0) to disable payloads. If an ICMP port unreachable error (type 3, code 3) is returned, the port is closed. Other ICMP unreachable errors (type 3, codes 1, 2, 9, 10, or 13) mark the port as filtered. Occasionally, a service will respond with a UDP packet, proving that it is open. If no response is received after retransmissions, the port is classified as open|filtered.

Command: nmap –sU –data-length=value target

SCTP INIT Scan

SCTP is a relatively new alternative to the TCP and UDP protocols, combining most characteristics of TCP and UDP, and also adding new features like multi-homing and multi-streaming. It is mostly being used for SS7/SIGTRAN related services but has the potential to be used for other applications as well. SCTP INIT scan is the SCTP equivalent of a TCP SYN scan. It can be performed quickly, scanning thousands of ports per second on a fast network not hampered by restrictive firewalls. Like SYN scan, INIT scan is relatively unobtrusive and stealthy, since it never completes SCTP associations.

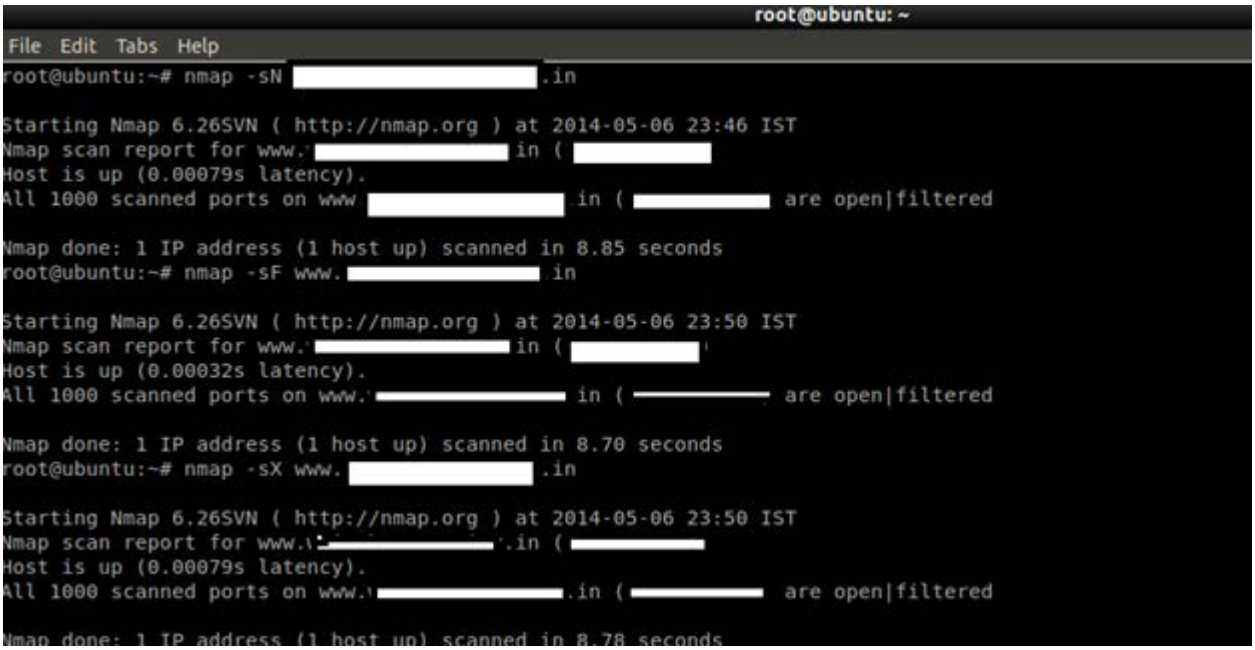
Command: nmap –sY target

TCP NULL, FIN, and Xmas scans

- NULL scan (-sN)
  Does not set any bits (TCP flag header is 0).

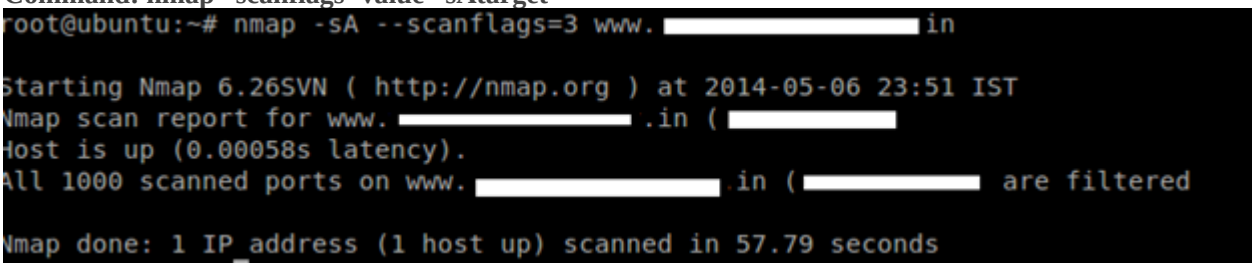
- FIN scan (-sF)
  Sets just the TCP FIN bit.

- Xmas scan (-sX)
  Sets the FIN, PSH, and URG flags, lighting the packet up like a Christmas tree.



TCP ACK Scan
This scan is different than the others discussed so far in that it never determines open (or even open|filtered) ports. It is used to map out firewall rulesets, determining whether they are stateful or not, and which ports are filtered.

Command: nmap –scanflags=value –sAtarget



The ACK scan probe packet has only the ACK flag set (unless you use –scanflags). When scanning unfiltered systems, open and closed ports will both return a RST packet. Nmap then labels them as unfiltered, meaning that they are reachable by the ACK packet.

TCP Window Scan
Window scan is exactly the same as ACK scan, except that it exploits an implementation detail of certain systems to differentiate open ports from closed ones, rather than always printing unfiltered when an RST is returned.

Command: nmap –sW target

See the valuable and juicy information which is useful for a hacker to attack further:

```
9/tcp     open  discard
13/tcp    open  daytime
17/tcp    open  qotd
19/tcp    open  chargen
20/tcp    open  ftp-data
21/tcp    open  ftp
22/tcp    open  ssh
23/tcp    open  telnet
24/tcp    open  priv-mail
25/tcp    open  smtp
26/tcp    open  rsftp
30/tcp    open  unknown
32/tcp    open  unknown
33/tcp    open  dsp
37/tcp    open  time
42/tcp    open  nameserver
43/tcp    open  whois
49/tcp    open  tacacs
53/tcp    open  domain
70/tcp    open  gopher
79/tcp    open  finger
80/tcp    open  http
81/tcp    open  hosts2-ns
82/tcp    open  xfer
83/tcp    open  mit-ml-dev
84/tcp    open  ctf
85/tcp    open  mit-ml-dev
88/tcp    open  kerberos-sec
89/tcp    open  su-mit-tg
90/tcp    open  dnsix
99/tcp    open  metagram
100/tcp   open  newacct
```

TCP Maimon Scan

The Maimon scan is named after its discoverer, Uriel Maimon. He described the technique in *Phrack* Magazine issue #49 (November 1996). Nmap, which included this technique, was released two issues later. This technique is exactly the same as NULL, FIN, and Xmas scans, except that the probe is FIN/ACK.

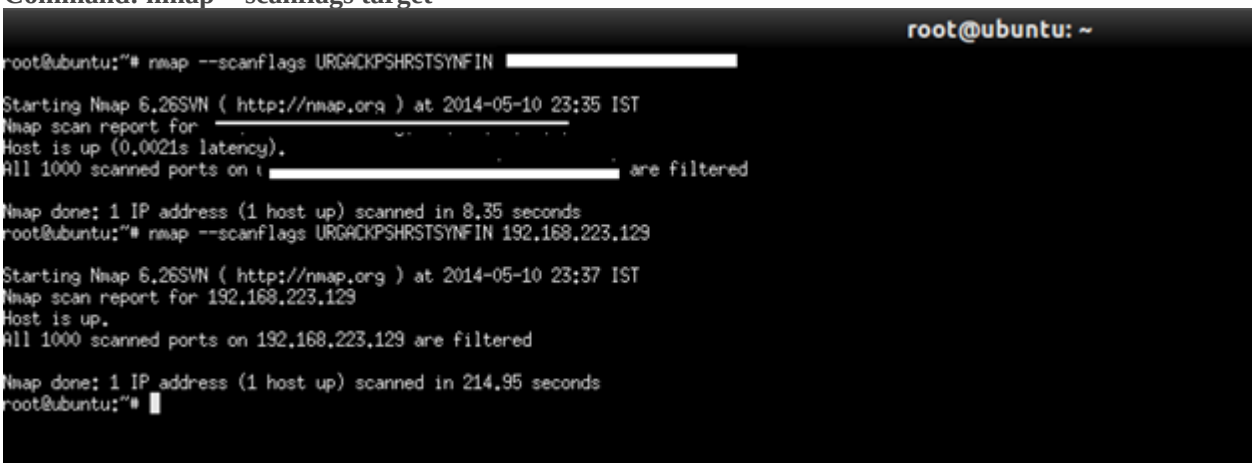
Command: nmap –sM target

Custom TCP Scan Using –scanflag Options

For advance pentesting, a pentester will not use a general TCP scan like ACK, FIN, etc. because these things may be blocked by IDS/IPS. So they will use some different techniques by specifying "-scanflag" options. This also can be used for firewall evading.

The –scanflags argument can be a numerical flag value such as 9 (PSH and FIN), but using symbolic names is easier. Just mash together any combination of URG, ACK, PSH, RST, SYN, and FIN. For example, –scanflags URGACKPSHRSTSYNFIN sets everything, though it's not very useful for scanning.

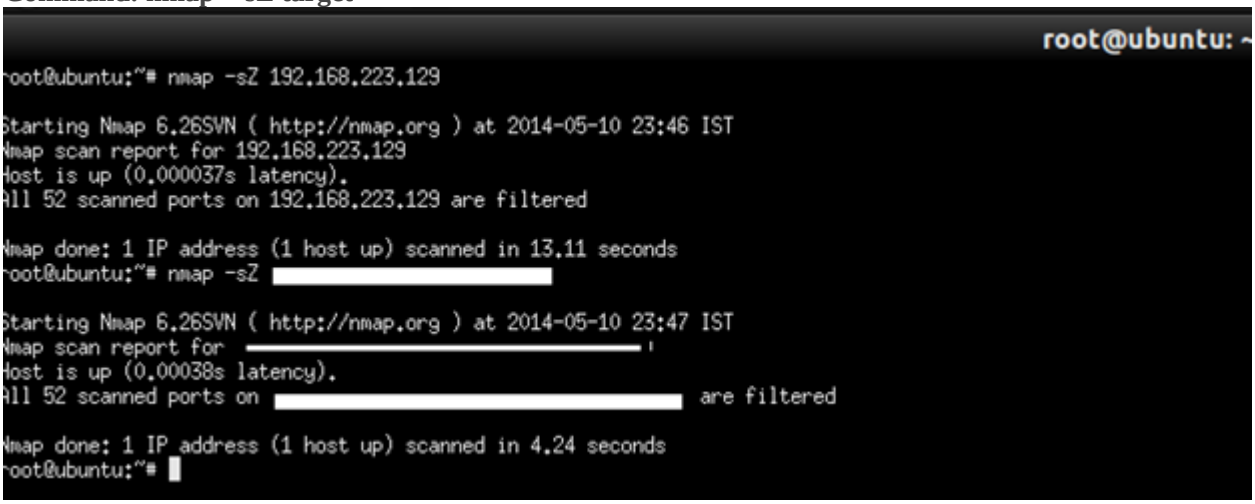
Command: nmap –-scanflags target



SCTP COOKIE ECHO Scan

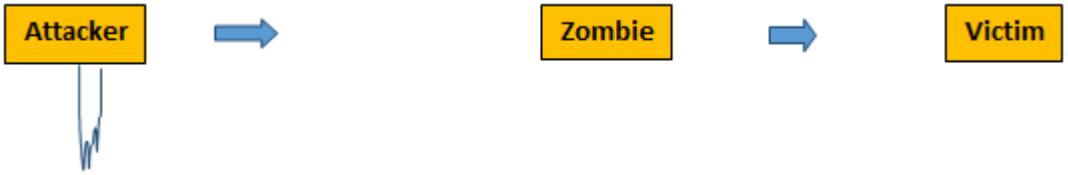
SCTP COOKIE ECHO scan is a more advanced SCTP scan. It takes advantage of the fact that SCTP implementations should silently drop packets containing COOKIE ECHO chunks on open ports, but send an ABORT if the port is closed. The advantage of this scan type is that it is not as obvious a port scan as an INIT scan. Also, there may be non-stateful firewall rulesets blocking INIT chunks, but not COOKIE ECHO chunks. A good IDS will be able to detect SCTP COOKIE ECHO scans too. The downside is that SCTP COOKIE ECHO scans cannot differentiate between open and filtered ports, leaving you with the state open|filtered in both cases.

Command: nmap –-sZ target



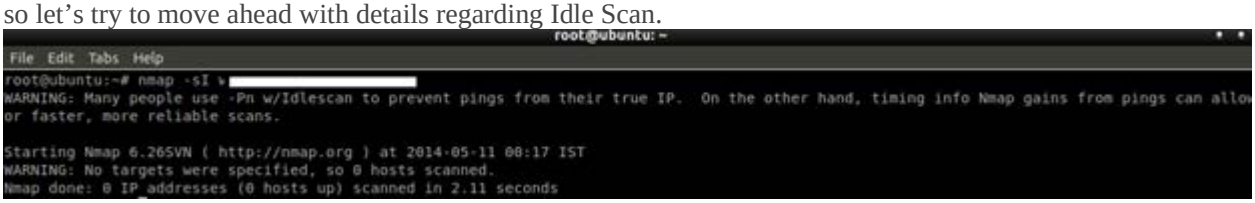
TCP Idle Scan
This advanced scan method allows for a truly blind TCP port scan of the target (meaning no packets are sent to the target from your real IP address). Instead, a unique side-channel attack exploits predictable IP fragmentation ID sequence generation on the zombie host to glean information about the open ports on the target. IDS systems will display the scan as coming from the zombie machine you specify. This is very useful for conducting MITM (Man In The Middle Attack).

Command: nmap –sI zombie target



Victim thought that Zombie was the Attacker machine, which it was actually not. So here the Attacker tried to fool the Victim.

- **Here Zombie means the middle man that you have trusted. Zombie can be any machine which acts like a middle machine between Attacker and Victim.**However, we are in advanced pentesting, so let's try to move ahead with details regarding Idle Scan.



History And Details In 1998, security researcher Antirez (who also wrote the hping2 tool used in parts of this book) posted to the Bugtraq mailing list an ingenious new port scanning technique. He called it "dumb scan". Attackers can actually scan a target without sending a single packet to the target from their own IP address! Instead, a clever side-channel attack allows for the scan to be bounced off a dumb "zombie host". Intrusion detection system (IDS) reports will finger the innocent zombie as the attacker. Besides being extraordinarily stealthy, this scan type permits discovery of IP-based trust relationships between machines.
What Is the Actual Game?

Actually, for an attacker to conduct this attack, he does not need to be an expert in TCP/IP, but it is more advanced than other techniques as discussed so far. The below steps are put together to conduct this attack.

- One way to determine whether a TCP port is open is to send a SYN (session establishment) packet to the port. The target machine will respond with a SYN/ACK (session request acknowledgment) packet if the port is open, and RST (reset) if the port is closed. This is the basis of the previously discussed SYN scan.
- A machine that receives an unsolicited SYN/ACK packet will respond with a RST. An unsolicited RST will be ignored.
- Every IP packet on the Internet has a fragment identification number (IP ID). Since many operating systems simply increment this number for each packet they send, probing for the IPID can tell an attacker how many packets have been sent since the last probe.
By combining these traits, it is possible to scan a target network while forging your identity so that it looks like an innocent zombie machine did the scanning.

Idle Scan Explained
To conduct this attack, the following steps may be followed for successful exploitation.

1. Probe the zombie's IP ID and record it.
2. Forge a SYN packet from the zombie and send it to the desired port on the target. Depending on the port state, the target's reaction may or may not cause the zombie's IP ID to be incremented.
3. Probe the zombie's IP ID again. The target port state is then determined by comparing this new IP ID with the one recorded in step 1.

How to Determine from IP ID

From the IP ID value, an attacker will try to learn about port status, whether it is open or filtered or closed. Read below for details:

- After the above idle scan process, the zombie's IP ID should have increased by either one or two. An increase of one indicates that the zombie hasn't sent out any packets, except for its reply to the attacker's probe. This lack of sent packets means that the port is not open (the target must have sent the zombie either a RST packet, which was ignored, or nothing at all).
- An increase of two indicates that the zombie sent out a packet between the two probes. This extra packet usually means that the port is open (the target presumably sent the zombie a SYN/ACK packet in response to the forged SYN, which induced a RST packet from the zombie).
- Increases larger than two usually signify a bad zombie host. It might not have predictable IP ID numbers, or might be engaged in communication unrelated to the idle scan.

See the below images that relate Attacker, Zombie, and Victim, and how the attack is conducted.

Idle Scan of an Open Port
Step 1: Probe The Zombie's IP address
SYN/ACK

RST IPID=31337

The Attacker sends SYN/ACK packets to Zombie. But Zombie is not expecting to get these packets. So in response, he discloses IP ID value by responding with an RST packet to the attacker.

Step 2: Forge SYN packet from zombie
SYN request from Zombie which is spoofed

SYN/ACK to Zombie

RST packet.IP ID:31338

The Victim sends a SYN/ACK packet in response to the SYN packet that appears to come from Zombie. The Zombie sends back RST by incrementing IP ID value.

Step3: Probe Zombie's IP ID again
SYN/ACK

Zombie

RST,IP ID:31339

The Zombie's IP ID increased by two, which is learned from step one.

So here we know that the port is open from the IPID value.

Note**:** If the port is closed, then the IPID value will be increased by one. If the Zombie's IP ID increased by one as in the first step, we can say that it may be closed or filtered. In the case of filtered, the Victim has no response to Zombie for the SYN request of Attacker.

In this situation, an Attacker will learn that there may be IDS/IPS which have rules to block some certain scan attempts by Zombie machines. For that, he will again use decoy options for Nmap to evade that. We will discuss that later.

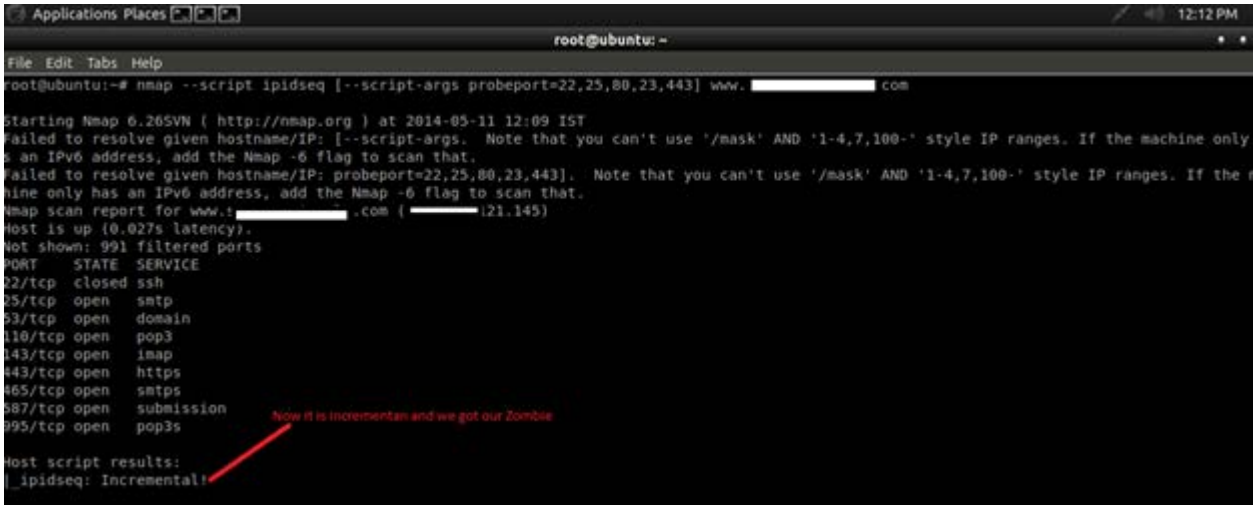
Step 1: Finding Zombie Host for Idle Scan
The first step in executing an IP ID idle scan is to find an appropriate zombie. It needs to assign IP ID packets incrementally on a global (rather than per-host it communicates with) basis. It should be idle (hence the scan name), as extraneous traffic will bump up its IP ID sequence, confusing the scan logic.

A common approach is to simply execute a Nmap ping scan of some network. We can use Nmap's random IP selection mode (-iR), but that is likely to result in far away zombies with substantial latency.

Performing a port scan and OS identification (-O) on the zombie candidate network, rather than just a ping scan, helps in selecting a good zombie. As long as verbose mode (-v) is enabled, OS detection will usually determine the IP ID sequence generation method and print a line such as "IP ID Sequence Generation: Incremental". If the type is given as Incremental or Broken little-endian incremental, the machine is a good zombie candidate.

Another approach to identifying zombie candidates is to run the ipidseq NSE script against a host. This script probes a host to classify its IP ID generation method, then prints the IP ID classification, much like the OS detection does.

Command: (nmap –script ipidseq [–script-args probeport=port] target)

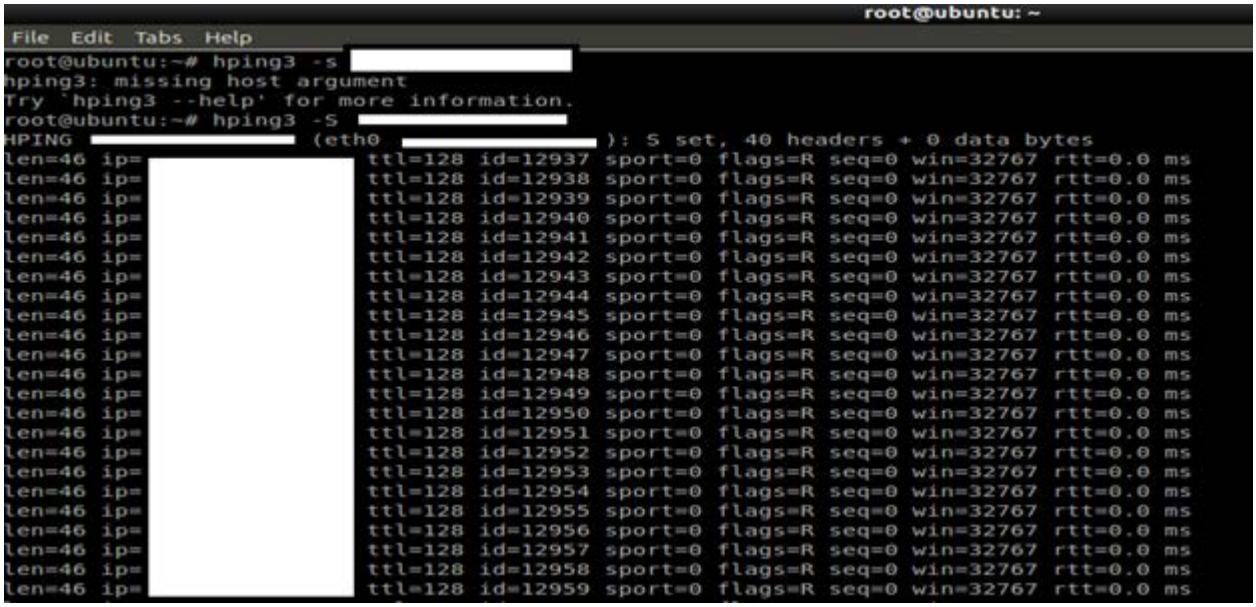


Now we can tell that it is incremental and a good candidate for Zombie.

Using Hping
We can also use hping for discovering a zombie. The hping method for idle scanning provides a lower level example for how idle scanning is performed. In this example, the target host (target1) will be scanned using an idle host (target2). An open and a closed port will be tested to see how each scenario plays out.

First, establish that the idle host is actually idle, send packets using hping2 and observe the ID numbers increase incrementally by one. If the ID numbers increase haphazardly, the host is not actually idle, or has an OS that has no predictable IP ID.

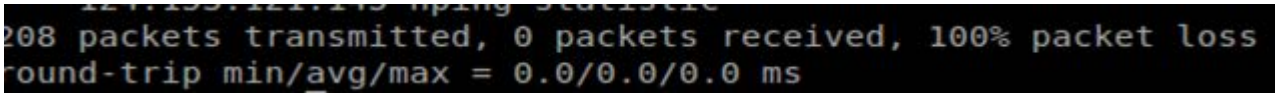
hping3 -S target



Send a spoofed SYN packet to the target host on a port you expect to be open.

hping3 –spoof Zombie -S p 22 target

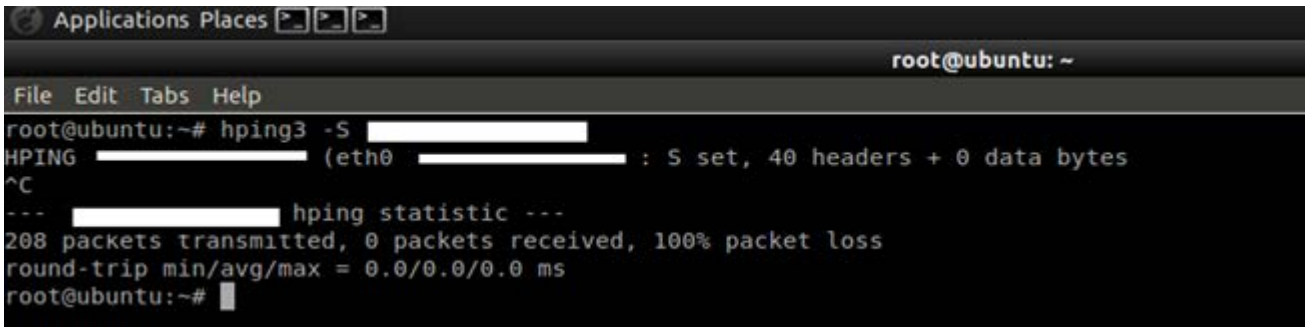
*Note: Here I do not want to include the screenshot. Though this is a part of research and this document is for only educational purposes, the owner of the website does not want to disclose it. If you have any doubt you can contact me here or email me.



As you can see, there is no response and it shows 100% packet loss. That means we have failed to find the zombie. Again we will check the following step for confirmation.

Check the IPID value for any increment:

hping3 -S target

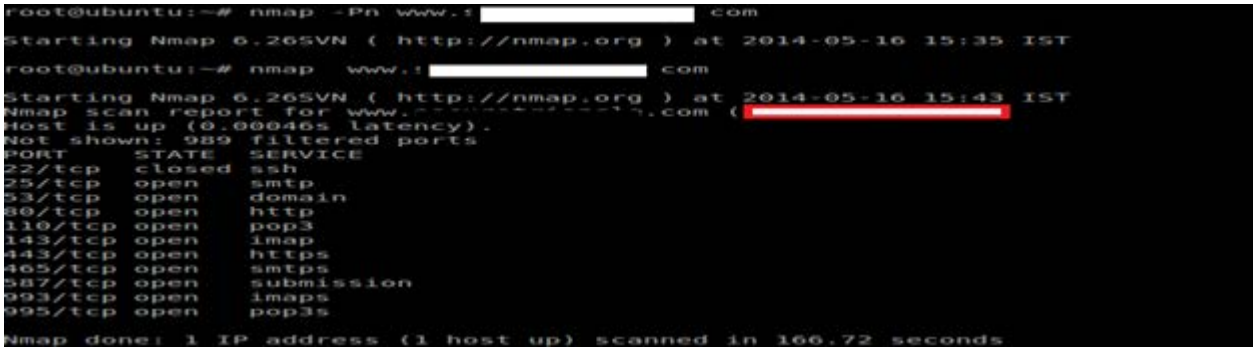


No response.

It includes that the port is filtered.

Attack using Nmap
See the image below. We are attacking the target machine using a zombie host.

Command: nmap –Pn –p- -sI zombie target
First we will do an Nmap scan for ports:



Based on that, let's try port 22, which is already running.

Here we are unable to attack to the target, as it is showing the port is already used for some other purpose. By default, Nmap forges probes to the target from the source port 80 of the zombie. You can choose a different port by appending a colon and port number to the zombie name (e.g. -sI zombie:113). The chosen port must not be filtered from the attacker or the target. A SYN scan of the zombie should show the port in the open or closed state.

Here –Pn: prevents Nmap from sending the initial packets to the target machine.

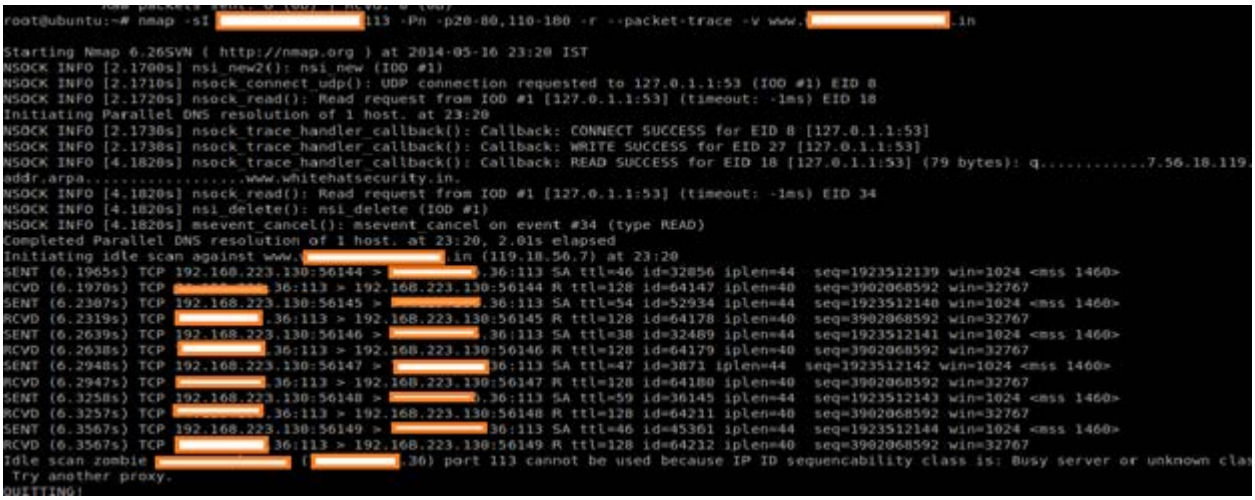
-p-: will scan all 65535 ports.

-sI: used for idle scan and sending spoof packets.

Here what happens, the attacker's IDS will think that the packet is coming from a zombie machine, not from the target machine. So he will be confused.

Understanding Nmap Internally
As a pentester, we must understand internal workings of Nmap's idle scan, so that we will craft the same thing in our own implementation. Even we can write our own code based on Python to do the same thing. We must understand the basic flow or algorithm of Nmap's idle scan. For that, we will use packet trace options in Nmap.

Command: nmap -sI Zombie:113 -Pn -p20-80,110-180 -r –packet-trace -v target
–Pn is necessary for stealth, otherwise pinged packets would be sent to the target from the attacker's real address. Version scanning would also expose the true address, -sV is *not* specified. The -r option (turns off port randomization) is only used to make this example easier.

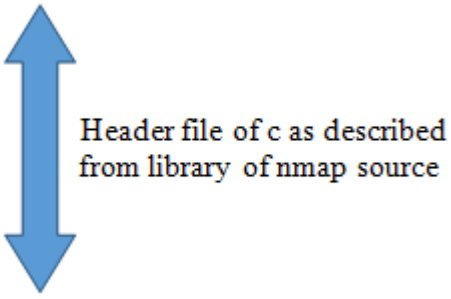


As I said before, use a suitable zombie port for successful attack.

Process of this attack:

Nmap firsts tests Zombie's IP ID sequence generation by sending six SYN/ACK packets to it and analyzing the responses. Here R means Reset packet. That means that is not reachable through that port, though that is already used for other services. For more details, follow the Idle Scan by Nmap Manual (http://nmap.org/book/idlescan.html). Here is a vulnerable machine with a suitable zombie for a successful attack.
So the below mentioned C code is for idle scan. Compile the C program and run the code.

This is an extraordinary scan code that can allow for completely blind scanning (eg. no packets sent to the target from your own IP address) and can also be used to penetrate firewalls and scope out router ACLs.




```
#include "idle_scan.h"

#include "timing.h"

#include "osscan2.h"

#include "nmap.h"

#include "NmapOps.h"
```

```
#include "services.h"

#include "Target.h"

#include "utils.h"

#include "output.h"


#include "struct_ip.h"


#include


extern NmapOps o;


struct idle_proxy_info {    create a constructer and take all variable into it

  Target host; /* contains name, IP, source IP, timing info, etc. */

  int seqclass; /* IP ID sequence class (IPID_SEQ_* defined in nmap.h) */

  u16 latestid; /* The most recent IP ID we have received from the proxy */

  u16 probe_port; /* The port we use for probing IP ID infoz */

  u16 max_groupsz; /* We won't test groups larger than this ... */

  u16 min_groupsz; /* We won't allow the group size to fall below this

                      level.  Affected by --min-parallelism */

  double current_groupsz; /* Current group size being used ... depends on

                 conditions ... won't be higher than

                 max_groupsz */
```

```
  int senddelay; /* Delay between sending pr0be SYN packets to target
                    (in microseconds) */
  int max_senddelay; /* Maximum time we are allowed to wait between
                        sending pr0bes (when we send a bunch in a row.
                        In microseconds. */

  pcap_t *pd; /* A Pcap descriptor which (starting in
                 initialize_idleproxy) listens for TCP packets from
                 the probe_port of the proxy box */
  int rawsd; /* Socket descriptor for sending probe packets to the proxy */
  struct eth_nfo eth; // For when we want to send probes via raw IP instead.
  struct eth_nfo *ethptr; // points to eth if filled out, otherwise NULL
};

/* Sends an IP ID probe to the proxy machine and returns the IP ID.
   This function handles retransmissions, and returns -1 if it fails.
   Proxy timing is adjusted, but proxy->latestid is NOT ADJUSTED --
   you'll have to do that yourself.  Probes_sent is set to the number
   of probe packets sent during execution */
static int ipid_proxy_probe(struct idle_proxy_info *proxy, int *probes_sent,
                            int *probes_rcvd) {
  struct timeval tv_end;
```

```
  int tries = 0;

  int trynum;

  int sent=0, rcvd=0;

  int maxtries = 3; /* The maximum number of tries before we give up */

  struct timeval tv_sent[3], rcvdtime;

  int ipid = -1;

  int to_usec;

  unsigned int bytes;

  int base_port;

  struct ip *ip;

  struct tcp_hdr *tcp;

  static u32 seq_base = 0;

  static u32 ack = 0;

  static int packet_send_count = 0; /* Total # of probes sent by this program -- to ensure that our sequence #
always changes */



  if (o.magic_port_set)

    base_port = o.magic_port;

  else base_port = o.magic_port + get_random_u8();



  if (seq_base == 0) seq_base = get_random_u32();

  if (!ack) ack = get_random_u32();
```

```
do {
  gettimeofday(&tv_sent[tries], NULL);

  /* Time to send the pr0be!*/
  send_tcp_raw(proxy->rawsd, proxy->ethptr,
                          proxy->host.v4sourceip(), proxy->host.v4hostip(),
                          o.ttl, false,
                          o.ipoptions, o.ipoptionslen,
                          base_port + tries, proxy->probe_port,
                          seq_base + (packet_send_count++ * 500) + 1, ack, 0, TH_SYN|TH_ACK, 0, 0,
                          (u8 *) "\x02\x04\x05\xb4", 4,
                          NULL, 0);
  sent++;
  tries++;

  /* Now it is time to wait for the response ... */
  to_usec = proxy->host.to.timeout;
  gettimeofday(&tv_end, NULL);
  while((ipid == -1 || sent > rcvd) && to_usec > 0) {

    to_usec = proxy->host.to.timeout - TIMEVAL_SUBTRACT(tv_end, tv_sent[tries-1]);
```

```
    if (to_usec < 0) to_usec = 0; // Final no-block poll       ip = (struct ip *) readipv4_pcap(proxy->pd, &bytes,
to_usec, &rcvdtime, NULL, true);

    gettimeofday(&tv_end, NULL);

    if (ip) {

            if (bytes < ( 4 * ip->ip_hl) + 14U)

              continue;



            if (ip->ip_p == IPPROTO_TCP) {



              tcp = ((struct tcp_hdr *) (((char *) ip) + 4 * ip->ip_hl));

              if (ntohs(tcp->th_dport) < base_port || ntohs(tcp->th_dport) - base_port >= tries  || ntohs(tcp-
>th_sport) != proxy->probe_port || ((tcp->th_flags & TH_RST) == 0)) {

                if (ntohs(tcp->th_dport) > o.magic_port && ntohs(tcp->th_dport) < (o.magic_port + 260)) {
if (o.debugging) {                    error("Received IP ID zombie probe response which probably came from an
earlier prober instance ... increasing rttvar from %d to %d",                    proxy->host.to.rttvar, (int)
(proxy->host.to.rttvar * 1.2));

                  }

                  proxy->host.to.rttvar = (int) (proxy->host.to.rttvar * 1.2);

                  rcvd++;

                }

                else if (o.debugging > 1) {

                  error("Received unexpected response packet from %s during IP ID zombie probing:",
inet_ntoa(ip->ip_src));

                  readtcppacket( (unsigned char *) ip,MIN(ntohs(ip->ip_len), bytes));

                }
```

```
                continue;

              }



              trynum = ntohs(tcp->th_dport) - base_port;

              rcvd++;



              ipid = ntohs(ip->ip_id);

              adjust_timeouts2(&(tv_sent[trynum]), &rcvdtime, &(proxy->host.to));

            }

    }

  }

 } while(ipid == -1 && tries < maxtries);  if (probes_sent) *probes_sent = sent;  if (probes_rcvd)
*probes_rcvd = rcvd;  return ipid; } /* Returns the number of increments between an early IP ID and a later
one, assuming the given IP ID Sequencing class.  Returns -1 if the  distance cannot be determined */ static int
ipid_distance(int seqclass , u16 startid, u16 endid) {  if (seqclass == IPID_SEQ_INCR)  return endid -
startid;  if (seqclass == IPID_SEQ_BROKEN_INCR) {  /* Convert to network byte order */  startid =
htons(startid);  endid = htons(endid);  return endid - startid;  }  return -1; } static void
initialize_proxy_struct(struct idle_proxy_info *proxy) {  proxy->seqclass = proxy->latestid = proxy-
>probe_port = 0;

 proxy->max_groupsz = proxy->min_groupsz = 0;

 proxy->current_groupsz = 0;

 proxy->senddelay = 0;

 proxy->max_senddelay = 0;

 proxy->pd = NULL;

 proxy->rawsd = -1;

 proxy->ethptr = NULL;
```

```
}


/* takes a proxy name/IP, resolves it if necessary, tests it for IP ID
   suitability, and fills out an idle_proxy_info structure.  If the
   proxy is determined to be unsuitable, the function whines and exits
   the program */
#define NUM_IPID_PROBES 6
static void initialize_idleproxy(struct idle_proxy_info *proxy, char *proxyName,
                                 const struct in_addr *first_target, const struct scan_lists * ports) {
  int probes_sent = 0, probes_returned = 0;
  int hardtimeout = 9000000; /* Generally don't wait more than 9 secs total */
  unsigned int bytes, to_usec;
  int timedout = 0;
  char *p, *q;
  char *endptr = NULL;
  int seq_response_num;
  int newipid;
  int i;
  char filter[512]; /* Libpcap filter string */
  char name[MAXHOSTNAMELEN + 1];
  struct sockaddr_storage ss;
  size_t sslen;
```

```
  u32 sequence_base;

  u32 ack = 0;

  struct timeval probe_send_times[NUM_IPID_PROBES], tmptv, rcvdtime;

  u16 lastipid = 0;

  struct ip *ip;

  struct tcp_hdr *tcp;

  int distance;

  int ipids[NUM_IPID_PROBES];

  u8 probe_returned[NUM_IPID_PROBES];

  struct route_nfo rnfo;

  assert(proxy);

  assert(proxyName);



  ack = get_random_u32();



  for(i=0; i < NUM_IPID_PROBES; i++) probe_returned[i] = 0;   initialize_proxy_struct(proxy);
initialize_timeout_info(&proxy->host.to);



  proxy->max_groupsz = (o.max_parallelism)? o.max_parallelism : 100;

  proxy->min_groupsz = (o.min_parallelism)? o.min_parallelism : 4;

  proxy->max_senddelay = 100000;
```

```
Strncpy(name, proxyName, sizeof(name));
q = strchr(name, ':');
if (q) {
  *q++ = '\0';
  proxy->probe_port = strtoul(q, &endptr, 10);
  if (*q==0 || !endptr || *endptr != '\0' || !proxy->probe_port) {
    fatal("Invalid port number given in IP ID zombie specification: %s", proxyName);
  }
} else {
  if (ports->syn_ping_count > 0) {
    proxy->probe_port = ports->syn_ping_ports[0];
  } else if (ports->ack_ping_count > 0) {
    proxy->probe_port = ports->ack_ping_ports[0];
  } else {
    u16 *ports;
    int count;

    getpts_simple(DEFAULT_TCP_PROBE_PORT_SPEC, SCAN_TCP_PORT, &ports, &count);
    assert(count > 0);
    proxy->probe_port = ports[0];
    free(ports);
  }
```

```
  }


  proxy->host.setHostName(name);
  if (resolve(name, 0, 0, &ss, &sslen, o.pf()) == 0) {
    fatal("Could not resolve idle scan zombie host: %s", name);
  }
  proxy->host.setTargetSockAddr(&ss, sslen);


  /* Lets figure out the appropriate source address to use when sending
     the pr0bez */
  proxy->host.TargetSockAddr(&ss, &sslen);
  if (!nmap_route_dst(&ss, &rnfo))
    fatal("Unable to find appropriate source address and device interface to use when sending packets to %s",
proxyName);


  if (o.spoofsource) {
    o.SourceSockAddr(&ss, &sslen);
    proxy->host.setSourceSockAddr(&ss, sslen);
    proxy->host.setDeviceNames(o.device, o.device);
  } else {
    proxy->host.setDeviceNames(rnfo.ii.devname, rnfo.ii.devfullname);
    proxy->host.setSourceSockAddr(&rnfo.srcaddr, sizeof(rnfo.srcaddr));
```

```
}
if (rnfo.direct_connect) {
  proxy->host.setDirectlyConnected(true);
} else {
  proxy->host.setDirectlyConnected(false);
  proxy->host.setNextHop(&rnfo.nexthop,
                                        sizeof(rnfo.nexthop));
}
proxy->host.setIfType(rnfo.ii.device_type);
if (rnfo.ii.device_type == devt_ethernet)
  proxy->host.setSrcMACAddress(rnfo.ii.mac);

/* Now lets send some probes to check IP ID algorithm ... */
/* First we need a raw socket ... */
if ((o.sendpref & PACKET_SEND_ETH) &&  proxy->host.ifType() == devt_ethernet) {
  if (!setTargetNextHopMAC(&proxy->host))
    fatal("%s: Failed to determine dst MAC address for Idle proxy",
                __func__);
  memcpy(proxy->eth.srcmac, proxy->host.SrcMACAddress(), 6);
  memcpy(proxy->eth.dstmac, proxy->host.NextHopMACAddress(), 6);
  proxy->eth.ethsd = eth_open_cached(proxy->host.deviceName());
  if (proxy->eth.ethsd == NULL)
```

```
      fatal("%s: Failed to open ethernet device (%s)", __func__, proxy->host.deviceName());

    proxy->rawsd = -1;

    proxy->ethptr = &proxy->eth;

  } else {

#ifdef WIN32

    win32_fatal_raw_sockets(proxy->host.deviceName());

#endif

    if ((proxy->rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0 )       pfatal("socket troubles in %s", __func__);     unblock_socket(proxy->rawsd);

    broadcast_socket(proxy->rawsd);

#ifndef WIN32

    sethdrinclude(proxy->rawsd);

#endif

    proxy->eth.ethsd = NULL;

    proxy->ethptr = NULL;

  }



/* Now for the pcap opening nonsense ... */

 /* Note that the snaplen is 152 = 64 byte max IPhdr + 24 byte max link_layer

  * header + 64 byte max TCP header. */

  if((proxy->pd=my_pcap_open_live(proxy->host.deviceName(), 152,  (o.spoofsource)? 1 : 0, 50))==NULL)

    fatal("%s", PCAP_OPEN_ERRMSG);
```

```
  p = strdup(proxy->host.targetipstr());

  q = strdup(inet_ntoa(proxy->host.v4source()));

  Snprintf(filter, sizeof(filter), "tcp and src host %s and dst host %s and src port %hu", p, q, proxy->probe_port);

 free(p);

 free(q);

 set_pcap_filter(proxy->host.deviceFullName(), proxy->pd,  filter);

 if (o.debugging)

   log_write(LOG_STDOUT, "Packet capture filter (device %s): %s\n", proxy->host.deviceFullName(), filter);

/* Windows nonsense -- I am not sure why this is needed, but I should

   get rid of it at sometime */



 sequence_base = get_random_u32();



 /* Yahoo!  It is finally time to send our pr0beZ! */



  while(probes_sent < NUM_IPID_PROBES) {      if (o.scan_delay) enforce_scan_delay(NULL);      else if (probes_sent) usleep(30000);      /* TH_SYN|TH_ACK is what the proxy will really be receiving from        the target, and is more likely to get through firewalls.  But        TH_SYN allows us to get a nonzero ACK back so we can associate        a response with the exact request for timing purposes.  So I        think I'll use TH_SYN, although it is a tough call. */      /* We can't use decoys 'cause that would screw up the IP IDs */ send_tcp_raw(proxy->rawsd, proxy->ethptr,

                                proxy->host.v4sourceip(), proxy->host.v4hostip(),

                                o.ttl, false,
```

```
                        o.ipoptions, o.ipoptionslen,
                         o.magic_port + probes_sent + 1, proxy->probe_port,
                        sequence_base + probes_sent + 1, ack, 0, TH_SYN|TH_ACK, 0, 0,
                        (u8 *) "\x02\x04\x05\xb4",4,
                        NULL, 0);
   gettimeofday(&probe_send_times[probes_sent], NULL);
   probes_sent++;

   /* Time to collect any replies */
   while(probes_returned < probes_sent && !timedout) {       to_usec = (probes_sent ==
NUM_IPID_PROBES)? hardtimeout : 1000;       ip = (struct ip *) readipv4_pcap(proxy->pd, &bytes, to_usec,
&rcvdtime, NULL, true);

     gettimeofday(&tmptv, NULL);

     if (!ip) {
              if (probes_sent < NUM_IPID_PROBES)      break;      if (TIMEVAL_SUBTRACT(tmptv,
probe_send_times[probes_sent - 1]) >= hardtimeout) {
                timedout = 1;
              }
              continue;
     } else if (TIMEVAL_SUBTRACT(tmptv, probe_send_times[probes_sent - 1]) >=
                        hardtimeout)  {
              timedout = 1;
```

```
}

if (lastipid != 0 && ip->ip_id == lastipid) {
        continue; /* probably a duplicate */
}
lastipid = ip->ip_id;

if (bytes < ( 4 * ip->ip_hl) + 14U)
        continue;

if (ip->ip_p == IPPROTO_TCP) {
        tcp = ((struct tcp_hdr *) (((char *) ip) + 4 * ip->ip_hl));
        if (ntohs(tcp->th_dport) < (o.magic_port+1) || ntohs(tcp->th_dport) - o.magic_port >
NUM_IPID_PROBES  || ntohs(tcp->th_sport) != proxy->probe_port || ((tcp->th_flags & TH_RST) == 0)) {
          if (o.debugging > 1) error("Received unexpected response packet from %s during initial IP ID
zombie testing", inet_ntoa(ip->ip_src));
          continue;
        }

        seq_response_num = probes_returned;

        /* The stuff below only works when we send SYN packets instead of
           SYN|ACK, but then are slightly less stealthy and have less chance
```

```
                of sneaking through the firewall.  Plus SYN|ACK is what they will
                be receiving back from the target */
              probes_returned++;
              ipids[seq_response_num] = (u16) ntohs(ip->ip_id);
              probe_returned[seq_response_num] = 1;
              adjust_timeouts2(&probe_send_times[seq_response_num], &rcvdtime, &(proxy->host.to));
    }
   }
 }

 /* Yeah!  We're done sending/receiving probes ... now lets ensure all of our responses are adjacent in the array */
 for(i=0,probes_returned=0; i < NUM_IPID_PROBES; i++) {    if (probe_returned[i]) {      if (i > probes_returned)
              ipids[probes_returned] = ipids[i];
    probes_returned++;
   }
 }

 if (probes_returned == 0)
   fatal("Idle scan zombie %s (%s) port %hu cannot be used because it has not returned any of our probes -- perhaps it is down or firewalled.",
                proxy->host.HostName(), proxy->host.targetipstr(),
                proxy->probe_port);
```

```
  proxy->seqclass = get_ipid_sequence(probes_returned, ipids, 0);

  switch(proxy->seqclass) {

  case IPID_SEQ_INCR:

  case IPID_SEQ_BROKEN_INCR:

    log_write(LOG_PLAIN, "Idle scan using zombie %s (%s:%hu); Class: %s\n", proxy->host.HostName(),
proxy->host.targetipstr(), proxy->probe_port, ipidclass2ascii(proxy->seqclass));

    break;

  default:

    fatal("Idle scan zombie %s (%s) port %hu cannot be used because IP ID sequencability class is: %s.  Try
another proxy.", proxy->host.HostName(), proxy->host.targetipstr(), proxy->probe_port, ipidclass2ascii(proxy-
>seqclass));

  }



  proxy->latestid = ipids[probes_returned - 1];

  proxy->current_groupsz = MIN(proxy->max_groupsz, 30);



  if (probes_returned < NUM_IPID_PROBES) {      /* Yikes!  We're already losing packets ... clamp down a bit
... */     if (o.debugging)       error("Idle scan initial zombie qualification test: %d probes sent, only %d
returned", NUM_IPID_PROBES, probes_returned);     proxy->current_groupsz = MIN(12, proxy-
>max_groupsz);

    proxy->current_groupsz = MAX(proxy->current_groupsz, proxy->min_groupsz);

    proxy->senddelay += 5000;

  }
```

```
  /* OK, through experimentation I have found that some hosts (*cough*
     Solaris) APPEAR to use simple IP ID incrementing, but in reality they
     assign a new IP ID base to each host which connects with them.  This
     is actually a good idea on several fronts, but it totally
     frustrates our efforts (which rely on side-channel IP ID info
     leaking to different hosts).  The good news is that we can easily
     detect the problem by sending some spoofed packets "from" the first
     target to the zombie and then probing to verify that the proxy IP ID
     changed.  This will also catch the case where the Nmap user is
     behind an egress filter or other measure that prevents this sort of
     sp00fery */
  if (first_target) {
    for (probes_sent = 0; probes_sent < 4; probes_sent++) {       if (probes_sent) usleep(50000);
send_tcp_raw(proxy->rawsd, proxy->ethptr,
                                first_target, proxy->host.v4hostip(),
                                o.ttl, false,
                                o.ipoptions, o.ipoptionslen,
                                o.magic_port, proxy->probe_port,
                                sequence_base + probes_sent + 1, ack, 0, TH_SYN|TH_ACK, 0, 0,
                                (u8 *) "\x02\x04\x05\xb4",
                                4, NULL, 0);
```

```
    }


    /* Sleep a little while to give packets time to reach their destination */
    usleep(300000);
    newipid = ipid_proxy_probe(proxy, NULL, NULL);
    if (newipid == -1)
      newipid = ipid_proxy_probe(proxy, NULL, NULL); /* OK, we'll give it one more try */


    if (newipid < 0) fatal("Your IP ID Zombie (%s; %s) is behaving strangely -- suddenly cannot obtain IP ID",
proxy->host.HostName(), proxy->host.targetipstr());


    distance = ipid_distance(proxy->seqclass, proxy->latestid, newipid);
    if (distance <= 0) {       fatal("Your IP ID Zombie (%s; %s) is behaving strangely -- suddenly cannot obtain
valid IP ID distance.", proxy->host.HostName(), proxy->host.targetipstr());
    } else if (distance == 1) {
      fatal("Even though your Zombie (%s; %s) appears to be vulnerable to IP ID sequence prediction (class:
%s), our attempts have failed.  This generally means that either the zombie uses a separate IP ID base for each
host (like Solaris), or because you cannot spoof IP packets (perhaps your ISP has enabled egress filtering to
prevent IP spoofing), or maybe the target network recognizes the packet source as bogus and drops them",
proxy->host.HostName(), proxy->host.targetipstr(), ipidclass2ascii(proxy->seqclass));
    }
    if (o.debugging && distance != 5) {
      error("WARNING: IP ID spoofing test sent 4 packets and expected a distance of 5, but instead got %d",
distance);
    }
    proxy->latestid = newipid;
```

```
  }


}


/* Adjust timing parameters up or down given that an idle scan found a
   count of 'testcount' while the 'realcount' is as given.  If the
   testcount was correct, timing is made more aggressive, while it is
   slowed down in the case of an error */
static void adjust_idle_timing(struct idle_proxy_info *proxy,
                               Target *target, int testcount,
                               int realcount) {


  static int notidlewarning = 0;


  if (o.debugging > 1)
    log_write(LOG_STDOUT,
              "%s: tested/true %d/%d -- old grpsz/delay: %f/%d ",
              __func__, testcount, realcount, proxy->current_groupsz, proxy->senddelay);
  else if (o.debugging && testcount != realcount) {
    error("%s: testcount: %d  realcount: %d -- old grpsz/delay: %f/%d", __func__, testcount, realcount, proxy-
>current_groupsz, proxy->senddelay);
  }
```

```
    if (testcount < realcount) {       /* We must have missed a port -- our probe could have been       dropped, the response to proxy could have been dropped, or we       didn't wait long enough before probing the proxy IP ID.  The          third case is covered elsewhere in the scan, so we worry most      about the first two.  The solution is to decrease our group                 size and add a sending delay */ /* packets could be dropped because too many were sent at once */       proxy->current_groupsz = MAX(proxy->min_groupsz, proxy->current_groupsz * 0.8);

      proxy->senddelay += 10000;

      proxy->senddelay = MIN(proxy->max_senddelay, proxy->senddelay);

       /* No group size should be greater than .5s of send delays */

      proxy->current_groupsz = MAX(proxy->min_groupsz, MIN(proxy->current_groupsz, 500000 / (proxy->senddelay + 1)));



    } else if (testcount > realcount) {

      /* Perhaps the proxy host is not really idle ... */

      /* I guess all I can do is decrease the group size, so that if the proxy is not really idle, at least we may be able to scan chunks more quickly in between outside packets */

      proxy->current_groupsz = MAX(proxy->min_groupsz, proxy->current_groupsz * 0.8);



      if (!notidlewarning && o.verbose) {

              notidlewarning = 1;

              error("WARNING: idle scan has erroneously detected phantom ports -- is the proxy %s (%s) really idle?", proxy->host.HostName(), proxy->host.targetipstr());

      }

    } else {

      /* W00p We got a perfect match.  That means we get a slight increase

               in allowed group size and we can lightly decrease the senddelay */
```

```
      proxy->senddelay = (int) (proxy->senddelay * 0.9);
      if (proxy->senddelay < 500) proxy->senddelay = 0;
      proxy->current_groupsz = MIN(proxy->current_groupsz * 1.1, 500000 / (proxy->senddelay + 1));
      proxy->current_groupsz = MIN(proxy->max_groupsz, proxy->current_groupsz);

    }
    if (o.debugging > 1)
      log_write(LOG_STDOUT, "-> %f/%d\n", proxy->current_groupsz, proxy->senddelay);
}

/* OK, now this is the hardcore idle scan function which actually does
   the testing (most of the other cruft in this file is just
   coordination, preparation, etc).  This function simply uses the
   idle scan technique to try and count the number of open ports in the
   given port array.  The sent_time and rcv_time are filled in with
   the times that the probe packet & response were sent/received.
   They can be NULL if you don't want to use them.  The purpose is for
   timing adjustments if the numbers turn out to be accurate. */

static int idlescan_countopen2(struct idle_proxy_info *proxy,
                               Target *target, u16 *ports, int numports,
```

```
                                struct timeval *sent_time, struct timeval *rcv_time)

{



#if 0 /* Testing code */

  int i;

  for(i=0; i < numports; i++)     if (ports[i] == 22)       return 1;   return 0; #endif   int openports;   int tries;   int
proxyprobes_sent = 0; /* diff. from tries 'cause sometimes we                                    skip tries */   int
proxyprobes_rcvd = 0; /* To determine if packets were dr0pped */   int sent, rcvd;   int ipid_dist;   struct
timeval start, end, latestchange, now;   struct timeval probe_times[4];   int pr0be;   static u32 seq = 0;   int
newipid = 0;   int sleeptime;   int lasttry = 0;   int dotry3 = 0;   struct eth_nfo eth;   if (seq == 0) seq =
get_random_u32();   memset(&end, 0, sizeof(end));   memset(&latestchange, 0, sizeof(latestchange));
gettimeofday(&start, NULL);   if (sent_time) memset(sent_time, 0, sizeof(*sent_time));   if (rcv_time)
memset(rcv_time, 0, sizeof(*rcv_time));   if (proxy->rawsd < 0) {     if (!setTargetNextHopMAC(target))
fatal("%s: Failed to determine dst MAC address for Idle proxy",      __func__);     memcpy(eth.srcmac, target-
>SrcMACAddress(), 6);

    memcpy(eth.dstmac, target->NextHopMACAddress(), 6);

    eth.ethsd = eth_open_cached(target->deviceName());

    if (eth.ethsd == NULL)

      fatal("%s: Failed to open ethernet device (%s)", __func__, target->deviceName());

  } else eth.ethsd = NULL;



  /* I start by sending out the SYN pr0bez */

  for(pr0be = 0; pr0be < numports; pr0be++) {     if (o.scan_delay) enforce_scan_delay(NULL);     else if
(proxy->senddelay && pr0be > 0) usleep(proxy->senddelay);



    /* Maybe I should involve decoys in the picture at some point --

       but doing it the straightforward way (using the same decoys as

       we use in probing the proxy box is risky.  I'll have to think
```

```
      about this more. */
    send_tcp_raw(proxy->rawsd, eth.ethsd? &eth : NULL,
                                 proxy->host.v4hostip(), target->v4hostip(),
                                 o.ttl, false,
                                 o.ipoptions, o.ipoptionslen,
                                 proxy->probe_port, ports[pr0be], seq, 0, 0, TH_SYN, 0, 0,
                                 (u8 *) "\x02\x04\x05\xb4", 4,
                                 o.extra_payload, o.extra_payload_length);
  }
  gettimeofday(&end, NULL);


  openports = -1;
  tries = 0;
  TIMEVAL_MSEC_ADD(probe_times[0], start, MAX(50, (target->to.srtt * 3/4) / 1000));
  TIMEVAL_MSEC_ADD(probe_times[1], start, target->to.srtt / 1000 );
  TIMEVAL_MSEC_ADD(probe_times[2], end, MAX(75, (2 * target->to.srtt +
                                                 target->to.rttvar) / 1000));
  TIMEVAL_MSEC_ADD(probe_times[3], end, MIN(4000, (2 * target->to.srtt +
                                                   (target->to.rttvar << 2 )) / 1000));   do {
if (tries == 2) dotry3 = (get_random_u8() > 200);
    if (tries == 3 && !dotry3)
      break; /* We usually want to skip the long-wait test */
```

```
  if (tries == 3 || (tries == 2 && !dotry3))

    lasttry = 1;



  gettimeofday(&now, NULL);

  sleeptime = TIMEVAL_SUBTRACT(probe_times[tries], now);

  if (!lasttry && proxyprobes_sent > 0 && sleeptime < 50000)

    continue; /* No point going again so soon */



  if (tries == 0 && sleeptime < 500)       sleeptime = 500;     if (o.debugging > 1) error("In preparation for idle scan probe try #%d, sleeping for %d usecs", tries, sleeptime);

  if (sleeptime > 0)

    usleep(sleeptime);



  newipid = ipid_proxy_probe(proxy, &sent, &rcvd);

  proxyprobes_sent += sent;

  proxyprobes_rcvd += rcvd;



  if (newipid > 0) {

    ipid_dist = ipid_distance(proxy->seqclass, proxy->latestid, newipid);

    /* I used to only do this if ipid_sit >= proxyprobes_sent, but I'd

           rather have a negative number in that case. */

    if (ipid_dist < proxyprobes_sent) {       if (o.debugging)           error("%s: Must have lost a sent packet because ipid_dist is %d while proxyprobes_sent is %d.", __func__, ipid_dist, proxyprobes_sent); /* I no longer whack timing here ... done at bottom. */       }       ipid_dist -= proxyprobes_sent;       if (ipid_dist > openports) {
```

```
            openports = ipid_dist;

            gettimeofday(&latestchange, NULL);

      } else if (ipid_dist < openports && ipid_dist >= 0) {

            /* Uh-oh.  Perhaps I dropped a packet this time */

            if (o.debugging > 1) {

              error("%s: Counted %d open ports in try #%d, but counted %d earlier ... probably a proxy_probe
problem", __func__, ipid_dist, tries, openports);

            }

            /* I no longer whack timing here ... done at bottom. */

      }

    }



    if (openports > numports || (numports <= 2 && (openports == numports)))

      break;

  } while(tries++ < 3);   if (proxyprobes_sent > proxyprobes_rcvd) {

    /* Uh-oh.  It looks like we lost at least one proxy probe packet */

    if (o.debugging) {

      error("%s: Sent %d probes; only %d responses.  Slowing scan.", __func__, proxyprobes_sent,
proxyprobes_rcvd);

    }

    proxy->senddelay += 5000;

    proxy->senddelay = MIN(proxy->max_senddelay, proxy->senddelay);

    /* No group size should be greater than .5s of send delays */
```

```
    proxy->current_groupsz = MAX(proxy->min_groupsz, MIN(proxy->current_groupsz, 500000 / (proxy->senddelay+1)));

  } else {

    /* Yeah, we got as many responses as we sent probes.  This calls for a

       very light timing acceleration ... */

    proxy->senddelay = (int) (proxy->senddelay * 0.95);

    if (proxy->senddelay < 500) proxy->senddelay = 0;

    proxy->current_groupsz = MAX(proxy->min_groupsz, MIN(proxy->current_groupsz, 500000 / (proxy->senddelay+1)));

  }



  if ((openports > 0) && (openports <= numports)) {      /* Yeah, we found open ports... lets adjust the timing ... */     if (o.debugging > 2) error("%s:  found %d open ports (out of %d) in %lu usecs", __func__, openports, numports, (unsigned long) TIMEVAL_SUBTRACT(latestchange, start));

    if (sent_time) *sent_time = start;

    if (rcv_time) *rcv_time = latestchange;

  }

  if (newipid > 0) proxy->latestid = newipid;

  if (eth.ethsd) { eth.ethsd = NULL; } /* don't need to close it due to caching */

  return openports;

}



/* The job of this function is to use the idle scan technique to count

   the number of open ports in the given list.  Under the covers, this
```

```
   function just farms out the hard work to another function. */

static int idlescan_countopen(struct idle_proxy_info *proxy,

                              Target *target, u16 *ports, int numports,

                              struct timeval *sent_time, struct timeval *rcv_time) {

  int tries = 0;

  int openports;



  do {

    openports = idlescan_countopen2(proxy, target, ports, numports, sent_time,

                                    rcv_time);

    tries++;

    if (tries == 6 || (openports >= 0 && openports <= numports))       break;     if (o.debugging) {       error("%s: In try #%d, counted %d open ports out of %d.  Retrying", __func__, tries, openports, numports);     }     /* Sleep for a little while -- maybe proxy host had brief birst of       traffic or similar problem */     sleep(tries * tries);     if (tries == 5)       sleep(45); /* We're gonna give up if this fails, so we will be a bit patient */     /* Since the host may have received packets while we were sleeping,       lets update our proxy IP ID counter */     proxy->latestid = ipid_proxy_probe(proxy, NULL, NULL);

  } while(1);



  if (openports < 0 || openports > numports ) {

    /* Oh f*ck!!!! */

    fatal("Idle scan is unable to obtain meaningful results from proxy %s (%s).  I'm sorry it didn't work out.", proxy->host.HostName(),

          proxy->host.targetipstr());

  }
```

```
  if (o.debugging > 2) error("%s: %d ports found open out of %d, starting with %hu", __func__, openports,
numports, ports[0]);



  return openports;

}



/* Recursively idle scans scans a group of ports using a depth-first

   divide-and-conquer strategy to find the open one(s). */



static int idle_treescan(struct idle_proxy_info *proxy, Target *target,

                         u16 *ports, int numports, int expectedopen) {



  int firstHalfSz = (numports + 1)/2;

  int secondHalfSz = numports - firstHalfSz;

  int flatcount1, flatcount2;

  int deepcount1 = -1, deepcount2 = -1;

  struct timeval sentTime1, rcvTime1, sentTime2, rcvTime2;

  int retrycount = -1, retry2 = -1;

  int totalfound = 0;

  /* Scan the first half of the range */
```

```
 if (o.debugging > 1) {

   error("%s: Called against %s with %d ports, starting with %hu. expectedopen: %d", __func__, target-
>targetipstr(), numports, ports[0], expectedopen);

   error("IDLE SCAN TIMING: grpsz: %.3f delay: %d srtt: %d rttvar: %d",

                 proxy->current_groupsz, proxy->senddelay, target->to.srtt,

                 target->to.rttvar);

 }



 flatcount1 = idlescan_countopen(proxy, target, ports, firstHalfSz,

                                                 &sentTime1, &rcvTime1);



 if (firstHalfSz > 1 && flatcount1 > 0) {

   /* A port appears open!  We dig down deeper to find it ... */

   deepcount1 = idle_treescan(proxy, target, ports, firstHalfSz, flatcount1);

   /* Now we assume deepcount1 is right, and adjust timing if flatcount1 was

      wrong. */

   adjust_idle_timing(proxy, target, flatcount1, deepcount1);

 }



 /* I guess we had better do the second half too ... */



 flatcount2 = idlescan_countopen(proxy, target, ports + firstHalfSz,
```

```
                                        secondHalfSz, &sentTime2, &rcvTime2);


 if ((secondHalfSz) > 1 && flatcount2 > 0) {

   /* A port appears open!  We dig down deeper to find it ... */

   deepcount2 = idle_treescan(proxy, target, ports + firstHalfSz,

                                        secondHalfSz, flatcount2);

   /* Now we assume deepcount1 is right, and adjust timing if flatcount1 was

      wrong */

   adjust_idle_timing(proxy, target, flatcount2, deepcount2);

 }


 totalfound = (deepcount1 == -1)? flatcount1 : deepcount1;

 totalfound += (deepcount2 == -1)? flatcount2 : deepcount2;


 if ((flatcount1 + flatcount2 == totalfound) &&

     (expectedopen == totalfound || expectedopen == -1)) {


   if (flatcount1 > 0) {

     if (o.debugging > 1) {

               error("Adjusting timing -- idlescan_countopen correctly found %d open ports (out of %d, starting
with %hu)", flatcount1, firstHalfSz, ports[0]);

     }
```

```
    adjust_timeouts2(&sentTime1, &rcvTime1, &(target->to));
   }


  if (flatcount2 > 0) {
    if (o.debugging > 2) {
            error("Adjusting timing -- idlescan_countopen correctly found %d open ports (out of %d, starting
with %hu)", flatcount2, secondHalfSz,
                ports[firstHalfSz]);
    }
    adjust_timeouts2(&sentTime2, &rcvTime2, &(target->to));
   }
 }


 if (totalfound != expectedopen) {
  if (deepcount1 == -1) {
    retrycount = idlescan_countopen(proxy, target, ports, firstHalfSz, NULL,
                                                    NULL);
    if (retrycount != flatcount1) {
            /* We have to do a deep count if new ports were found and
               there are more than 1 total */
            if (firstHalfSz > 1 && retrycount > 0) {
              retry2 = retrycount;
```

```
                retrycount = idle_treescan(proxy, target, ports, firstHalfSz,

                                                        retrycount);

                adjust_idle_timing(proxy, target, retry2, retrycount);

              } else {

                if (o.debugging)

                  error("Adjusting timing because my first scan of %d ports, starting with %hu found %d open,
while second scan yielded %d", firstHalfSz, ports[0], flatcount1, retrycount);

                adjust_idle_timing(proxy, target, flatcount1, retrycount);

              }

              totalfound += retrycount - flatcount1;

              flatcount1 = retrycount;



              /* If our first count erroneously found and added an open port,

                 we must delete it */

              if (firstHalfSz == 1 && flatcount1 == 1 && retrycount == 0)

                target->ports.forgetPort(ports[0], IPPROTO_TCP);



    }

  }



  if (deepcount2 == -1) {

    retrycount = idlescan_countopen(proxy, target, ports + firstHalfSz,
```

```
                                                secondHalfSz, NULL, NULL);

    if (retrycount != flatcount2) {

            if (secondHalfSz > 1 && retrycount > 0) {

              retry2 = retrycount;

              retrycount = idle_treescan(proxy, target, ports + firstHalfSz,

                                                    secondHalfSz, retrycount);

              adjust_idle_timing(proxy, target, retry2, retrycount);

            } else {

              if (o.debugging)

                error("Adjusting timing because my first scan of %d ports, starting with %hu found %d open,
while second scan yeilded %d", secondHalfSz, ports[firstHalfSz], flatcount2, retrycount);

              adjust_idle_timing(proxy, target, flatcount2, retrycount);

            }



            totalfound += retrycount - flatcount2;

            flatcount2 = retrycount;



            /* If our first count erroneously found and added an open port,

               we must delete it. */

            if (secondHalfSz == 1 && flatcount2 == 1 && retrycount == 0)

              target->ports.forgetPort(ports[firstHalfSz], IPPROTO_TCP);
```

```
      }
    }
  }

  if (firstHalfSz == 1 && flatcount1 == 1)
    target->ports.setPortState(ports[0], IPPROTO_TCP, PORT_OPEN);

  if ((secondHalfSz == 1) && flatcount2 == 1)
    target->ports.setPortState(ports[firstHalfSz], IPPROTO_TCP, PORT_OPEN);
  return totalfound;

}

/* The very top-level idle scan function -- scans the given target
   host using the given proxy -- the proxy is cached so that you can keep
   calling this function with different targets. */
void idle_scan(Target *target, u16 *portarray, int numports,
                        char *proxyName, const struct scan_lists * ports) {

  static char lastproxy[MAXHOSTNAMELEN + 1] = ""; /* The proxy used in any previous call */
  static struct idle_proxy_info proxy;
  int groupsz;
```

```
  int portidx = 0; /* Used for splitting the port array into chunks */

  int portsleft;

  char scanname[128];

  Snprintf(scanname, sizeof(scanname), "idle scan against %s", target->NameIP());

  ScanProgressMeter SPM(scanname);



  if (numports == 0) return; /* nothing to scan for */

  if (!proxyName) fatal("idle scan requires a proxy host");



  if (*lastproxy && strcmp(proxyName, lastproxy))

    fatal("%s: You are not allowed to change proxies midstream.  Sorry", __func__);

  assert(target);



  if (target->timedOut(NULL))

    return;



  if (target->ifType() == devt_loopback) {

    log_write(LOG_STDOUT, "Skipping Idle Scan against %s -- you can't idle scan your own machine
(localhost).\n", target->NameIP());

    return;

  }
```

```
  target->startTimeOutClock(NULL);



  /* If this is the first call,  */

  if (!*lastproxy) {

    initialize_idleproxy(&proxy, proxyName, target->v4hostip(), ports);

    strncpy(lastproxy, proxyName, sizeof(lastproxy));

  }



  /* If we don't have timing infoz for the new target, we'll use values

     derived from the proxy */

  if (target->to.srtt == -1 && target->to.rttvar == -1) {

    target->to.srtt = MAX(200000,2 * proxy.host.to.srtt);

    target->to.rttvar = MAX(10000, MIN(proxy.host.to.rttvar, 2000000));

    target->to.timeout = target->to.srtt + (target->to.rttvar << 2);    } else {      target->to.srtt = MAX(target->to.srtt, proxy.host.to.srtt);

    target->to.rttvar = MAX(target->to.rttvar, proxy.host.to.rttvar);

    target->to.timeout = target->to.srtt + (target->to.rttvar << 2);

  }



  /* Now I guess it is time to let the scanning begin!  Since idle

     scan is sort of tree structured (we scan a group and then divide

     it up and drill down in subscans of the group), we split the port
```

```
    space into smaller groups and then call a recursive
    divide-and-counquer function to find the open ports */
  while(portidx < numports) {
    portsleft = numports - portidx;
    /* current_groupsz is doubled below because idle_subscan cuts in half */
    groupsz = MIN(portsleft, (int) (proxy.current_groupsz * 2));
    idle_treescan(&proxy, target, portarray + portidx, groupsz, -1);
    portidx += groupsz;
  }

  char additional_info[14];
  Snprintf(additional_info, sizeof(additional_info), "%d ports", numports);
  SPM.endTask(NULL, additional_info);

  /* Now we go through the ports which were scanned but not determined
     to be open, and add them in the "closed|filtered" state */
  for(portidx = 0; portidx < numports; portidx++) {      if (target->ports.portIsDefault(portarray[portidx],
IPPROTO_TCP)) {
      target->ports.setPortState(portarray[portidx], IPPROTO_TCP, PORT_CLOSEDFILTERED);
      target->ports.setStateReason(portarray[portidx], IPPROTO_TCP, ER_NOIPIDCHANGE, 0, NULL);
    } else
      target->ports.setStateReason(portarray[portidx], IPPROTO_TCP, ER_IPIDCHANGE, 0, NULL);
```

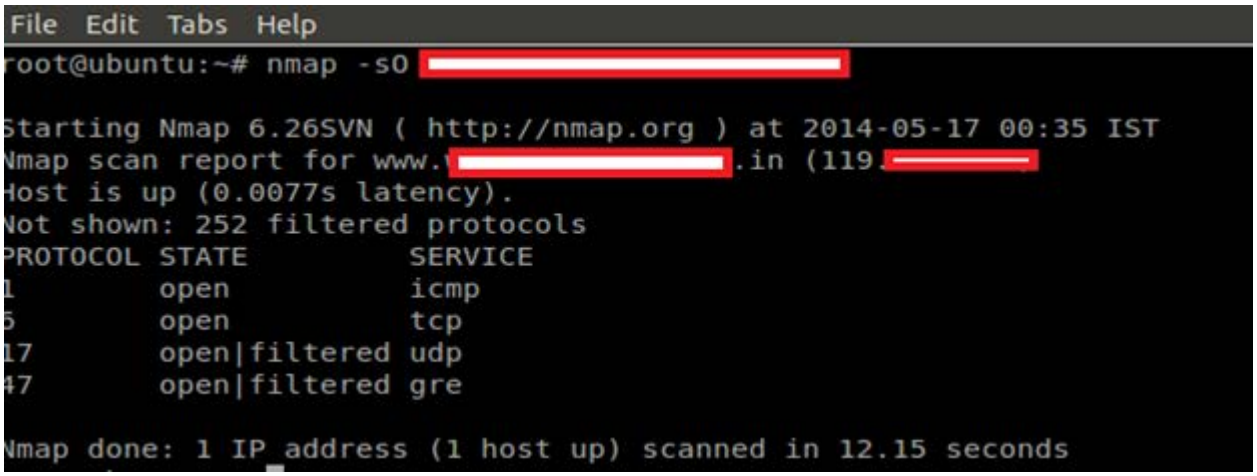
```
}


target->stopTimeOutClock(NULL);

return;

}
```

As you can see, this is my screenshot of the C++ source file. Just compile it and run it to get the results.

OK, let’s start with our original scan method.

IP Protocol Scan
IP protocol scan allows you to determine which IP protocols (TCP, ICMP, IGMP, etc.) are supported by target machines. This isn’t technically a port scan, since it cycles through IP protocol numbers rather than TCP or UDP port numbers.

Command: nmap –sO target



FTP Bounce Scan
This allows a user to connect to one FTP server, then ask that files be sent to a third-party server. Simply it asks the FTP server to send a file to each interesting port of a target host in turn. The error message will describe whether the port is open or not. This is a good way to bypass firewalls, because organizational FTP servers are often placed where they have more access to other internal hosts than any old Internet host would. It takes an argument of the form <username>:<password>@<server>:<port>. <Server> is the name or IP address of a vulnerable FTP server.

Command: nmap –b ftp rely host. Also, this can be used for port bounce attack.
nmap -T0 -b username:password@ftpserver.tld:21 victim.tld
This uses the username “username”, the password “password”, the FTP server “ftpserver.tld” and port 21 on said server to scan victim.tld.

If the FTP server supports anonymous logins, just forget about the username:password@ part and Nmap will assume it allows -anonymous. You may omit :21 if the FTP port is 21, however, some people configure FTP on weird ports as an attempt at “security”.

Port Specification and Scan Order
In addition to all of the scan methods discussed previously, Nmap offers options for specifying which ports are scanned and whether the scan order is randomized or sequential. By default, Nmap scans the most common 1,000 ports for each protocol.

-p <port ranges> (Only scan specified ports)

This option specifies which ports you want to scan and overrides the default. Individual port numbers are OK, as are ranges separated by a hyphen (e.g. 1-1023). The beginning and/or end values of a range may be omitted, causing Nmap to use 1 and 65535, respectively. So you can specify -p- to scan ports from 1 through 65535. Scanning port zero is allowed if you specify it explicitly.

Nmap –p 1-1023 target

When scanning a combination of protocols (e.g. TCP and UDP), you can specify a particular protocol by preceding the port numbers by T: for TCP, U: for UDP, S: for SCTP, or P: for IP Protocol.

nmap -p U:53,111,137,T:21-25,80,139,8080 target

-F (Fast (limited port) scan)

Specifies that you wish to scan fewer ports than the default. Normally Nmap scans the most common 1,000 ports for each scanned protocol. With -F, this is reduced to 100.

nmap –F target
-r (Don't randomize ports)
By default, Nmap randomizes the scanned port order (except that certain commonly accessible ports are moved near the beginning for efficiency reasons). This randomization is normally desirable, but you can specify -r for sequential (sorted from lowest to highest) port scanning instead.

nmap –r target

So this is the end of this part of the series. In the next part, I will go through advanced firewall evasion and custom creation of exploits with Nmap.

Warning: The above mentioned malicious attack conducted on the lab has been given prior permission by the owner of website or admin. The above is meant for only educational purposes. So do not use for any personal intent, as it is prone to cyber attack.
References:
http://www.kyuzz.org/antirez/papers/dumbscan.html
http://www.kyuzz.org/antirez/papers/moreipid.html
http://en.wikipedia.org/wiki/Idle_scan

Nmap Cheat Sheet: From Discovery to Exploits, Part 3: Gathering Additional Information about Host and Network

As we discussed before, this is our third installment in our Nmap series.

Nmap is well known for port scanning, port discovery, and port mapping. But we can do many more things by the Nmap NSE script. We can do email fingerprinting, retrieve a Whois record, use UDP services, etc.

Discovering Geographical Location
Gorjan Petrovski submitted Nmap NSE scripts that help us geo locate a remote IP address: ip-geolocation-maxmind, ip-geolocation-ipinfodb, and ipgeolocation-geobytes.

This will show us how to set up and use the geo location scripts included with Nmap NSE.

ip-geolocation-maxmind
For the NSE script to be run under Nmap, download Maxmind's city database from http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz. Extract it to your local Nmap data folder ($NMAP_DATA/nselib/data/).
Fire up the command line and enter the command to download the scripts:

Wget http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
Put into the /usr/local/share/nmap/nselib/data and run the following command:

nmap –script ip-geolocation-* ip

Submitting a new geo-location provider
Sometimes the location may be wrong, because the location depends upon the maxmind database. So we have to submit a new database to Nmap [NSE script which is Nmap Dev, so that we can develop our own Nmap].

Getting information from WHOIS records
WHOIS records often contain important data such as the registrar name and contact information. System administrators have been using WHOIS for years now, and although here are many tools available to query this protocol, Nmap proves itself invaluable because of its ability to deal with IP ranges and hostname lists. Open up the command line and write the commands like:

nmap –script whois target

```
File  Edit  Tabs  Help
root@ubuntu:~# nmap --script whois scanme.nmap.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-06-14 17:33 IST
Nmap scan report for scanme.nmap.org (74.207.244.221)
Host is up (0.00051s latency).
All 1000 scanned ports on scanme.nmap.org (74.207.244.221) are filtered

Host script results:
| whois: Record found at whois.arin.net
| netrange: 74.207.224.0 - 74.207.255.255
| netname: LINODE-US
| orgname: Linode
| orgid: LINOD
| country: US stateprov: NJ
|
| orgtechname: Linode Network Operations
|_orgtechemail: support@linode.com

Nmap done: 1 IP address (1 host up) scanned in 63.90 seconds
root@ubuntu:~#
```

The argument –script whois tells Nmap to query a Regional Internet Registries WHOIS database in order to obtain the records of a given target. This script uses the IANA's Assignments Data to select the RIR and it caches the results locally. Alternatively, we could override this behavior and select the order of the service providers to use in the argument whodb:

nmap –script whois –script-args whois.whodb=arin+ripe+afrinic <target>

```
map done: 1 IP address (1 host up) scanned in 63.90 seconds
oot@ubuntu:~# nmap --script whois --script-args whois.whodb=arin+ripe+afrinic scanme.nmap.org

tarting Nmap 6.26SVN ( http://nmap.org ) at 2014-06-14 17:37 IST
arning: 74.207.244.221 giving up on port because retransmission cap hit (10).
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44191 > 74.207.244.221:25734 S ttl=44 id=62139 iplen=44  seq=2999317366 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44192 > 74.207.244.221:25734 S ttl=59 id=43566 iplen=44  seq=2999251831 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44193 > 74.207.244.221:25734 S ttl=43 id=35713 iplen=44  seq=2999448436 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44194 > 74.207.244.221:25734 S ttl=55 id=26992 iplen=44  seq=2999382901 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44195 > 74.207.244.221:25734 S ttl=50 id=46543 iplen=44  seq=2999055218 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44196 > 74.207.244.221:25734 S ttl=37 id=15195 iplen=44  seq=2998989683 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44197 > 74.207.244.221:25734 S ttl=37 id=11312 iplen=44  seq=2999186288 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44198 > 74.207.244.221:25734 S ttl=43 id=55877 iplen=44  seq=2999120753 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44199 > 74.207.244.221:25734 S ttl=51 id=13394 iplen=44  seq=2999841662 win=1024 <mss 1460>
endto in send_ip_packet_sd: sendto(5, packet, 44, 0, 74.207.244.221, 16) => Network is unreachable
ffending packet: TCP 192.168.223.131:44200 > 74.207.244.221:25734 S ttl=56 id=63116 iplen=44  seq=2999776127 win=1024 <mss 1460>
mitting future Sendto error messages now that 10 have been shown.  Use -d2 if you really want to see them.
```

This script will query, sequentially, a list of WHOIS providers until the record or a referral to the record is found. To ignore the referral records, use the value nofollow:

nmap –script whois –script-args whois.whodb=nofollow <target>

To query the WHOIS records of a hostname list (-iL <input file>) without launching a port scan (-sn). We can also use other options for Nmap. Enter the following Nmap command:

nmap -sn –script whois -v -iL hosts.txt (hosts.txt contains a list of hosts or IP addresses)

Disabling cache

Sometimes cached responses will be preferred over querying the WHOIS service, and this might prevent the discovery of an IP address assignment. To disable the cache we could set the script argument whodbto nocache:

nmap -sn –script whois –script-args whois.whodb=nocache scanme.nmap.org

Checking if a host is known for malicious activities

Nmap allows us to systematically check if a host is known for distributing malware or being used in phishing attacks, with some help from the Google Safe Browsing API.

This recipe shows system administrators how to check if a host has been flagged by Google's Safe Browsing Service as being used in phishing attacks or distributing malware.

We have to use the script http-google-malware which depends on Google's Safe Browsing service and it requires you to register to get an API key. Register at:

http://code.google.com/apis/safebrowsing/key_signup.html

Open your favorite terminal and type:

nmap -p80 –script http-google-malware –script-args http-google-malware.api=<API> <target>

The script http-google-malwarequeries Google Safe Browsing Service determines if a host is suspected to be malicious. This service is used by web browsers such as Mozilla Firefox and Google Chrome to protect its users, and the lists are updated very frequently. Check the following command:

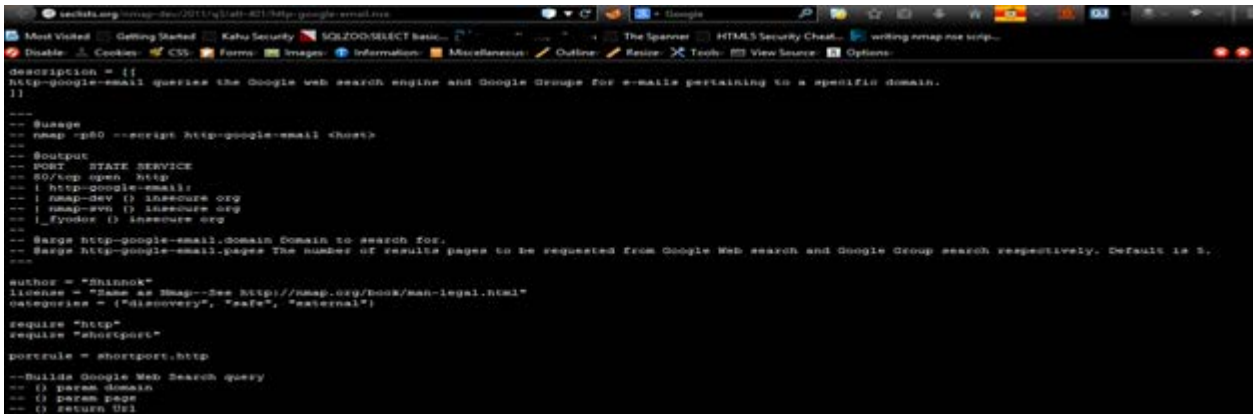
nmap -p80 –script http-google-malware -v scanme.nmap.org

```
root@ubuntu:~# nmap -p80 --script http-google-malware -v scanme.nmap.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-06-27 20:33 IST
NSE: Loaded 1 scripts for scanning.
NSE: Script Pre-scanning.
Initiating Ping Scan at 20:34
Scanning scanme.nmap.org (74.207.244.221) [4 ports]
Completed Ping Scan at 20:34, 0.05s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 20:34
Completed Parallel DNS resolution of 1 host. at 20:34, 2.41s elapsed
Initiating SYN Stealth Scan at 20:34
Scanning scanme.nmap.org (74.207.244.221) [1 port]
Completed SYN Stealth Scan at 20:34, 0.21s elapsed (1 total ports)
NSE: Script scanning 74.207.244.221.
Nmap scan report for scanme.nmap.org (74.207.244.221)
Host is up (0.00098s latency).
PORT   STATE    SERVICE
80/tcp filtered http

NSE: Script Post-scanning.
Read data files from: /usr/local/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 5.40 seconds
           Raw packets sent: 6 (240B) | Rcvd: 1 (40B)
```

Collecting valid e-mail accounts

Checking for an email address is very useful for penetration tester, since this information can also useful for further attacks like phishing attack, brute force attack, etc. Nmap gives the facility to perform discovery of email address.

To run these methods we have to run the NMAP script. The script http-google-email is not included in Nmap's official repository. So we need to download it from http://seclists.org/nmap-dev/2011/q3/att-401/http-google-email.nse and copy it to our local scripts directory. As you can see, the following screenshot has details for the email collector script:



But as a security researcher we must understand the script, because the script may contain malicious things. So the following covers details regarding the script. I have already downloaded the script below so that you do not have to download again, you can directly use this script:

description = [[

http-google-email queries the Google web search engine and Google Groups for e-mails pertaining to a specific domain.

]]

-- @usage

-- nmap -p80 --script http-google-email <host>

--

-- @output

-- PORT   STATE SERVICE

-- 80/tcp open http

-- | http-google-email:

-- | nmap-dev () insecure org

-- | nmap-svn () insecure org

-- |_fyodor () insecure org

--

-- @args http-google-email.domain Domain to search for.

-- @args http-google-email.pages The number of results pages to be requested from Google Web search and Google Group search respectively. Default is 5.

---

author = "Shinnok"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

```
categories = {"discovery", "safe", "external"}
```

-- This tells about HTTP protocol:

```
require "http"
require "shortport"

portrule = shortport.http
```

-- Declare the variable to perform the search:

```
--Builds Google Web Search query
-- () param domain
-- () param page
-- () return Url
```

-- HTTP query method to perform the search:

```
local function google_search_query(domain, page)
  return string.format("http://www.google.com/search?q=%%40%s&hl=en&lr=&ie=UTF-8&start=%s&sa=N",
domain, page)
end
```

-- Building Google group search:

```
--Builds Google Groups Search query
-- () param domain
-- () param page
-- () return Url
local function google_groups_query(domain, page)
  return string.format("http://groups.google.com/groups?q=%s&hl=en&lr=&ie=UTF-8&start=%s&sa=N",
domain, page)
end
```

-- Here the main program starts. It will retrieve 50 pages per search:

```
---
--MAIN
---
action = function(host, port)
  local pages = 50
  local target
  local emails = {}
```

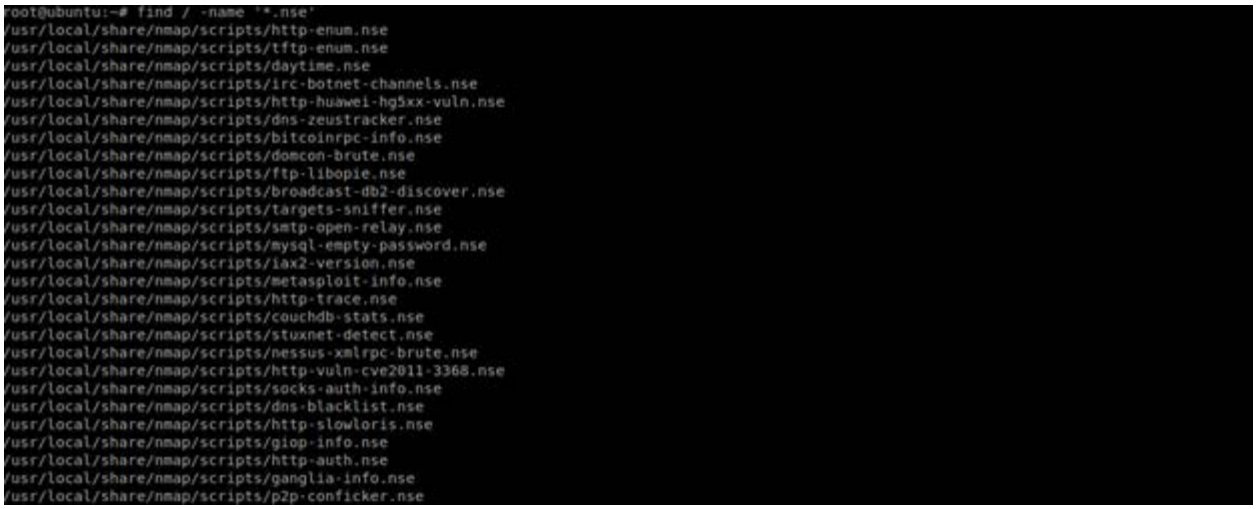
```
 if(stdnse.get_script_args("http-google-email.pages")) then
    pages = stdnse.get_script_args("http-google-email.pages")*10
 end

 -- Check if we have the domain argument passed
 if(stdnse.get_script_args("http-google-email.domain")) then
  target = stdnse.get_script_args("http-google-email.domain")
 else
  -- Verify that we have a hostname available
  if not(host.targetname) then
   return string.format("[ERROR] Host can not be resolved to a domain name.")
  else
   target = host.targetname
  end
 end

 stdnse.print_debug(1, "%s: Checking domain %s", SCRIPT_NAME, target)

-- Local search through Google:

 -- Google Web search

 for page=0, pages, 10 do
  local qry = google_search_query(target, page)
  local req = http.get_url(qry)
  stdnse.print_debug(2, "%s", qry)
  stdnse.print_debug(2, "%s", req.body)

  body = req.body:gsub('<em>', '')
  body = body:gsub('</em>', '')
  if body then
   local found = false
   for email in body:gmatch('[A-Za-z0-9%.%%%+%-]+@' .. target) do
    for _, value in pairs(emails) do
     if value == email then
      found = true
     end
    end
    if not found then
     emails[#emails+1] = email
    end
   end
  end
 end

 -- Google Groups search

 for page=0, pages, 10 do
  local qry = google_groups_query(target, page)
  local req = http.get_url(qry)
  stdnse.print_debug(2, "%s", qry)
  stdnse.print_debug(2, "%s", req.body)
```

```
    body = req.body:gsub('<b>', '')
    body = body:gsub('</b>', '')
    if body then
      local found = false
      for email in body:gmatch('[A-Za-z0-9%.%%%+%-]+@' .. target) do
        for _, value in pairs(emails) do
          if value == email then
            found = true
          end
        end
        if not found then
          emails[#emails+1] = email
        end
      end
    end
  end


  if #emails > 0 then
    return "\n" .. stdnse.strjoin("\n", emails)
  end
end
```

The easiest method to find all nse scripts is to use the find command like below.

```
find / -name '*.nse'
```

The above command will enlist all .nse files for the directory. In general the path may be usr/share/nmap/scripts. Please find the below commands and screenshot for reference:



Clearly it mentions all scripts and relevant directory paths. After getting the path, just download the script with the “wget” command. You can also directly copy the script to the relevant directory.

```
root@ubuntu:/usr/local/share/nmap/scripts# wget http://seclists.org/nmap-dev/2011/q3/att-401/http-google-email.nse
--2014-07-01 20:48:28--  http://seclists.org/nmap-dev/2011/q3/att-401/http-google-email.nse
Resolving seclists.org (seclists.org)... 173.255.243.189, 2600:3c01::f03c:91ff:fe70:d085
Connecting to seclists.org (seclists.org)|173.255.243.189|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3328 (3.2K) [text/plain]
Saving to: `http-google-email.nse'

100%[======================================>] 3,328       --.-K/s   in 0.03s

2014-07-01 20:48:36 (119 KB/s) - `http-google-email.nse' saved [3328/3328]

root@ubuntu:/usr/local/share/nmap/scripts#
```

Check the script for confirmation. Follow the below screenshot.

```
Applications Places                                                    8:50 PM
root@ubuntu: /usr/local/share/nmap/scripts
File  Edit  Tabs  Help
http-google-email.nse
root@ubuntu:/usr/local/share/nmap/scripts# cat "http-google-email.nse"
description = [[
http-google-email queries the Google web search engine and Google Groups for e-mails pertaining to a specific domain.
]]

---
-- @usage
-- nmap -p80 --script http-google-email <host>
--
-- @output
-- PORT   STATE SERVICE
-- 80/tcp open  http
-- | http-google-email:
-- | nmap-dev () insecure org
-- | nmap-svn () insecure org
-- |_fyodor () insecure org
--
-- @args http-google-email.domain Domain to search for.
-- @args http-google-email.pages The number of results pages to be requested from Google Web search and Google Group search respectively. Defa
ult is 5.
---

author = "Shinnok"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"discovery", "safe", "external"}

require "http"
require "shortport"

portrule = shortport.http

--Builds Google Web Search query
-- () param domain
-- () param page
-- () return Url
local function google_search_query(domain, page)
  return string.format("http://www.google.com/search?q=%%40%s&hl=en&lr=&ie=UTF-8&start=%s&sa=N", domain, page)

root@ubuntu: /usr/loc...  [scripts]
```

After copying http-google-email.nse, we should update the script database with the following command:

nmap –script-updatedb

After updating the scrip,t fire up the command line with the following script:

```
nmap -p80 –script http-google-email <target>
```

NSE Script Argument

The flag –script–args argument is used to set the argument of NSE script. For example:

nmap -sV –script http-title –script-args http.useragent=”Mozilla 999″ <target>

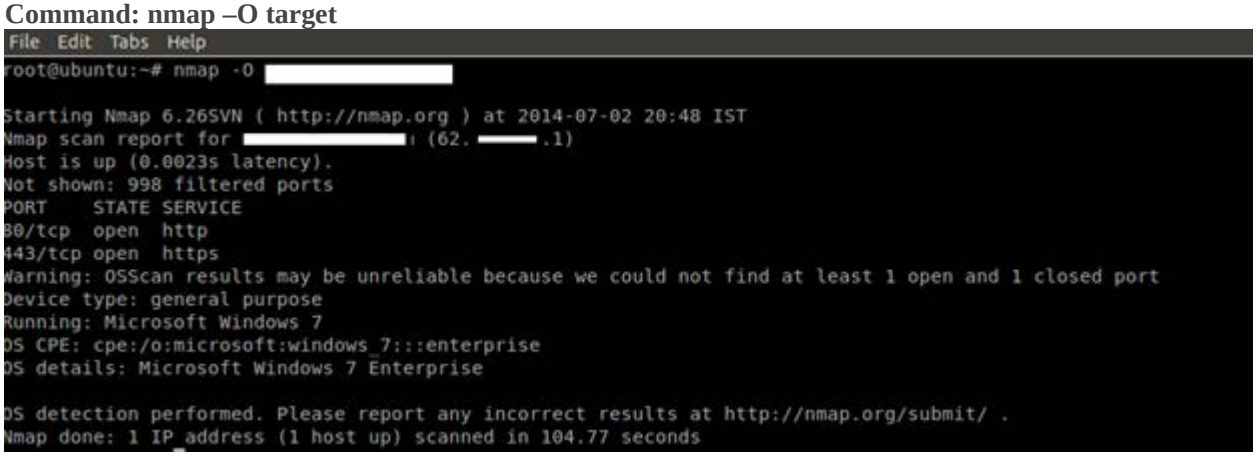
The above will work on HTTP Library argument “useragent”.

Additional Host Information And Further Pentesting

As a professional security tester we should go a further step to gain additional information about the network or host which will boost our pentesting.

The –O parameter used to detect the target operating system:

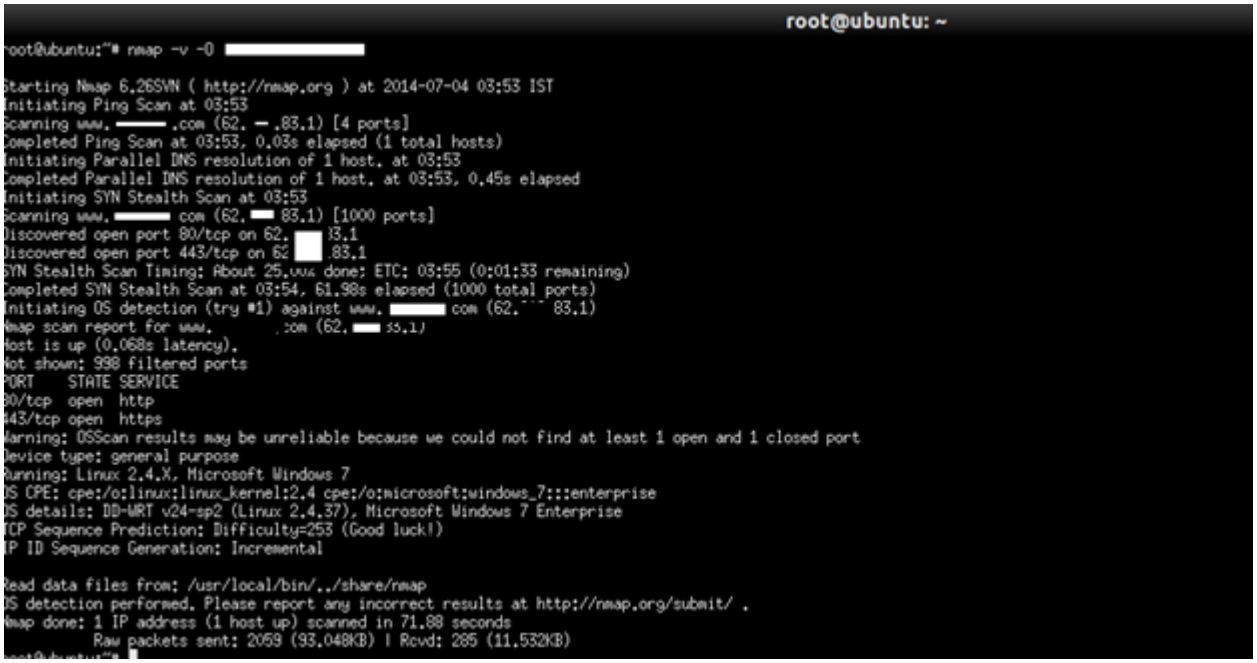
Command: nmap –O target



Operating system detection is performed by analyzing responses from the target for a set of predictable characteristics which can be used to identify the type of OS on the remote system. In order for the OS scan work perfectly there must be at least one open and one closed port on the target system. When scanning multiple targets, the –osscan-limit option can be combined with -O to instruct Nmap not to OS scan hosts that do not meet this criteria.

Multiple options for nmap can be used, like –v.

Ex: nmap –v –O <target>



In some cases, Nmap will not be able to determine the OS. it will provide a fingerprint which can be submitted to Nmap' s OS database at www.nmap.org/submit/
By submitting the fingerprint generated and correctly identifying the target system's operating system, we can improve the accuracy of Nmap' s OS detection feature in future releases.

Guessing the Operating System
If Nmap is unable to determine the operating system, we can use the –osscan option to force Nmap into discovering the OS.

Note: This option is useful when Nmap is unable to determine the discovered OS
Command: nmap -O –osscan-guess target

```
File  Edit  Tabs  Help
root@ubuntu:~# nmap -O --osscan-guess [redacted]

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-04 04:05 IST
Nmap scan report for www.[redacted].com (106.[redacted].240)
Host is up (0.055s latency).
Other addresses for www.[redacted].com (not scanned): 106.[redacted].246
rDNS record for 106.[redacted].240: [redacted]
Not shown: 998 filtered ports
PORT    STATE SERVICE
80/tcp  open  http
443/tcp open  https
Warning: OSScan results may be unreliable because we could not find at least 1 o
pen and 1 closed port
Device type: general purpose|storage-misc|VoIP phone
Running (JUST GUESSING): Linux 2.4.X (98%), Microsoft Windows 7 (94%), BlueArc e
mbedded (93%), Pirelli embedded (90%)
OS CPE: cpe:/o:linux:linux_kernel:2.4 cpe:/o:microsoft:windows_7:::enterprise cp
e:/h:bluearc:titan_2100 cpe:/h:pirelli:dp-10
Aggressive OS guesses: DD-WRT v24-sp2 (Linux 2.4.37) (98%), Microsoft Windows 7
Enterprise (94%), BlueArc Titan 2100 NAS device (93%), Pirelli DP-10 VoIP phone
(90%)
No exact OS matches for host (test conditions non-ideal).

OS detection performed. Please report any incorrect results at http://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 65.98 seconds
```

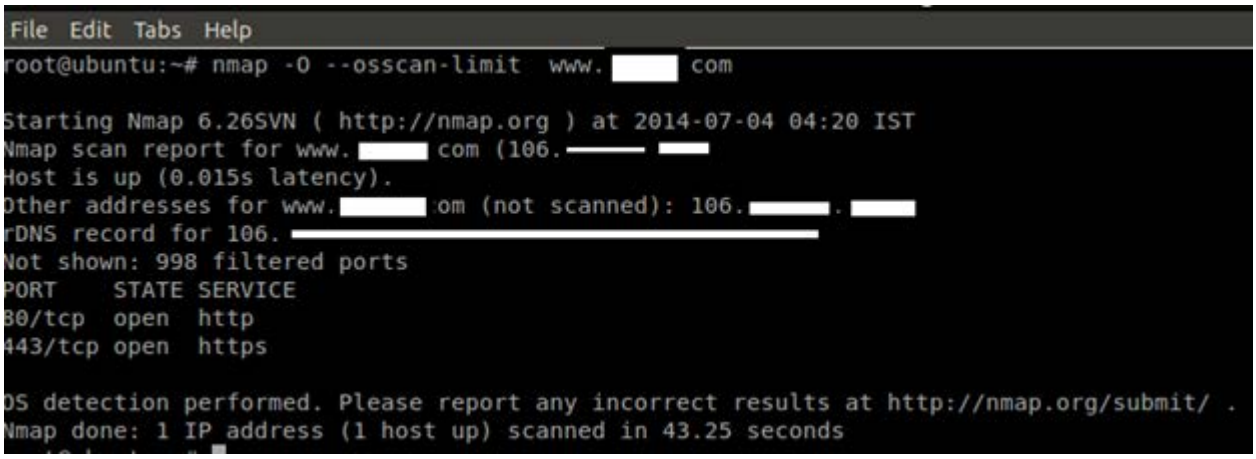
It will list all possible matches of operating system in Nmap's script database.

The –fuzzy option can be used as a shortcut for the above.

```
root@ubuntu:~# nmap -O --fuzzy www.[redacted].com

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-04 04:10 IST
Nmap scan report for www.[redacted].com (106.[redacted])
Host is up (0.051s latency).
Other addresses for www.[redacted].com (not scanned): 106.[redacted]
rDNS record for 106.[redacted]
Not shown: 998 filtered ports
PORT    STATE SERVICE
80/tcp  open  http
443/tcp open  https
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running: Linux 2.4.X
OS CPE: cpe:/o:linux:linux_kernel:2.4
OS details: DD-WRT v24-sp2 (Linux 2.4.37)

OS detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 60.42 seconds
```

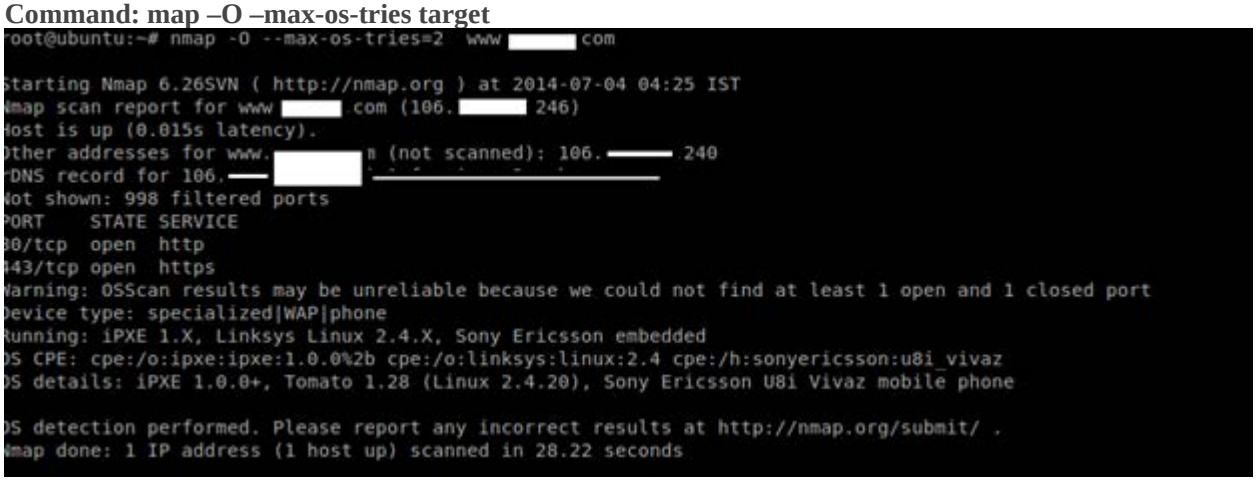
–osscan-limit (Limit OS detection to promising targets)

OS detection is far more effective if at least one open and one closed TCP port are found. Set this option and Nmap will not even try OS detection against hosts that do not meet this criteria. This can save substantial time, particularly on -Pn scans against many hosts. It only matters when OS detection is requested with -O or -A.



–max-os-tries (Set the maximum number of OS detection tries against a target)
When Nmap performs OS detection against a target and fails to find a perfect match, it usually repeats the attempt. By default, Nmap tries five times if conditions are favorable for OS fingerprint submission, and twice when conditions aren't so good. Specifying a lower –max-os-tries value (such as 1) speeds Nmap up, though we miss out on retries which could potentially identify the OS.

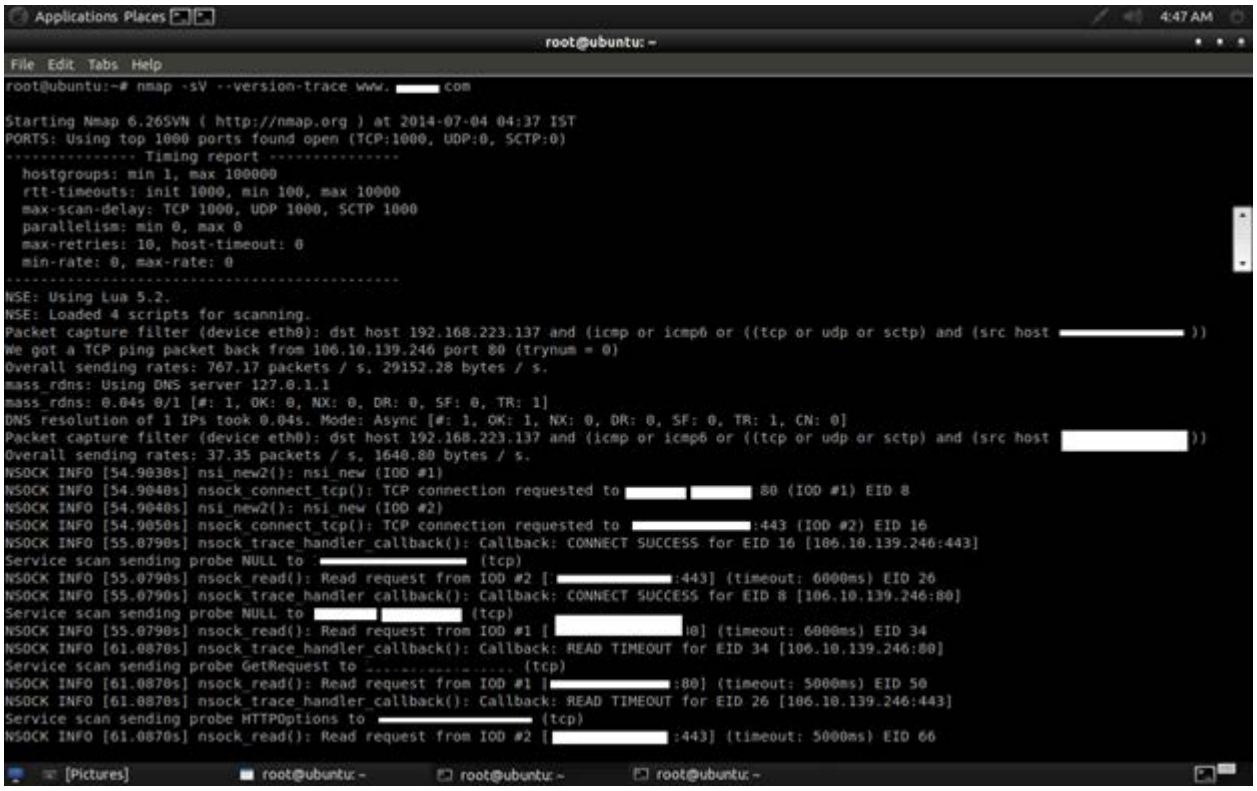
Command: map –O –max-os-tries target



The above options show the extra discovery options for OS discovery. We can easily discover the difference between the above options.

Troubleshooting Version Scans
The –version-trace option can be enabled to display verbose version scan activity.

Usage syntax: nmap -sV –version-trace [target]



The –version-trace option can be helpful for debugging problems or to gain additional information about the target system.

Perform an RPC Scan
The -sR option performs a RPC (Remote Procedure Call) scan on the specified target.

```
root@ubuntu:~# nmap -sR [redacted]
WARNING: -sR is now an alias for -sV and activates version detection as well as
RPC scan.

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-13 23:41 IST
Nmap scan report for www.[redacted] [redacted];)
Host is up (0.018s latency).
Other addresses for www.[redacted] (not scanned): 106.[redacted]
rDNS record for [redacted]
Not shown: 998 filtered ports
PORT    STATE SERVICE VERSION
80/tcp  open  http?
443/tcp open  https?

Service detection performed. Please report any incorrect results at http://nmap.
org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 75.32 seconds
```

Usage syntax: nmap -sR [target]
The output of the -sR scan above displays information about RPC services running on the target system. RCP is most commonly associated with Unix and Linux systems specifically for the NFS (Network File System) service.

Nmap Scanning with –hostmap version
The script hostmap depends on external services, and the official version only supports BFK's

DNS Logger. Previously I told how to download and update the database with the local directory.

Download hostmap.nse with Bing support at https://secwiki.org/w/Nmap/.

External_Script_Library.

After copying it to our local script directory, update your script database by running the following command:

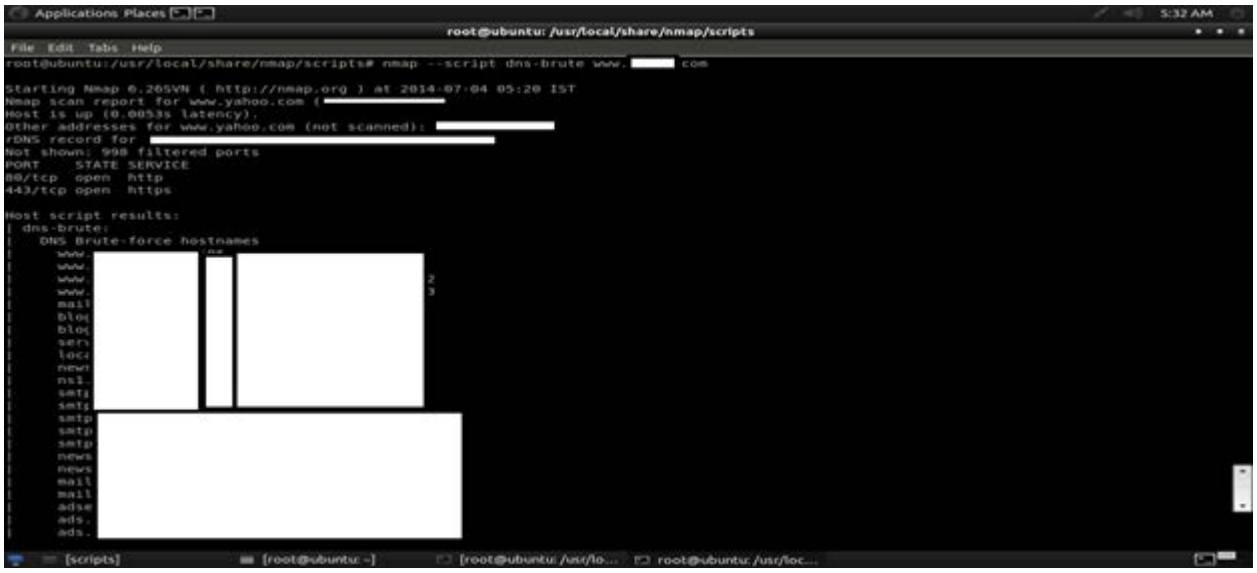
#nmap –script-updatedb
Open a terminal and enter the following command:
nmap -p80 –script hostmap nmap.org
The arguments –script hostmap -p80 tell Nmap to start the HTTP script hostmap and limit port scanning to port 80 to speed up this task.

This version of hostmap.nse queries two different web services: BFK's DNS Logger and ip2hosts.com. BFK's DNS Logger is a free service that collects its information from public DNS data and ip2hosts. Both of these services are free, and abusing them will most likely get you banned from the service.

Different search engine arguments are used for pentesting about hostname as follows:

nmap -p80 –script hostmap –script-args hostmap.provider=BING <target>
nmap -p80 –script hostmap –script-args hostmap.provider=BFK <target>
nmap -p80 –script hostmap –script-args hostmap.provider=ALL <target>
To save a hostname list for each IP scanned, use the argument hostmap.prefix. Setting this argument will create a file with a filename of <prefix><target> in our working directory:

nmap -p80 –script hostmap –script-args hostmap.prefix=HOSTSFILE <target>
Brute forcing DNS record
This is used for attempts to enumerate DNS hostnames by brute force guessing of common subdomains. With the dns-brute.srv argument, dns-brute will also try to enumerate common DNS SRV records.



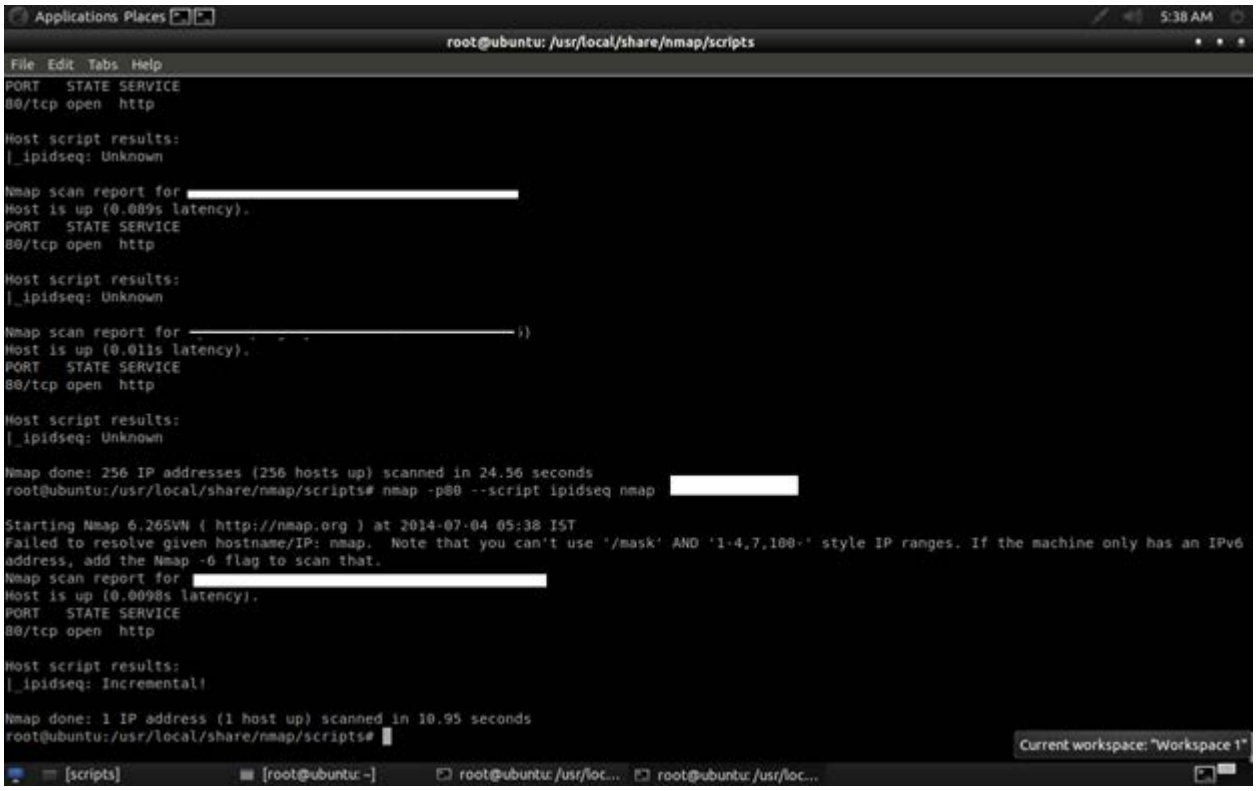
Spoofing the Origin of IP Port scan
Idle scanning is a very powerful technique, where Nmap takes advantage of an idle host with a predictable IP ID sequence number to spoof the origin IP of a port scan.

This technique illustrates how to find zombie hosts and use them to spoof your IP address when scanning a remote host with Nmap.

To launch an idle scan we need a zombie host. A zombie host is a machine with a predictable

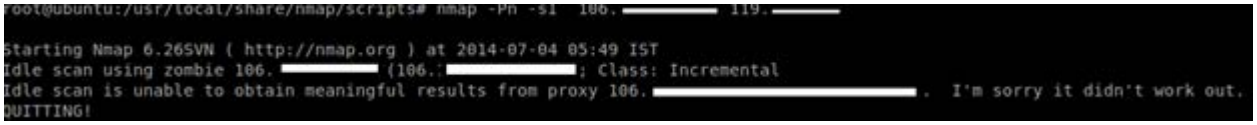
IP ID sequence number that will be used as the spoofed IP address. A good candidate must not be communicating with other hosts, in order to maintain the correct IP ID sequence number and avoid false positives. To find hosts with an incremental IP ID sequence, you could use the script ipidseq as follows:

#nmap -p80 –script ipidseq < ip>/24
#nmap -p80 –script ipidseq -iR 1000
Possible candidates will return the text incrementally in the script's output section:



To launch an idle scan, open your terminal and type the following command:

#nmap -Pn -sI <zombie host> <target>



I have already discussed idle scan in the previous part. Please go through it.
Idle scanning should work if the zombie host meets the previously-discussed requirements. If something did not work as expected, the returned error message should give us an idea of what went wrong.

Timing options for Nmap
What are timing options and why?
As a pentester we are using timing options for Nmap, but we should know why we are using timing options and why.

When we are doing Nmap many times we should come up across a firewall which may block our request for a certain time response. To speed up Nmap scanning and for good performance we should use timing options.
These timing options can be used to speed up or slow down scanning operations, depending on our needs.

When scanning a large number of hosts on a fast network, we may want to increase the number of parallel operations to get faster results. Alternatively, when scanning slow networks (or across the Internet) you may want to slow down a scan to get more accurate results or to evade intrusion detection systems. Below are some timing options for Nmap.

Timing parameter
By default when we scan using Nmap it is scanning in seconds. But we can further increase the performance by setting up timing format. Nmap can be used to with the following timing parameters:

m-minutes

s-seconds

ms-miliseconds

h-hours

Sometimes while choosing timing options, we may be confused about how much time we will set for the scanning. To resolve these issues, Nmap offers a variety of timing options for scanning as below.

Commands: nmap -T[0-5] [target]
There are six templates (numbered 0-5) that can be used to speed up scanning (for faster results) or to slow down scanning (to evade firewalls).

0-paranoid

1-sneaky

2-polite

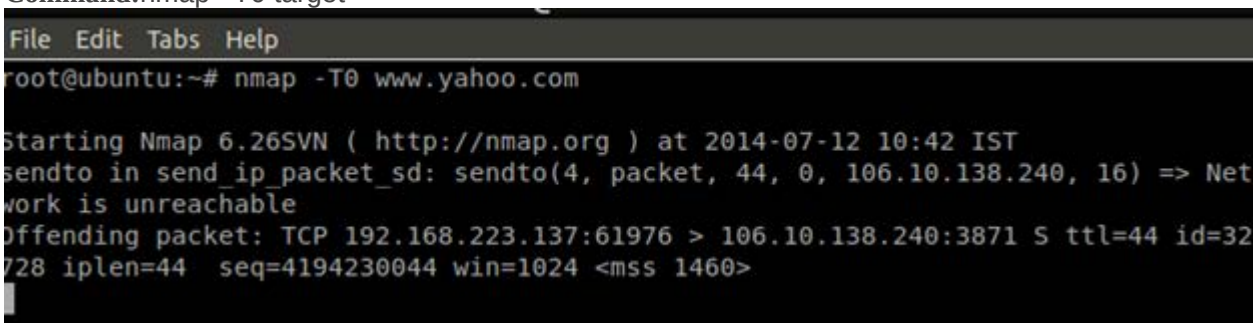
3-normal

4-aggressive

5-insane

With 0 option:
With 0 option we can do a paranoid scan for Nmap, which is a very slow scanning option so that the firewall or IDs are not able to block that request and will decrease the noise for the Nmap probe.

Command:nmap –T0 target



With 1 option:
The sneaky option is used for firewall bypass or IDS evade options.

Nmap –T1 target

While -T0 and -T1 may be useful for avoiding IDS alerts, they will take an extraordinarily long time to scan thousands of machines or ports.

With 2 option:
This is used for the polite option, which is to interfere with the target system.
Polite mode slows down the scan to use less bandwidth and target machine resources.

Nmap –T2 target

```
File  Edit  Tabs  Help
root@ubuntu:~# nmap -T2 www.google.com

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 10:57 IST
Nmap scan report for www.google.com (74.125.236.177)
Host is up (0.00046s latency).
Not shown: 999 filtered ports
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 909.74 seconds
```

With 3 option:
This is a normal scan, as every time Nmap uses this template as a default scan method.

Nmap –T3 target

With 4 and 5 option:
The t4 and t5 option is a very fast and aggressive scan.
Aggressive (4) mode speeds scans up by making the assumption that you are on a reasonably fast and reliable network. Finally, insane mode (5) assumes that you are on an extraordinarily fast network or are willing to sacrifice some accuracy for speed.

Nmap –T4 target

Nmap –T5 target

```
File  Edit  Tabs  Help
root@ubuntu:~# nmap -T4 www.google.com

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 11:02 IST
Nmap scan report for www.google.com (74.125.236.177)
Host is up (0.00070s latency).
All 1000 scanned ports on www.google.com (74.125.236.177) are filtered

Nmap done: 1 IP address (1 host up) scanned in 56.71 seconds
root@ubuntu:~#
```

Parallel option:
As a pentester we should not waste our time by scanning one by one. Instead we can do optimization by scanning many at a time. Nmap does this by dividing the target IP space into groups and then scanning one group at a time. In general, larger groups are more efficient. The downside is that host results can't be provided until the whole group is finished.

There are two options for Nmap to do parallelism, like min and max.

Min:

The –min-parallelism option is used to specify the minimum number of parallel port scan operations.

nmap –min-parallelism [number of operation] [target]

```
root@ubuntu:~# nmap --min-parallelism 100 192.168.223.1

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 12:07 IST
Nmap scan report for 192.168.223.1
Host is up (0.00091s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE
135/tcp  open  msrpc
1027/tcp open  IIS
MAC Address: 00:50:56:C0:00:08 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 3.35 seconds
```

While manually setting the –min-parallelism option may increase scan performance, setting it too high may produce inaccurate results.

Max:
The –max-parallelism operation is used to specify the maximum number of parallel port scan operations.

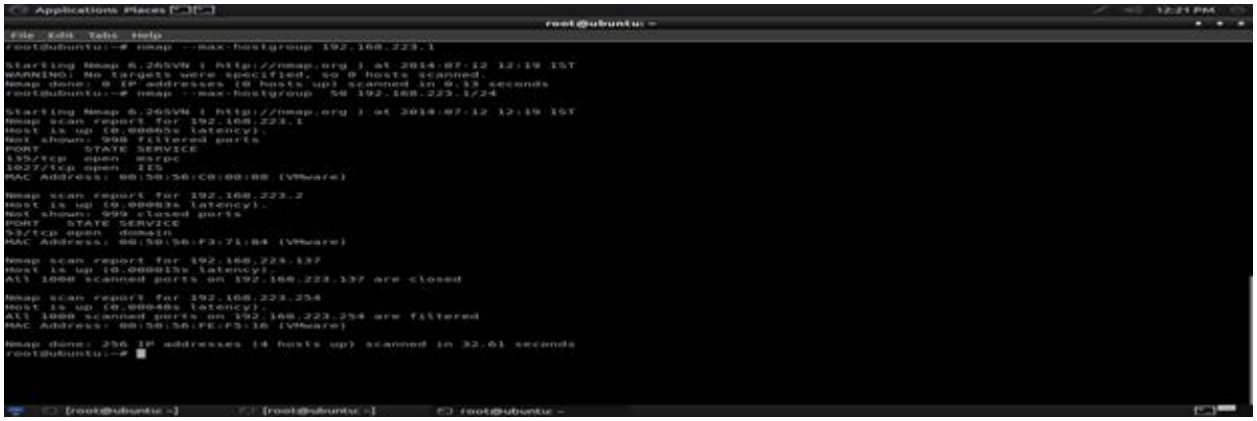
Nmap –max-pallelism [number of operation] [target]

```
File  Edit  Tabs  Help
root@ubuntu:~# nmap --min-parallelism 100 192.168.223.1

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 12:07 IST
Nmap scan report for 192.168.223.1
Host is up (0.00091s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE
135/tcp  open  msrpc
1027/tcp open  IIS
MAC Address: 00:50:56:C0:00:08 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 3.35 seconds
root@ubuntu:~# nmap --max-parallelism 10 192.168.223.1

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 12:13 IST
Nmap scan report for 192.168.223.1
Host is up (0.00075s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE
135/tcp  open  msrpc
1027/tcp open  IIS
MAC Address: 00:50:56:C0:00:08 (VMware)

Nmap done: 1 IP address (1 host up) scanned in 21.46 seconds
```

Host Group size options:
Nmap has the ability to port scan or version scan multiple hosts in parallel. It is used for setting the number of hosts in the targets. It has also two options, max and min.

Max:
The –max-hostgroup option is used to specify the maximum number of targets Nmap should scan in parallel.

nmap –max-hostgroup [number] [targets]



Min:
The –min-hostgroup option is used to specify the minimum number of targets Nmap should scan in parallel.

nmap –min-hostgroup [number] [targets]

```
root@ubuntu:~# nmap --min-hostgroup  50 192.168.223.1/24

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 12:21 IST
Nmap scan report for 192.168.223.1
Host is up (0.00071s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE
135/tcp  open  msrpc
1027/tcp open  IIS
MAC Address: 00:50:56:C0:00:08 (VMware)

Nmap scan report for 192.168.223.2
Host is up (0.00041s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
53/tcp open  domain
MAC Address: 00:50:56:F3:71:84 (VMware)

Nmap scan report for 192.168.223.137
Host is up (0.000026s latency).
All 1000 scanned ports on 192.168.223.137 are closed

Nmap scan report for 192.168.223.254
Host is up (0.00013s latency).
All 1000 scanned ports on 192.168.223.254 are filtered
MAC Address: 00:50:56:FE:F5:16 (VMware)

Nmap done: 256 IP addresses (4 hosts up) scanned in 32.41 seconds
```

Nmap will perform scans in parallel to save time when scanning multiple targets such as a range or entire subnet. By default, Nmap will automatically adjust the size of the host groups based on the type of scan being performed and network conditions. By specifying the –min-hostgroup option, Nmap will attempt to keep the group sizes above the specified number.

The max option is helpful if you want to reduce the load on a network or to avoid triggering any red flags with various network security products.

RTT TIME-OUT

In the TCP connection, RTT or Round Trip Timeout is a measurement for timeout value for the sliding window protocol in the communication, and which depends on the below points.

If the timeout value is too small, the source will time out too fast, resulting in unnecessary retransmissions. On the other hand, if the timeout value is too large, the source will take too long to recover from errors.

For details regarding this follow the link:
http://www.pcvr.nl/tcpip/tcp_time.htm
Nmap maintains a running timeout value for determining how long it will wait for a probe response before giving up or retransmitting the probe. This is calculated based on the response times of previous probes.

During the scan, Nmap's internal processes calculate these response time values:

srtt – Smoothed Round Trip Time – This estimate of response time is based on the observed traffic between the Nmap station and the remote device. This value is provided in microseconds.

rttvar – Round Trip Time Variance – Network communication can be very unpredictable, and Nmap compensates for this uncertainty by creating a range of timeout values. If a response isn't received within this variance, then Nmap concludes that a response isn't likely to ever appear.

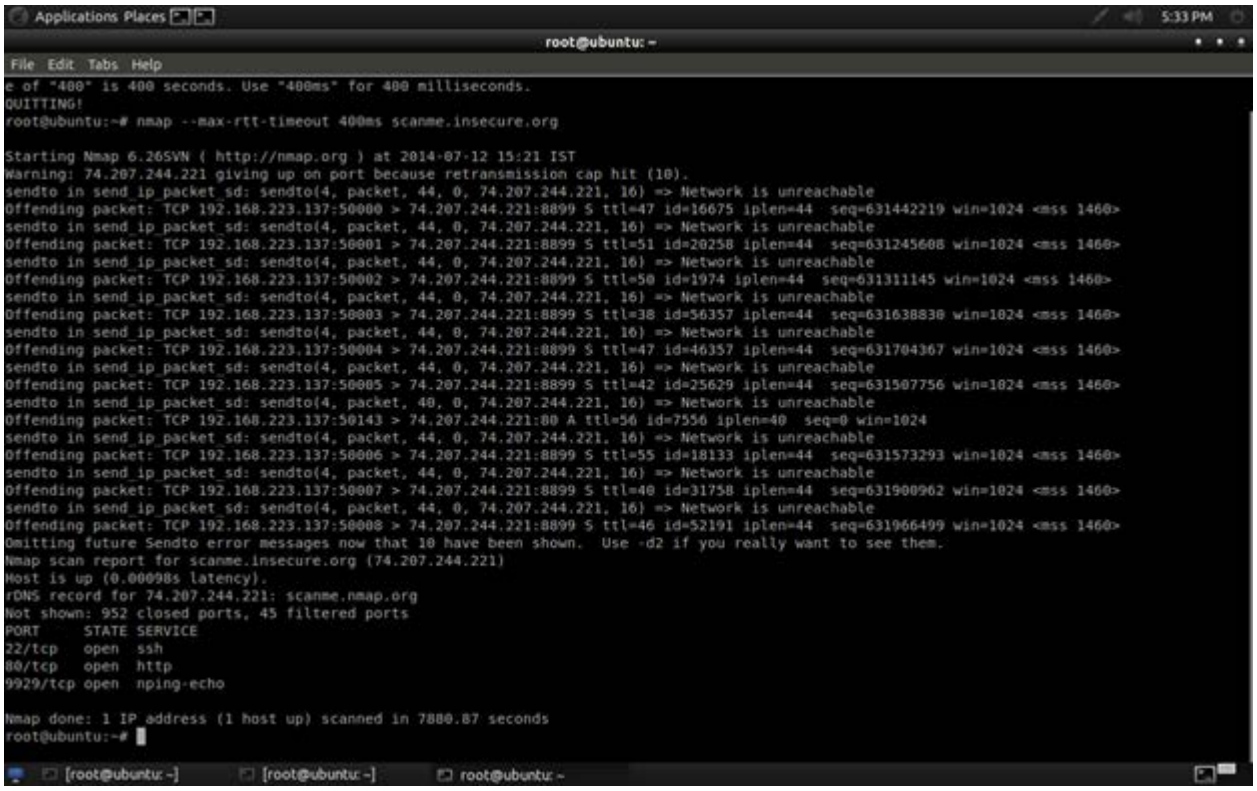
Initial option for Nmap
It controls initial timeout for network response:

nmap –initial-rtt-timeout [time] [target]

Increasing the time value will reduce the number of packet retransmissions due to timeouts. By decreasing the value we can speed up scans, but we have to do so with caution. Setting the RTT timeout value too low can negate any potential performance gains and lead to inaccurate results.

Max options for RTT value
The –max-rtt-timeout option is used to specify the maximum RTT (Round-Trip Time) timeout for a packet response.

nmap –max-rtt-timeout [time] [target]

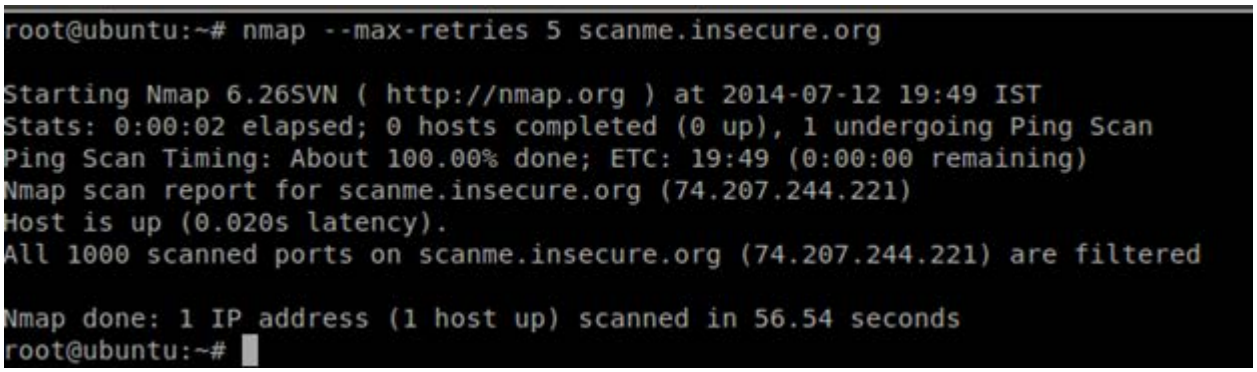


Nmap dynamically adjusts RTT timeout options for best results by default. The default maximum RTT timeout is 10 seconds. Manually adjusting the maximum RTT timeout lower will allow for faster scan times (especially when scanning large blocks of addresses). Specifying a high maximum RTT timeout will prevent Nmap from giving up too soon when scanning over slow/unreliable connections. Typical values are between 100 milliseconds for fast/reliable networks and 10000 milliseconds for slow/unreliable connections.

Max Retries

This option is used to perform a maximum number of probes by Nmap for pentesting.

nmap –max-retries [number] [target]



By default, Nmap will automatically adjust the number of probe retransmissions based on network conditions. The –max-retries option can be used if we want to override the default settings or troubleshoot a connectivity problem. Specifying a high number can increase the time it takes for a

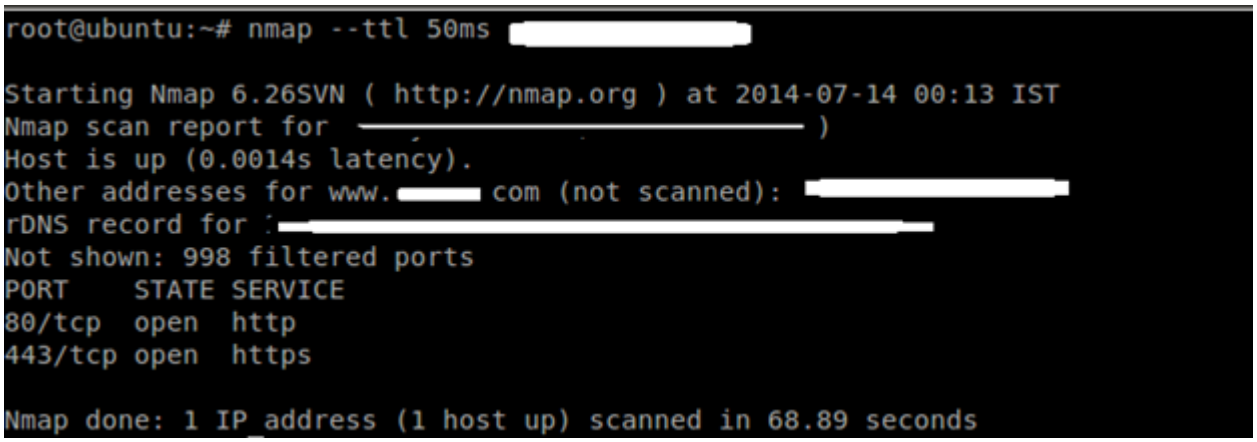
scan to complete, but will produce more accurate results. By lowering the –max-retries we can speed up a scan, although we may not get accurate results if Nmap gives up too quickly.

The TTL option
Every time doing pentesting while doing reconnaissance we came across a TTL value, that is time to live value option. But we should know what a TTL value is.

TTL
Time To Live is a value in an Internet Protocol (IP) packet that tells a network router whether or not the packet has been in the network too long and should be discarded. From the perspective of a pentester, a TTL value can help to determine a lot of information about a target.
Nmap can be used as great measure to find all hosts with regards of TTL value. With the help of TTL value, Nmap will do a more comprehensive and reliable scan against the target.

The –ttl option is used to specify the TTL (time-to-live) for the specified scan (in milli seconds).

nmap –ttl [time] [target]



Packets sent using this option will have the specified TTL value. This option is useful when scanning targets on slow connections where normal packets may time out before receiving a response.

Host timeout option:
The –host-timeout option causes Nmap to give up on slow hosts after the specified time.

nmap –host-timeout [time] [target]

```
root@ubuntu:~# nmap --host-timeout 1m 192.168.223.137

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:08 IST
Nmap scan report for 192.168.223.137
Host is up (0.000019s latency).
All 1000 scanned ports on 192.168.223.137 are closed

Nmap done: 1 IP address (1 host up) scanned in 4.13 seconds
root@ubuntu:~# nmap --host-timeout 1m 192.168.223.137-255

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:09 IST
Nmap scan report for 192.168.223.137
Host is up (0.000017s latency).
All 1000 scanned ports on 192.168.223.137 are closed

Nmap scan report for 192.168.223.254
Host is up (0.00041s latency).
All 1000 scanned ports on 192.168.223.254 are filtered
MAC Address: 00:50:56:FE:F5:16 (VMware)

Nmap done: 119 IP addresses (2 hosts up) scanned in 30.26 seconds
```

A host may take a long time to scan if it is located on a slow or unreliable network. Systems that are protected by rate limiting firewalls may also take a considerable amount of time to scan. The –host-timeout option instructs Nmap to give up on the target system if it fails to complete after the specified time interval. In the above example, the scan takes longer than one minute to complete (as specified by the 1m parameter), which causes Nmap to terminate the scan. This option is particularly useful when scanning multiple systems across a WAN or Internet connection.

Minimum Scan Delay

The –scan-delay option instructs Nmap to pause for the specified time interval between probes.

syntax: nmap –scan-delay [time] [target]

```
root@ubuntu:~# nmap --scan-delay 5s scanme.insecure.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:12 IST
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
```

Some systems employ rate limiting, which can hamper Nmap scanning attempts. Nmap will automatically adjust the scan delay by default on systems where rate limiting is detected. In some cases we need our own scan delay if any rate limiting or IDS are in the actions.

Maximum Scan Delay

The –max-scan-delay is used to specify the maximum amount of time Nmap should wait between probes.

syntax: nmap –max-scan-delay [time] [target]

```
root@ubuntu:~# nmap --max-scan-delay 300 scanme.insecure.org
Since April 2010, the default unit for --max-scan-delay is seconds, so your time
 of "300" is 5.0 minutes. If this is what you want, use "300s".
QUITTING!
root@ubuntu:~# nmap --max-scan-delay 300s scanme.insecure.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:16 IST
Nmap scan report for scanme.insecure.org (74.207.244.221)
Host is up (0.035s latency).
rDNS record for 74.207.244.221: scanme.nmap.org
All 1000 scanned ports on scanme.insecure.org (74.207.244.221) are filtered

Nmap done: 1 IP address (1 host up) scanned in 58.53 seconds
```

The –max-scan-delay option can be used to provide an upper limit to the amount of time between probes. This can speed up a scan, but comes at the expense of accurate results and added network stress.

Minimum Packet Rate
The –min-rate option is used to specify the minimum number of packets Nmap should send per second.

syntax: nmap –min-rate [number] [target]

```
root@ubuntu:~# nmap --min-rate 30 scanme.insecure.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:18 IST
Nmap scan report for scanme.insecure.org (74.207.244.221)
Host is up (0.00089s latency).
rDNS record for 74.207.244.221: scanme.nmap.org
All 1000 scanned ports on scanme.insecure.org (74.207.244.221) are filtered

Nmap done: 1 IP address (1 host up) scanned in 9.27 seconds
root@ubuntu:~#
```

Maximum Packet Rate
The –max-rate option is used to specify the maximum number of packets Nmap

should send per second.

syntax: nmap –max-rate [number] [target]

```
root@ubuntu:~# nmap --max-rate 30 scanme.insecure.org
]
Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:19 IST
Stats: 0:00:02 elapsed; 0 hosts completed (0 up), 1 undergoing Ping Scan
Ping Scan Timing: About 12.50% done; ETC: 22:19 (0:00:00 remaining)
Nmap scan report for scanme.insecure.org (74.207.244.221)
Host is up (0.00056s latency).
rDNS record for 74.207.244.221: scanme.nmap.org
All 1000 scanned ports on scanme.insecure.org (74.207.244.221) are filtered

Nmap done: 1 IP address (1 host up) scanned in 102.39 seconds
root@ubuntu:~#
```

In the example above, specifying –max-rate 30 instructs Nmap to send no more than 30 packets per second. This can dramatically slow down a scan but can be helpful when attempting to avoid intrusion detection systems or a target that uses rate limiting.

Defeat Reset Rate Limits
The –defeat-rst-ratelimit is used to defeat targets that apply rate limiting to RST (reset) packets.

syntax: nmap –defeat-rst-ratelimit [target]

```
root@ubuntu:~# nmap --defeat-rst-ratelimit scanme.insecure.org

Starting Nmap 6.26SVN ( http://nmap.org ) at 2014-07-12 22:21 IST
Nmap scan report for scanme.insecure.org (74.207.244.221)
Host is up (0.035s latency).
rDNS record for 74.207.244.221: scanme.nmap.org
All 1000 scanned ports on scanme.insecure.org (74.207.244.221) are filtered

Nmap done: 1 IP address (1 host up) scanned in 57.79 seconds
```

The –defeat-rst-ratelimit option can be useful if you want to speed up scans on targets that implement RST packet rate limits. It can, however, lead to inaccurate results, and as such, it is rarely used.

This is the end of the document. I will cover **“Evading Firewall, Pentesting with Nmap, Web Service Auditing, Web Application Pentesting, Nmap Script Engine development”** in the upcoming installment.
References
http://www.professormesser.com/nmap/hacking-nmap-using-nmap-to-calculate-network-response-time/5/
http://nmap.org/book/man-performance.html

Nmap Cheat Sheet: Part 4

This is the fourth part of our Nmap Cheat Sheet. Here we will discuss more about firewall scanning, IDS/IPS Evasion, web server pen testing, etc. Before that, we should know some basics about firewall so that it will easy to bypass it.

What is a Firewall?
A **firewall** is nothing but a software or hardware used to access or forbid unauthorized access to or from a network. As a pen tester, a security researcher is always trying to find the firewall installed on the infrastructure, so that he/she can try to bypass the firewall. There are lots of public exploits and o-a day vulnerabilities available on the Internet which helps for well-known exploitation.
Basically there are two category of firewall:

- Host based firewall
- Network based firewall

Host Based Firewall:
These are software running on a single host (read, computer system), which are used to control inbound traffic (traffic from the network toward the host) and outbound traffic (from the host toward the network). These are installed on the operating systems of individual computers. Examples include IPTables and Firestarter for Linux, and Zone Alarm and Tiny Personal Firewall for Windows.

Network Based Firewall:
These are either hardware devices, software, or combinations of hardware and software, which are used to control inbound traffic from the external, unprotected network.

Firewalls are installed in between the protected and unprotected network. They watch all traffic going to and fro, and are configured by setting rules to allow only the required inbound and outbound traffic.

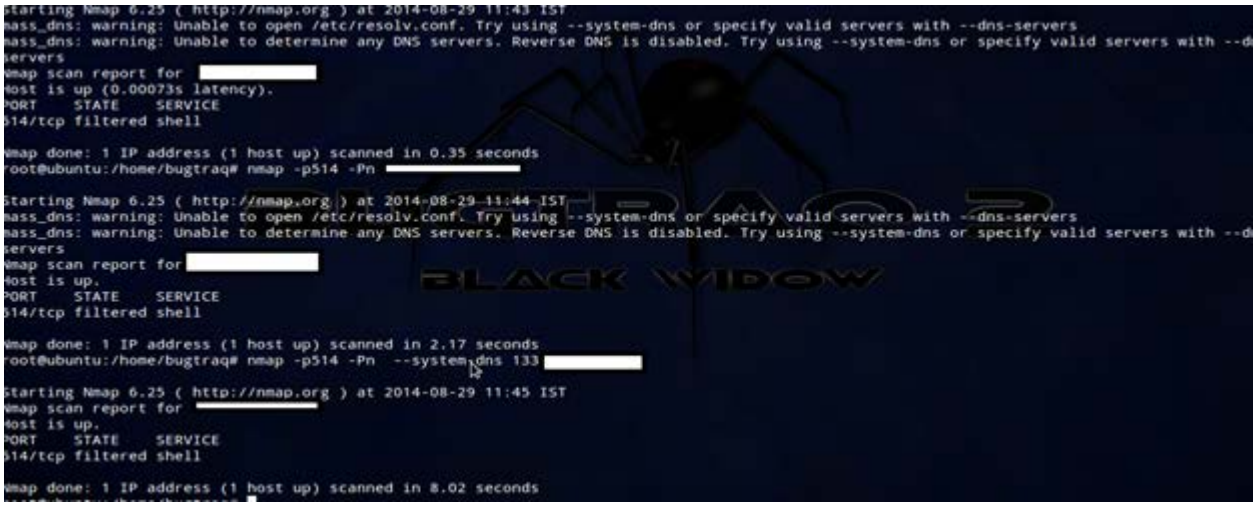
Scanning firewall:
To effectively scan a firewall we must check all open ports, services, and states. While scanning for Nmap also behavior should be taken, so timing options should be seen to determine the firewall presence. So you can see below details of Nmap results. From Nmap scan results we can easily know that there is a firewall.

Port status	Port type	Details
Blocked	Closed port	Most of the firewall ports should be in a closed state
Filtered	Filtered port	A few ports may be filtered to restrict access of the running services to a few IP addresses
Allowed	Open port	Very few ports should be in an open state. Whenever you find them, do not forget to probe further and close non-required ports

By advance Googling I came to know that the following IP address is protected by WAF (web application firewall) as well as some kind of IDS. We tried with some kind of brute force attack, SQL injection. Every time we put some special character, it wass showing “Firewall authentication failed”. We came to know that this thing can be bypassed with HTTP verb tampering. We will discuss that later.

So firewall is present. Let's do Nmap for more reefing results. So first do an Nmap scan with –Pn.

Now we confirmed that remote shell is running in the remote server which is filtered. So we are sure there is a firewall behind the scene. Let's do the scan with specified port to get details.



Let’s do an internal network scan. First we will check version scan:

We came to know that there are lots of services running in the network with port specification and timing options.

Observation:

The scanned firewall runs various services for the inside network, including DNS, SSH, HTTPS and Web proxy. These are accessible to all PCs on the internal network. It also runs a transparent proxy on port 80, so that client browser settings are not required to be changed.

Evade or Evasion or Bypass of a Firewall:
Well a bypass or evasion or evade is nothing but another way to get into the system. To block malicious attack or spam, admin uses firewall or IDS/IPS. But from an attacker's point of view, he will find a way to bypass the rule for firewall; there are lots of way to bypass the firewall for an Nmap scan. We will discuss everything below. But one thing is sure shot "**skill and common sense**" that is used by an attacker.

1. **Fragmentation:**

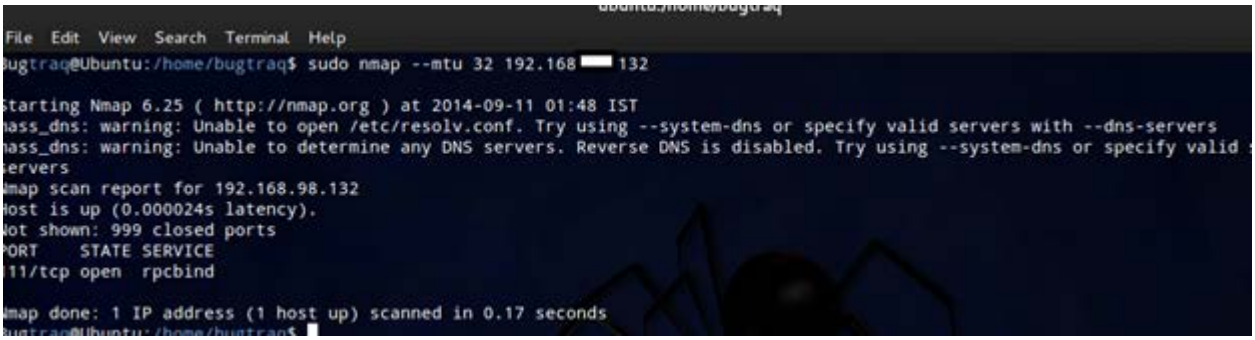
In general, the word fragmentation means dividing large objects into small parts. Similarly, Nmap uses 8 bytes of packet for bypassing firewall/ids/ips. Nmap will split into small small packets for bypassing firewall. This technique is very old, still it will work if there is a misconfiguration of firewall.

Nmap –f host

MTU:

MAXIMUM Transmission Unit. It is an alias of fragmentation, but here we can specify the scan for a custom amount of packets.

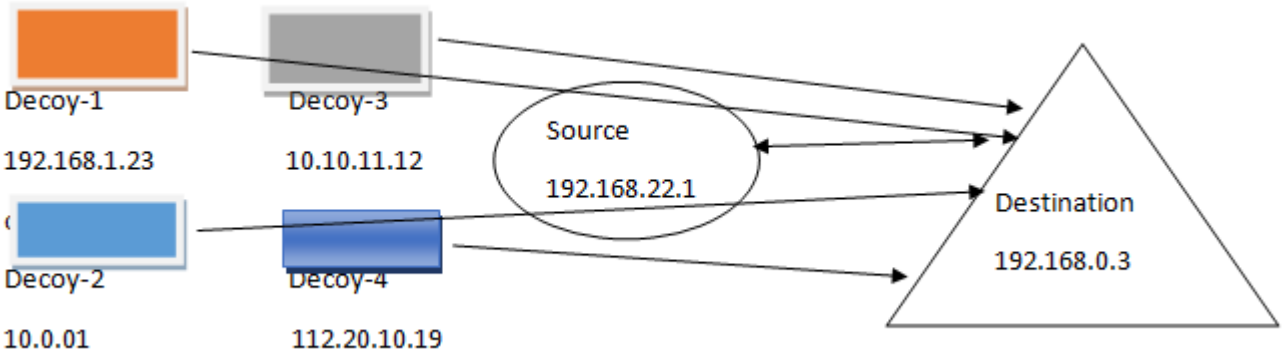
Nmap –mtu 16 host



The above Nmap scan instructs to use 16 bytes of packets instead of 8 bytes in the previous scan. So we can specify the custom packets which should be multiple of 8.

2. Decoy

This type of scan is very stealthy and undetectable. In this scan, targets are scanned by multiple fake or spoofed IP addresses. This way the firewall can be bypassed and the firewall will think that the attack or scan is done by multiple resources or ip addresses.

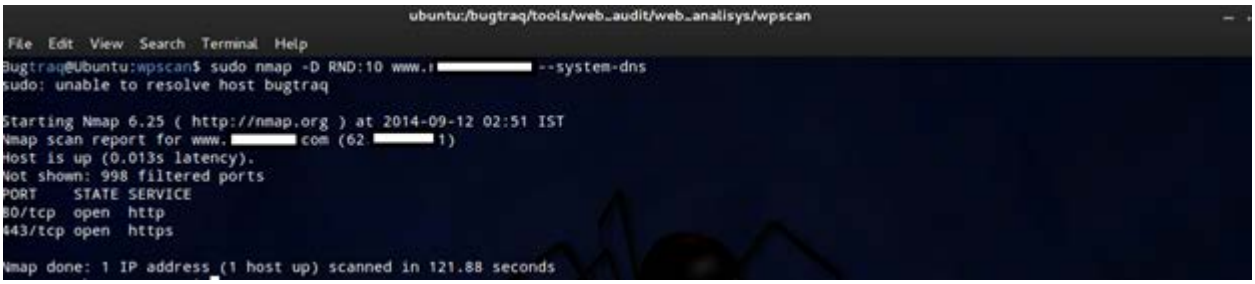


Decoys are used both in the initial ping scan (using ICMP, SYN, ACK, or whatever) and during the actual port scanning phase. Decoys are also used during remote OS detection (-O). Decoys do not work with version detection or TCP connect scan.

This effectively makes it appear that the target is being scanned by multiple systems simultaneously. Using decoys allows the actual source of the scan to “blend into the crowd”, which makes it harder to trace where the scan is coming from.

There are two ways to perform decoy scan:

1.nmap –D RND:10 TARGET

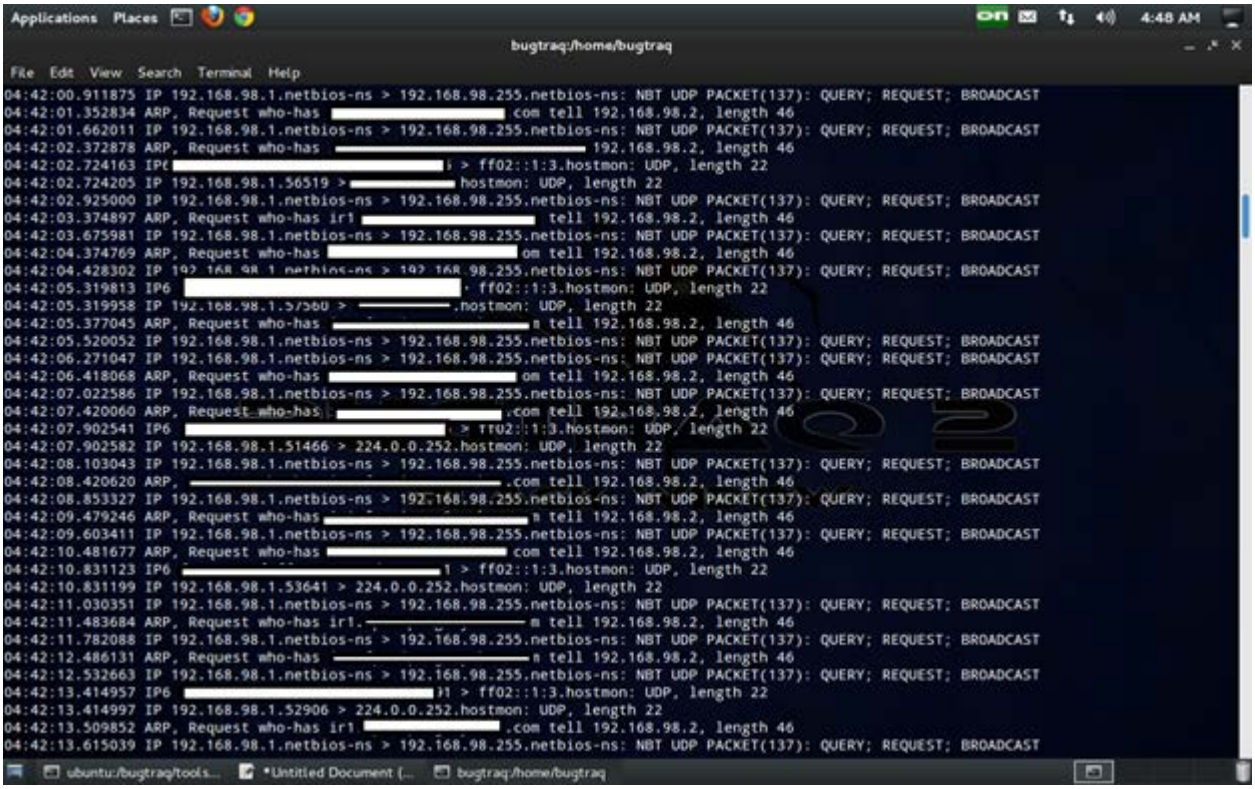


Here Nmap will generate random 10 IPs and it will scan the target using 10 IP and source.

2.nmap –D decoy1,decoy2,decoy3 target

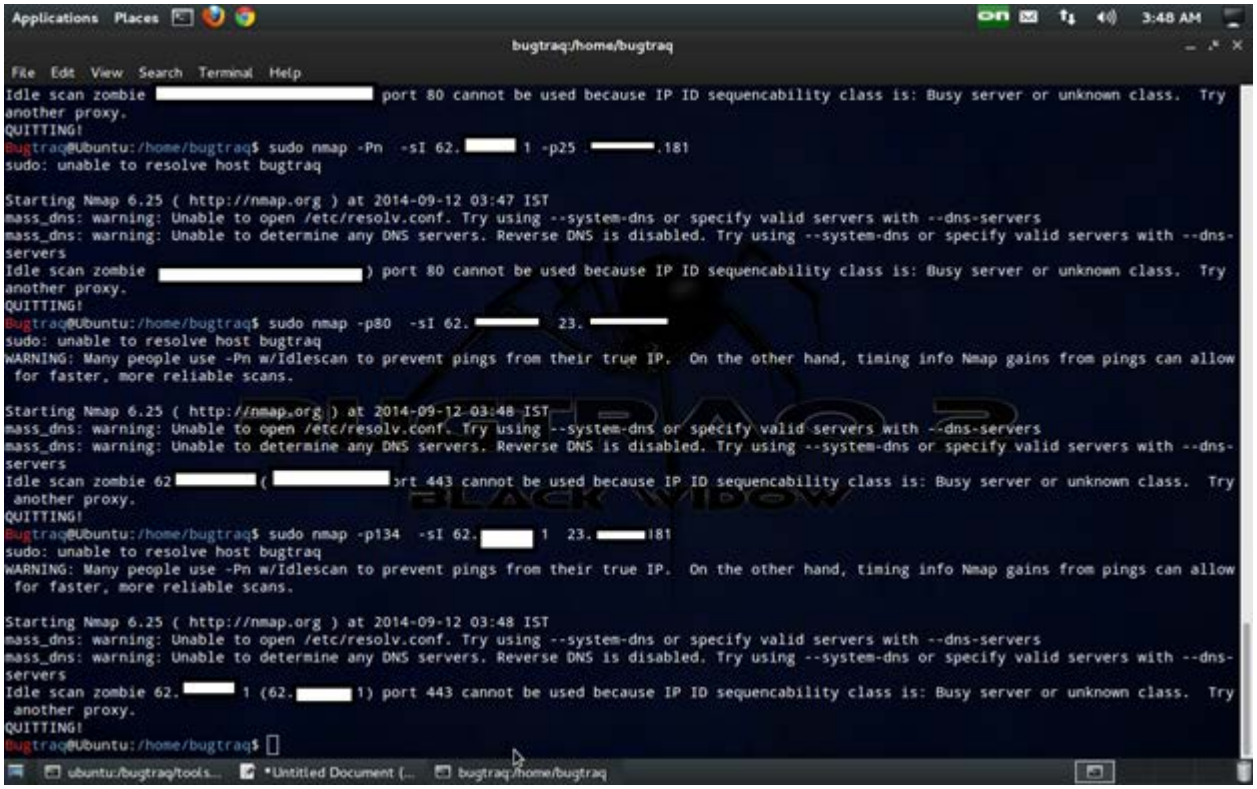
Here decoys are specified by the attacker.

The below network capture show multiple decoys which will fool the firewall.



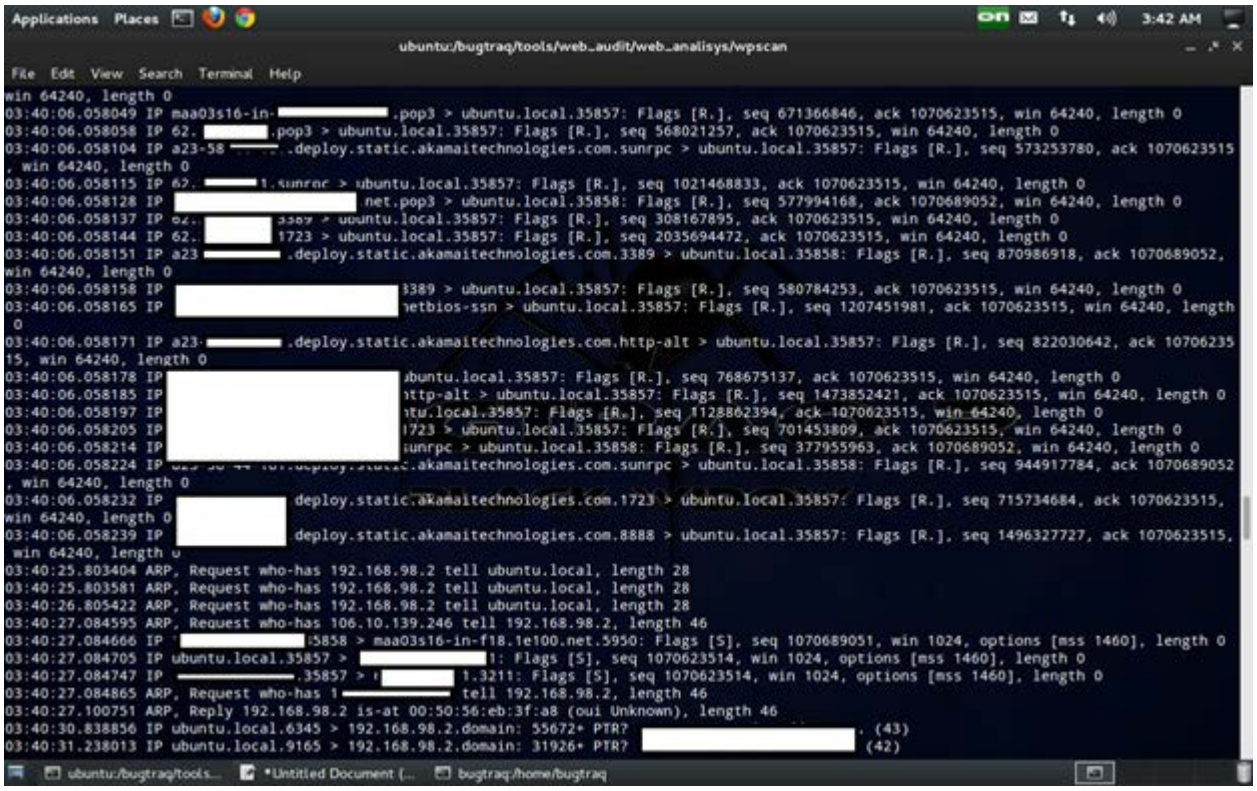
Idle Zombie Scan:
Using this technique, the attacker will first exploit an idle system and use it to scan the target system.
The scan works by exploiting the predictable IP sequence ID generation employed by some systems. In order for an idle scan to be successful, the zombie system must truly be idle at the time of scanning. **For any doubt please refer to the previous installments.**
In this technique the IP of the attacker will be hidden.

Nmap –P0 -sI zombie target

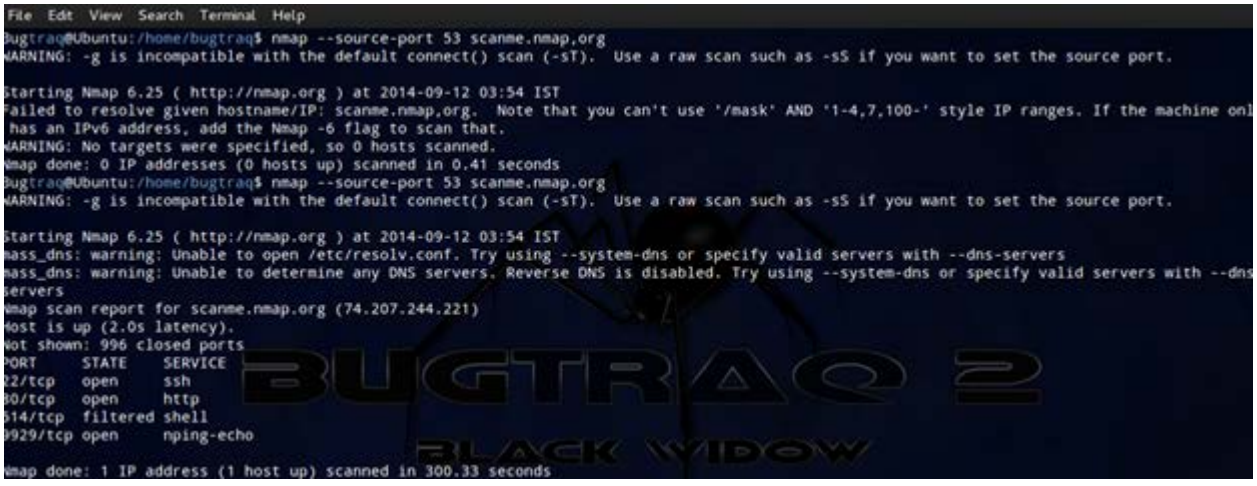


For confirmation we will use tcpdump to dump all network traffic and also to be able to capture the traffic.

Tcpdump –I interface

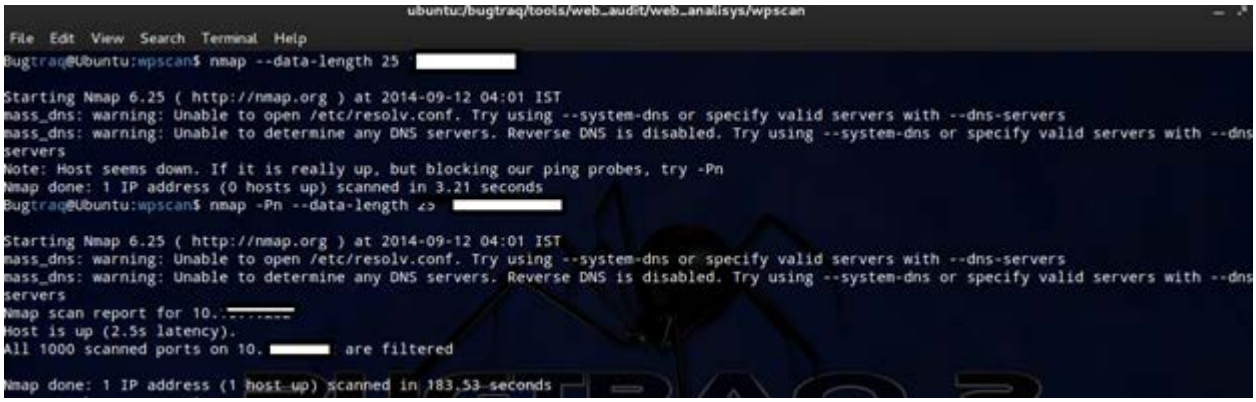


–source-port option:
Every TCP segment contains a source port number in addition to a destination. By default, Nmap will randomly pick an available outgoing source port to probe a target. The **–source-port** option will force Nmap to use the specified port as the source for all packets. This technique can be used to exploit weaknesses in firewalls that are improperly configured to blindly accept incoming traffic based on a specific port number. Port 20 (FTP), port 53 (DNS), and 67 (DHCP) are common ports susceptible to this type of scan.
Nmap –source-port port target

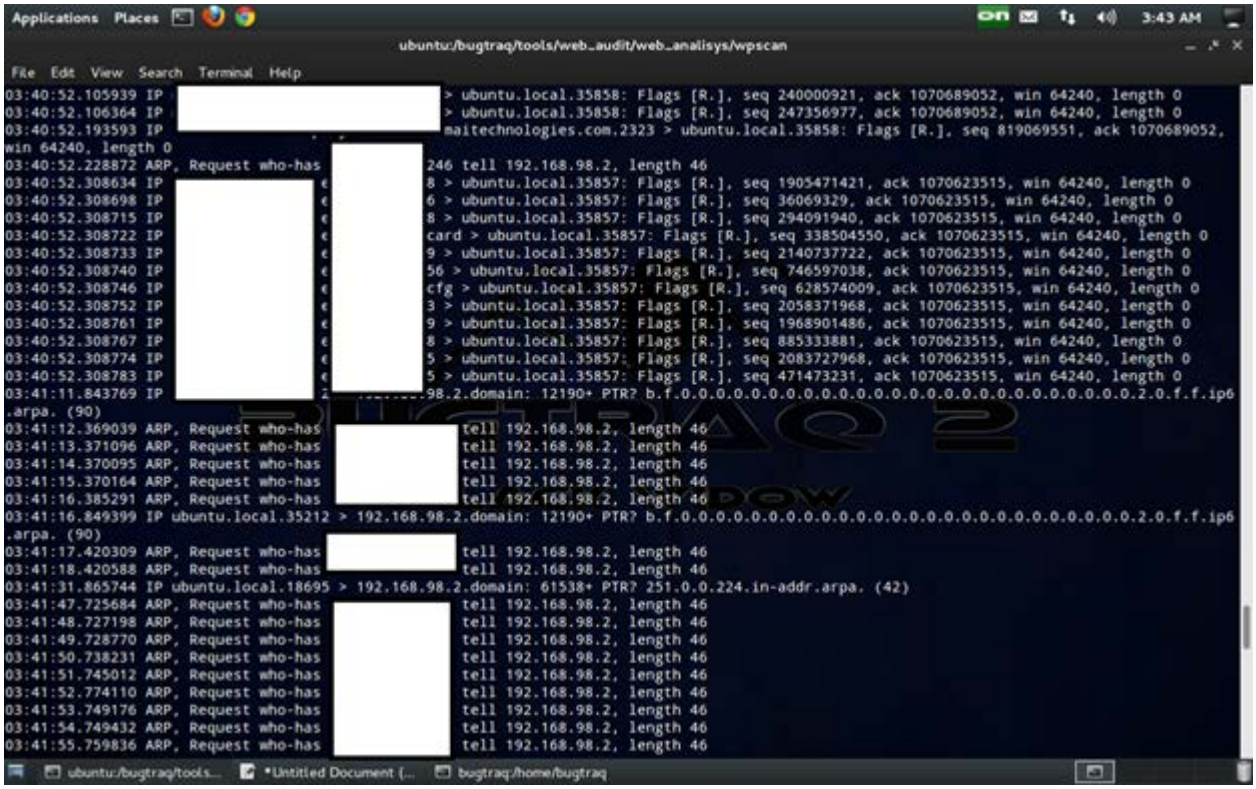


Random Data length:
Appending random data length, we can also bypass firewall. Many firewalls are inspecting packets by looking at their size in order to identify a potential port scan. This is because many scanners are

sending packets that have specific size. In order to avoid that kind of detection we can use the command –**data-length** to add additional data and to send packets with a different size than the default. In the image below, we have changed the packet size by adding 25 more bytes.
nmap –data-length target



Capture traffic as below:



Randomize Target Scan Order:
The –randomize-hosts option is used to randomize the scanning order of the specified targets. The –randomize-hosts option helps prevent scans of multiple targets from being detected by firewalls and intrusion detection systems. This is done by scanning them in a random order instead of sequential.

nmap –randomize-hosts targets

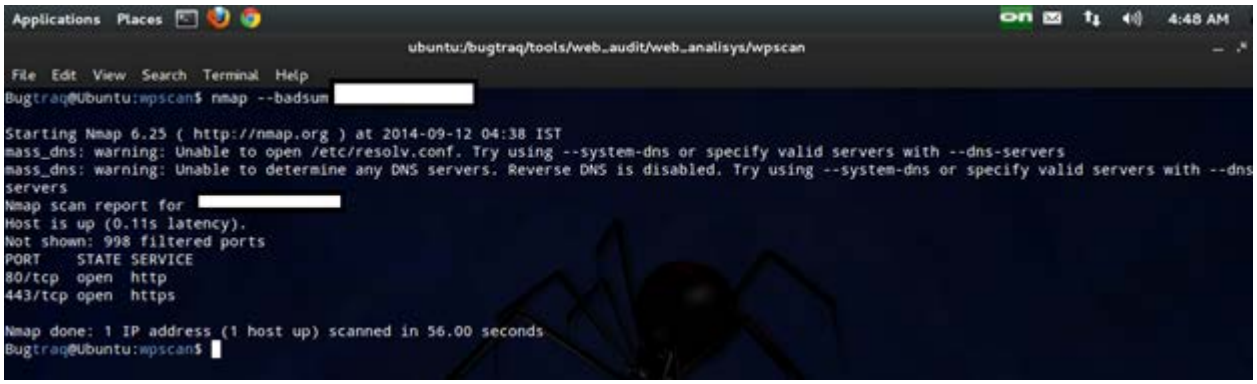
Spoof MAC address:

MAC (Media Access Control) is nothing but the unique physical address for a machine. So this is also another method for bypassing the firewall. In some firewalls, the MAC address rule is applied, so it is very useful for this rule. You will need to discover which MAC address you need to set in order to obtain results. This can be easily done either manually or by advanced fuzzing. I like fuzzing, which can be done in Python very easily. Everything we will do in manual can be imported into Python as regex and can start to automate.
Specifically the **–spoof-mac** option gives you the ability to choose a MAC address from a specific vendor, to choose a random MAC address, or to set a specific MAC address of your choice. Another advantage of MAC address spoofing is that you make your scan stealthier, because your real MAC address it will not appear on the firewall log files.
nmap -sT -PN –spoof-mac aa:bb:cc:dd:ee:ff target

The spoof mac address takes the following arguments:

Argument	Function
0 (zero)	Generates a random MAC address
Specific MAC Address	Uses the specified MAC address
Vendor Name	Generates a MAC address from the specified vendor (such as Apple, Dell, 3Com, etc)

Send Bad Checksum:
Checksum is nothing but the integrity check. In some firewalls or IDS/IPS, packets are checked by the checksum of packets. So the attacker will fool the IDS/IPS by sending bad checksums.



Sun-RPC Grind Scan:

What is Sun RPC?

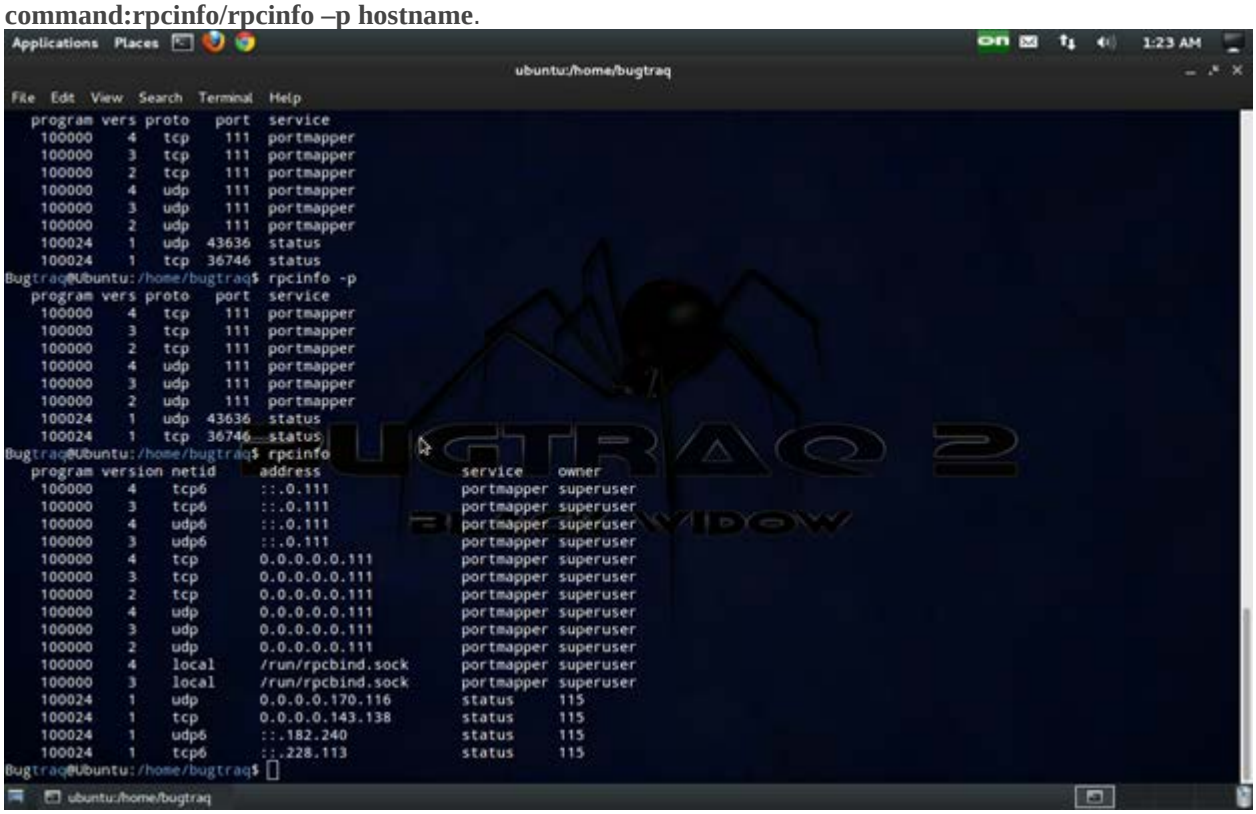
Sun RPC (remote procedure Call) is a Unix protocol used to implement many services including NFS. Originally developed by Sun, but now widely available on other platforms (including Digital Unix). Also known as Open Network Computing (ONC).

Sun RPC package has an RPC compiler (rpcgen) that automatically generates the client and server stubs.

Nmap comes with an nmap-rpc database of almost 600 RPC programs. Many RPC services use high-numbered ports and/or the UDP transport protocol, making them available through many poorly configured firewalls. RPC programs (and the infrastructure libraries themselves) also have a long history of serious remotely exploitable security holes. So network administrators and security auditors often wish to learn more about any RPC programs on their networks.

Basically we can get details about RPC by:

command:rpcinfo/rpcinfo –p hostname.



the rpc server must be mapped to the host which can be confirmed by port mapper.
if the port mapper (rpcbind) service (UDP or TCP port 111) is available
This shows that hosts frequently offer many RPC services, which increases the probability that one is exploitable. We also noticed that most of the services are on strange high-numbered ports (which may change for any number of reasons) and split between UDP and TCP transport protocols.

Nmap can determine all of the information by directly communicating with open RPC ports through the following three-step process.

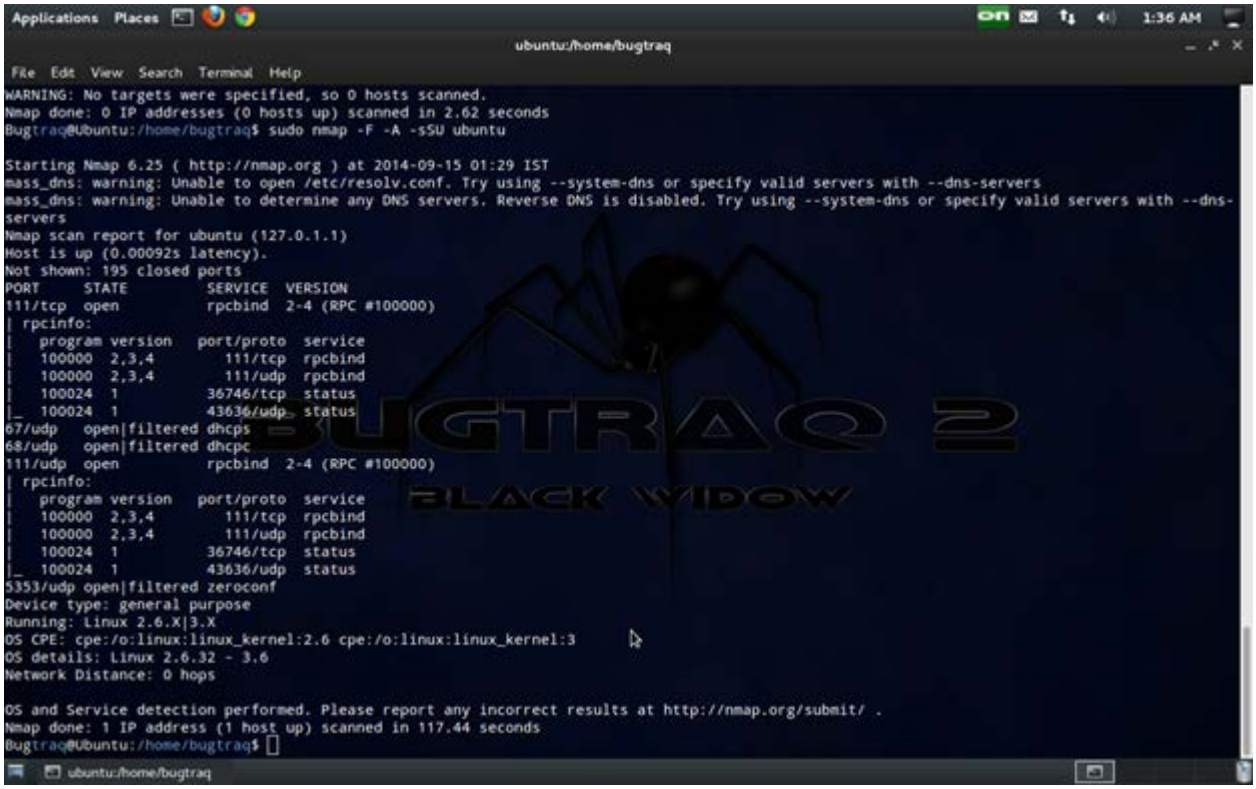
1. The TCP and/or UDP port scan finds all of the open ports.

2. Version detection determines which of the open ports use the SunRPC protocol.
3. The RPC brute force engine determines the program identity of each RPC port by trying a null command against each of the 600 programs numbers in nmap-rpc. Most of the time Nmap guesses wrong and receives an error message stating that the requested program number is not listening on the port. Nmap continues trying each number in its list until success is returned for one of them. Nmap gives up in the unlikely event that it exhausts all of its known program numbers or if the port sends malformed responses that suggest it is not really RPC.

The above scan has results for RPC services, but unfortunately we did not get any SUN RPC, because we have an Ubuntu machine. In a real pen testing environment, we can gather information about SUN if we have a SUN server.

SSL Post-processor Scan

Nmap has the capacity to detect the SSL encryption protocol and then launch an encrypted session through which it executes normal version detection. As with the RPC grinder discussed previously, the SSL post-processor/scan is automatically executed whenever an appropriate (SSL) port is detected.
Command: nmap -Pn -sSV -T4 –F target



nmap-service-probes File Format:
As with remote OS detection (-O), Nmap uses a flat file to store the version detection probes and match strings. While the version of nmap-services distributed with Nmap is sufficient for most users, understanding the file format allows advanced pen testers to add their own services to the detection engine. Like many Unix files, nmap-service-probes is line-oriented. Lines starting with a hash (#) are treated as comments and ignored by the parser. Blank lines are ignored as well.

Exclude Directive

This directive excludes the specified ports from the version scan. It can only be used once and should be near the top of the file, above any Probe directives. Port should be separated by a comma.

Syntax: Exclude <port specification>
Probe Directive

Syntax: Probe <protocol> <probename> <probestring>
Examples:

```
Probe TCP GetRequest q|GET / HTTP/1.0\r\n\r\n|
Probe UDP DNSStatusRequest q|\0\0\x10\0\0\0\0\0\0\0\0\0|
Probe TCP NULL q||
```

The Probe directive tells Nmap what string to send to recognize various services. The arguments are as follows:

<protocol>

> This must be either TCP or UDP. Nmap only uses probes that match the protocol of the service it is trying to scan.

<probename>

> This is a plain English name for the probe. It is used in service fingerprints to describe which probes elicited responses.

<probestring>

> Tells Nmap what to send. It must start with a q, then a delimiter character which begins and ends the string. Between the delimiter characters is the string that is actually sent. It is formatted similarly to a C or Perl string in that it allows the following standard escape characters: \\ \0, \a, \b, \f, \n, \r, \t, \v, and \xHH (whereH is any hexadecimal digit). One Probe line in nmap-service-probes has an empty probe string, as shown in the third example above. This is the TCP NULL probe which just listens for the initial banners that many services send. If your delimiter character (| in these examples) is needed for your probe string, you need to choose a different delimiter.

match Directive

Syntax: match <service> <pattern> [<versioninfo>]

Examples:

```
match ftp m/^220.*Welcome to .*Pure-?FTPd (\d\S+\s*)/ p/Pure-FTPd/ v/$1/ cpe:/a:pureftpd:pure-ftpd:$1/
match ssh m/^SSH-([\d.]+)-OpenSSH[_-]([\w.]+)\r?\n/i p/OpenSSH/ v/$2/ i/protocol $1/ cpe:/a:openbsd:openssh:$2/
match mysql m|^\x10\0\0\x01\xff\x13\x04Bad handshake$| p/MySQL/ cpe:/a:mysql:mysql/
match chargen m|@ABCDEFGHIJKLMNOPQRSTUVWXYZ|
match uucp m|^login: login: login: $| p/NetBSD uucpd/ o/NetBSD/ cpe:/o:netbsd:netbsd/a
match printer m|^([\w-_.]+): lpd: Illegal service request\n$| p/lpd/ h/$1/
match afs m|^[\d\D]{28}\s*(OpenAFS)([\d\.]{3}[^\s\0]*)\0| p/$1/ v/$2/
```

The match directive tells Nmap how to recognize services based on responses to the string sent by the previous Probe directive. A single Probe line may be followed by dozens or hundreds of match statements. If the given pattern matches, an optional version specifier builds the application name,

version number, and additional info for Nmap to report. The arguments to this directive follow:
<service>

This is simply the service name that the pattern matches. Examples would be ssh, smtp, http, or snmp.

<pattern>

This pattern is used to determine whether the response received matches the service given in the previous parameter. The format is like Perl, with the syntax being m/[regex]/[opts]. The "m" tells Nmap that a match string is beginning. The forward slash (/) is a delimiter, which can be substituted by almost any printable character as long as the second slash is also replaced to match. The regex is a Perl-style regular expression. This is made possible by the excellent Perl Compatible Regular Expressions (PCRE) library (http://www.pcre.org). The only options currently supported are 'i', which makes a match case-insensitive and 's' which includes newlines in the '.' specifier. As you might expect, these two options have the same semantics as in Perl. Subexpressions to be captured (such as version numbers) are surrounded by parentheses as shown in most of the examples above.

<versioninfo>

The <versioninfo> section actually contains several optional fields. Each field begins with an identifying letter (such as h for "hostname"). Next comes a delimiter character which the signature writer chooses. The preferred delimiter is slash ('/') unless that is used in the field itself. Next comes the field value, followed by the delimiter character. The following table describes the six fields:

Field format	Value description
p/vendorproductname/	Includes the vendor and often service name and is of the form "Sun Solaris rexecd", "ISC BIND named", or "Apache httpd".
v/version/	The application version "number", which may include non-numeric characters and even multiple words.
i/info/	Miscellaneous further information which was immediately available and might be useful. Examples include whether an X server is open to unauthenticated connections, or the protocol number of SSH servers.
h/hostname/	The hostname (if any) offered up by a service. This is common for protocols such as SMTP and POP3 and is useful because these hostnames may be for internal networks or otherwise differ from the straightforward reverse DNS responses.
o/operatingsystem/	The operating system the service is running on. This may legitimately be different than the OS reported by Nmap IP stack based OS detection. For example, the target IP might be a Linux box which uses network address translation to forward requests to an Microsoft IIS server in the DMZ. In this case, stack OS detection should report the OS as Linux, while service detection reports port 80 as being Windows.
d/devicetype/	The type of device the service is running on, a string like "print server" or "webcam". Some services disclose this information, and it can be inferred in

	many more cases. For example, the HP-ChaiServer web server only runs on printers. For a full list of device types, see the section called “Device Types”.
cpe:/cpename/[a]	A CPE name for some aspect of the service. This may be used multiple times; it’s conceivable to be able to identify not only the service (cpe:/a names) but also the operating system (cpe:/o names) and hardware platform (cpe:/h names) as well. The trailing slash is not part of CPE syntax but is included to match the format of other fields. See the section called “Common Platform Enumeration (CPE)” for more information about CPE.

Softmatch Directive

Syntax: softmatch <service> <pattern>
Examples:

```
softmatch ftp m/^220 [-.\w ]+ftp.*\r\n$/i
softmatch smtp m|^220 [-.\w ]+SMTP.*\r\n|
softmatch pop3 m|^\+OK [-\[\]\(\)!,/+:<>@.\w ]+\r\n$|
```

The softmatch directive is similar in format to the match directive discussed above. The main difference is that scanning continues after a softmatch, but it is limited to probes that are known to match the given service. This allows for a normal (“hard”) match to be found later, which may provide useful version information.

ports and sslports Directives

Syntax: ports <portlist>
Examples:

```
ports 21,43,110,113,199,505,540,1248,5432,30444
ports 111,4045,32750-32810,38978
```

This line tells Nmap what ports the services identified by this probe are commonly found on. It should only be used once within each Probe section. The syntax is a slightly simplified version of that taken by the Nmap -p option. Syntax: sslports <portlist>

Example:

```
sslports 443
```

This is the same as ‘ports’ directive described above, except that these ports are often used to wrap a service in SSL. For example, the HTTP probe declares “sslports 443” and SMTP-detecting probes have an “sslports 465” line because those are the standard ports for HTTPS and SMTPS respectively. The <portlist> format is the same as with ports. This optional directive cannot appear more than once per Probe.

totalwaitms Directive

Syntax: totalwaitms <milliseconds>
Example:

```
totalwaitms 5000
```

This rarely necessary directive specifies the amount of time Nmap should wait before giving up on the most recently defined Probe against a particular service. The Nmap default is usually fine.

Rarity Directive

Syntax: rarity <value between 1 and 9>
Example:

```
rarity 6
```

The rarity directive roughly corresponds to how infrequently this probe can be expected to return useful results. The higher the number, the rarer the probe is considered and the less likely it is to be tried against a service.

Fallback Directive

Syntax: fallback <Comma separated list of probes>
Example:

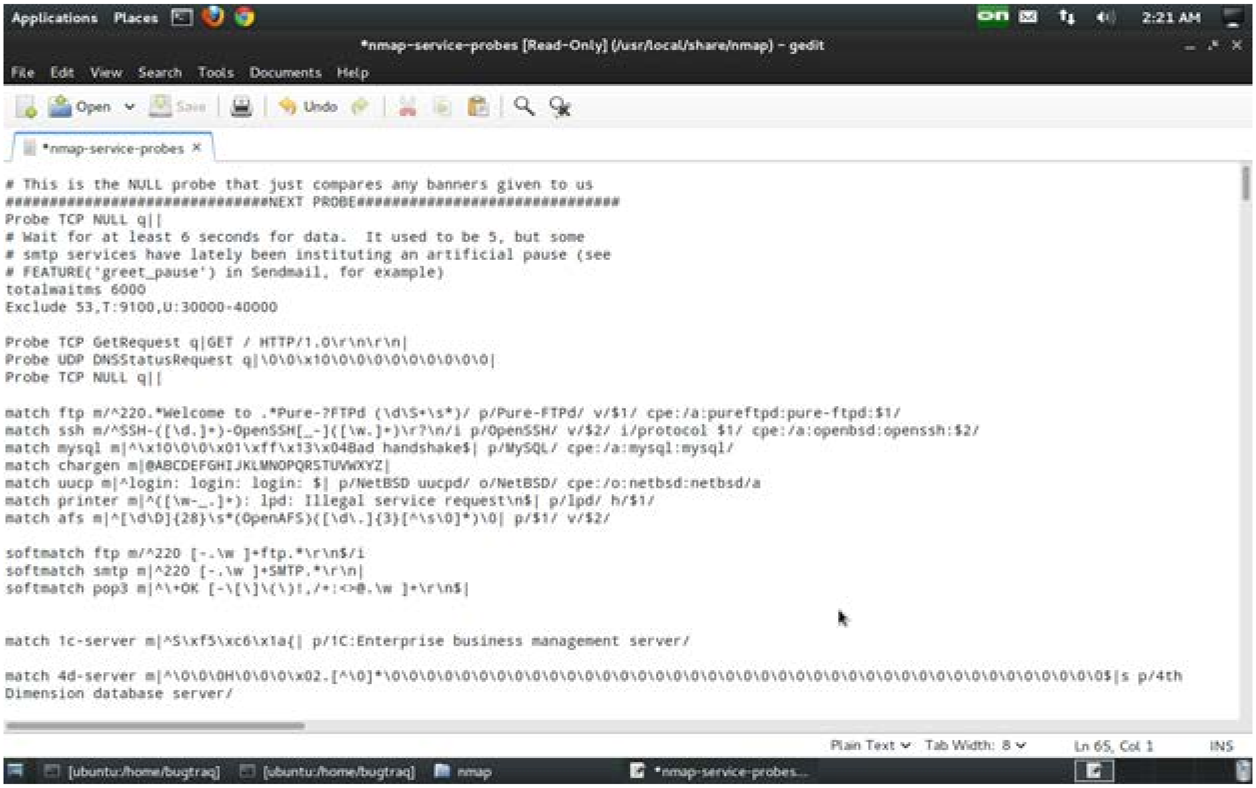
```
fallback GetRequest,GenericLines
```

This optional directive specifies which probes should be used as fallbacks if there are no matches in the current Probe section. For TCP probes without a fallback directive, Nmap first tries match lines in the probe itself and then does an implicit fallback to the NULL probe. If the fallback directive is present, Nmap first tries to match lines from the probe itself, then those from the probes specified in the fallback directive (from left to right). Finally, Nmap will try the NULL probe. For UDP, the behavior is identical except that the NULL probe is never tried**.**

Practice for above:
Ok we have studied enough theory, it is time for practice, because theory is boring unless and until practice is done. So let us do some pen testing. Following are the steps a pen tester can follow for real life hacking or security assessments.

1. Locate the Nmap file probes: command:locate nmap-service-probes

2. Edit the file:



Here we are editing via gedit and we will put all rules as we need for our pen testing. Follow the below picture. Save the file and go for pen testing.

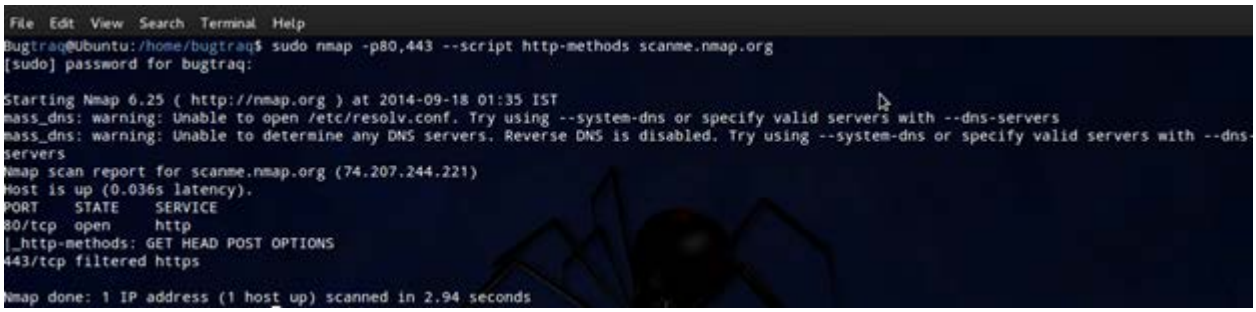
Now it is the best time to start our ninja skill for pen testing using Nmap. Obviously using Nmap we can do lot of things. Here we will start with basic web app pen testing.

First we will start with web-server pen-testing. Basically a web server is a space or a server which handles all scripts, web pages and validation, if required and then sends to client's browser. I a web server has vulnerablilities, then it is prone to different types of attack like sqli, xss, csrf, HPP, RCE, etc. So it is better to do server side validation. Here we will discuss some common vulnerabilities that we will pen test using Nmap.

HTTP Methods by Nmap:
HTTP methods are nothing but different types of requests handled by a web server to deliver web pages.
Web servers support different HTTP methods according to their configuration and software, and some of them could be dangerous under certain conditions. Pen testers need a way of quickly listing the available methods. Some methods are GET, HEAD, POST, TRACE, DEBUG, OPTION, DELETE, TRACK, PUT, etc. A full list of methods that are handled by the browser can be found at the link: http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

As previously described, Nmap can do easy work with an NSE script. Check the below script:

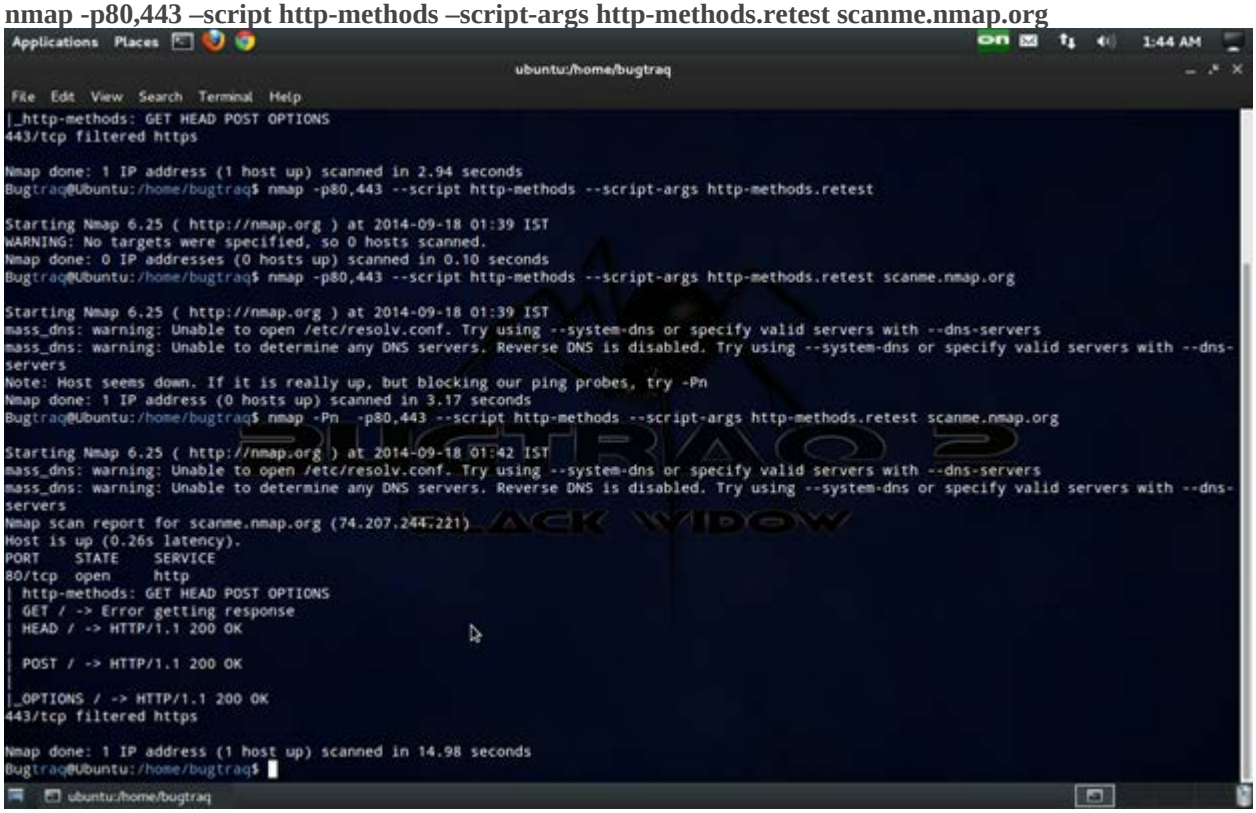
Cmd: **nmap -p80,443 –script http-methods scanme.nmap.org**



The argument -p80,443 –script http-methods makes Nmap launch the httpmethods script if a web server is found on ports 80 or 443 (-p80,443).

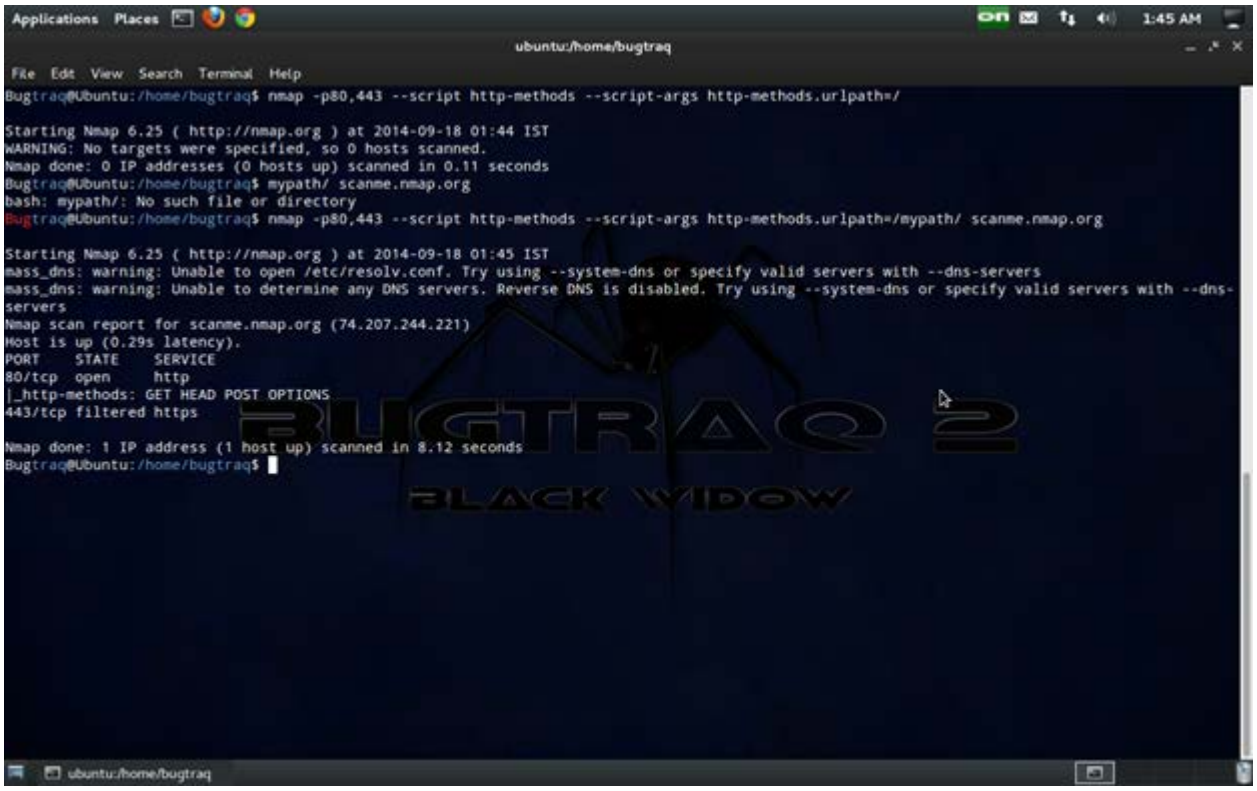
To individually check for the HTTP methods, the following scripts are needful.

Cmd:
nmap -p80,443 –script http-methods –script-args http-methods.retest scanme.nmap.org



By default, the script http-methods uses the root folder as the base path (/). If we wish to set a different base path, set the argument http-methods.url-path:

Cmd:
nmap -p80,443 –script http-methods –script-args http-methods.urlpath=/mypath/ scanme.nmap.org

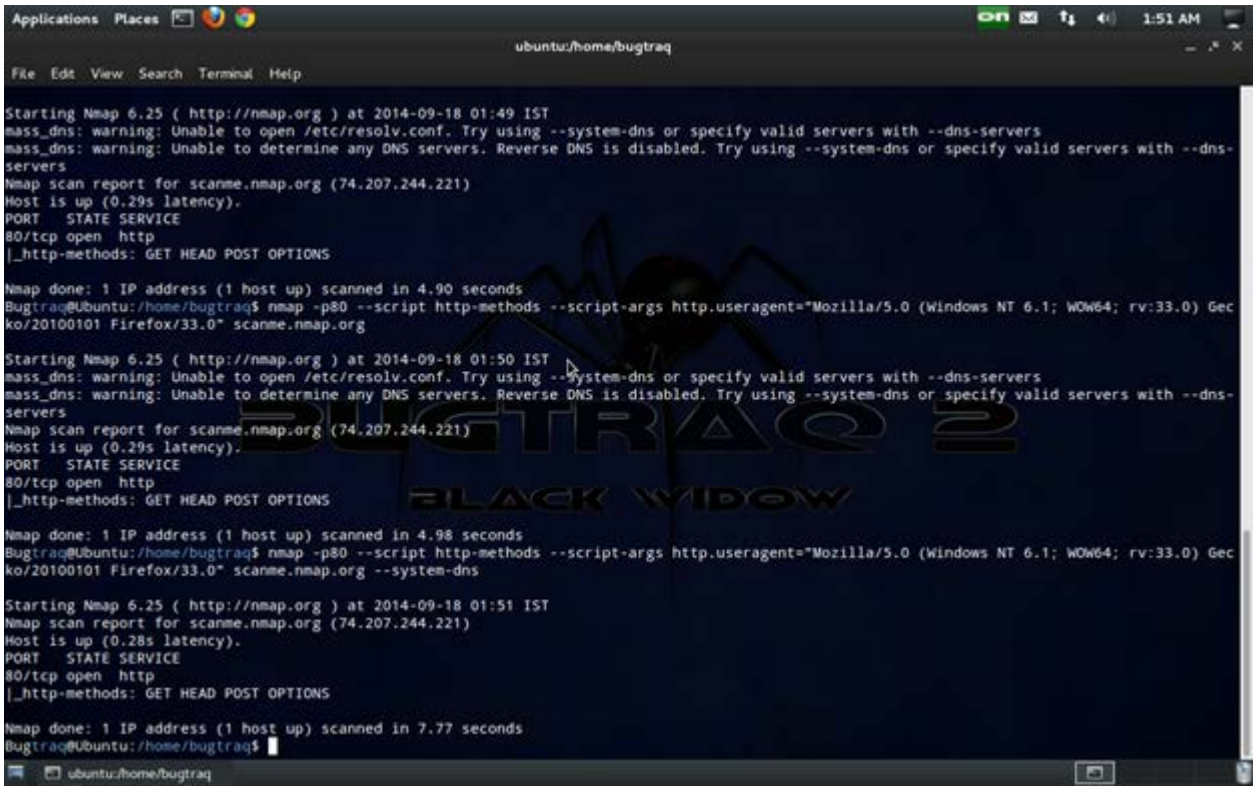


The HTTP methods TRACE, CONNECT, PUT, and DELETE might present a security risk, and they need to be tested thoroughly if supported by a web server or application. TRACE makes applications susceptible to Cross Site Tracing (XST) attacks and could lead to attackers accessing cookies marked as httpOnly. The CONNECT method might allow the web server to be used as an unauthorized web proxy. The methods PUT and DELETE have the ability to change the contents of a folder, and this could obviously be abused if the permissions are not set properly.

You can learn more about common risks associated with each method here: https://www.owasp.org/index.php/Test_HTTP_Methods_%28OTG-CONFIG-006%29

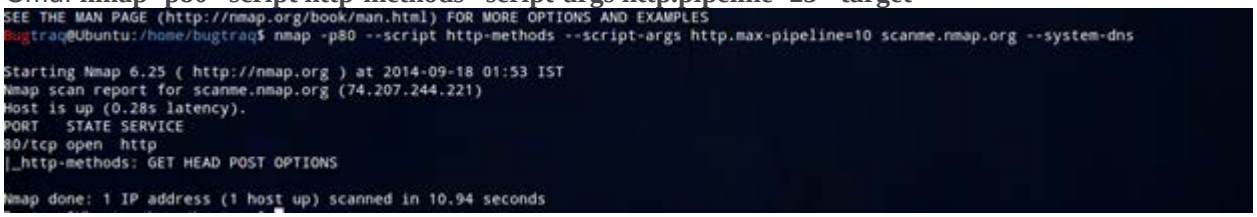
HTTP User Agent:
There are some packet filtering products that block requests that use Nmap's default HTTP User Agent. You can use a different HTTP User Agent by setting the argument http.useragent:

cmd: nmap -p80 –script http-methods –script-args http.useragent="Mozilla 5″ <target>



HTTP pipelining
Some web servers allow the encapsulation of more than one HTTP request in a single packet. This may speed up the execution of an NSE HTTP script, and it is recommended that it is used if the web server supports it. The HTTP library, by default, tries to pipeline 40 requests and auto adjusts the number of requests according to the traffic conditions, based on the Keep-Alive header.

Cmd: **nmap -p80 –script http-methods –script-args http.pipeline=25 <target>**



Additionally, we can use the argument http.max-pipeline to set the maximum number of HTTP requests to be added to the pipeline. If the script parameter http.pipeline is set, this argument will be ignored:

cmd:nmap -p80 –script http-methods –script-args http.max-pipeline=10 <target>
HTTP-Proxy scanning with Nmap:
HTTP proxies are used to make requests through their addresses, therefore hiding our real IP address from the target. Detecting them is important if you are a system administrator who needs to keep the network secure, or an attacker who spoofs his real origin. The following command shows how to detect open proxy:

cmd: **nmap –script http-open-proxy -p8080 <target>**

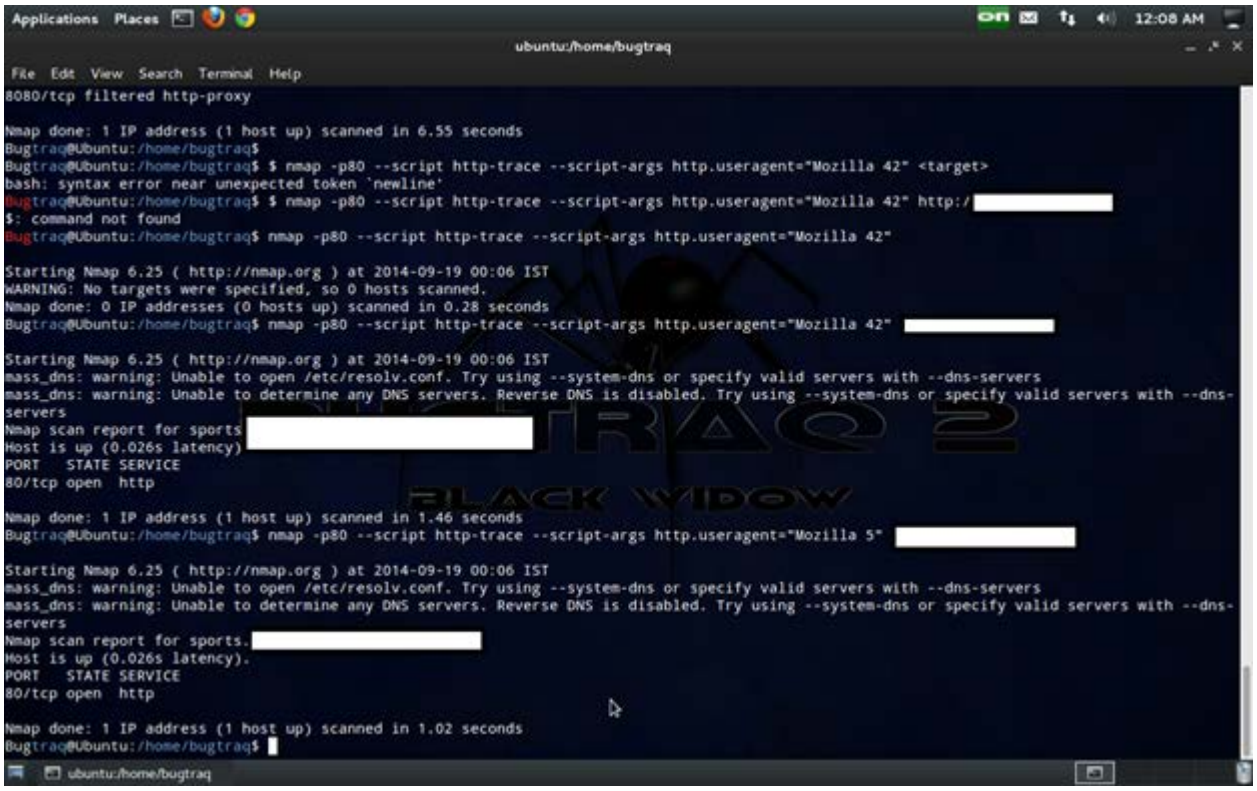
Here the argument –script http-open-proxy -p8080 to launch the NSE script http-open-proxy if a web server is found running on port 8080.

Different pattern:
We may use a different pattern by a specified URL to target for scanning. It can be done by a specified NSE Script. Follow the below command:

Cmd: **nmap –script http-open-proxy –script-args http-open-proxy.url=http://whatsmyip.org,http-open-.pattern=”Your IP address is” -p8080 <target>**

HTTP User Agent:
There are some packet filtering products that block requests that use Nmap’s default HTTP user agent. You can use a different HTTP User Agent by setting the argument http.useragent:

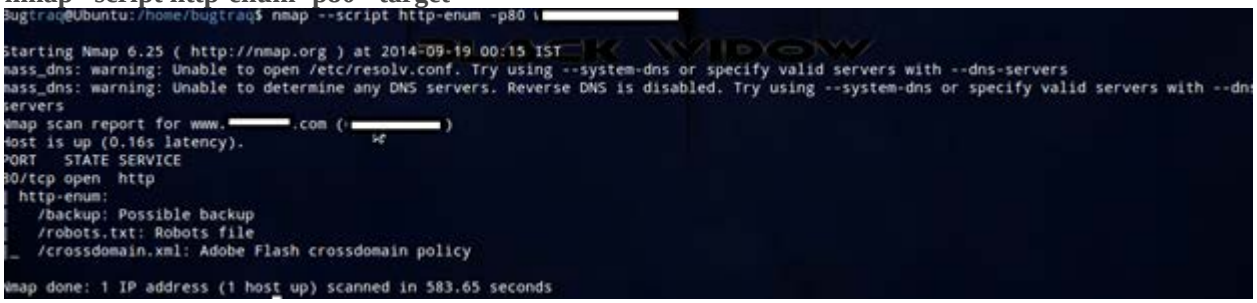
nmap -p80 –script http-trace –script-args http.useragent=”Mozilla 5″<target>



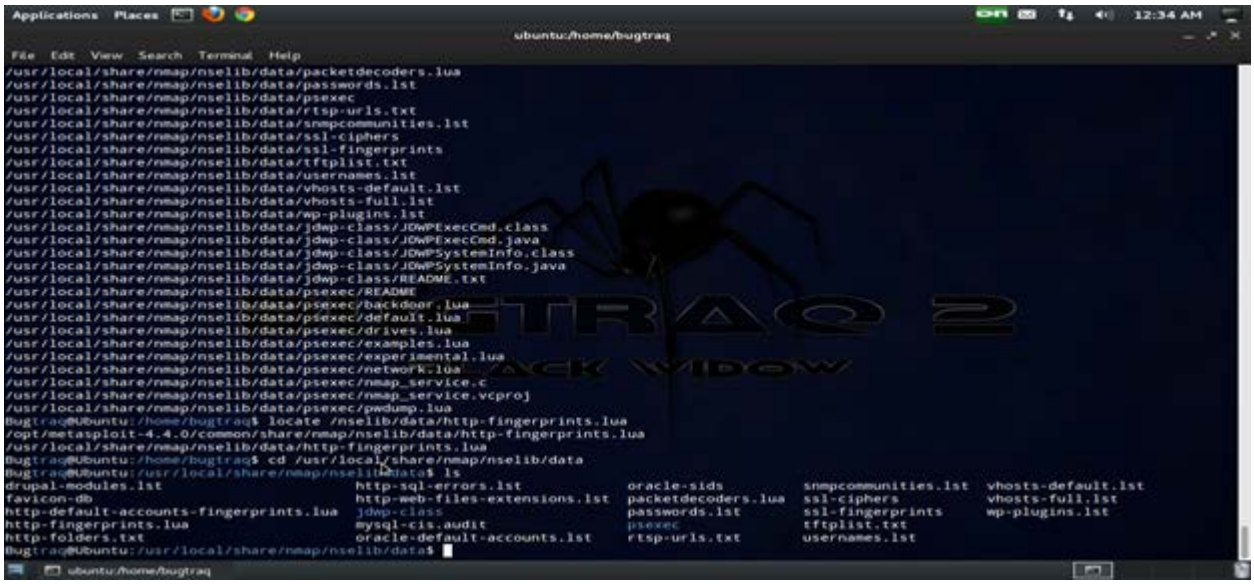
Remember: we can specify different browser agents like Mozilla, Chrome, Apple Webkit, etc.

Discovering interesting files and directories on admin accounts:
One of the common tasks during penetration tests that cannot be done manually is file and directory discovery. Web application vulns often disclosed directory listing, user account enumeration, account panel, config files, etc. For doing our work faster, Nmap gives an ideal way to discover through NSE SCRIPTS.

nmap –script http-enum -p80 <target>



Discovering LUA scripts

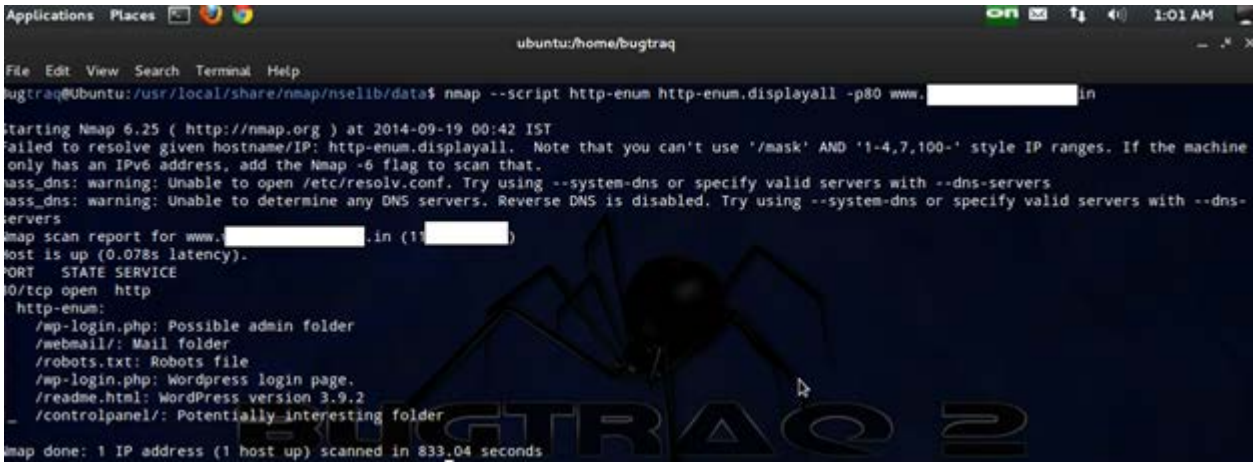


Entry in LUA table:

The fingerprints are stored in the file http-fingerprints.lua in /nselib/data/, and they are actually LUA tables. An entry looks like something like the following:

To display all the entries that returned a status code that could possibly indicate a page exists, use the script argument http-enum.displayall:

nmap script http-enum http-enum.displayall -p80 — <target>



BY HTTP User Agent

There are some packet filtering products that block requests made using Nmap's default HTTP User Agent. We can use a different HTTP User Agent by setting the argument http.useragent:

nmap -p80 –script http-enum –script-args http.useragent="Mozilla 5″<target>

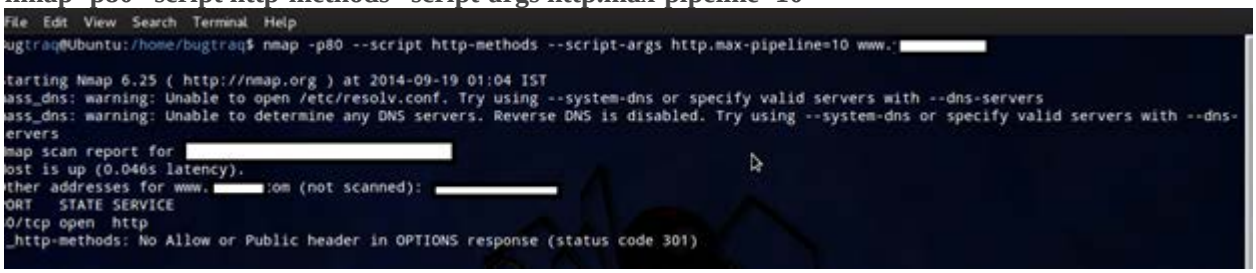
BY HTTP pipelining

Some web servers allow the encapsulation of more than one HTTP request in a single packet. This may speed up the execution of an NSE HTTP script, and it is recommended that it is used if the web server supports it. The HTTP library, by default, tries to pipeline 40 requests and automatically adjusts that number according to the traffic conditions, based on the Keep-Alive header.

$ nmap -p80 –script http-enum –script-args http.pipeline=25 <target>

Additionally, we can use the argument http.max-pipeline to set the maximum number of HTTP requests to be added to the pipeline. If the script parameter http.pipeline is set, this argument will be ignored:

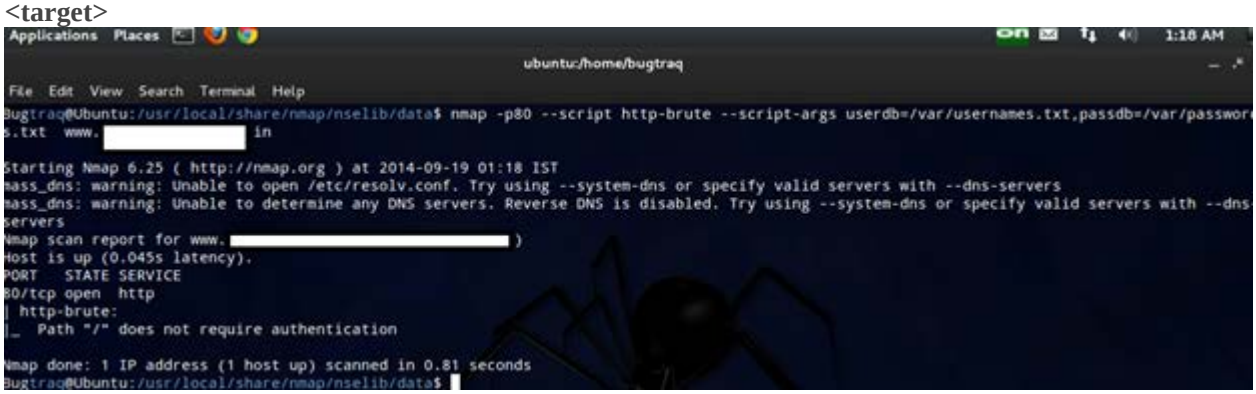
nmap -p80 –script http-methods –script-args http.max-pipeline=10



Brute forcing HTTP authentication:
Many home routers, IP webcams, and even web applications still rely on HTTP authentication these days, and penetration testers need to try a word list of weak passwords to make sure the system or user accounts are safe. Now, thanks to the NSE script http-brute, we can perform robust dictionary attacks against HTTPAuth protected resources. See below commands:

nmap -p80 –script http-brute –script-args http-brute.path=/admin/ <target>
The script http-brute depends on the NSE libraries unpwdb and brute. These libraries have several script arguments that can be used to tune the auditing for our brute force password. To use different username and password lists, set the arguments userdb and passdb:

nmap -p80 –script http-brute –script-args userdb=/var/usernames.txt,passdb=/var/passwords.txt <target>



Using HTTP User Agent

There are some packet filtering products that block requests made using Nmap's default

HTTP User Agent. You can use a different User Agent value by setting the argument

http.useragent:

$ nmap -p80 –script http-brute –script-args http.useragent="Mozilla 5″ <target>
Brute-force modes
The brute library supports different modes that alter the combinations used in the attack.basically a pentester will try to bruteforce the different parameters.they are using Burp Proxy and Intruder to perform the attack. The same thing can be used by Nmap.

The available modes are:

user: In this mode, for each user listed in userdb, every password in passdb will be tried.

nmap –script http-brute –script-args brute.mode=user <target>

pass: In this mode, for each password listed in passdb, every user in usedb will be tried.

nmap –script http-brute –script-args brute.mode=pass <target>

fcreds: This mode requires the additional argument brute.credfile.

nmap –script http-brute –script-args brute.mode=creds,brute.credfile=./creds.txt <target>
mod_userdir Pentesting:
Apache's module UserDir provides access to the user directories by using URIs with the syntax /~username/. With Nmap we can perform dictionary attacks and determine a list of valid usernames on the web server. To try to enumerate valid users in a web server with mod_userdir; use Nmap with these

arguments:

nmap -p80 –script http-userdir-enum <target>

HTTP User Agent

There are some packet filtering products that block requests made using Nmap's default HTTP User Agent. You can use a different User Agent value by setting the argument http.

useragent:

nmap -p80 –script http-brute –script-args http.useragent="Mozilla 42″ <target>
HTTP pipelining

Some web servers allow the encapsulation of more than one HTTP request in a single packet. This may speed up the execution of an NSE HTTP script, and it is recommended that it is used if the web server supports it. The HTTP library, by default, tries to pipeline 40 requests and auto adjusts that number according to the traffic conditions, based on the Keep-Alive header.

nmap -p80 –script http-methods –script-args http.pipeline=25 <target>
Testing for Default credentials:
Often default credentials are found in the web applications.NSE scripts made easy to find the vulnerable application.

nmap -p80 –script http-default-accounts <target>
The script detects web applications by looking at known paths and initiating a login

routine using the stored, default credentials. It depends on a fingerprint file located at /

nselib/data/http-default-accounts.nse

WordPress Auditing:
To find accounts with weak passwords in WordPress installations, use the following Nmap command:

$ nmap -p80 –script http-wordpress-brute <target>

To set the number of threads, use the script argument http-wordpress-brute.threads:

$ nmap -p80 –script http-wordpress-brute –script-args http-wordpressbrute.
threads=5 <target>
If the server has virtual hosting, set the host field by using the argument http-wordpressbrute.

hostname:

nmap -p80 –script http-wordpress-brute –script-args http-
wordpressbrute.hostname=”ahostname.wordpress.com” <target>
To set a different login URI, use the argument http-wordpress-brute.uri:

$ nmap -p80 –script http-wordpress-brute –script-args http-wordpressbrute.
uri=”/hidden-wp-login.php” <target>
To change the name of the POST variable that stores the usernames and passwords, set the arguments http-wordpress-brute.uservar and http-wordpress-brute.passvar:

$ nmap -p80 –script http-wordpress-brute –script-args http-wordpressbrute.uservar=usuario,http-
wordpress-brute.passvar=pasguord <target>
Joomla Auditing:

Joomla! is a very popular CMS that is used for many different purposes, including

e-commerce. Detecting user accounts with weak passwords is a common task for penetration

testers, and Nmap helps with that by using the NSE script http-joomla-brute.

nmap -p80 –script http-joomla-brute <target>

NB: the same method for WordPress will apply to Joomla
Detecting Web Application Firewall:
Like a firewall app, WAF can also be used by the server for protecting against malicious attacks. Web servers are often protected by packet filtering systems that drop or redirect suspected malicious packets. Web penetration testers benefit from knowing that there is a traffic filtering system between them and the target application. If that is the case, they can try more rare or stealthy techniques to try to bypass the Web Application Firewall (WAF) or Intrusion Prevention System (IPS). It also helps them to determine if a vulnerability is
actually exploitable in the current environment.

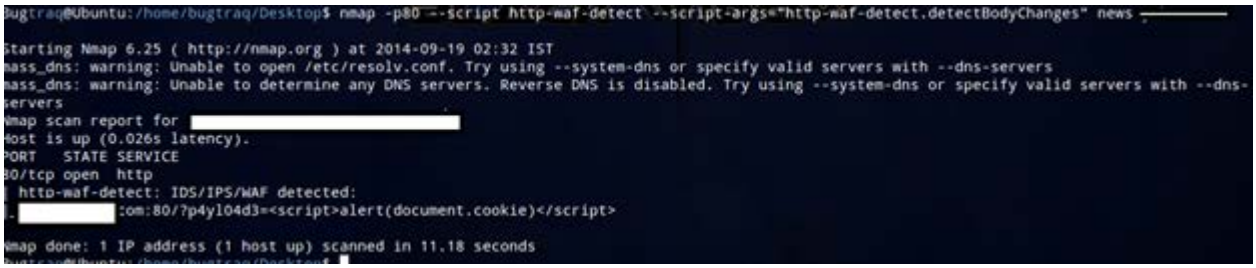
To detect a Web Application Firewall or Intrusion Prevention System:

nmap p80 –script http-waf-detect <target>-

As we can see, there is a firewall mod_security which throws an error:

To detect changes in the response body, use the argument http-waf-detect. detectBodyChanges. I recommend that you enable it when dealing with pages with little dynamic content:

nmap -p80 –script http-waf-detect –script-args=”http-waf-detect.detectBodyChanges” <target>



To include more attack payloads, use the script argument http-waf-detect.aggro. This mode generates more HTTP requests but can also trigger more products:

nmap -p80 –script http-waf-detect –script-args=”http-waf-detect.aggro” <target>

HTTP User Agent

There are some packet filtering products that block requests made using Nmap’s default HTTP User Agent. You can use a different User Agent value by setting the argument http.useragent:

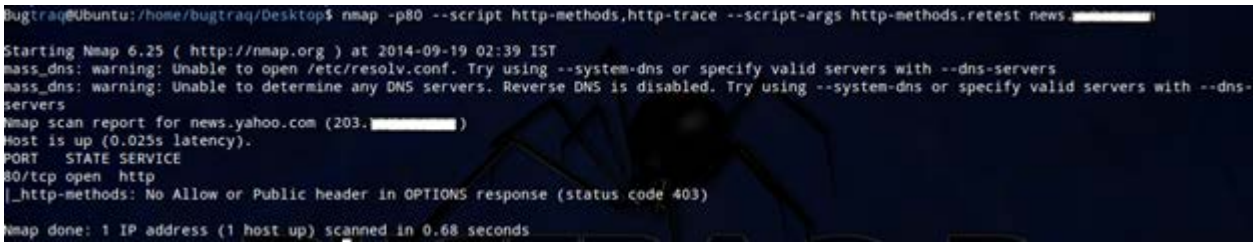
nmap -p80 –script http-waf-detect –script-args http.useragent=”Mozilla 42″ <target>
HTTP pipelining

Some web servers allow the encapsulation of more than one HTTP request in a single packet. This may speed up the execution of an NSE HTTP script, and it is recommended that it is used if the web server supports it. The HTTP library, by default, tries to pipeline 40 requests and automatically adjusts that number according to the traffic conditions, based on the Keep-Alive header.

$ nmap -p80 –script http-methods –script-args http.pipeline=25 <target>
Additionally, you can use the argument http.max-pipeline to set the maximum number of HTTP requests to be added to the pipeline. If the script parameter http.pipeline is set, this argument will be ignored:

$.nmap -p80 –script http-methods –script-args http.max-pipeline=10 <target>
Detecting XST Vulnerabilities:
Cross Site Tracing (XST) vulnerabilities are caused by the existence of Cross Site Scripting vulnerabilities (XSS) in web servers where the HTTP method TRACE is enabled. This technique is mainly used to bypass cookie restrictions imposed by the directive httpOnly. Pen testers can save time by using Nmap to quickly determine if the web server has the method TRACE enabled.

nmap -p80 –script http-methods,http-trace –script-args http-methods.retest <target>



Detecting XSS Vulnerabilities:
XSS or cross site scripting is a well known attack where an attacker can execute JavaScript. Cross Site Scripting vulnerabilities allow attackers to spoof content, steal user cookies, and even execute malicious code on the user's browsers. There are even advanced exploitation frameworks such as Beef that allow attackers to perform complex attacks through JavaScript hooks. Web pen testers can use Nmap to discover these vulnerabilities in web servers in an automated manner. For that, Nmap has a solution, which is NSE.
nmap -p80 –script http-unsafe-output-escaping <target>

The script http-unsafe-output-escaping was written by Martin Holst Swende, and it spiders a web server to detect the possible problems with the way web applications return output based on user input. The script inserts the following payload into all the parameters it finds:

ghz%3Ehzx%22zxc%27xcv
The payload shown above is designed to detect the characters > " ', which could cause Cross Site Scripting vulnerabilities. The official documentation of the scripts http-unsafe-output-escaping and httpphpself-xss can be found at the following URLs:

http://nmap.org/nsedoc/scripts/http-phpself-xss.html

http://nmap.org/nsedoc/scripts/http-unsafe-output-escaping.html

Detecting SQL Injection:
SQL injection vulnerabilities are caused by the lack of sanitation of user input, and they allow attackers to execute DBMS queries that could compromise the entire system.

To scan a web server looking for files vulnerable to SQL injection by using Nmap, use the following command:

nmap -p80 –script http-sql-injection <target>

The httpspider library behavior can be configured via library arguments. By default, it uses pretty conservative values to save resources, but during a comprehensive test, we need to tweak several of them to achieve optimum results. For example, the library will only crawl 20 pages by default, but we can set the argument httpspider.maxpagecount accordingly for bigger sites, as shown in the following command:

nmap -p80 –script http-sql-injection –script-args httpspider.maxpagecount=200 <target>
Another interesting argument is httpspider.withinhost, which limits the web crawler to a given host. This is turned on by default, but if you need to test a collection of web applications linked to each other, you could use the following command:

nmap -p80 –script http-sql-injection –script-args httpspider.withinhost=false <target>
The official documentation for the library can be found at

http://nmap.org/nsedoc/lib/httpspider.html

HTTP User Agent

There are some packet filtering products that block requests made using Nmap's default HTTP User Agent. We can use a different User Agent value by setting the argument http.useragent:

nmap -p80 –script http-sql-injection –script-args http.useragent="Mozilla 42″ <target>
HTTP pipelining

Some web servers allow the encapsulation of more than one HTTP request in a single packet. This may speed up the execution of an NSE HTTP script, and it is recommended that this is used if the web server supports it. The HTTP library, by default, tries to pipeline 40 requests and automatically adjusts that number according to the traffic conditions, based on the Keep-Alive header.

nmap -p80 –script http-sql-injection –script-args http.pipeline=25<target
N.B: there are some examples for which a screenshot is unavailable because it's impossible for everything to be put here. My recommendation is for all readers to try all commands and let me know if any problem occurs. There are lots of NSE scripts available for pen testing. The above are a few examples. All the scripts will be discussed in the upcoming installments.
CYBER THREAT DISCLOSURE POLICY:

The above scanning /malicious activity is done by the proper permission of the respective website domain owner. It is recommended to not use these skills for attacking/hacking a website.