

Nmap Network Scanning

Nmap Network Scanning is the official guide to the Nmap Security Scanner, a free and open source utility used by millions of people for network discovery, administration, and security auditing. From explaining port scanning basics for novices to detailing low-level packet crafting methods used by advanced hackers, this book suits all levels of security and networking professionals. A 42-page reference guide documents every Nmap feature and option, while the rest of the book demonstrates how to apply those features to quickly solve real-world tasks. Examples and diagrams show actual communication on the wire.

Topics include subverting firewalls and intrusion detection systems, optimizing Nmap performance, and automating common networking tasks with the Nmap Scripting Engine. Hints and instructions are provided for common uses such as taking network inventory, penetration testing, detecting rogue wireless access points, and quashing network worm outbreaks. Nmap runs on Windows, Linux, and Mac OS X.

Nmap's original author, Gordon “Fyodor” Lyon, wrote this book to share everything he has learned about network scanning during more than a decade of Nmap development. It was briefly the #1 selling computer book on Amazon ([screenshot](#)). The book is in English, though [several translations](#) are in the works.

Key facts: The ISBN is 978-0-9799587-1-7 (ISBN-10 is 0-9799587-1-7) and suggested retail prices are \$49.95 in the U.S., £34.95 in the U.K., and €39.95 in Europe. Like most books, it costs less online (as little as \$32.97 - see [purchasing options](#)). It is 468 pages long. The official release date was January 1, 2009, though Amazon managed to beat that by a couple weeks.

About half of the content is [available in the free online edition](#). Chapters exclusive to the print edition include “Detecting and Subverting Firewalls and Intrusion Detection Systems”, “Optimizing Nmap Performance”, “Port Scanning Techniques and Algorithms”, “Host Discovery (Ping Scanning)”, and more. The solution selections which provide detailed instructions on the best way to solve common networking tasks are also exclusive to the printed book. The [final table of contents](#) and [cover art](#) are available.

Chapter 1. Getting Started with Nmap

Table of Contents

[Introduction](#)

[Nmap Overview and Demonstration](#)

[Avatar Online](#)

[Saving the Human Race](#)

[MadHat in Wonderland](#)

[The Phases of an Nmap Scan](#)

[Legal Issues](#)

[Is Unauthorized Port Scanning a Crime?](#)

[Can Port Scanning Crash the Target Computer/Networks?](#)

[Nmap Copyright](#)

[The History and Future of Nmap](#)

Introduction

Nmap (“Network Mapper”) is a free and open source utility for network exploration 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 both console and graphical versions are available.

This chapter uses fictional stories to provide a broad overview of Nmap and how it is typically used. An important legal section helps users avoid (or at least be aware of) controversial usage that could lead to ISP account cancellation or even civil and criminal charges. It also discusses the risks of crashing remote machines as well as miscellaneous issues such as the Nmap license (GNU GPL), and copyright.

Nmap Overview and Demonstration

Sometimes the best way to understand something is to see it in action. This section includes examples of Nmap used in (mostly) fictional yet typical circumstances. Nmap newbies should not expect to understand everything at once. This is simply a broad overview of features that are described in depth in later chapters. The “solutions” included throughout this book demonstrate many other common Nmap tasks for security auditors and network administrators.

Avatar Online

Felix dutifully arrives at work on December 15th, although he does not expect many structured tasks. The small San Francisco penetration-testing firm he works for has been quiet lately due to impending holidays. Felix spends business hours pursuing his latest hobby of building powerful Wi-Fi antennas for wireless assessments and war driving exploration. Nevertheless, Felix is hoping for more business. Hacking has been his hobby and fascination since a childhood spent learning everything he could about networking, security, Unix, and phone systems. Occasionally his curiosity took him too far, and Felix was almost swept up in the 1990 Operation Sundevil prosecutions. Fortunately Felix emerged from adolescence without a criminal record, while retaining his expert knowledge of security weaknesses. As a professional, he is able to perform the same types of network intrusions as before, but with the added benefit of contractual immunity from prosecution and even a paycheck! Rather than keeping his creative exploits secret, he can brag about them to client management when presenting his reports. So Felix was not disappointed when his boss interrupted his antenna soldering to announce that the sales department finally closed a pen-testing deal with the Avatar Online gaming company.

Avatar Online (AO) is a small company working to create the next generation of massive multi-player online role-playing games (MMORPGs). Their product, inspired by the Metaverse envisioned in Neil Stevenson's *Snow Crash*, is fascinating but still highly confidential. After witnessing the [high-profile leak](#) of Valve Software's upcoming game source code, AO quickly hired the security consultants. Felix's task is to initiate an external (from outside the firewall) vulnerability assessment while his partners work on physical security, source code auditing, social engineering, and so forth. Felix is permitted to exploit any vulnerabilities found.

The first step in a vulnerability assessment is network discovery. This reconnaissance stage determines what IP address ranges the target is using, what hosts are available, what services those hosts are offering, general network topology details, and what firewall/filtering policies are in effect.

Determining the IP ranges to scan would normally be an elaborate process involving ARIN (or another geographical registry) lookups, DNS queries and zone transfer attempts, various web sleuthing techniques, and more. But in this case, Avatar Online explicitly specified what networks they want tested: the corporate network on 6.209.24.0/24 and their production/DMZ systems residing on 6.207.0.0/22. Felix checks the ARIN IP allocation records anyway and confirms that these IP ranges belong to AO^[2]. Felix subconsciously decodes the CIDR notation^[3] and recognizes this as 1,280 IP addresses. No problem.

Being the careful type, Felix first starts out with what is known as an Nmap list scan (-sL option). This feature simply enumerates every IP address in the given target netblock(s) and does a reverse-DNS lookup (unless -n was specified) on each. One reason to do this first is stealth. The names of the hosts can hint at potential vulnerabilities and allow for a better understanding of the target network, all without raising alarm bells^[4]. Felix is doing this for another reason—to double-check that the IP ranges are correct. The systems administrator who provided the IPs might have made a mistake, and scanning the wrong company would be a disaster. The contract signed with Avatar Online may act as a get-out-of-jail-free card for penetrating their networks, but will not help if Felix accidentally roots another company's server! The command he uses and an excerpt of the results are shown in [Example 1.1](#).

Example 1.1. Nmap list scan against Avatar Online IP addresses

```
felix> nmap -sL 6.209.24.0/24 6.207.0.0/22

Starting Nmap ( http://nmap.org )
Host 6.209.24.0 not scanned
Host fw.corp.avataronline.com (6.209.24.1) not scanned
Host dev2.corp.avataronline.com (6.209.24.2) not scanned
Host 6.209.24.3 not scanned
Host 6.209.24.4 not scanned
Host 6.209.24.5 not scanned
```

```
...
Host dhcp-21.corp.avataronline.com (6.209.24.21) not
scanned
Host dhcp-22.corp.avataronline.com (6.209.24.22) not
scanned
Host dhcp-23.corp.avataronline.com (6.209.24.23) not
scanned
Host dhcp-24.corp.avataronline.com (6.209.24.24) not
scanned
Host dhcp-25.corp.avataronline.com (6.209.24.25) not
scanned
Host dhcp-26.corp.avataronline.com (6.209.24.26) not
scanned
...
Host 6.207.0.0 not scanned
Host gw.avataronline.com (6.207.0.1) not scanned
Host ns1.avataronline.com (6.207.0.2) not scanned
Host ns2.avataronline.com (6.207.0.3) not scanned
Host ftp.avataronline.com (6.207.0.4) not scanned
Host 6.207.0.5 not scanned
Host 6.207.0.6 not scanned
Host www.avataronline.com (6.207.0.7) not scanned
Host 6.207.0.8 not scanned
...
Host cluster-c120.avataronline.com (6.207.2.120) not
scanned
Host cluster-c121.avataronline.com (6.207.2.121) not
scanned
Host cluster-c122.avataronline.com (6.207.2.122) not
scanned
Host cluster-c123.avataronline.com (6.207.2.123) not
scanned
Host cluster-c124.avataronline.com (6.207.2.124) not
scanned
...
Host 6.207.3.253 not scanned
Host 6.207.3.254 not scanned
Host 6.207.3.255 not scanned
Nmap done: 1280 IP addresses scanned in 331.49 seconds
felix>
```

Reading over the results, Felix finds that all of the machines with reverse-DNS entries resolve to Avatar Online. No other businesses

seem to share the IP space. Moreover, these results give Felix a rough idea of how many machines are in use and a good idea of what many are used for. He is now ready to get a bit more intrusive and try a port scan. He uses Nmap features that try to determine the application and version number of each service listening on the network. He also requests that Nmap try to guess the remote operating system via a series of low-level TCP/IP probes known as OS fingerprinting. This sort of scan is not at all stealthy, but that does not concern Felix. He is interested in whether the administrators of AO even notice these blatant scans. After a bit of consideration, Felix settles on the following command:

```
nmap -sS -p- -PS22,80,113,33334 -PA80,113,21000 -PU19000  
-PE -A -T4 -oA avatartcpscan-121503 6.209.24.0/24  
6.207.0.0/22
```

These options are described in later chapters, but here is a quick summary of them.

-sS

Enables the efficient TCP port scanning technique known as SYN scan. Felix would have added a U at the end if he also wanted to do a UDP scan, but he is saving that for later. SYN scan is the default scan type, but stating it explicitly does not hurt.

-p-

Requests that Nmap scan every port from 1-65535. The default is to scan only ports one through 1024, plus about 600 others explicitly mentioned in the nmap-services database. This option format is simply a short cut for -p1-65535. He could have specified -p0-65535 if he wanted to scan the rather illegitimate port zero as well. The -p option has a very flexible syntax, even allowing the specification of a differing set of UDP and TCP ports.

```
-PS22,80,113,33334 -PA80,113,21000 -PU19000 -PE
```

These are all *ping types* used in combination to determine whether a host is really available and avoid wasting a lot of time scanning IP addresses that are not in use. This particular incantation sends a TCP SYN packet to ports 22, 80, 113, and

33334; a TCP ACK packet to ports 80, 113, and 21000; a UDP packet to port 19000; and a normal ICMP echo request packet. If Nmap receives a response from the target host itself to any of these probes, it considers the host to be up and available for scanning. This is more extensive than the Nmap default, which simply sends an echo request and an ACK packet to port 80. In a pen-testing situation, you often want to scan every host even if they do not seem to be up. After all, they could just be heavily filtered in such a way that the probes you selected are ignored but some other obscure port may be available. To scan every IP whether it shows an available host or not, specify the `-PN` option instead of all of the above. Felix starts such a scan in the background, though it may take a day to complete.

`-A`

This shortcut option turns on Advanced and Aggressive features such as OS and service detection. At the time of this writing it is equivalent to `-sV -sC -O --traceroute` (version detection, Nmap Scripting Engine, remote OS detection, and traceroute). More features may be added to `-A` later.

`-T4`

Adjusts timing to the aggressive level (#4 of 5). This is the same as specifying `-T aggressive`, but is easier to type and spell. In general, the `-T4` option is recommended if the connection between you and the target networks are faster than dialup modems.

`-oA avatartcpscan-121503`

Outputs results in every format (normal, XML, greppable) to files named `avatartcpscan-121503.<extension>` where the extensions are `.nmap`, `.xml`, and `.gnmap` respectively. All of the output formats include the start date and time, but Felix likes to note the date explicitly in the filename. Normal output and errors are still sent to stdout^[5] as well.

`6.209.24.0/24 6.207.0.0/22`

These are the Avatar Online netblocks discussed above. They are given in CIDR notation, but Nmap allows them to be

specified in many other formats. For example, 6.209.24.0/24 could instead be specified as 6.209.24.0-255.

Since such a comprehensive scan against more than a thousand IP addresses could take a while, Felix simply starts it executing and resumes work on his Yagi antenna. A couple hours later he notices that it has finished and takes a peek at the results. [Example 1.2](#) shows one of the machines discovered.

Example 1.2. Nmap results against an AO firewall

```
Interesting ports on fw.corp.avataronline.com
(6.209.24.1):
(The 65530 ports scanned but not shown below are in
state: filtered)
PORT      STATE  SERVICE      VERSION
22/tcp    open   ssh          OpenSSH 3.7.1p2 (protocol
1.99)
53/tcp    open   domain       ISC BIND 9.2.1
110/tcp   open   pop3         Courier pop3d
113/tcp   closed auth
143/tcp   open   imap         Courier Imap 1.6.X - 1.7.X
3128/tcp  open   http-proxy   Squid webproxy 2.2.STABLE5
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux Kernel 2.4.0 - 2.5.20
Uptime 3.134 days
```

To the trained eye, this conveys substantial information about AO's security posture. Felix first notes the reverse DNS name—this machine is apparently meant to be a firewall for their corporate network. The next line is important, but all too often ignored. It states that the vast majority of the ports on this machine are in the filtered state. This means that Nmap is unable to reach the port because it is blocked by firewall rules. The fact that all ports except for a few chosen ones are in this state is a sign of security competence. Deny-by-default is a security mantra for good reasons—it means that even if someone accidentally left SunRPC (port 111) open on this machine, the firewall rules would prevent attackers from communicating with it.

Felix then looks at every port line in turn. The first port is Secure Shell (OpenSSH). Version 3.7.1p2 is common, as many

administrators upgraded to this version due to potentially exploitable buffer management bugs affecting previous versions. Nmap also notes that the SSH protocol is 1.99, suggesting that the inferior legacy SSHv1 protocol is supported. A truly paranoid sysadmin would only allow SSH connections from certain trusted IP addresses, but one can argue for open access in case the administrator needs emergency access while far from home. Security often involves trade-offs, and this one may be justifiable. Felix makes a note to try his brute force password cracker and especially his private timing-based SSH user enumeration tool against the server.

Felix also notes port 53. It is running ISC BIND, which has a long history of remotely exploitable security holes. Visit the [BIND security page](#) for further details. BIND 9.2.1 even has a potentially exploitable buffer overflow, although the default build is not vulnerable. Felix checks and finds that this server is not vulnerable to the libbind issue, but that is beside the point. This server almost certainly should not be running an externally-accessible nameserver. A firewall should only run the bare essentials to minimize the risk of a disastrous compromise. Besides, this server is not authoritative for any domains—the real nameservers are on the production network. An administrator probably only meant for clients within the firewall to contact this nameserver, but did not bother locking it down to only the internal interface. Felix will later try to gather important information from this unnecessary server using zone transfer requests and intrusive queries. He may attempt cache poisoning as well. By spoofing the IP of windowsupdate.microsoft.com or another important download server, Felix may be able to trick unsuspecting internal client users into running a trojan-horse program that provides him with full network access behind the firewall.

The next two open ports are 110 (POP3) and 143 (IMAP). Note that 113 (auth) between them is closed instead of open. POP3 and IMAP are mail retrieval services which, like BIND, have no legitimate place on this server. They are also a security risk in that they generally transfer the mail and (even worse) authentication credentials unencrypted. Users should probably VPN in and check their mail from an internal server. These ports could also be wrapped in SSL encryption. Nmap would have then listed the services as ssl/pop3 and ssl/imap. Felix will try his user enumeration and password guessing attacks on these services, which will probably be much more effective than against SSH.

The final open port is a Squid proxy. This is another service that may have been intended for internal client use and should not be accessible from the outside (and particularly not on the firewall). Felix's initially positive opinion of the AO security administrators drops further. Felix will test whether he can abuse this proxy to connect to other sites on the Internet. Spammers and malicious hackers often use proxies in this way to hide their tracks. Even more critical, Felix will try to proxy his way into the *internal* network. This common attack is how [Adrian Lamo](#) broke into the New York Times internal network in 2002. Lamo was caught after he called reporters to brag about his exploits against the NY Times and other companies [in detail](#).

The following lines disclose that this is a Linux box, which is valuable information when attempting exploitation. The low three-day uptime was detected during OS fingerprinting by sending several probes for the TCP timestamp option value and extrapolating the line back to zero.

Felix then examines the Nmap output for another machine, as shown in [Example 1.3](#).

Example 1.3. Another interesting AO machine

```
Interesting ports on dhcp-23.corp.avataonline.com
(6.209.24.23):
(The 65526 ports scanned but not shown below are in
state: closed)
PORT      STATE      SERVICE      VERSION
135/tcp    filtered  msrpc
136/tcp    filtered  profile
137/tcp    filtered  netbios-ns
138/tcp    filtered  netbios-dgm
139/tcp    filtered  netbios-ssn
445/tcp    open      microsoft-ds  Microsoft Windows XP
microsoft-ds
1002/tcp   open      windows-icfw?
1025/tcp   open      msrpc        Microsoft Windows msrpc
16552/tcp  open      unknown
Device type: general purpose
Running: Microsoft Windows NT/2K/XP
OS details: Microsoft Windows XP Professional RC1+
through final release
```

Felix smiles when he spies this Windows XP box on the Network. Thanks to a spate of MS RPC vulnerabilities, those machines are trivial to compromise if the OS patches aren't up-to-date. The second line shows that the default state is closed, meaning the firewall does not have the same deny-by-default policy for this machine as for itself. Instead they tried to specifically block the Windows ports they consider dangerous on 135-139. This filter is woefully inadequate, as MS exports MS RPC functionality on many other ports in Windows XP. TCP ports 445 and 1025 are two examples from this scan. While Nmap failed to recognize 16552, Felix has seen this pattern enough to know that it is probably the MS Messenger Service. If AO had been using deny-by-default filtering, port 16552 would not be accessible in the first place. Looking through the results page, Felix sees several other Windows machines on this DHCP network. Felix cannot wait to try his favorite DCOM RPC exploit against them. It was written by HD Moore and is available at <http://www.metasploit.com/tools/dcom.c>. If that fails, there are a couple newer MS RPC vulnerabilities he will try.

Felix continues poring over the results for vulnerabilities he can leverage to compromise the network. On the production network, he sees that gw.avataronline.com is a Cisco router that also acts as a rudimentary firewall for the systems. They fall into the trap of only blocking privileged ports (those under 1024), which leaves a bunch of vulnerable SunRPC and other services accessible on that network. The machines with names like clust-* each have dozens of ports open that Nmap does not recognize. They are probably custom daemons running the AO game engine. www.avataronline.com is a Linux box with an open Apache server on the HTTP and HTTPS ports. Unfortunately, it is linked with an exploitable version of the OpenSSL library. Oops! Before the sun sets, Felix has gained privileged access to hosts on both the corporate and production networks.

As Felix has demonstrated, Nmap is frequently used by security auditors and network administrators to help locate vulnerabilities on client/corporate networks. Subsequent chapters describe the techniques used by Felix, as well as many other Nmap features, in much greater detail.

Saving the Human Race

Figure 1.1. Trinity begins her assault



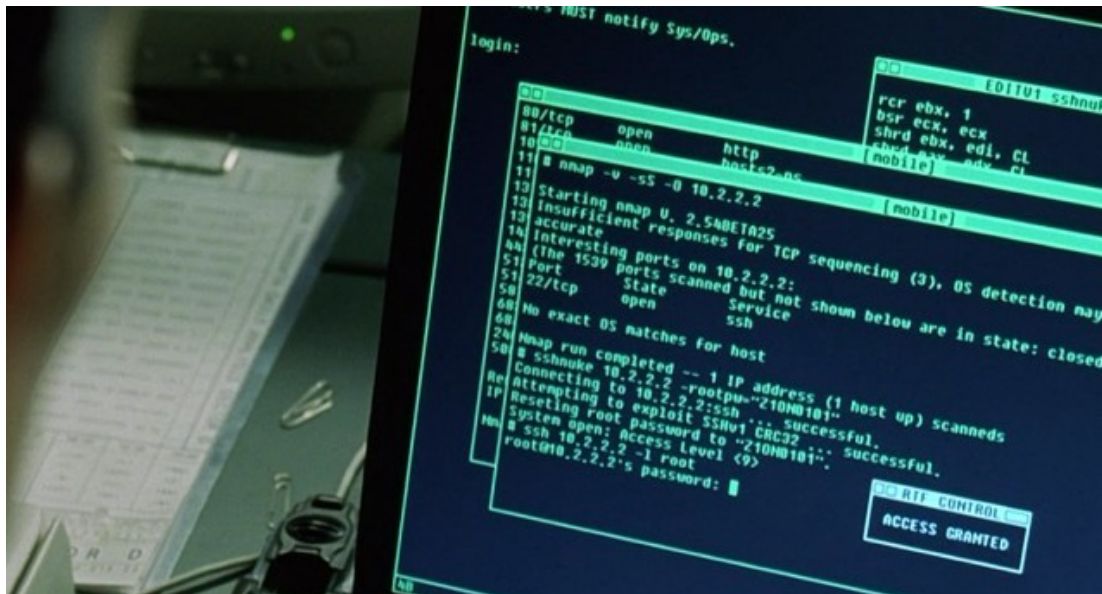
Trinity is in quite a pickle! Having discovered that the world we take for granted is really a virtual “Matrix” run by machine overlords, Trinity decides to fight back and free the human race from this mental slavery. Making matters worse, her underground colony of freed humans (Zion) is under attack by 250,000 powerful alien sentinels. Her only hope involves deactivating the emergency power system for 27 city blocks in less than five minutes. The previous team died trying. In life's bleakest moments when all hope seems to be lost, what should you turn to? Nmap, of course! But not quite yet.

She first must defeat the perimeter security, which on many networks involves firewalls and intrusion detection systems (IDS). She is well aware of advanced techniques for circumventing these devices (covered later in this book). Unfortunately, the emergency power system administrators knew better than to connect such a critical system to the Internet, even indirectly. No amount of source routing or IP ID spoofed scanning will help Trinity overcome this “air gap” security. Thinking fast, she devises a clever plan that involves jumping her motorcycle off the rooftop of a nearby building, landing on the power station guard post, and then beating up all of the security guards. This advanced technique is not covered in any physical security manual, but proves highly effective. This demonstrates how clever hackers research and devise their own

attacks, rather than always utilizing the script-kiddie approach of canned exploits.

Trinity fights her way to the computer room and sits down at a terminal. She quickly determines that the network is using the private 10.0.0.0/8 network address space. A ping to the network address generates responses from dozens of machines. An Nmap ping scan would have provided a more comprehensive list of available machines, but using the broadcast technique saved precious seconds. Then she whips out Nmap^[6]. The terminal has version 2.54BETA25 installed. This version is ancient (2001) and less efficient than newer releases, but Trinity had no time to install a better version from the future. This job will not take long anyway. She runs the command **nmap -v -sS -O 10.2.1.3**. This executes a TCP SYN scan and OS detection against 10.2.1.3 and provides verbose output. The host appears to be a security disaster—AIX 3.2 with well over a dozen ports open. Unfortunately, this is not the machine she needs to compromise. So she runs the same command against 10.2.2.2. This time the target OS is unrecognized (she should have upgraded Nmap!) and only has port 22 open. This is the Secure Shell encrypted administration service. As any sexy PVC-clad hacker goddess knows, many SSH servers from around that time (2001) have an exploitable vulnerability in the CRC32 compensation attack detector. Trinity whips out an all-assembly-code exploit and utilizes it to change the root password of the target box to Z10N0101. Trinity uses much more secure passwords under normal circumstances. She logs in as root and issues a command to disable the emergency backup power system for 27 city blocks, finishing just in time! Here is a shot of the action—squint just right and you should be able to read the text.

Figure 1.2. Trinity scans the Matrix



In addition, a terminal-view video showing the whole hack is available [on the Internet](#). At least it will be until the MPAA finds out and sends sentinels or lawyers after the webmasters.

MadHat in Wonderland

This story differs from the previous ones in that it is actually true. Written by frequent Nmap user and contributor Lee “MadHat” Heath, it describes how he enhanced and customized Nmap for daily use in a large enterprise. In true open source spirit, he has released these valuable scripts [on his Web site](#). IP addresses have been changed to protect the corporate identity. The remainder of this section is in his own words.

After spending the past couple of decades learning computers and working my way up from tech support through sysadmin and into my dream job of Information Security Officer for a major Internet company, I found myself with a problem. I was handed the sole responsibility of security monitoring for our entire IP space. This was almost 50,000 hosts worldwide when I started several years ago, and it has doubled since then.

Scanning all of these machines for potential vulnerabilities as part of monthly or quarterly assessments would be tough enough, but management wanted it done daily. Attackers will not wait a week or

month to exploit a newly exposed vulnerability, so I can't wait that long to find and patch it either.

Looking around for tools, I quickly chose Nmap as my port scanner. It is widely considered to be the best scanner, and I had already been using it for years to troubleshoot networks and test security. Next I needed software to aggregate Nmap output and print differences between runs. I considered several existing tools, including HD Moore's [Nlog](#). Unfortunately none of these monitored changes in the way I desired. I had to know whenever a router or firewall access control list was misconfigured or a host was publicly sharing inappropriate content. I also worried about the scalability of these other solutions, so I decided to tackle the problem myself.

The first issue to come up was speed. Our networks are located worldwide, yet I was provided with only a single U.S.-based host to do the scanning. In many cases, firewalls between the sites slowed the scanning down significantly. Scanning all 100,000 hosts took over 30 hours, which is unacceptable for a daily scan. So I wrote a script called nmap-wrapper which runs dozens of Nmap processes in parallel, reducing the scan time to fifteen hours, even including OS detection.

The next problem was dealing with so much data. A SQL database seemed like the best approach for scalability and data-mining reasons, but I had to abandon that idea due to time pressures. A future version may add this support. Instead, I used a flat file to store the results of each class C address range for each day. The most powerful and extensible way to parse and store this information was the Nmap XML format, but I chose the “grepable” (-oG option) format because it is so easy to parse from simple scripts. Per-host timestamps are also stored for reporting purposes. These have proven quite helpful when administrators try to blame machine or service crashes on the scanner. They cannot credibly claim a service crash at 7:12AM when I have proof that the scan ran at 9:45AM.

The scan produces copious data, with no convenient access method. The standard Unix **diff** tool is not smart enough to report only the changes I care about, so I wrote a Perl script named nmap-diff to provide daily change reports. A typical output report is shown in [Example 1.4](#).

Example 1.4. nmap-diff typical output

```

> nmap-diff.pl -c3
  5 IPs showed changes

10.12.4.8 (ftp-box.foocompany.biz)
    21/tcp    open    ftp
    80/tcp    open    http
    443/tcp   open    https
    1027/tcp   open    IIS
+ 1029/tcp   open    ms-lsa
    38292/tcp  open    landesk-cba
OS: Microsoft Windows Millennium Edition (Me)
    Windows 2000 Professional or Advanced Server
    or Windows XP

10.16.234.3 (media.foocompany.biz)
    80/tcp    open    http
+ 554/tcp    open    rtsp
+ 7070/tcp   open    realserver

192.168.10.186 (testbox.foocompany.biz)
+ 8082/tcp   open    blackice-alerts
OS: Linux Kernel 2.4.0 - 2.5.20

172.24.12.58 (mtafoocompany.biz)
+ 25/tcp     open    smtp
OS: FreeBSD 4.3 - 4.4PRERELEASE

172.23.76.22 (media2.fooCorp.biz)
    80/tcp    open    http
    1027/tcp   open    IIS
+ 1040/tcp   open    netsaint
    1755/tcp   open    wms
    3372/tcp   open    msdtc
    6666/tcp   open    irc-serv
    7007/tcp   open    afs3-bos
OS: Microsoft Windows Millennium Edition (Me)
    Windows 2000 Professional or Advanced Server
    or Windows XP

```

Management and staff were impressed when I demonstrated this new system at an internal company security symposium. But instead of allowing me to rest on my laurels, they began asking for new features. They wanted counts of mail and web servers, growth

estimates, and more. This data was all available from the scans, but was difficult to access. So I created yet another Perl script, `nmap-report`, which made querying the data much easier. It takes specifications such as open ports or operating systems and finds all the systems that matched on a given day.

One problem with this approach to security monitoring is that employees do not always place services on their IANA-registered official ports. For example, they might put a web server on port 22 (SSH) or vice versa. Just as I was debating how to address this problem, Nmap came out with an advanced service and version detection system (see [Chapter 7, Service and Application Version Detection](#)). `nmap-report` now has a rescan feature that uses version scanning to report the true services rather than guessing based on port number. I hope to further integrate version detection in future versions. [Example 1.5](#) shows `nmap-report` listing FTP servers.

Example 1.5. `nmap-report` execution

```
> nmap-report -p21 -rV
[...]
172.21.199.76 (ftp1.foocorp.biz)
    21/tcp    open    ssl|ftp Serv-U ftpd 4.0

192.168.12.56 (ftp2.foocorp.biz)
    21/tcp    open    ftp      NcFTPd

192.168.13.130 (dropbox.foocorp.biz)
    21/tcp    open    ftp      WU-FTPD 6.00LS
```

While being far from perfect, these scripts have proven themselves quite valuable at monitoring large networks for security-impacting changes. Since Nmap itself is open source, it only seemed fair to release my scripts to the public as well. I have made them freely available at <http://www.unspecific.com/nmap>.

^[2] These IP addresses are actually registered to the United States Army Yuma Proving Ground, which is used to test a wide variety of artillery, missiles, tanks, and other deadly weapons. The moral is to be very careful about who you scan, lest you accidentally hit a

highly sensitive network. The scan results in this story are not actually from this IP range.

[3] Classless Inter-Domain Routing (CIDR) notation is a method for describing networks with more granularity than class A (CIDR /8), class B (CIDR /16), or class C (CIDR /24) notation. An excellent description is available at <http://public.pacbell.net/dedicated/cidr.html>.

[4] It is possible that the target nameserver will log a suspicious bunch of reverse-DNS queries from Felix's nameserver, but most organizations don't even keep such logs, much less analyze them.

[5] stdout is the “C” notation for representing the standard output mechanism for a system, such as to the Unix xterm or Windows command window in which Nmap was initiated.

[6] A sexy leather-clad attacker from the previous team actually started the session. It is unclear at what point she died and left the remaining tasks to Trinity.

The Phases of an Nmap Scan

Now that we've seen some applications of Nmap, let's look at what happens when an Nmap scan runs. Scans proceed in phases, with each phase finishing before the next one begins. As you can see from the phase descriptions below, there is far more to Nmap than just port scanning.

Target enumeration. In this phase, Nmap researches the host specifiers provided by the user, which may be a combination of host DNS names, IP addresses, CIDR network notations, and more. You can even use (-iR) to ask Nmap to choose your targets for you! Nmap resolves these specifiers into a list of IPv4 or IPv6 addresses for scanning. This phase cannot be skipped since it is essential for further scanning, but you can simplify the processing by passing just IP addresses so Nmap doesn't have to do forward resolution. If you pass the -sL -n options (list scan with no reverse-DNS resolution), Nmap will print out the targets and perform no further scanning. This phase is discussed in [the section called “Specifying Target Hosts and Networks”](#) and [the section called “List Scan \(-sL\)”](#).

Host discovery (ping scanning). Network scans usually begin by discovering which targets on the network are online and thus worth deeper investigation. This process is called *host discovery* or *ping scanning*. Nmap offers many host discovery techniques, ranging from quick ARP requests to elaborate combinations of TCP, ICMP, and other types of probes. This phase is run by default, though you can skip it (simply assume all target IPs are online) using the `-PN` (no ping) option. To quit after host discovery, specify `-sP -n`. Host discovery is the subject of [Chapter 3](#).

Reverse-DNS resolution. Once Nmap has determined which hosts to scan, it looks up the reverse-DNS names of all hosts found online by the ping scan. Sometimes a host's name provides clues to its function, and names make reports more readable than providing only IP numbers. This step may be skipped with the `-n` (no resolution) option, or expanded to cover all target IPs (even down ones) with `-R` (resolve all). Name resolution is covered in [the section called "DNS Resolution"](#).

Port scanning. This is Nmap's fundamental operation. Probes are sent, and the responses (or non-responses) to those probes are used to classify remote ports into states such as open, closed, or filtered. That brief description doesn't begin to encompass Nmap's many scan types, configurability of scans, and algorithms for improving speed and accuracy. An overview of port scanning is in [Chapter 4](#). Detailed information on algorithms and command-line options are in [Chapter 5](#). Port scanning is performed by default, though you can skip it and still perform some of the later traceroute and partial Nmap Scripting Engine phases by specifying their particular command-line options (such as `--traceroute` and `--script`) along with a ping scan (`-sP`).

Version detection. If some ports are found to be open, Nmap may be able to determine what server software is running on the remote system. It does this by sending a variety of probes and matching the responses against a database of thousands of known service signatures. Version detection is enabled by the `-sV` option. It is fully described in [Chapter 7](#).

OS detection. If requested with the `-O` option, Nmap proceeds to OS detection. Different operating systems implement network standards in subtly different ways. By measuring these differences it is often possible to determine the operating system running on a remote host. Nmap matches responses to a standard set of probes

against a database of more than a thousand known operating system responses. OS detection is covered in [Chapter 8](#).

Traceroute. Nmap contains an optimized traceroute implementation, enabled by the `--traceroute` option. It can find the network routes to many hosts in parallel, using the best available probe packets as determined by Nmap's previous discovery phases. Traceroute usually involves another round of reverse-DNS resolution for the intermediate hosts. More information is found in [the section called "Host Discovery"](#).

Script scanning. The Nmap Scripting Engine (NSE) uses a collection of special-purpose scripts to gain even more information about remote systems. NSE is powered by the Lua programming language and a standard library designed for network information gathering. Among the facilities offered are advanced version detection, notification of service vulnerabilities, and discovery of backdoors and other malware. NSE is a large subject, fully discussed in [Chapter 9](#). NSE is not executed unless you request it with options such as `--script` or `-sC`.

Output. Finally, Nmap collects all the information it has gathered and writes it to the screen or to a file. Nmap can write output in several formats. Its default, human-readable format (interactive format) is usually presented in this book. Nmap also offers an XML-based output format, among others. The ins and outs of output are the subject of [Chapter 13](#).

As already discussed, Nmap offers many options for controlling which of these phases are run. For scans of large networks, each phase is repeated many times since Nmap deals with the hosts in smaller groups. It scans each group completely and outputs those results, then moves on to the next batch of hosts.

Legal Issues

When used properly, Nmap helps protect your network from invaders. But when used improperly, Nmap can (in rare cases) get you sued, fired, expelled, jailed, or banned by your ISP. Reduce your risk by reading this legal guide before launching Nmap.

Is Unauthorized Port Scanning a Crime?

The legal ramifications of scanning networks with Nmap are complex and so controversial that third-party organizations have even printed T-shirts and bumper stickers promulgating opinions on the matter^[7], as shown in Figure 1.3. The topic also draws many passionate but often unproductive debates and flame wars. If you ever participate in such discussions, try to avoid the overused and ill-fitting analogies to knocking on someone's home door or testing whether his door and windows are locked.

Figure 1.3. Strong opinions on port scanning legality and morality



While I agree with the sentiment that port scanning *should not* be illegal, it is rarely wise to take legal advice from a T-shirt. Indeed, taking it from a software engineer and author is only slightly better. Speak to a competent lawyer within your jurisdiction for a better understanding of how the law applies to your particular situation.

With that important disclaimer out of the way, here is some general information that may prove helpful.

The best way to avoid controversy when using Nmap is to always secure written authorization from the target network representatives before initiating any scanning. There is still a chance that your ISP will give you trouble if they notice it (or if the target administrators accidentally send them an abuse report), but this is usually easy to resolve. When you are performing a penetration test, this authorization should be in the Statement of Work. When testing your own company, make certain that this activity clearly falls within your job description. Security consultants should be familiar with the excellent [Open Source Security Testing Methodology Manual \(OSSTMM\)](#), which provides best practices for these situations.

While civil and (especially) criminal court cases are the nightmare scenario for Nmap users, these are very rare. After all, no United States federal laws explicitly make port scanning illegal. A much more frequent occurrence is that the target network will notice a scan and send a complaint to the network service provider where the scan initiated (your ISP). Most network administrators do not seem to care or notice the many scans bouncing off their networks daily, but a few complain. The scan source ISP may track down the user corresponding to the reported IP address and time, then chide the user or even kick them off the service. Port scanning without authorization is sometimes against the provider's acceptable use policy (AUP). For example, the AUP for the huge cable-modem ISP Comcast says:

Network probing or port scanning tools are only permitted when used in conjunction with a residential home network, or if explicitly authorized by the destination host and/or network. Unauthorized port scanning, for any reason, is strictly prohibited.

Even if an ISP does not explicitly ban unauthorized port scanning, they might claim that some “anti-hacking” provision applies. Of course this does *not* make port scanning illegal. Many perfectly legal and (in the United States) constitutionally protected activities are banned by ISPs. For example, the AUP quoted above also prohibits users from transmitting, storing, or posting “any information or material which a reasonable person could deem to be objectionable, offensive, indecent, pornographic, ... embarrassing, distressing, vulgar, hateful, racially or ethnically offensive, or otherwise inappropriate, regardless of whether this material or its

dissemination is unlawful". In other words, some ISPs ban any behavior that could possibly offend or annoy someone^[8]. Indiscriminate scanning of other people's networks/computers does have that potential. If you decide to perform such controversial scanning anyway, never do it from work, school, or any other service provider that has substantial control over your well-being. Use a dialup or commercial broadband provider instead. Losing your DSL connection and having to change providers is a slight nuisance, but it is immeasurably preferable to being expelled or fired.

While legal cases involving port scanning (without follow-up hacking attacks) are rare, they do happen. One of the most notable cases involved a man named Scott Moulton who had an ongoing consulting contract to maintain the Cherokee County, Georgia emergency 911 system. In December 1999, he was tasked with setting up a router connecting the Canton, Georgia Police Department with the E911 Center. Concerned that this might jeopardize the E911 Center security, Scott initiated some preliminary port scanning of the networks involved. In the process he scanned a Cherokee County web server that was owned and maintained by a competing consulting firm named VC3. They noticed the scan and emailed Scott, who replied that he worked for the 911 Center and was testing security. VC3 then reported the activity to the police. Scott lost his E911 maintenance contract and was arrested for allegedly violating the Computer Fraud and Abuse Act of America [Section 1030\(a\)\(5\)\(B\)](#). This act applies against anyone who "intentionally accesses a protected computer without authorization, and as a result of such conduct, causes damage" (and meets other requirements). The damage claimed by VC3 involved time spent investigating the port scan and related activity. Scott sued VC3 for defamation, and VC3 countersued for violation of the Computer Fraud and Abuse Act as well as the Georgia Computer Systems Protection Act.

The civil case against Scott was dismissed before trial, implying a complete lack of merit. The ruling made many Nmap users smile:

"Court holds that plaintiff's act of conducting an unauthorized port scan and throughput test of defendant's servers does not constitute a violation of either the Georgia Computer Systems Protection Act or the Computer Fraud and Abuse Act."—Civ. Act. No. 1:00-CV-434-TWT (N.D. Ga. November 6, 2000)

This was an exciting victory in the civil case, but Scott still had the criminal charges hanging over his head. Fortunately he kept his spirits high, sending the following [note](#) to the *nmap-hackers* mailing list:

I am proud that I could be of some benefit to the computer society in defending and protecting the rights of specialists in the computer field, however it is EXTREMELY costly to support such an effort, of which I am not happy about. But I will continue to fight and prove that there is nothing illegal about port scanning especially when I was just doing my job.

Eventually, the criminal court came to the same conclusion and all charges were dropped. While Scott was vindicated in the end, he suffered six-figure legal bills and endured stressful years battling through the court system. The silver lining is that after spending so much time educating his lawyers about the technical issues involved, Scott started a [successful forensics services company](#).

While the Moulton case sets a good example (if not legal precedent), different courts or situations could still lead to worse outcomes. Remember that many states have their own computer abuse laws, some of which can arguably make even pinging a remote machine without authorization illegal^[9].

Laws in other nations obviously differ as well. For example, A 17-year-old youth was [convicted in Finland](#) of attempted computer intrusion for simply port scanning a bank. He was fined to cover the target's investigation expenses. The Moulton ruling might have differed if the VC3 machine had actually crashed and they were able to justify the \$5,000 damage figure required by the act.

At the other extreme, an Israeli judge [acquitted](#) Avi Mizrahi in early 2004 for vulnerability scanning the Mossad secret service. Judge Abraham Tennenbaum even praised Avi as follows:

In a way, Internet surfers who check the vulnerabilities of Web sites are acting in the public good. If their intentions are not malicious and they do not cause any damage, they should even be praised.

In 2007 and 2008, broad new cybercrime laws took effect in [Germany](#) and [England](#). These laws are meant to ban the distribution, use, and even possession of “hacking tools”. For example, the UK amendment to the Computer Misuse Act makes it

illegal to “supply or offer to supply, believing that it is likely to be used to commit, or to assist in the commission of [a Computer Misuse Act violation]”. These laws have already led some security tool authors to close shop or move their projects to other countries. The problem is that most security tools can be used by both ethical professionals (white-hats) to defend their networks and black-hats to attack. These dangerous laws are based on the tool author or user's intent, which is subjective and hard to divine. Nmap was designed to help secure the Internet, but I'd hate to be arrested and forced to defend my intentions to a judge and jury. These laws are unlikely to affect tools as widespread and popular as Nmap, but they have had a chilling effect on smaller tools and those which are more commonly abused by computer criminals (such as exploitation frameworks).

Regardless of the legal status of port scanning, ISP accounts will continue to be terminated if many complaints are generated. The best way to avoid ISP abuse reports or civil/criminal charges is to avoid annoying the target network administrators in the first place. Here are some practical suggestions:

- Probably at least 90% of network scanning is non-controversial. You are rarely badgered for scanning your own machine or the networks you administer. The controversy comes when scanning other networks. There are many reasons (good and bad) for doing this sort of network exploration. Perhaps you are scanning the other systems in your dorm or department to look for publicly shared files (FTP, SMB, WWW, etc.). Or maybe you are just trying to find the IP of a certain printer. You might have scanned your favorite web site to see if they are offering any other services, or because you were curious what OS they run. Perhaps you are just trying to test connectivity, or maybe you wanted to do a quick security sanity check before handing off your credit card details to that e-commerce company. You might be conducting Internet research. Or are you performing initial reconnaissance in preparation for a break-in attempt? The remote administrators rarely know your true intentions, and do sometimes get suspicious. The best approach is to get permission first. I have seen a few people with non-administrative roles land in hot water after deciding to “prove” network insecurity by launching an intrusive scan of the entire company or campus. Administrators tend to be more cooperative when asked in advance than when woken up at

3AM by an IDS alarm claiming they are under massive attack. So whenever possible, obtain written authorization before scanning a network. Adrian Lamo would probably have avoided jail if he had asked the New York Times to test their security rather than telling reporters about the flaws afterward. Unfortunately they would likely have said no. Be prepared for this answer.

- Target your scan as tightly as possible. Any machine connected to the Internet is scanned regularly enough that most administrators ignore such Internet white noise. But scanning enough networks or executing very noisy/intrusive scans increases the probability of generating complaints. So if you are only looking for web servers, specify -p80 rather than scanning all 65,536 TCP ports on each machine. If you are only trying to find available hosts, do an Nmap ping scan rather than full port scan. Do not scan a CIDR /16 (65K hosts) when a /24 netblock suffices. The random scan mode now takes an argument specifying the number of hosts, rather than running forever. So consider -iR 1000 rather than -iR 10000 if the former is sufficient. Use the default timing (or even -T polite) rather than -T insane. Avoid noisy and relatively intrusive scans such as version detection (-sV). Similarly, a SYN scan (-sS) is quieter than a connect scan (-sT) while providing the same information and often being faster.
- As noted previously, do not do anything controversial from your work or school connections. Even though your intentions may be good, you have too much to lose if someone in power (e.g. boss, dean) decides you are a malicious cracker. Do you really want to explain your actions to someone who may not even understand the terms packet or port scanner? Spend \$40 a month for a dialup, shell, or residential broadband account. Not only are the repercussions less severe if you offend someone from such an account, but target network administrators are less likely to even bother complaining to mass-market providers. Also read the relevant AUP and choose a provider accordingly. If your provider (like Comcast discussed above) bans any unauthorized port scanning and posting of "offensive" material, do not be surprised if you are kicked off for this activity. In general, the more you pay to a service provider the more accommodating they are. A T1 provider is highly unlikely to yank your connection without notice because someone reported being port scanned. A dialup or residential DSL/cable provider very well might. This can happen even when the scan was forged by someone else.

- Nmap offers many options for stealthy scans, including source-IP spoofing, decoy scanning, and the more recent idle scan technique. These are discussed in the IDS evasion chapter. But remember that there is always a trade-off. You are harder to find if you launch scans from an open WAP far from your house, with 17 decoys, while doing subsequent probes through a chain of nine open proxies. But if anyone does track you down, they will be mighty suspicious of your intentions.
- Always have a legitimate reason for performing scans. An offended administrator might write to you first (or your ISP might forward his complaint to you) expecting some sort of justification for the activity. In the Scott Moulton case discussed above, VC3 first emailed Scott to ask what was going on. If they had been satisfied with his answer, matters might have stopped there rather than escalating into civil and criminal litigation. Groups scanning large portions of the Internet for research purposes often use a reverse-DNS name that describes their project and run a web server with detailed information and opt-out forms.

Also remember that ancillary and subsequent actions are often used as evidence of intent. A port scan by itself does not always signify an attack. A port scan followed closely by an IIS exploit, however, broadcasts the intention loud and clear. This is important because decisions to prosecute (or fire, expel, complain, etc.) are often based on the whole event and not just one component (such as a port scan).

One dramatic case involved a Canadian man named Walter Nowakowski, who was apparently the first person to be charged in Canada with theft of communications ([Canadian Criminal Code Section S.342.1](#)) for accessing the Internet through someone's unsecured Wi-Fi network. Thousands of Canadian "war drivers" do this every day, so why was he singled out? Because of ancillary actions and intent. He was allegedly [caught](#) driving the wrong way on a one-way street, naked from the waist down, with laptop in hand, while downloading child pornography through the aforementioned unsecured wireless access point. The police apparently considered his activity egregious enough that they brainstormed for relevant charges and tacked on theft of communications to the many child pornography-related charges.

Similarly, charges involving port scanning are usually reserved for the most egregious cases. Even when paranoid administrators notify the police that they have been scanned, prosecution (or any further action) is exceedingly rare. The fact that a 911 emergency service was involved is likely what motivated prosecutors in the Moulton case. Your author has scanned hundreds of thousands of Internet hosts while writing this book and received no complaints.

To summarize this whole section, the question of whether port scanning is legal does not have a simple answer. I cannot unequivocally say “port scanning is never a crime”, as much as I would like to. Laws differ dramatically between jurisdictions, and cases hinge on their particular details. Even when facts are nearly identical, different judges and prosecutors do not always interpret them the same way. I can only urge caution and reiterate the suggestions above.

For testing purposes, you have permission to scan the host `scanme.nmap.org`. You may have noticed that it was used in several examples already. Note that this permission only includes scanning via Nmap and not testing exploits or denial of service attacks. To conserve bandwidth, please do not initiate more than a dozen scans against that host per day. If this free scanning target service is abused, it will be taken down and Nmap will report Failed to resolve given hostname/IP: `scanme.nmap.org`.

Can Port Scanning Crash the Target

Computer/Networks?

Nmap does not have any features designed to crash target networks. It usually tries to tread lightly. For example, Nmap detects dropped packets and slows down when they occur in order to avoid overloading the network. Nmap also does not send any corrupt packets. The IP, TCP, UDP, and ICMP headers are always appropriate, though the destination host is not necessarily expecting the packets. For these reasons, no application, host, or network component *should* ever crash based on an Nmap scan. If they do, that is a bug in the system which should be repaired by the vendor.

Reports of systems being crashed by Nmap are rare, but they do happen. Many of these systems were probably unstable in the first

place and Nmap either pushed them over the top or they crashed at the same time as an Nmap scan by pure coincidence. In other cases, poorly written applications, TCP/IP stacks, and even operating systems have been demonstrated to crash reproducibly given a certain Nmap command. These are usually older legacy devices, as newer equipment is rarely released with these problems. Smart companies use Nmap and many other common network tools to test devices prior to shipment. Those who omit such pre-release testing often find out about the problem in early beta tests when a box is first deployed on the Internet. It rarely takes long for a given IP to be scanned as part of Internet white noise. Keeping systems and devices up-to-date with the latest vendor patches and firmware should reduce the susceptibility of your machines to these problems, while also improving the security and usability of your network.

In many cases, finding that a machine crashes from a certain scan is valuable information. After all, attackers can do anything Nmap can do by using Nmap itself or their own custom scripts. Devices should not crash from being scanned and if they do, vendors should be pressured to provide a patch. In some usage scenarios, detecting fragile machines by crashing them is undesirable. In those cases you may want to perform very light scanning to reduce the risk of adverse effects. Here are a few suggestions:

- Use SYN scan (-sS) instead of connect scan (-sT). User-mode applications such as web servers can rarely even detect the former because it is all handled in kernel space (some older Linux kernels are an exception) and thus the services have no excuse to crash.
- Version scanning (-sV) risks crashing poorly written applications. Similarly, some pathetic operating systems have been reported to crash when OS fingerprinted (-O). Omit these options for particularly sensitive environments or where you do not need the results.
- Using -T2 or slower (-T1, -T0) timing modes can reduce the chances that a port scan will harm a system, though they slow your scan dramatically. Older Linux boxes had an `identd` daemon that would block services temporarily if they were accessed too frequently. This could happen in a port scan, as well as during legitimate high-load situations. Slower timing might help here. These slow timing modes should only be used as a last resort as they can slow scans by an order of magnitude or more.

- Limit the number of ports and machines scanned to the fewest that are required. Every machine scanned has a minuscule chance of crashing, and so cutting the number of machines down improves your odds. Reducing the number of ports scanned reduces the risks to end hosts as well as network devices. Many NAT/firewall devices keep a state entry for every port probe. Most of them expire old entries when the table fills up, but occasional (pathetic) implementations crash instead. Reducing the ports/hosts scanned reduces the number of state entries and thus might help those sorry devices stay up.

Nmap Copyright

While Nmap is open source, it still has a copyright license that must be respected. As free software, Nmap also carries no warranty. These issues are covered in much greater detail in [the section called "Legal Notices"](#). Companies wishing to bundle and use Nmap within proprietary software and appliances are especially encouraged to read this section so they don't inadvertently violate the Nmap license. Fortunately the Nmap Project sells commercial redistribution licenses for companies which need one.

^[7] These are from the now-defunct AmericanSushi.Com.

^[8] The Comcast AUP was improved after this was first published. The latest version is available at <http://www.comcast.net/terms/use/>

^[9] An excellent paper on this topic by lawyer Ethan Preston is available at <http://grove.ufl.edu/~techlaw/vol6/issue1/preston.html>. He has also written an excellent paper relating to the legal risks of publishing security information and exploits at <http://www.mcandl.com/computer-security.html>.

The History and Future of Nmap

Many ancient and well loved security tools, such as Netcat, tcpdump, and John the Ripper, haven't changed much over the years. Others, including Nessus, Wireshark, Cain and Abel, and Snort

have been under constant development since the day they were released. Nmap is in that second category. It was released as a simple Linux-only port scanner in 1997. Over the next 10+ years it sprouted a myriad of valuable features, including OS detection, version detection, the Nmap Scripting Engine, a Windows port, a graphical user interface, and more. This section provides a timeline of the most important events over a decade of Nmap history, followed by brief predictions on the future of Nmap. For all significant Nmap changes (thousands of them), read the [Nmap Changelog](#). Old releases of Nmap can be found at <http://nmap.org/dist/>, and ancient versions at <http://nmap.org/dist-old/>.

- **September 1, 1997** — Nmap is first released in [Phrack Magazine Issue 51, article 11](#). It doesn't have a version number because new releases aren't planned. Nmap is about 2,000 lines long, and compilation is as simple as `gcc -O6 -o nmap nmap.c -lm`.
- **September 5, 1997** — Due to popular demand, a slightly modified version of the *Phrack* code is released, calling itself version 1.25. The gzipped tarball is 28KB. Version 1.26 (48KB) is released 19 days later.
- **January 11, 1998** — Insecure.Org is registered and Nmap moves there from its previous home at the [DataHaven Project](#) ISP.
- **March 14, 1998** — Renaud Deraison writes to inform me that he is writing a security scanner, and asks if he can use some Nmap source code. Of course I say yes. Nine days later he sends me a pre-release version of Nessus, noting that it “is designed for sysadmins, not 3l33t H4ck3rZ”.
- **September 1, 1998** — Inspired by Nmap's first anniversary, I begin work on adding remote OS detection for the upcoming Nmap 2.00. On October 7 I release the first private beta version to a handful of top Nmap developers. We quietly work on this for several months.
- **December 12, 1998** — Nmap version 2.00 is publicly released, introducing Nmap OS detection for the first time. An article describing the techniques was released in [Phrack 54, Article 9](#). By this point Nmap is broken up into many files, consists of about 8,000 lines of code, is kept in a private CVS revision control system, and the tarball size is 275KB. The *nmap-hackers* mailing list is started, and later grows to more than 55,000 members.

- **April 11, 1999** — Nmap 2.11BETA1 is released. This is the first version to contain a graphical user interface as an alternative to the traditional command-line usage. The bundled Unix-only GUI named NmapFE was originally written by Zach Smith. Some people like it, but most prefer command-line execution.
- **April 28, 2000** — Nmap 2.50 is [released](#). By this point the tarball has grown to 461KB. This release includes timing modes such as -T aggressive, direct SunRPC scanning, and Window and ACK scan methods.
- **May 28, 2000** — Gerhard Rieger sends a [message](#) to the *nmap-dev* list describing a new “protocol scan” he has developed for Nmap, and he even includes a patch. This is so cool that I [release](#) Nmap 2.54BETA1 with his patch less than 12 hours later.
- **December 7, 2000** — Nmap 2.54BETA16 is [released](#) as the first official version to compile and run on Microsoft Windows. The Windows porting work was done by Ryan Permeah and Andy Lutomirski.
- **July 9, 2001** — The Nmap IP ID idle scan is introduced with Nmap 2.54BETA26. A paper describing the technique is released concurrently. This extremely cool (though not always practical) scan technique is described in [the section called “TCP Idle Scan \(-sI\)”](#).
- **July 25, 2002** — I quit my job at Netscape/AOL and start my dream job working on Nmap full time.
- **July 31, 2002** — Nmap 3.00 is [released](#). The tarball is 922K. This release includes Mac OS X support, XML output, and uptime detection.
- **August 28, 2002** — Nmap is converted from C to C++ and IPv6 supported is added as part of the [Nmap 3.10ALPHA1 release](#).
- **May 15, 2003** — Nmap is featured in the movie *The Matrix Reloaded*, where Trinity uses it (followed by a real SSH exploit) to hack a power station and save the world. This leads to more publicity for Nmap than it had ever seen before or has seen since then. Details and screen shots are available at <http://nmap.org/movies.html>.
- **July 21, 2003** — I finish a first implementation of Nmap service/version detection ([Chapter 7, Service and Application Version Detection](#)) and release it to a couple dozen top Nmap developers and users as Nmap 3.40PVT1. That is followed up by 16 more private releases over the next couple months as we improve the system and add signatures.

- **September 16, 2003** — Nmap service detection is finally [released](#) publicly as part of Nmap 3.45. A detailed paper is released concurrently.
- **February 20, 2004** — Nmap 3.50 is [released](#). The tarball is now 1,571KB. SCO Corporation is banned from redistributing Nmap because they refuse to comply with the GPL. They have to rebuild their Caldera release ISOs to remove Nmap. This release includes the packet tracing and UDP ping options. It also includes the OS classification system which classifies each of the hundreds of detected operating systems by vendor name, operating system name, OS generation, and device type.
- **August 31, 2004** — The core Nmap port scanning engine is rewritten for [Nmap 3.70](#). The new engine, named `ultra_scan` features dramatically improved algorithms and parallelization support to improve both accuracy and speed. The differences are particularly dramatic for hosts behind strict firewalls.
- **June 25, 2005** — Google sponsors 10 college and graduate students to work on Nmap full time for the summer as part of Google's [Summer of Code](#) initiative. Projects include a second generation OS detection system (Zhao Lei), a new cross-platform GUI named Umit (Adriano Monteiro Marques), and many other cool projects described at <http://seclists.org/nmap-hackers/2005/0008.html>.
- **September 8, 2005** — Nmap gains raw ethernet frame sending support with the release of version [3.90](#). This allows for ARP scanning (see [the section called "ARP Scan \(-PR\)"](#)) and MAC address spoofing as well as evading the raw IP packet ban introduced by Microsoft in Windows XP SP2.
- **January 31, 2006** — Nmap 4.00 is [released](#). The tarball is now 2,388KB. This release includes runtime interaction to provide on-demand completion estimates, a Windows executable installer, NmapFE updates to support GTK2, and much more.
- **May 24, 2006** — Google sponsors 10 more Nmap summer developers as part of their SoC program. Zhao and Adriano return as part of 2006 SoC to further develop their respective projects. Diman Todorov is sponsored to help develop the Nmap Scripting Engines. These and seven other talented students and their projects are described at <http://seclists.org/nmap-hackers/2006/0009.html>.
- **June 24, 2006** — After two years of development and testing, the 2nd generation OS detection system is integrated into [Nmap 4.20ALPHA1](#). This new system is based on everything

we've learned and the new ideas we've conceived since the 1st generation system debuted 8 years earlier. After a bit of time to grow the DB, the new system proves much more accurate and granular than the old one. It is described in [Chapter 8, Remote OS Detection](#).

- **December 10, 2006** — The Nmap Scripting Engine is [released](#) as part of Nmap 4.21ALPHA1. NSE allows users to write (and share) simple scripts to automate a wide variety of networking tasks. The system is a huge success, and is described in [Chapter 9, Nmap Scripting Engine](#).
- **December 20, 2006** — Nmap's Subversion source code repository [opens to the public](#). Until this time, only a handful of developers had access to the private source repository. Everyone else had to wait for releases. Now everyone can follow Nmap development day by day. There is even an nmap-svn mailing list providing real-time change notification by email. Details are provided in [the section called "Obtaining Nmap from the Subversion \(SVN\) Repository"](#).
- **May 28, 2007** — Google sponsors six summer Nmap developers as part of their SoC program. Meanwhile, Adriano's Umit GUI for Nmap is approved as an independent program for SoC sponsorship. Among the sponsored students was David Fifield, who continued long after the summer ended and became one of Nmap's top developers. The Nmap students and their projects are listed at <http://seclists.org/nmap-hackers/2007/0003.html>.
- **June 27, 2007** — [Die Hard 4: Live Free or Die Hard](#) is released in theaters. It includes a brief scene of hacker Matthew Farrell (Justin Long) demonstrating his Nmap skills. Then he leaves his computer to join Bruce Willis in fighting a diabolical terrorist mastermind. One week later, [The Bourne Ultimatum](#) is released and also contains an Nmap scene! The CIA uses Nmap in this movie to hack a newspaper's mail server and read the email of a reporter they assassinated (nice guys)! Screen shots of Nmap movie cameos are all available on the [Nmap movies page](#).
- **July 8, 2007** — The Umit graphical front end is improved and integrated into the [Nmap 4.22SOC1 release](#) for testing. Umit is later renamed to Zenmap, and the venerable NmapFE GUI is removed. Zenmap is covered in [Chapter 12, Zenmap GUI Users' Guide](#).
- **December 13, 2007** — Nmap 4.50 is [released](#) to celebrate Nmap's 10th anniversary!

- **June 1, 2008** — Nmap 4.65 is [released](#) and includes, for the first time, an executable Mac OS X installer. The Nmap source tarball is now four megabytes. This release includes 41 NSE scripts, 1,307 OS fingerprints, and 4,706 version detection signatures.
- **August 18, 2008** — The Nmap project completes its fourth Summer of Code, with our highest success percentage ever (six out of seven sponsored students). They greatly improved Zenmap, the Nmap Scripting Engine, OS detection, and Ncat, as described at <http://seclists.org/nmap-dev/2008/q4/0193.html>.
- **September 8, 2008** — Nmap 4.75 is [released](#) with almost 100 significant improvements over 4.68. These include the Zenmap network topology and scan aggregation features (see [Chapter 12, Zenmap GUI Users' Guide](#)). It also includes port-frequency data from my Worldscan project, which I [presented](#) at Black Hat and Defcon in August.

While it is easy to catalogue the history of Nmap, the future is uncertain. Nmap didn't start off with any grand development plan, and most of the milestones in the preceding timeline were not planned more than a year in advance. Instead of trying to predict the shape of the Internet and networking way out in the future, I closely study where it is now and decide what will be most useful for Nmap now and in the near future. So I have no idea where Nmap will be 10 years from now, though I expect it to be as popular and vibrant as ever. The Nmap community is large enough that we will be able to guide Nmap wherever it needs to go. Nmap has faced curve balls before, such as the sudden removal of raw packet support in Windows XP SP2, dramatic changes in network filtering practices and technology, and the slow emergence of IPv6. Each of those required significant changes to Nmap, and we'll have to do the same to embrace or at least cope with networking changes in the future.

While the 10-year plan is up in the air, the coming year is easier to predict. As exciting as big new features are, they won't be a focus. None of us want to see Nmap get bloated and disorganized. So this will be a year of consolidation. The Zenmap and NSE systems are not as mature as the rest of Nmap, so improving these is a big priority. New NSE scripts are great because they extend Nmap's functionality without the stability risks of incorporating new source code into Nmap proper. Meanwhile, Zenmap needs usability and stability improvements, as well as better results visualization.

Another focus is the Nmap web site, which will become more useful and dynamic. A web discussion system, Nmap demo site, and wiki are planned.

Nmap may also grow in its ability to handle web scanning. When Nmap was first developed, different services were often provided as separate daemons identified by the port number they listen on. Now, many new services simply run over HTTP and are identified by a URL path name rather than port number. Scanning for known URL paths is similar in many ways to port scanning (and to the SunRPC scanning which Nmap has also done for many years). Nmap already does some web scanning using the Nmap Scripting Engine (see [Chapter 9, Nmap Scripting Engine](#)), but it would be faster and more efficient if basic support was built into Nmap itself.

Some of the coolest Nmap features in the past, such as OS detection and version scanning, were developed in secret and given a surprise release. You can expect more of these in coming years because they are so much fun!

Chapter 2. Obtaining, Compiling, Installing, and Removing Nmap

Table of Contents

[Introduction](#)

- [Testing Whether Nmap is Already Installed](#)
- [Command-line and Graphical Interfaces](#)
- [Downloading Nmap](#)
- [Verifying the Integrity of Nmap Downloads](#)
- [Obtaining Nmap from the Subversion \(SVN\) Repository](#)

[Unix Compilation and Installation from Source Code](#)

- [Configure Directives](#)
- [If You Encounter Compilation Problems](#)

[Linux Distributions](#)

- [RPM-based Distributions \(Red Hat, Mandrake, SUSE, Fedora\)](#)
- [Updating Red Hat, Fedora, Mandrake, and Yellow Dog Linux with Yum](#)
- [Debian Linux and Derivatives such as Ubuntu](#)
- [Other Linux Distributions](#)

[Windows](#)

- [Windows 2000 Dependencies](#)
- [Windows Self-installer](#)

- [Command-line Zip Binaries](#)
- [Installing the Nmap zip binaries](#)
- [Compile from Source Code](#)
- [Executing Nmap on Windows](#)
- [Sun Solaris](#)
- [Apple Mac OS X](#)
 - [Executable Installer](#)
 - [Compile from Source Code](#)
 - [Compile Nmap from source code](#)
 - [Compile Zenmap from source code](#)
 - [Third-party Packages](#)
 - [Executing Nmap on Mac OS X](#)
- [FreeBSD / OpenBSD / NetBSD](#)
 - [OpenBSD Binary Packages and Source Ports Instructions](#)
 - [FreeBSD Binary Package and Source Ports Instructions](#)
 - [Installation of the binary package](#)
 - [Installation using the source ports tree](#)
 - [NetBSD Binary Package Instructions](#)
- [Amiga, HP-UX, IRIX, and Other Platforms](#)
- [Removing Nmap](#)

Introduction

Nmap can often be installed or upgraded with a single command, so don't let the length of this chapter scare you. Most readers will use the table of contents to skip directly to sections that concern them. This chapter describes how to install Nmap on many platforms, including both source code compilation and binary installation methods. Graphical and command-line versions of Nmap are described and contrasted. Nmap removal instructions are also provided in case you change your mind.

Testing Whether Nmap is Already Installed

The first step toward obtaining Nmap is to check whether you already have it. Many free operating system distributions (including most Linux and BSD systems) come with Nmap packages, although they may not be installed by default. On Unix systems, open a terminal window and try executing the command **nmap --version**. If Nmap exists and is in your PATH, you should see output similar to that in [Example 2.1](#).

Example 2.1. Checking for Nmap and determining its version number

```
felix~>nmap --version  
  
Nmap version 4.76 ( http://nmap.org )  
felix~>
```

If Nmap does *not* exist on the system (or if your PATH is incorrectly set), an error message such as `nmap: Command not found` is reported. As the example above shows, Nmap responds to the command by printing its version number (here 4.76).

Even if your system already has a copy of Nmap, you should consider upgrading to the latest version available from <http://nmap.org/download.html>. Newer versions often run faster, fix important bugs, and feature updated operating system and service version detection databases. A list of changes since the version already on your system can be found at <http://nmap.org/changelog.html>.

Command-line and Graphical Interfaces

Nmap has traditionally been a command-line tool run from a Unix shell or (more recently) Windows command prompt. This allows experts to quickly execute a command that does exactly what they want without having to maneuver through a bunch of configuration panels and scattered option fields. This also makes Nmap easier to script and enables easy sharing of useful commands among the user community.

One downside of the command-line approach is that it can be intimidating for new and infrequent users. Nmap offers more than a hundred command-line options, although many are obscure features or debugging controls that most users can ignore. Many graphical frontends have been created for those users who prefer a GUI interface. Nmap has traditionally included a simple GUI for Unix named NmapFE, but that was replaced in 2007 by Zenmap, which we have been developing since 2005. Zenmap is far more powerful and effective than NmapFE, particularly in results viewing. Zenmap's tab-based interface lets you search and sort results, and also browse them in several ways (host details, raw Nmap output,

and ports/hosts). It works on Linux, Windows, Mac OS X, and other platforms. Zenmap is covered in depth in [Chapter 12, Zenmap GUI Users' Guide](#). The rest of this book focuses on command-line Nmap invocations. Once you understand how the command-line options work and can interpret the output, using Zenmap or the other available Nmap GUIs is easy. Nmap's options work the same way whether you choose them from radio buttons and menus or type them at a command-line.

Downloading Nmap

Nmap.Org is the official source for downloading Nmap source code and binaries for Nmap and Zenmap. Source code is distributed in bzip2 and gzip compressed tar files, and binaries are available for Linux (RPM format), Windows (NSIS executable installer) and Mac OS X (.dmg disk image). Find all of this at <http://nmap.org/download.html>.

Verifying the Integrity of Nmap Downloads

It often pays to be paranoid about the integrity of files downloaded from the Internet. Popular packages such as Sendmail ([example](#)), OpenSSH ([example](#)), tcpdump, Libpcap, BitchX, Fragrouter, and many others have been infected with malicious trojans. Software distributions sites at the Free Software Foundation, Debian, and SourceForge have also been successfully compromised. This has never happened to Nmap, but one should always be careful. To verify the authenticity of an Nmap release, consult the PGP detached signatures or cryptographic hashes (including SHA1 and MD5) posted for the release in the Nmap signatures directory at <http://nmap.org/dist/sigs/?C=M&O=D>.

The most secure verification mechanism is detached PGP signatures. As the signing key is never stored on production servers, even someone who successfully compromises the web server couldn't forge and properly sign a trojan release. While numerous applications are able to verify PGP signatures, I recommend [GNU Privacy Guard \(GPG\)](#).

Nmap releases are signed with a special Nmap Project Signing Key, which can be obtained from the major key servers or http://nmap.org/data/nmap_gpgkeys.txt. My key is included in that

file too. The keys can be imported with the command **gpg --import nmap_gpgkeys.txt**. You only need to do this once, then you can verify all future Nmap releases from that machine. Before trusting the keys, verify that the fingerprints match the values shown in [Example 2.2](#).

Example 2.2. Verifying the Nmap and Fyodor PGP Key Fingerprints

```
flog~> gpg --fingerprint nmap fyodor
pub 1024D/33599B5F 2005-04-24
    Key fingerprint = BB61 D057 C0D7 DCEF E730 996C 1AF6
    EC50 3359 9B5F
uid                               Fyodor <fyodor@insecure.org>
sub 2048g/D3C2241C 2005-04-24

pub 1024D/6B9355D0 2005-04-24
    Key fingerprint = 436D 66AB 9A79 8425 FDA0 E3F8 01AF
    9F03 6B93 55D0
uid                               Nmap Project Signing Key
    (http://insecure.org/)
sub 2048g/A50A6A94 2005-04-24
```

For every Nmap package download file (e.g. nmap-4.76.tar.bz2 and nmap-4.76-win32.zip), there is a corresponding file in the sigs directory with .asc appended to the name (e.g. nmap-4.76.tar.bz2.asc). This is the detached signature file.

With the proper PGP key in your keyring and the detached signature file downloaded, verifying an Nmap release takes a single GPG command, as shown in [Example 2.3](#). That example assumes that the verified file can be found in the same directory by simply removing “.asc” from the signature filename. When that isn't the case, simply pass the target filename as the final argument to GPG. If the file has been tampered with, the results will look like [Example 2.4](#).

Example 2.3. Verifying PGP key fingerprints (Successful)

```
flog> gpg --verify nmap-4.76.tar.bz2.asc
gpg: Signature made Fri 12 Sep 2008 02:03:59 AM PDT using
DSA key ID 6B9355D0
```



```
gpg: Good signature from "Nmap Project Signing Key
(http://www.insecure.org/)"
```

Example 2.4. Detecting a bogus file

```
flog> gpg --verify nmap-4.76.tar.bz2.asc nmap-4.76-
hacked.tar.bz2
gpg: Signature made Fri 12 Sep 2008 02:03:59 AM PDT using
DSA key ID 6B9355D0
gpg: BAD signature from "Nmap Project Signing Key
(http://www.insecure.org/)"
```

While PGP signatures are the recommended validation technique, SHA2, SHA1, and MD5 (among other) hashes are made available for more casual validation. An attacker who can manipulate your Internet traffic in real time (and is extremely skilled) or who compromises Nmap.Org and replaces both the distribution file and digest file, could defeat this test. However, it can be useful to check the authoritative Nmap.Org hashes if you obtain Nmap from a third party or feel it might have been accidentally corrupted. For every Nmap package download file, there is a corresponding file in the sigs directory with .digest.txt appended to the name (e.g. nmap-4.76.tar.bz2.digest.txt). An example is shown in [Example 2.5](#). This is the detached signature file. The hashes from the digest file can be verified using common tools such as gpg, sha1sum, or md5sum, as shown in [Example 2.6, "Verifying Nmap hashes"](#).

Example 2.5. A typical Nmap release digest file

```
flog> cat sigs/nmap-4.76.tgz.digest.txt
nmap-4.76.tgz: MD5 = 54 B5 C9 E3 F4 4C 1A DD E1 7D F6
81 70 EB 7C FE
nmap-4.76.tgz: SHA1 = 4374 CF9C A882 2C28 5DE9 D00E
8F67 06D0 BCFA A403
nmap-4.76.tgz: RMD160 = AE7B 80EF 4CE6 DBAA 6E65 76F9
CA38 4A22 3B89 BD3A
nmap-4.76.tgz: SHA224 = 524D479E 717D98D0 2FB0A42B
9A4E6E52 4027C9B6 1D843F95
D419F87F
nmap-4.76.tgz: SHA256 = 0E960E05 53EB7647 0C8517A0
038092A3 969DB65C BE23C03F
D6DAEF1A CDCC9658
```

```
nmap-4.76.tgz: SHA384 = D52917FD 9EE6EE62 F5F456BF
E245675D B6EEEEBC5 0A287B27
                        3CAA4F50 B171DC23 FE7808A8
C5E3A49A 4A78ACBE A5AEED33
nmap-4.76.tgz: SHA512 = 826CD89F 7930A765 C9FE9B41
1DAFD113 2C883857 2A3A9503
                        E4C1E690 20A37FC8 37564DC3
45FF0C97 EF45ABE6 6CEA49FF
                        E262B403 A52F4ECE C23333A0
48DEDA66
```

Example 2.6. Verifying Nmap hashes

```
flog> gpg --print-md sha256 nmap-4.76.tgz
nmap-4.76.tgz: 0E960E05 53EB7647 0C8517A0 038092A3
969DB65C BE23C03F D6DAEF1A
                CDCC9658
flog> shasum nmap-4.76.tgz
4374cf9ca8822c285de9d00e8f6706d0bcfaa403  nmap-4.76.tgz
flog> md5sum nmap-4.76.tgz
54b5c9e3f44c1adde17df68170eb7cfe  nmap-4.76.tgz
```

While releases from Nmap.Org are signed as described in this section, certain Nmap add-ons, interfaces, and platform-specific binaries are developed and distributed by other parties. They have different mechanisms for establishing the authenticity of their downloads.

Obtaining Nmap from the Subversion (SVN)

Repository

In addition to regular stable and development releases, the latest Nmap source code is always available using the [Subversion \(SVN\) revision control system](#). This delivers new features and version/OS detection database updates immediately as they are developed. The downside is that SVN head revisions aren't always as stable as official releases. So SVN is most useful for Nmap developers and users who need a fix which hasn't yet been formally released.

SVN write access is strictly limited to top Nmap developers, but everyone has read access to the repository. Check out the latest code using the command **svn co --username guest --password "" svn://svn.insecure.org/nmap/**. Then you can later update your source code by typing **svn up** in your working directory. The “guest” username is required due to an svnserve authorization bug.

While most users only follow the /nmap directory in svn (which pulls in /nbase, /nsock, and /zenmap on its own), there is one other interesting directory: /nmap-exp. This directory contains *experimental* Nmap branches which Nmap developers create when they wish to try new things without destabilizing Nmap proper. When developers feel that an experimental branch is ready for wider-scale testing, they will generally email the location to the *nmap-dev* mailing list.

Once Nmap is checked out, you can build it from source code just as you would with the Nmap tarball (described later in this chapter).

If you would like real-time (or digested) notification and diffs by email when any changes are made to Nmap, sign up for the nmap-svn mailing list at <http://cgi.insecure.org/mailman/listinfo/nmap-svn>.

Unix Compilation and Installation from Source Code

While binary packages (discussed in later sections) are available for most platforms, compilation and installation from source code is the traditional and most powerful way to install Nmap. This ensures that the latest version is available and allows Nmap to adapt to the library availability and directory structure of your system. For example, Nmap uses the OpenSSL cryptography libraries for version detection when available, but most binary packages do not include this functionality. On the other hand, binary packages are generally quicker and easier to install, and allow for consistent management (installation, removal, upgrading, etc.) of all packaged software on the system.

Source installation is usually a painless process—the build system is designed to auto-detect as much as possible. Here are the steps required for a default install:

1. Download the latest version of Nmap in .tar.bz2 (bzip2 compression) or .tgz (gzip compression) format from <http://nmap.org/download.html>.
2. Decompress the downloaded tarball with a command such as:

```
bzip2 -cd nmap-<VERSION>.tar.bz2 | tar xvf -
```

With GNU tar, the simpler command **tar xvjf nmap-*<VERSION>*.tar.bz2** does the trick. If you downloaded the .tgz version, replace bzip2 with gzip in the decompression command.

3. Change into the newly created directory: **cd nmap-*<VERSION>***
4. Configure the build system: **./configure**

If the configuration succeeds, an ASCII art dragon appears to congratulate you on successful configuration and warn you to be careful, as shown in [Example 2.7](#).

Example 2.7. Successful configuration screen

```
flog~/nmap> ./configure
checking build system type... x86_64-unknown-linux-
gnu
[hundreds of lines cut]
configure: creating ./config.status
config.status: creating Makefile
config.status: creating nsock_config.h
config.status: nsock_config.h is unchanged
      ( )      /\      _      (
      \ |      ( \ ( \.(      )
_____
 \ \ \ \ ` ` ` ` ) \      ( ____
/ _ \ \
( _ ` \+ . x ( .\      \ /
\ _____-----/ (o) \_
- .-      \+ ;      ( o
\ _____
      )      \ _____
\ \ /
( _____ +- .( -' .- <. - _ vvvvvvvv vv v\
\ /
```

```
( _____ . _ . : < _ - < _ _ ( --
AAAAAAAA A _ / |
. / . / . + - . . - / + - - .
\ _____ // \ _____
( _ ' / x / x _ / (
\ _ ' \ _ /
, x / ( ' . / . /
| \ /
/ / _ / / +
/ \ \
' ( _ /
/ \

NMAP IS A POWERFUL TOOL -- USE
CAREFULLY AND RESPONSIBLY
Configuration complete. Type make (gmake on some
*BSD machines) to compile.
```

5. Build Nmap (and the Zenmap GUI if its requirements are met):
make

Note that GNU Make is required. On BSD-derived Unix systems, this is often installed as *gmake*. So if **make** returns a bunch of errors such as “Makefile, line 1: Need an operator”, try running **gmake** instead.

6. Become a privileged user for system-wide install: **su root**

This step may be skipped if you only have an unprivileged shell account on the system. In that case, you will likely need to pass the `--prefix` option to `configure` in step four as described in the next section.

7. Install Nmap, support files, docs, etc.: **make install**

Congratulations! Nmap is now installed as `/usr/local/bin/nmap`! Run it with no arguments for a quick help screen.

As you can see above, a simple source compilation and install consists of little more than running **`./configure;make;make install`** as root. However, there are a number of options available to configure that affect the way Nmap is built.

Configure Directives

Most of the Unix build options are controlled by the configure script, as used in step number four above. There are dozens of command-line parameters and environmental variables which affect the way Nmap is built. Run **./configure --help** for a huge list with brief descriptions. These are not applicable to building Nmap on Windows. Here are the options which are either specific to Nmap or particularly important:

--prefix=<dirname>

This option, which is standard to the configure scripts of most software, determines where Nmap and its components are installed. By default, the prefix is /usr/local, meaning that nmap is installed in /usr/local/bin, the man page (nmap.1) is installed in /usr/local/man/man1, and the data files (nmap-os-db, nmap-services, nmap-service-probes, etc.) are installed under /usr/local/share/nmap. If you only wish to change the path of certain components, use the options --bindir, --datadir, and/or --mandir. An example usage of --prefix would be to install Nmap in my account as an unprivileged user. I would run **./configure --prefix=~/fyodor**. Nmap creates subdirectories like ~/fyodor/man/man1 in the install stage if they do not already exist.

--without-zenmap

This option prevents the Zenmap graphical frontend from being installed. Normally the build system checks your system for requirements such as the Python scripting language and then installs Zenmap if they are all available.

--with-openssl=<dirname>

The version detection system and Nmap Scripting Engine are able to probe SSL-encrypted services using the free OpenSSL libraries. Normally the Nmap build system looks for these libraries on your system and include this capability if they are found. If they are in a location your compiler does not search for by default, but you still want them to be used, specify **--with-openssl=<dirname>**. Nmap then looks in <dirname>/libs for the OpenSSL libraries themselves

and `<dirname>/include` for the necessary header files. Specify `--without-openssl` to disable SSL entirely.

Some distributions ship with user OpenSSL libraries that allow running programs, but not the developer files needed to compile them. Without these developer packages, Nmap will not have OpenSSL support. On Debian-based systems, install the `libssl-dev` package. On Red Hat-based systems, install `libopenssl-devel`.

`--with-libpcap=<dirname>`

Nmap uses the [Libpcap library](#) for capturing raw IP packets. Nmap normally looks for an existing copy of Libpcap on your system and uses that if the version number and platform is appropriate. Otherwise Nmap includes its own recent copy of Libpcap, which has been modified for improved Linux functionality. The specific changes are described in `libpcap/NMAP_MODIFICATIONS` in the Nmap source directory. Because of these Linux-related changes, Nmap always uses its own Libpcap by default on that platform. If you wish to force Nmap to link with your own Libpcap, pass the option `--with-libpcap=<dirname>` to configure. Nmap then expects the Libpcap library to be in `<dirname>/lib/libpcap.a` and the include files to be in `<dirname>/include`. Nmap will always use the version of Libpcap included in its tarball if you specify `--with-libpcap=included`.

`--with-libpcre=<dirname>`

PCRE is a Perl-compatible regular expression library available from <http://www.pcre.org>. Nmap normally looks for a copy on your system, and then falls back to its own copy if that fails. If your PCRE library is not in your compiler's standard search path, Nmap probably will not find it. In that case you can tell Nmap where it can be found by specifying the option `--with-libpcre=<dirname>` to configure. Nmap then expects the library files to be in `<dirname>/lib` and the include files to be in `<dirname>/include`. In some cases, you may wish to use the PCRE libraries included with Nmap in preference to those already on your system. In that case, specify `--with-libpcre=included`.

`--with-libdnet=<dirname>`

Libdnet is an excellent networking library that Nmap uses for sending raw ethernet frames. The version in the Nmap tree is heavily modified (particularly the Windows code), so the default is to use that included version. If you wish to use a version already installed on your system instead, specify `--with-libdnet=<directoryname>`. Nmap then expects the library files to be in `<directoryname>/lib` and the include files to be in `<directoryname>/include`.

`--with-localdirs`

This simple option tells Nmap to look in `/usr/local/lib` and `/usr/local/include` for important library and header files. This should never be necessary, except that some people put such libraries in `/usr/local` without configuring their compiler to find them. If you are one of those people, use this option.

If You Encounter Compilation Problems

In an ideal world, software would always compile perfectly (and quickly) on every system. Unfortunately, society has not yet reached that state of nirvana. Despite all our efforts to make Nmap portable, compilation issues occasionally arise. Here are some suggestions in case the source distribution compilation fails.

Upgrade to the latest Nmap

Check <http://nmap.org/download.html> to make sure you are using the latest version of Nmap. The problem may have already been fixed.

Read the error message carefully

Scroll up in the output screen and examine the error messages given when commands fail. It is often best to find the first error message, as that often causes a cascade of further errors. Read the error message carefully, as it could indicate a system problem such as low disk space or a broken compiler. Users with programming skills may be able to resolve a wider range of problems themselves. If you make code changes to fix the problem, please send a patch (created with `diff -uw <oldfile> <newfile>`) and any details about your problem and platform to `nmap-dev` as described in [the](#)

[section called “Bugs”](#). Integrating the change into the base Nmap distribution allows many other users to benefit, and prevents you from having to make the changes with each new Nmap version.

Ask Google and other Internet resources

Try searching for the exact error message on Google or other search engines. You might also want to browse recent activity on the Nmap development (*nmap-dev*) list—archives and a search interface are available at <http://seclists.org>.

Ask *nmap-dev*

If none of your research leads to a solution, try sending a report to the Nmap development (*nmap-dev*) mailing list, as described in [the section called “Bugs”](#).

Consider binary packages

Binary packages of Nmap are available on most platforms and are usually easy to install. The downsides are that they may not be as up-to-date and you lose some of the flexibility of self-compilation. Later sections of this chapter describe how to find binary packages on many platforms, and even more are available via Internet searching. Obviously you should only install binary packages from reputable sources.

Linux Distributions

Linux is the most popular platform for running Nmap. In one user survey, 86% said that Linux was at least one of the platforms on which they run Nmap. The first release of Nmap in 1997 *only* ran on Linux.

Linux users can choose between a source code install or using binary packages provided by their distribution or Insecure.Org. The binary packages are generally quicker and easier to install, and are often slightly customized to use the distribution's standard directory paths and such. These packages also allow for consistent management in terms of upgrading, removing, or surveying software on the system. A downside is that packages created by the distributions are necessarily behind the Nmap.Org source releases.

Most Linux distributions (particularly Debian and Gentoo) keep their Nmap package relatively current, though a few are way out of date. Choosing the source install allows for more flexibility in determining how Nmap is built and optimized for your system. To build Nmap from source, see [the section called “Unix Compilation and Installation from Source Code”](#). Here are simple package instructions for the most common distributions.

RPM-based Distributions (Red Hat, Mandrake, SUSE, Fedora)

I build RPM packages for every release of Nmap and post them to the Nmap download page at <http://nmap.org/download.html>. I build two packages: The nmap package contains just the command-line executable and data files, while the zenmap package contains the optional Zenmap graphical frontend (see [Chapter 12, Zenmap GUI Users' Guide](#)). The zenmap package requires that the nmap package be installed first. One down side to installing the RPMs rather than compiling from source is that the RPMs don't support OpenSSL for version detection and Nmap Scripting Engine probing of SSL services.

Installing via RPM is quite easy—it even downloads the package for you when given the proper URLs. The following example downloads and installs Nmap 4.68, including the frontend. Of course you should use the latest version at the download site above instead. Any existing RPM-installed versions are upgraded. [Example 2.8](#) demonstrates this installation process.

Example 2.8. Installing Nmap from binary RPMs

```
# rpm -vhU http://nmap.org/dist/nmap-4.68-1.i386.rpm
Retrieving http://nmap.org/dist/nmap-4.68-1.i386.rpm
Preparing...
##### [100%]
  1:nmap
##### [100%]
# rpm -vhU http://nmap.org/dist/zenmap-4.68-1.noarch.rpm
Retrieving http://nmap.org/dist/zenmap-4.68-1.noarch.rpm
Preparing...
##### [100%]
```

```
1:zenmap
##### [100%]
```

As the filenames above imply, these binary RPMs were created for normal PCs (x86 architecture). I also distribute x86_64 binaries for 64-bit Linux users. These binaries won't work for the relatively few Linux users on other platforms such as SPARC, Alpha, or PowerPC. They also may refuse to install if your library versions are sufficiently different from what the RPMs were initially built on. One option in these cases would be to find binary RPMs prepared by your Linux vendor for your specific distribution. The original install CDs or DVD are a good place to start. Unfortunately, those may not be current or available. Another option is to install Nmap from source code as described previously, though you lose the binary package maintenance consistency benefits. A third option is to build and install your own binary RPMs from the source RPMs distributed from the download page above. [Example 2.9](#) demonstrates this technique with Nmap 4.68.

Example 2.9. Building and installing Nmap from source RPMs

```
> rpmbuild --rebuild http://nmap.org/dist/nmap-4.68-1.src.rpm
[ hundreds of lines cut ]
Wrote: /home/fyodor/rpmdir/RPMS/i386/nmap-4.68-1.i386.rpm
[ cut ]
> su
Password:
# rpm -vhU /home/fyodor/rpmdir/RPMS/i386/nmap-4.68-1.i386.rpm
Preparing...
##### [100%]
1:nmap
##### [100%]
#
```

It is not necessary to rebuild Zenmap in this fashion because the Zenmap RPM is architecture-independent ("noarch"). For that reason there are no Zenmap source RPMs.

Removing RPM packages is as easy as **rpm -e nmap zenmap**.

Updating Red Hat, Fedora, Mandrake, and Yellow Dog

Linux with Yum

The Red Hat, Fedora, Mandrake, and Yellow Dog Linux distributions have an application named Yum which manages software installation and updates from central RPM repositories. This makes software installation and updates trivial. Since distribution-specific Yum repositories are normally used, you know the software has already been tested for compatibility with your particular distribution. Most distributions do maintain Nmap in their Yum repository, but they don't always keep it up to date. This is particularly problematic if you (like most people) don't always quickly update to the latest release of your distribution. If you are running a two-year old Linux release, Yum will often give you a two-year-old version of Nmap. Even the latest version of distributions often take months to update to a new Nmap release. So for the latest version of Nmap on these systems, try the RPMs we distribute as described in the previous section. But if our RPMs aren't compatible with your system or you are in a great hurry, installing Nmap from Yum is usually as simple as executing **yum install nmap** (run **yum install nmap zenmap** if you would like the GUI too, though some distributions don't yet package Zenmap). Yum takes care of contacting a repository on the Internet, finding the appropriate package for your architecture, and then installing it along with any necessary dependencies. This is shown (edited for brevity) in [Example 2.10](#). You can later perform **yum update** to install available updates to Nmap and other packages in the repository.

Example 2.10. Installing Nmap from a system Yum repository

```
flog~#yum install nmap
Setting up Install Process
Parsing package install arguments
Resolving Dependencies
--> Running transaction check
--> Package nmap.x86_64 2:4.52-1.fc8 set to be updated
--> Finished Dependency Resolution
Dependencies Resolved

=====
=====
```

```

Package           Arch      Version
Repository        Size
=====
Installing:
  nmap             x86_64    2:4.52-1.fc8
updates           1.0 M

Transaction Summary
=====
Install          1 Package(s)
Update           0 Package(s)
Remove           0 Package(s)

Total download size: 1.0 M
Is this ok [y/N]: y
Downloading Packages:
(1/1): nmap-4.52-1.fc8.x86_64 100% |
=====| 1.0 MB      00:02
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing: nmap
##### [1/1]

Installed: nmap.x86_64 2:4.52-1.fc8
Complete!

```

Debian Linux and Derivatives such as Ubuntu

LaMont Jones does a fabulous job maintaining the Nmap .deb packages, including keeping them reasonably up-to-date. The proper upgrade/install command is **apt-get install nmap**. This works for Debian derivatives such as Ubuntu too. Information on the latest Debian “stable” Nmap package is available at <http://packages.debian.org/stable/nmap> and the development (“unstable”) Nmap and Zenmap packages are available from <http://packages.debian.org/unstable/nmap> and <http://packages.debian.org/unstable/zenmap>.

Other Linux Distributions

There are far too many Linux distributions available to list here, but even many of the obscure ones include Nmap in their package tree. If they don't, you can simply compile from source code as described in [the section called "Unix Compilation and Installation from Source Code"](#).

Windows

While Nmap was once a Unix-only tool, a Windows version was released in 2000 and has since become the second most popular Nmap platform (behind Linux). Because of this popularity and the fact that many Windows users do not have a compiler, binary executables are distributed for each major Nmap release. While it has improved dramatically, the Windows port is not quite as efficient or stable as on Unix. Here are some known limitations:

- You cannot generally scan your own machine from itself (using a loopback IP such as 127.0.0.1 or any of its registered IP addresses). This is a Windows limitation that we haven't yet worked around. If you really want to do this, use a TCP connect scan without pinging (-sT -PN) as that uses the high level socket API rather than sending raw packets.
- Nmap only supports ethernet interfaces (including most 802.11 wireless cards and many VPN clients) for raw packet scans. Unless you use the -sT -PN options, RAS connections (such as PPP dialups) and certain VPN clients are not supported. This support was dropped when Microsoft removed raw TCP/IP socket support in Windows XP SP2. Now Nmap must send lower-level ethernet frames instead.

Scans speeds on Windows are generally comparable to those on Unix, though the latter often has a slight performance edge. One exception to this is connect scan (-sT), which is often much slower on Windows because of deficiencies in the Windows networking API. This is a shame, since that is the one TCP scan that works against localhost and over all networking types (not just ethernet, like the raw packet scans). Connect scan performance can be improved substantially by applying the Registry changes in the nmap_performance.reg file included with Nmap. By default these changes are applied for you by the Nmap executable installer. This

registry file is in the `nmap-<version>` directory of the Windows binary zip file, and `nmap-<version>/mswin32` in the source tarball (where `<version>` is the version number of the specific release). These changes increase the number of ephemeral ports reserved for user applications (such as Nmap) and reduce the time delay before a closed connection can be reused. Most people simply check the box to apply these changes in the executable Nmap installer, but you can also apply them by double-clicking on `nmap_performance.reg`, or by running the command **regedit32 nmap_performance.reg**. To make the changes by hand, add these three Registry DWORD values to `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters`:

MaxUserPort

Set a large value such as 65534 (0x0000fffe). See [MS KB Q196271](#).

TCPTimedWaitDelay

Set the minimum value (0x0000001e). See [MS KB Q149532](#).

StrictTimeWaitSeqCheck

Set to 1 so TCPTimedWaitDelay is checked.



Note

I would like to thank Ryan Permech of eEye, Andy Lutomirski, and Jens Vogt for their hard work on the Nmap Windows port. For many years, Nmap was a Unix-only tool, and it would likely still be that way if not for their efforts.

Windows users have three choices for installing Nmap, all of which are available from the download page at <http://nmap.org/download.html>.

Windows 2000 Dependencies

Nmap supports Windows 2000, but a couple dependencies from Microsoft must be installed first. Those are the [Windows Installer 3.1 \(v2\)](#) and the [Security Update for Windows 2000 \(KB835732\)](#). After installing these, follow the general instructions in the following two sections to install Nmap.

Windows Self-installer

Every Nmap release includes a Windows self-installer named `nmap-<version>-setup.exe` (where `<version>` is the version number of the specific release). Most Nmap users choose this option since it is so easy. Another advantage of the self-installer is that it provides the option to install the Zenmap GUI. Simply run the installer file and let it walk you through panels for choosing an install path and installing WinPcap. The installer was created with the open-source [Nullsoft Scriptable Install System](#). After it completes, read [the section called “Executing Nmap on Windows”](#) for instructions on executing Nmap on the command-line or through Zenmap.

Command-line Zip Binaries



Note

Most users prefer installing Nmap with the self-installer discussed previously.

Every stable Nmap release comes with Windows command-line binaries and associated files in a Zip archive. No graphical interface is included, so you need to run `nmap.exe` from a DOS/command window. Or you can download and install a superior command shell such as those included with the free Cygwin system available from <http://www.cygwin.com>. Here are the step-by-step instructions for installing and executing the Nmap .zip binaries.

Installing the Nmap zip binaries

1. Download the .zip binaries from <http://nmap.org/download.html>.
2. Uncompress the zip file into the directory you want Nmap to reside in. An example would be `C:\Program Files`. A directory called `nmap-<version>` should be created, which includes the Nmap executable and data files. Microsoft Windows XP and Vista include zip extraction—just right-click on the file in Explorer. If you do not have a Zip decompression program, there is one (called `unzip`) in Cygwin described above, or you can download the open-source and free [7-Zip utility](#). Commercial alternatives are [WinZip](#) and [PKZIP](#).
3. For improved performance, apply the Nmap Registry changes discussed previously.

4. Nmap requires the free WinPcap packet capture library. We build our own WinPcap installer which is available in the zip file as winpcap-nmap-<version>.exe, where <version> is the WinPcap version rather than the Nmap version. Alternatively, you can obtain and install the latest version from <http://www.winpcap.org>. You must install version 4.0 or later.
5. Due to the way Nmap is compiled, it requires the [Microsoft Visual C++ 2008 Redistributable Package](#) of runtime components. Many systems already have this installed from other packages, but you should run vcredist_x86.exe from the zip file just in case you need it.
6. Instructions for executing your compiled Nmap are given in [the section called "Executing Nmap on Windows"](#).

Compile from Source Code

Most Windows users prefer to use the Nmap binary self-installer, but compilation from source code is an option, particularly if you plan to help with Nmap development. Compilation requires Microsoft Visual C++ 2008, which is part of their commercial Visual Studio suite. Any of the Visual Studio editions should work, including the free [Visual C++ 2008 Express SP1](#).

Compiling Nmap on Windows from Source

1. Download the latest Nmap source distribution from <http://nmap.org/download.html>. It has the name nmap-<version>.tar.bz2 or nmap-<version>.tgz. Those are the same tar file compressed using bzip2 or gzip, respectively. The bzip2-compressed version is smaller.
2. Uncompress the source code file you just downloaded. Recent releases of the free [Cygwin distribution](#) can handle both the .tar.bz2 and .tgz formats. Use the command **tar xvjf nmap-version.tar.bz2** or **tar xvzf nmap-version.tgz**, respectively. Alternatively, the common WinZip application can decompress these files.
3. Open Visual Studio and the Nmap solution file (nmap-<version>/mswin32/nmap.sln).
4. Choose "Build Solution" from the "Build Menu". Nmap should begin compiling, and end with the line "-- Done --" saying that all projects built successfully and there were zero failures.

5. The executable and data files can be found in `nmap-<version>/mswin32/Release/`. You can copy them to a preferred directory as long as they are all kept together.
6. Ensure that you have WinPcap installed. You can obtain it by installing our binary self-installer or executing `winpcap-nmap-<version>.exe` from our zip package. Alternatively, you can obtain the official installer at <http://www.winpcap.org>.
7. Instructions for executing your compiled Nmap are given in the next section.

If you wish to build an Nmap executable Windows installer or Zenmap executable, see `docs/win32-installer-zenmap-buildguide.txt` in Nmap SVN (also [available on the Nmap Web site](#)).

Many people have asked whether Nmap can be compiled with the gcc/g++ included with Cygwin or other compilers. Some users have reported success with this, but we don't maintain instructions for building Nmap under Cygwin.

Executing Nmap on Windows

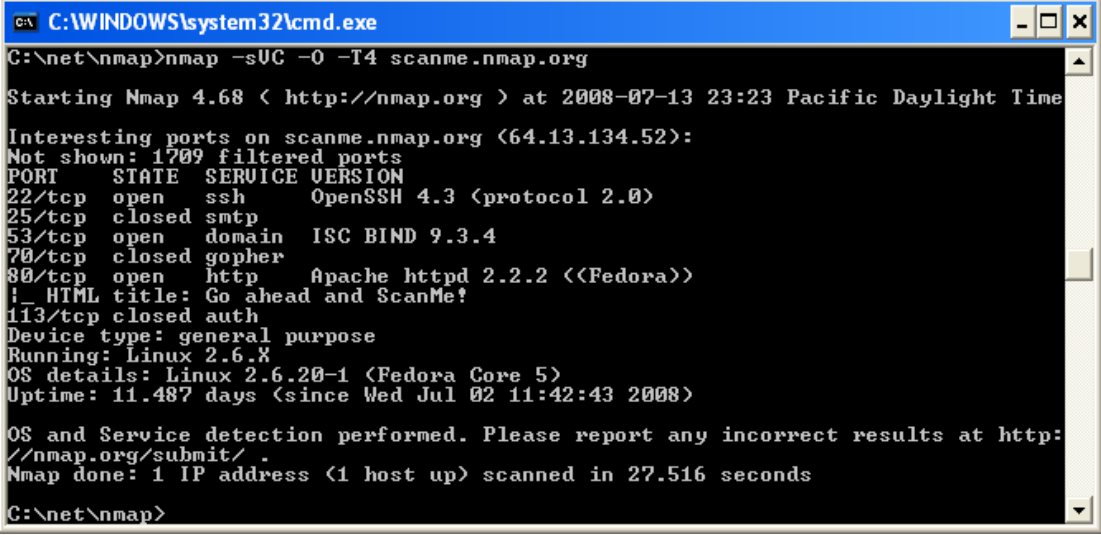
Nmap releases now include the Zenmap graphical user interface for Nmap. If you used the Nmap installer and left the Zenmap field checked, there should be a new Zenmap entry on your desktop and Start Menu. Click this to get started. Zenmap is fully documented in [Chapter 12, Zenmap GUI Users' Guide](#). While many users love Zenmap, others prefer the traditional command-line approach to executing Nmap. Here are detailed instructions for users who are unfamiliar with command-line interfaces:

1. Make sure the user you are logged in as has administrative privileges on the computer (user should be a member of the administrators group).
2. Open a command/DOS Window. Though it can be found in the program menu tree, the simplest approach is to choose "Start" -> "Run" and type **cmd<enter>**. Opening a Cygwin window (if you installed it) by clicking on the Cygwin icon on the desktop works too, although the necessary commands differ slightly from those shown here.
3. Change to the directory you installed Nmap into. Assuming you used the default path, type the following commands.

```
4. c:
5. cd "\Program Files\Nmap"
```

6. Execute **nmap.exe**. [Figure 2.1](#) is a screen shot showing a simple example.

Figure 2.1. Executing Nmap from a Windows command shell



```
C:\WINDOWS\system32\cmd.exe
C:\net\nmap>nmap -sUC -O -T4 scanme.nmap.org

Starting Nmap 4.68 < http://nmap.org > at 2008-07-13 23:23 Pacific Daylight Time
Interesting ports on scanme.nmap.org (64.13.134.52):
Not shown: 1709 filtered ports
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 4.3 <protocol 2.0>
25/tcp    closed smtp
53/tcp    open  domain   ISC BIND 9.3.4
70/tcp    closed gopher
80/tcp    open  http      Apache httpd 2.2.2 <<Fedora>>
!_ HTML title: Go ahead and ScanMe!
113/tcp   closed auth
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.20-1 <Fedora Core 5>
Uptime: 11.487 days <since Wed Jul 02 11:42:43 2008>

OS and Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address <1 host up> scanned in 27.516 seconds

C:\net\nmap>
```

If you execute Nmap frequently, you can add the Nmap directory (c:\Program Files\Nmap by default) to your command execution path. The exact place to set this varies by Windows platform. On my Windows XP box, I do the following:

1. From the desktop, right click on My Computer and then click “properties”.
2. In the System Properties window, click the “Advanced” tab.
3. Click the “Environment Variables” button.
4. Choose Path from the System variables section, then hit edit.
5. Add a semi-colon and then your Nmap directory (c:\Program Files\Nmap by default) to the end of the value.
6. Open a new DOS window and you should be able to execute a command such as **nmap scanme.nmap.org** from any directory.

Sun Solaris

Solaris has long been well-supported by Nmap. Sun even donated a complete SPARCstation to the project, which is still being used to test new Nmap builds. For this reason, many Solaris users compile and install from source code as described in [the section called “Unix Compilation and Installation from Source Code”](#).

Users who prefer native Solaris packages will be pleased to learn that Steven Christensen does an excellent job of maintaining Nmap packages at <http://www.sunfreeware.com> for all modern Solaris versions and architectures. Instructions are on his site, and are generally very simple: download the appropriate Nmap package for your version of Solaris, decompress it, and then run **pkgadd -d <packagename>**. As is generally the case with contributed binary packages, these Solaris packages are simple and quick to install. The advantages of compiling from source are that a newer version may be available and you have more flexibility in the build process.

Apple Mac OS X

Thanks to several people graciously donating shell accounts on their Mac OS X boxes, Nmap usually compiles on that platform without problems. Because not everyone has the development tools necessary to compile from source, there is an executable installer as well. Nmap is also available through systems such as MacPorts and Fink which package Unix software for Mac OS X.

Executable Installer

The easiest way to install Nmap and Zenmap on Mac OS X is to use our installer. The [Mac OS X section of the Nmap download page](#) provides a file named `nmap-<version>.dmg`, where `<version>` is the version number of the most recent release. The `.dmg` file is known as a “disk image”. Installation instructions follow:

1. Download the file `nmap-<version>.dmg`. Double-click the icon to open it. (Depending on how you downloaded the file, it may be opened automatically.)
2. The contents of the disk image will be displayed. One of the files will be a Mac meta-package file named `nmap-<version>.mpkg`. Double-click it to start the installer.
3. Follow the instructions in the installer. You will be asked for your password since Nmap installs in a system directory.
4. Once the installer is finished, eject the disk image by control-clicking on its icon and selecting “Eject”. The disk image may now be placed in the trash.

See the instructions in [the section called “Executing Nmap on Mac OS X”](#) for help on running Nmap and Zenmap after they are installed.

The programs installed by the installer are universal binaries that will run on Mac OS X 10.4 (Tiger) or later. Users of earlier versions will have to compile from source or use a third-party package.

Compile from Source Code

Compiling Nmap from source on Mac OS X is no more difficult than on other platforms once a proper build environment is in place.

Compile Nmap from source code

Compiling Nmap on Mac OS X requires [Xcode](#), Apple's developer tools that include GCC and the rest of the usual build system. Xcode is not installed by default, but is available as an optional install on the Mac OS X installation discs. If you do not have the installation discs or if you want a newer version, you can download Xcode free of charge by following these steps.

1. Apple restricts downloads of Xcode to members of the Apple Developer Connection. Browse to <http://connect.apple.com> and fill out some forms to create an account. Skip to the next step if you already have an account.
2. Return to <http://connect.apple.com> and log in with your account credentials.
3. Hit the Download link and then choose Developer Tools.
4. Download and install the most recent Xcode.

These exact steps may change, but it is hoped that this general approach will continue to work.

Once you have installed Xcode, follow the compilation instructions found in [the section called “Unix Compilation and Installation from Source Code”](#). Note that on some older versions of Mac OS X, you may have to replace the command `./configure` with `./configure CPP=/usr/bin/cpp`.

Compile Zenmap from source code

Zenmap depends on some external libraries that do not come with Mac OS X, including GTK+ and PyGTK. These libraries have many

dependencies of their own. A convenient way to install all of them is to use a third-party packaging system as described in [Section](#) . Once the dependencies are installed, follow the instructions in [the section called “Unix Compilation and Installation from Source Code”](#) to install Zenmap as usual.

Third-party Packages

Another option for installing Nmap is to use a system which packages Unix software for Mac OS X. The two discussed here are [Fink](#) and [MacPorts](#). See the respective projects' web sites for how to install the package managers.

To install using Fink, run the command **fink install nmap**. Nmap will be installed as `/sw/bin/nmap`. To uninstall use the command **fink remove nmap**.

To install using MacPorts, run **sudo port install nmap**. Nmap will be installed as `/opt/local/bin/nmap`. To uninstall, run **sudo port uninstall nmap**.

These systems install the nmap executable outside the global PATH. To enable Zenmap to find it, set the `nmap_command_path` variable in `zenmap.conf` to `/sw/bin/nmap` or `/opt/local/bin/nmap` as described in [the section called “The nmap Executable”](#).

Executing Nmap on Mac OS X

The terminal emulator in Mac OS X is called Terminal, and is located in the directory `/Applications/Utilities`. Open it and a terminal window appears. This is where you will type your commands.

By default the root user is disabled on Mac OS X. To run a scan with root privileges prefix the command name with `sudo`, as in **sudo nmap -sS <target>**. You will be asked for a password, which is just your normal login password. Only users with administrator privileges can do this.

Zenmap requires the X11 application to be installed. If it was not installed by default it may be available as an optional install on the Mac OS X installation discs.

When Zenmap is started, a dialog is displayed requesting that you type your password. Users with administrator privileges may enter their password to allow Zenmap to run as the root user and run more advanced scans. To run Zenmap in unprivileged mode, select the “Cancel” button on this authentication dialog.

FreeBSD / OpenBSD / NetBSD

The BSD flavors are well supported by Nmap, so you can simply compile it from source as described in [the section called “Unix Compilation and Installation from Source Code”](#). This provides the normal advantages of always having the latest version and a flexible build process. If you prefer binary packages, these *BSD variants each maintain their own Nmap packages. Many BSD systems also have a *ports tree* which standardizes the compilation of popular applications. Instructions for installing Nmap on the most popular *BSD variants follow.

OpenBSD Binary Packages and Source Ports

Instructions

According to the [OpenBSD FAQ](#), users “are HIGHLY advised to use packages over building an application from ports. The OpenBSD ports team considers packages to be the goal of their porting work, not the ports themselves.” That same FAQ contains detailed instructions for each method. Here is a summary:

Installation using binary packages

1. Choose a mirror from <http://www.openbsd.org/ftp.html>, then FTP in and grab the Nmap package from `/pub/OpenBSD/<version>/packages/<platform>/nmap-<version>.tgz`. Or obtain it from the OpenBSD distribution CD-ROM.
2. As root, execute: **pkg_add -v nmap-<version>.tgz**

Installation using the source ports tree

1. If you do not already have a copy of the ports tree, obtain it via CVS using instructions at <http://openbsd.org/faq/faq15.html>.

2. As root, execute the following command (replace /usr/ports with your local ports directory if it differs):

```
cd /usr/ports/net/nmap && make install clean
```

FreeBSD Binary Package and Source Ports

Instructions

The FreeBSD project has a whole [chapter](#) in their Handbook describing the package and port installation processes. A brief summary of the process follows.

Installation of the binary package

The easiest way to install the binary Nmap package is to run **pkg_add -r nmap**. You can then run the same command with the zenmap argument if you want the X-Window front-end. If you wish to obtain the package manually instead, retrieve it from <http://freshports.org/security/nmap> and <http://freshports.org/security/zenmap> or the CDROM and run **pkg_add <packagename.tgz>**.

Installation using the source ports tree

1. The ports tree is often installed with the system itself (usually in /usr/ports). If you do not already have it, specific installation instructions are provided in the FreeBSD Handbook chapter referenced above.
2. As root, execute the following command (replace /usr/ports with your local ports directory if it differs):

```
cd /usr/ports/security/nmap && make install clean
```

NetBSD Binary Package Instructions

NetBSD has packaged Nmap for an enormous number of platforms, from the normal i386 to PlayStation 2, PowerPC, VAX, SPARC, MIPS, Amiga, ARM, and several platforms that I have never even heard of! Unfortunately they are not very up-to-date. A list of NetBSD Nmap packages is available from

<ftp://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc/net/nmap/README.html> and a description of using their package system to install

applications is available at
<http://netbsd.org/Documentation/pkgsrc/using.html>.

Amiga, HP-UX, IRIX, and Other Platforms

One of the wonders of Open Source development is that resources are often directed towards what people find exciting rather than having an exclusive focus on profits as most corporations do. It is along those lines that the Amiga port came about. Diego Casorran performed most of the work and sent in a clean patch which was integrated into the main Nmap distribution. In general, AmigaOS users should be able to simply follow the source compilation instructions in [the section called “Unix Compilation and Installation from Source Code”](#). You may encounter a few hurdles on some systems, but I presume that must be part of the fun for Amiga fanatics.

Nmap supports many proprietary Unix flavors such as HP-UX and SGI IRIX. The Nmap project depends on the user community to help maintain adequate support for these systems. If you have trouble, try sending a report with full details to the *nmap-dev* mailing list, as described in [the section called “Bugs”](#). Also let us know if you develop a patch which improves support on your platform so we can incorporate it into Nmap.

Removing Nmap

If your purpose for removing Nmap is simply to upgrade to the latest version, you can usually use the upgrade option provided by most binary package managers. Similarly, installing the latest source code (as described in [the section called “Unix Compilation and Installation from Source Code”](#)) generally overwrites any previous from-source installations. Removing Nmap is a good idea if you are changing install methods (such as from source to RPM or vice versa) or if you are not using Nmap anymore and you care about the few megabytes of disk space it consumes.

How to remove Nmap depends on how you installed it initially (see previous sections). Ease of removal (and other maintenance) is a major advantage of most binary packages. For example, when Nmap is installed using the RPM system common on Linux distributions, it can be removed by running the command **rpm -e**

nmap zenmap as root. Analogous options are offered by most other package managers—consult their documentation for further information.

If you installed Nmap from the Windows installer, simply open the Control Panel, select “Add or Remove Programs” and select the “Remove” button for Nmap. You can also remove WinPcap unless you need it for other applications such as Wireshark.

If you installed Nmap from source code, removal is slightly more difficult. If you still have the build directory available (where you initially ran **make install**), you can remove Nmap by running **make uninstall**. If you no longer have that build directory, type **nmap -V** to obtain the Nmap version number. Then download that source tarball for that version of Nmap from <http://nmap.org/dist/> or <http://nmap.org/dist-old/>. Uncompress the tarball and change into the newly created directory (nmap-<version>). Run **./configure**, including any install-path options that you specified the first time (such as **--prefix** or **--datadir**). Then run **make uninstall**. Alternatively, you can simply delete all the Nmap-related files. If you used a default source install of Nmap versions 4.50 or higher, the following commands remove it.

```
# cd /usr/local
# rm -f bin/nmap bin/nmapfe bin/xnmap
# rm -f man/man1/nmap.1 man/man1/zenmap.1
# rm -rf share/nmap
# ./bin/uninstall_zenmap
```

You may have to adjust the above commands slightly if you specified **--prefix** or other install-path option when first installing Nmap. The files relating to zenmap, nmapfe, and xnmap do not exist if you did not install the Zenmap frontend.

Chapter 3. Host Discovery (“Ping Scanning”)

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

ARP Scan (-PR)

DNS Resolution

Specifying Target Hosts and Networks

List Scan (-sL)

Chapter 4. Port Scanning Overview

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

Chapter 5. Port Scanning Techniques and Algorithms

Table of Contents

[A Few Blank Sections](#)

[Idle Scan Implementation Algorithms](#)

[IP Protocol Scan \(-sO\)](#)

[Disambiguating Open from Filtered UDP Ports](#)

[Adaptive Retransmission](#)

[TCP Idle Scan \(-sI\)](#)

[Idle Scan Step by Step](#)

[Finding a Working Idle Scan Zombie Host](#)

[Executing an Idle Scan](#)

[Idle Scan Implementation Algorithms](#)

A Few Blank Sections

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

Idle Scan Implementation Algorithms

IP Protocol Scan (-sO)

Disambiguating Open from Filtered UDP Ports

Adaptive Retransmission

TCP Idle Scan (-sI)



Note

Volunteers have translated this section into [Spanish](#) and [Portuguese \(Brazil\)](#)

In 1998, security researcher Antirez (who also wrote the **hping2** tool frequently used in this book) posted to the Bugtraq mailing list an ingenious new port scanning technique. Idle scan, as it has become known, allows for completely blind port scanning. 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.

While idle scanning is more complex than any of the techniques discussed so far, you don't need to be a TCP/IP expert to understand it. It can be put together from these basic facts:

- 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 Step by Step

Fundamentally, an idle scan consists of three steps that are repeated for each port:

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.

After this 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.

Even though what happens with a closed port is slightly different from what happens with a filtered port, the attacker measures the same result in both cases, namely, an IP ID increase of 1. Therefore it is not possible for the idle scan to distinguish between closed and filtered ports. When Nmap records an IP ID increase of 1 it marks the port closed|filtered.

For those wanting more detail, the following three diagrams show exactly what happens in the three cases of an open, closed, and filtered port. The actors in each are:




 the attacker,  the zombie, and  the target.

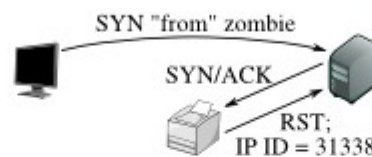
Figure 5.1. Idle scan of an open port

Step 1: Probe the zombie's IP ID.



The attacker sends a SYN/ACK to the zombie. The zombie, not expecting the SYN/ACK, sends back a RST, disclosing its IP ID.

Step 2: Forge a SYN packet from the zombie.



The target sends a SYN/ACK in response to the SYN that appears to come from the zombie. The zombie, not expecting it, sends back a RST, incrementing its IP ID in the process.

Step 3: Probe the zombie's IP ID again.



The zombie's IP ID has increased by 2 since step 1, so the port is open!

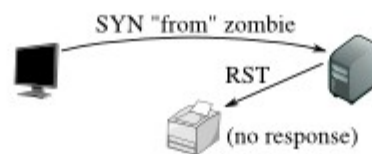
Figure 5.2. Idle scan of a closed port

Step 1: Probe the zombie's IP ID.



The attacker sends a SYN/ACK to the zombie. The zombie, not expecting the SYN/ACK, sends back a RST, disclosing its IP ID. This step is always the same.

Step 2: Forge a SYN packet from the zombie.



The target sends a RST (the port is closed) in response to the SYN that appears to come from the zombie. The zombie ignores the unsolicited RST, leaving its IP ID unchanged.

Step 3: Probe the zombie's IP ID again.



The zombie's IP ID has increased by only 1 since step 1, so the port is not open.

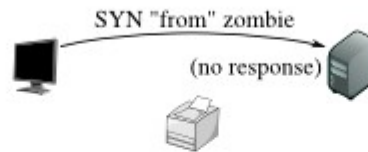
Figure 5.3. Idle scan of a filtered port

Step 1: Probe the zombie's IP ID.



Just as in the other two cases, the attacker sends a SYN/ACK to the zombie. The zombie discloses its IP ID.

Step 2: Forge a SYN packet from the zombie.



The target, obstinately filtering its port, ignores the SYN that appears to come from the zombie. The zombie, unaware that anything has happened, does not increment its IP ID.

Step 3: Probe the zombie's IP ID again.



The zombie's IP ID has incremented by only 1 since step 1, so the port is not open. From the attacker's point of view this filtered port is indistinguishable from a closed port.

Idle scan is the ultimate stealth scan. Nmap offers decoy scanning (-D) to help users shield their identity, but that (unlike idle scan) still requires an attacker to send some packets to the target from his real IP address in order to get scan results back. One upshot of idle scan is that intrusion detection systems will generally send alerts claiming that the zombie machine has launched a scan against them. So it can be used to frame some other party for a scan. Keep this possibility in mind when reading alerts from your IDS.

A unique advantage of idle scan is that it can be used to defeat certain packet filtering firewalls and routers. IP source address filtering is a common (though weak) security mechanism for limiting machines that may connect to a sensitive host or network. For example, a company database server might only allow connections from the public web server that accesses it. Or a home user might only allow SSH (interactive login) connections from his work machines.

A more disturbing scenario occurs when some company bigwig demands that network administrators open a firewall hole so he can access internal network resources from his home IP address. This can happen when executives are unwilling or unable to use secure VPN alternatives.

Idle scanning can sometimes be used to map out these trust relationships. The key factor is that idle scan results list open ports from the zombie host's perspective. A normal scan against the aforementioned database server might show no ports open, but performing an idle scan while using the web server's IP as the

zombie could expose the trust relationship by showing the database-related service ports as open.

Mapping out these trust relationships can be very useful to attackers for prioritizing targets. The web server discussed above may seem mundane to an attacker until she notices its special database access.

A disadvantage to idle scanning is that it takes far longer than most other scan types. Despite the optimized algorithms described in [the section called “Idle Scan Implementation Algorithms”](#), A 15-second SYN scan could take 15 minutes or more as an idle scan. Another issue is that you must be able to spoof packets as if they are coming from the zombie and have them reach the target machine. Many ISPs (particularly dialup and residential broadband providers) now implement egress filtering to prevent this sort of packet spoofing. Higher end providers (such as colocation and T1 services) are much less likely to do this. If this filtering is in effect, Nmap will print a quick error message for every zombie you try. If changing ISPs is not an option, you might try using another IP on the same ISP network. Sometimes the filtering only blocks spoofing of IP addresses that are *outside* the range used by customers. Another challenge with idle scan is that you must find a working zombie host, as described in the next section.

Finding a Working Idle Scan Zombie Host

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. The lower the latency between the attacker and the zombie, and between the zombie and the target, the faster the scan will proceed.

When an idle scan is attempted, Nmap tests the proposed zombie and reports any problems with it. If one doesn't work, try another. Enough Internet hosts are vulnerable that zombie candidates aren't hard to find. Since the hosts need to be idle, choosing a well-known host such as `www.yahoo.com` or `google.com` will almost never work.

A common approach is to simply execute a Nmap ping scan of some network. You could use Nmap's random IP selection mode (`-iR`), but

that is likely to result in far away zombies with substantial latency. Choosing a network near your source address, or near the target, produces better results. You can try an idle scan using each available host from the ping scan results until you find one that works. As usual, it is best to ask permission before using someone's machines for unexpected purposes such as idle scanning.

We didn't just choose a printer icon to represent a zombie in our illustrations to be funny—simple network devices often make great zombies because they are commonly both underused (idle) and built with simple network stacks which are vulnerable to IP ID traffic detection.

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. That is still no guarantee that it will work, as Solaris and some other systems create a new IP ID sequence for each host they communicate with. The host could also be too busy. OS detection and the open port list can also help in identifying systems that are likely to be idle.

While identifying a suitable zombie takes some initial work, you can keep re-using the good ones.

Executing an Idle Scan

Once a suitable zombie has been found, performing a scan is easy. Simply specify the zombie hostname to the -sI option and Nmap does the rest. [Example 5.1](#) shows an example of Ereet scanning the Recording Industry Association of America by bouncing an idle scan off an Adobe machine named Kiosk.

Example 5.1. An idle scan against the RIAA

```
# nmap -PN -p- -sI kiosk.adobe.com www.riaa.com

Starting Nmap ( http://nmap.org )
Idlescan using zombie kiosk.adobe.com
(192.150.13.111:80); Class: Incremental
```

```
Interesting ports on 208.225.90.120:
(The 65522 ports scanned but not shown below are in
state: closed)
Port      State      Service
21/tcp    open      ftp
25/tcp    open      smtp
80/tcp    open      http
111/tcp   open      sunrpc
135/tcp   open      loc-srv
443/tcp   open      https
1027/tcp  open      IIS
1030/tcp  open      iad1
2306/tcp  open      unknown
5631/tcp  open      pcanywheredata
7937/tcp  open      unknown
7938/tcp  open      unknown
36890/tcp open      unknown

Nmap done: 1 IP address (1 host up) scanned in 2594.47
seconds
```

From the scan above, we learn that the RIAA is not very security conscious (note the open PC Anywhere, portmapper, and Legato nsrexec ports). Since they apparently have no firewall, it is unlikely that they have an IDS. But if they do, it will show kiosk.adobe.com as the scan culprit. The -PN option prevents Nmap from sending an initial ping packet to the RIAA machine. That would have disclosed Ereet's true address. The scan took a long time because -p- was specified to scan all 65K ports. Don't try to use kiosk for your scans, as it has already been removed.

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. -sl kiosk.adobe.com: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.

Idle Scan Implementation Algorithms

While [the section called "Idle Scan Step by Step"](#) describes idle scan at the fundamental level, the Nmap implementation is far more

complex. Key differences are parallelism for quick execution and redundancy to reduce false positives.

Parallelizing idle scan is trickier than with other scan techniques due to indirect method of deducing port states. If Nmap sends probes to many ports on the target and then checks the new IP ID value of the zombie, the number of IP ID increments will expose how many target ports are open, but not which ones. This isn't actually a major problem, as the vast majority of ports in a large scan will be closed|filtered. Since only open ports cause the IP ID value to increment, Nmap will see no intervening increments and can mark the whole group of ports as closed|filtered. Nmap can scan groups of up to 100 ports in parallel. If Nmap probes a group then finds that the zombie IP ID has increased $<N>$ times, there must be $<N>$ open ports among that group. Nmap then finds the open ports with a binary search. It splits the group into two and separately sends probes to each. If a subgroup shows zero open ports, that group's ports are all marked closed|filtered. If a subgroup shows one or more open ports, it is divided again and the process continues until those ports are identified. While this technique adds complexity, it can reduce scan times by an order of magnitude over scanning just one port at a time.

Reliability is another major idle scanning concern. If the zombie host sends packets to any unrelated machines during the scan, its IP ID increments. This causes Nmap to think it has found an open port. Fortunately, parallel scanning helps here too. If Nmap scans 100 ports in a group and the IP ID increase signals two open ports, Nmap splits the group into two fifty-port subgroups. When Nmap does an IP ID scan on both subgroups, the total zombie IP ID increase better be two again! Otherwise, Nmap will detect the inconsistency and rescan the groups. It also modifies group size and scan timing based on the detected reliability rate of the zombie. If Nmap detects too many inconsistent results, it will quit and ask the user to provide a better zombie.

Sometimes a packet trace is the best way to understand complex algorithms and techniques such as these. Once again, the Nmap `--packet-trace` makes these trivial to produce when desired. The remainder of this section provides an annotated packet trace of an actual seven port idle scan. The IP addresses have been changed to Attacker, Zombie, and Target and some irrelevant aspects of the trace lines (such as TCP window size) have been removed for clarity.

```
Attacker# nmap -sI Zombie -PN -p20-25,110 -r --packet-  
trace -v Target  
Starting Nmap ( http://nmap.org )
```

-PN is necessary for stealth, otherwise ping packets would be sent to the target from Attacker's real address. Version scanning would also expose the true address, and so -sV is *not* specified. The -r option (turns off port randomization) is only used to make this example easier to follow.

Nmap firsts tests Zombie's IP ID sequence generation by sending six SYN/ACK packets to it and analyzing the responses. This helps Nmap immediately weed out bad zombies. It is also necessary because some systems (usually Microsoft Windows machines, though not all Windows boxes do this) increment the IP ID by 256 for each packet sent rather than by one. This happens on little-endian machines when they don't convert the IP ID to network byte order (big-endian). Nmap uses these initial probes to detect and work around this problem.

```
SENT (0.0060s) TCP Attacker:51824 > Zombie:80 SA id=35996  
SENT (0.0900s) TCP Attacker:51825 > Zombie:80 SA id=25914  
SENT (0.1800s) TCP Attacker:51826 > Zombie:80 SA id=39591  
RCVD (0.1550s) TCP Zombie:80 > Attacker:51824 R id=15669  
SENT (0.2700s) TCP Attacker:51827 > Zombie:80 SA id=43604  
RCVD (0.2380s) TCP Zombie:80 > Attacker:51825 R id=15670  
SENT (0.3600s) TCP Attacker:51828 > Zombie:80 SA id=34186  
RCVD (0.3280s) TCP Zombie:80 > Attacker:51826 R id=15671  
SENT (0.4510s) TCP Attacker:51829 > Zombie:80 SA id=27949  
RCVD (0.4190s) TCP Zombie:80 > Attacker:51827 R id=15672  
RCVD (0.5090s) TCP Zombie:80 > Attacker:51828 R id=15673  
RCVD (0.5990s) TCP Zombie:80 > Attacker:51829 R id=15674  
Idlescan using zombie Zombie (Zombie:80); Class:  
Incremental
```

This test demonstrates that the zombie is working fine. Every IP ID was an increase of one over the previous one. So the system appears to be idle and vulnerable to IP ID traffic detection. These promising results are still subject to the next test, in which Nmap spoofs four packets to Zombie as if they are coming from Target. Then it probes the zombie to ensure that the IP ID increased. If it hasn't, then it is likely that either the attacker's ISP is blocking the spoofed packets or the zombie uses a separate IP ID sequence counter for each host it communicates with. Both are common

occurrences, so Nmap always performs this test. The last-known Zombie IP ID was 15674, as shown above.

```
SENT (0.5990s) TCP Target:51823 > Zombie:80 SA id=1390
SENT (0.6510s) TCP Target:51823 > Zombie:80 SA id=24025
SENT (0.7110s) TCP Target:51823 > Zombie:80 SA id=15046
SENT (0.7710s) TCP Target:51823 > Zombie:80 SA id=48658
SENT (1.0800s) TCP Attacker:51987 > Zombie:80 SA id=27659
RCVD (1.2290s) TCP Zombie:80 > Attacker:51987 R id=15679
```

The four spoofed packets coupled with the probe from Attacker caused the Zombie to increase its IP ID from 15674 to 15679. Perfect! Now the real scanning begins. Remember that 15679 is the latest Zombie IP ID.

```
Initiating Idlescan against Target
SENT (1.2290s) TCP Zombie:80 > Target:20 S id=13200
SENT (1.2290s) TCP Zombie:80 > Target:21 S id=3737
SENT (1.2290s) TCP Zombie:80 > Target:22 S id=65290
SENT (1.2290s) TCP Zombie:80 > Target:23 S id=10516
SENT (1.4610s) TCP Attacker:52050 > Zombie:80 SA id=33202
RCVD (1.6090s) TCP Zombie:80 > Attacker:52050 R id=15680
```

Nmap probes ports 20-23. Then it probes Zombie and finds that the new IP ID is 15680, only one higher than the previous value of 15679. There were no IP ID increments in between those two known packets, meaning ports 20-23 are probably closed|filtered. It is also possible that a SYN/ACK from a Target port has simply not arrived yet. In that case, Zombie has not responded with a RST and thus its IP ID has not incremented. To ensure accuracy, Nmap will try these ports again later.

```
SENT (1.8510s) TCP Attacker:51986 > Zombie:80 SA id=49278
RCVD (1.9990s) TCP Zombie:80 > Attacker:51986 R id=15681
```

Nmap probes again because four tenths of a second has gone by since the last probe it sent. The Zombie (if not truly idle) could have communicated with other hosts during this period, which would cause inaccuracies later if not detected here. Fortunately, that has not happened: the next IP ID is 15681 as expected.

```
SENT (2.0000s) TCP Zombie:80 > Target:24 S id=23928
SENT (2.0000s) TCP Zombie:80 > Target:25 S id=50425
SENT (2.0000s) TCP Zombie:80 > Target:110 S id=14207
```

```
SENT (2.2300s) TCP Attacker:52026 > Zombie:80 SA id=26941
RCVD (2.3800s) TCP Zombie:80 > Attacker:52026 R id=15684
```

Nmap probes ports 24, 25, and 110 then queries the Zombie IP ID. It has jumped from 15681 to 15684. It skipped 15682 and 15683, meaning that two of those three ports are likely open. Nmap cannot tell which two are open, and it could also be a false positive. So Nmap drills down deeper, dividing the scan into subgroups.

```
SENT (2.6210s) TCP Attacker:51867 > Zombie:80 SA id=18869
RCVD (2.7690s) TCP Zombie:80 > Attacker:51867 R id=15685
SENT (2.7690s) TCP Zombie:80 > Target:24 S id=30023
SENT (2.7690s) TCP Zombie:80 > Target:25 S id=47253
SENT (3.0000s) TCP Attacker:51979 > Zombie:80 SA id=12077
RCVD (3.1480s) TCP Zombie:80 > Attacker:51979 R id=15687
```

The first subgroup is ports 24 and 25. The IP ID jumps from 15685 to 15687, meaning that one of these two ports is most likely open. Nmap tries the divide and conquer approach again, probing each port separately.

```
SENT (3.3910s) TCP Attacker:51826 > Zombie:80 SA id=32515
RCVD (3.5390s) TCP Zombie:80 > Attacker:51826 R id=15688
SENT (3.5390s) TCP Zombie:80 > Target:24 S id=47868
SENT (3.7710s) TCP Attacker:52012 > Zombie:80 SA id=14042
RCVD (3.9190s) TCP Zombie:80 > Attacker:52012 R id=15689
```

A port 24 probe shows no jump in the IP ID. So that port is not open. From the results so far, Nmap has tentatively determined:

- Ports 20-23 are closed|filtered
- Two of the ports 24, 25, and 110 are open
- One of the ports 24 and 25 are open
- Port 24 is closed|filtered

Stare at this puzzle long enough and you'll find only one solution: ports 25 and 110 are open while the other five are closed|filtered. Using this logic, Nmap could cease scanning and print results now. It used to do so, but that produced too many false positive open ports when the Zombie wasn't truly idle. So Nmap continues scanning to verify its results:

```
SENT (4.1600s) TCP Attacker:51858 > Zombie:80 SA id=6225
RCVD (4.3080s) TCP Zombie:80 > Attacker:51858 R id=15690
```

```
SENT (4.3080s) TCP Zombie:80 > Target:25 S id=35713
SENT (4.5410s) TCP Attacker:51856 > Zombie:80 SA id=28118
RCVD (4.6890s) TCP Zombie:80 > Attacker:51856 R id=15692
Discovered open port 25/tcp on Target
SENT (4.6900s) TCP Zombie:80 > Target:110 S id=9943
SENT (4.9210s) TCP Attacker:51836 > Zombie:80 SA id=62254
RCVD (5.0690s) TCP Zombie:80 > Attacker:51836 R id=15694
Discovered open port 110/tcp on Target
```

Probes of ports 25 and 110 show that they are open, as we deduced previously.

```
SENT (5.0690s) TCP Zombie:80 > Target:20 S id=8168
SENT (5.0690s) TCP Zombie:80 > Target:21 S id=36717
SENT (5.0690s) TCP Zombie:80 > Target:22 S id=4063
SENT (5.0690s) TCP Zombie:80 > Target:23 S id=54771
SENT (5.3200s) TCP Attacker:51962 > Zombie:80 SA id=38763
RCVD (5.4690s) TCP Zombie:80 > Attacker:51962 R id=15695
SENT (5.7910s) TCP Attacker:51887 > Zombie:80 SA id=61034
RCVD (5.9390s) TCP Zombie:80 > Attacker:51887 R id=15696
```

Just to be sure, Nmap tries ports 20-23 again. A Zombie IP ID query shows no sequence jump. On the off chance that a SYN/ACK from Target to Zombie came in late, Nmap tries another IP ID query. This again shows no open ports. Nmap is now sufficiently confident with its results to print them.

```
The Idlescan took 5 seconds to scan 7 ports.
Interesting ports on Target:
PORT      STATE      SERVICE
20/tcp    closed|filtered ftp-data
21/tcp    closed|filtered ftp
22/tcp    closed|filtered ssh
23/tcp    closed|filtered telnet
24/tcp    closed|filtered priv-mail
25/tcp    open       smtp
110/tcp   open       pop3

Nmap finished: 1 IP address (1 host up) scanned in 5.949
seconds
```

For complete details on the Nmap idle scan implementation, read `idle_scan.cc` from the Nmap source code distribution.

While port scanning is a clever abuse of predictable IP ID sequences, they can be exploited for many other purposes as well. Examples are peppered throughout this book, particularly in [Chapter 10, *Detecting and Subverting Firewalls and Intrusion Detection Systems*](#).

Chapter 6. Optimizing Nmap Performance

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

Chapter 7. Service and Application Version Detection

Table of Contents

[Introduction](#)

[Usage and Examples](#)

[Technique Described](#)

[Cheats and Fallbacks](#)

[Probe Selection and Rarity](#)

[Technique Demonstrated](#)

[Post-processors](#)

[Nmap Scripting Engine Integration](#)

[RPC Grinding](#)

[SSL Post-processor Notes](#)

[nmap-service-probes File Format](#)

[Exclude Directive](#)

[Probe Directive](#)

[match Directive](#)

[softmatch Directive](#)

[ports and sslports Directives](#)

[totalwaitms Directive](#)

[rarity Directive](#)

[fallback Directive](#)

[Putting It All Together](#)

[Community Contributions](#)

[Submit Service Fingerprints](#)

[Submit Database Corrections](#)

[Submit New Probes](#)

[SOLUTION: Hack Version Detection to Suit Custom Needs, such as Open Proxy Detection](#)

[SOLUTION: Find All Servers Running an Insecure or Nonstandard Application Version](#)

Introduction

While Nmap does many things, its most fundamental feature is port scanning. Point Nmap at a remote machine, and it might tell you that ports 25/tcp, 80/tcp, and 53/udp are open. Using its nmap-services database of more than 2,200 well-known services, Nmap would report that those ports probably correspond to a mail server (SMTP), web server (HTTP), and name server (DNS) respectively. This lookup is usually accurate—the vast majority of daemons listening on TCP port 25 are, in fact, mail servers. However, you should not bet your security on this! People can and do run services on strange ports. Perhaps their main web server was already on port 80, so they picked a different port for a staging or test server. Maybe they think hiding a vulnerable service on some obscure port prevents “evil hackers” from finding it. Even more common lately is that people choose ports based not on the service they want to run, but on what gets through the firewall. When ISPs blocked port 80 after major Microsoft IIS worms CodeRed and Nimda, hordes of users responded by moving their personal web servers to another port. When companies block Telnet access due to its horrific security risks, I have seen users simply run telnetd on the Secure Shell (SSH) port instead.

Even if Nmap is right, and the hypothetical server above is running SMTP, HTTP, and DNS servers, that is not a lot of information. When doing vulnerability assessments (or even simple network inventories) of your companies or clients, you really want to know which mail and DNS servers and versions are running. Having an accurate version number helps dramatically in determining which exploits a server is vulnerable to. Do keep in mind that security fixes are often back-ported to earlier versions of software, so you cannot rely solely on the version number to prove a service is vulnerable. False negatives are rarer, but can happen when silly administrators spoof the version number of a vulnerable service to make it appear patched.

Another good reason for determining the service types and version numbers is that many services share the same port number. For

example, port 258/tcp is used by both the Checkpoint Firewall-1 GUI management interface and the yak Windows chat client. This makes a guess based on the nmap-services table even less accurate. Anyone who has done much scanning knows that you also often find services listening on unregistered ports—these are a complete mystery without version detection. A final problem is that filtered UDP ports often look the same to a simple port scanner as open ports. But if they respond to the service-specific probes sent by Nmap version detection, you know for sure that they are open (and often exactly what is running).

Service scans sometimes reveal information about a target beyond the service type and version number. Miscellaneous information discovered about a service is collected in the “info” field. This is displayed in the VERSION column inside parentheses following the product name and version number. This field can include SSH protocol numbers, Apache modules, and much more.

Some services also report their configured hostnames, which differ from machines' reverse DNS hostnames surprisingly often. The hostname field is reported on a Service Info line following the port table. It sounds like a minor information leak, but can have consequences. One year at the CanSecWest security conference, I was huddled up in my room with my laptop. Suddenly the tcpdump window in the corner of my screen went wild and I realized my machine was under attack. I scanned back and found an unusual high port sitting open. Upon connecting, the port spewed a bunch of binary characters, but one ASCII field in the output gave a configured domain name. The domain was for a small enough security company that I knew exactly who was responsible. I had the front desk ring his hotel room, and boy was he surprised when I asked him to stop probing my box.

Two more fields that version detection can discover are operating system and device type. These are also reported on the Service Info line. We use two techniques here. One is application exclusivity. If we identify a service as Microsoft Exchange, we know the operating system is Windows since Exchange doesn't run on anything else. The other technique is to persuade more portable applications to divulge the platform information. Many servers (especially web servers) require very little coaxing. This type of OS detection is intended to complement Nmap's OS detection system (-O) and can sometimes report differing results. Consider a Microsoft Exchange server hidden behind a port-forwarding Unix firewall.

The Nmap version scanning subsystem obtains all of this data by connecting to open ports and interrogating them for further information using probes that the specific services understand. This allows Nmap to give a detailed assessment of what is really running, rather than just what port numbers are open. [Example 7.1](#) shows the actual output.

Example 7.1. Simple usage of version detection

```
# nmap -A -T4 -F insecure.org

Starting Nmap ( http://nmap.org )
Interesting ports on insecure.org (205.217.153.53):
(The 1206 ports scanned but not shown below are in state:
filtered)
PORT      STATE  SERVICE VERSION
22/tcp    open   ssh      OpenSSH 3.1p1 (protocol 1.99)
25/tcp    open   smtp      Qmail smtpd
53/tcp    open   domain    ISC BIND 9.2.1
80/tcp    open   http      Apache httpd 2.0.39 ((Unix)
mod_perl/1.99_07-dev)
113/tcp   closed auth
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux Kernel 2.4.0 - 2.5.20

Nmap finished: 1 IP address (1 host up) scanned in 34.962
seconds
```

Nmap version detection offers the following advanced features (fully described later):

- High speed, parallel operation via non-blocking sockets and a probe/match definition grammar designed for efficient yet powerful implementation.
- Determines the application name and version number where available—not just the service protocol.
- Supports both the TCP and UDP protocols, as well as both textual ASCII and packed binary services.
- Multi-platform support, including Linux, Windows, Mac OS X, FreeBSD/NetBSD/OpenBSD, Solaris, and all the other platforms on which Nmap is known to work.

- If SSL is detected, Nmap connects using OpenSSL (if available) and tries to determine what service is listening behind that encryption layer. This allows it to discover services like HTTPS, POP3S, IMAPS, etc. as well as providing version details.
- If a SunRPC service is discovered, Nmap launches its brute-force RPC grinder to find the program number, name, and version number.
- IPv6 is supported, including TCP, UDP, and SSL over TCP.
- Community contributions: if Nmap gets data back from a service that it does not recognize, a *service fingerprint* is printed along with a submission URL. This system is patterned after the extremely successful Nmap OS Detection fingerprint submission process. New probes and corrections can also be submitted.
- Comprehensive database: Nmap recognizes more than one thousand service signatures, covering more than 180 unique service protocols from ACAP, AFP, and AIM to XML-RPC, Zebedee, and Zebra.

Usage and Examples

Before delving into the technical details of how version detection is implemented, here are some examples demonstrating its usage and capabilities. To enable version detection, just add `-sV` to whatever Nmap flags you normally use. Or use the `-A` option, which turns on version detection and other Advanced and Aggressive features later. It is really that simple, as shown in [Example 7.2](#).

Example 7.2. Version detection against `www.microsoft.com`

```
# nmap -A -T4 -F www.microsoft.com

Starting Nmap ( http://nmap.org )
Interesting ports on 80.67.68.30:
(The 1208 ports scanned but not shown below are in state:
closed)
PORT      STATE      SERVICE      VERSION
22/tcp    open      ssh          Akamai-I SSH (protocol 1.5)
80/tcp    open      http         AkamaiGHost (Akamai's HTTP
Acceleration service)
443/tcp   open      ssl/http     AkamaiGHost (Akamai's HTTP
Acceleration service)
Device type: general purpose
```

```
Running: Linux 2.1.X|2.2.X
OS details: Linux 2.1.19 - 2.2.25

Nmap finished: 1 IP address (1 host up) scanned in 19.223
seconds
```

This preceding scan demonstrates a couple things. First of all, it is gratifying to see www.Microsoft.Com served off one of Akamai's Linux boxes. More relevant to this chapter is that the listed service for port 443 is ssl/http. That means that service detection first discovered that the port was SSL, then it loaded up OpenSSL and performed service detection again through SSL connections to discover a web server running AkamiGHost behind the encryption. Recall that -T4 causes Nmap to go faster (more aggressive timing) and -F tells Nmap to scan only ports registered in nmap-services.

[Example 7.3](#) is a longer and more diverse example.

Example 7.3. Complex version detection

```
# nmap -A -T4 localhost

Starting Nmap ( http://nmap.org )
Interesting ports on felix (127.0.0.1):
(The 1640 ports scanned but not shown below are in state:
closed)
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          WU-FTPD wu-2.6.1-20
22/tcp    open  ssh          OpenSSH 3.1p1 (protocol 1.99)
53/tcp    open  domain       ISC BIND 9.2.1
79/tcp    open  finger       Linux fingerd
111/tcp   open  rpcbind      2 (rpc #100000)
443/tcp   open  ssl/http     Apache httpd 2.0.39 ((Unix)
mod_perl/1.99_04-dev)
515/tcp   open  printer      CUPS 1.1
631/tcp   open  ipp          CUPS 1.1
953/tcp   open  rndc?
5000/tcp  open  ssl/ftp      WU-FTPD wu-2.6.1-20
5001/tcp  open  ssl/ssh      OpenSSH 3.1p1 (protocol 1.99)
5002/tcp  open  ssl/domain   ISC BIND 9.2.1
5003/tcp  open  ssl/finger   Linux fingerd
6000/tcp  open  X11          (access denied)
8000/tcp  open  http-proxy   Junkbuster webproxy
```

```
8080/tcp open  http      Apache httpd 2.0.39 ((Unix)
mod_perl/1.99_04-dev)
8081/tcp open  http      Apache httpd 2.0.39 ((Unix)
mod_perl/1.99_04-dev)
Device type: general purpose
Running: Linux 2.4.X|2.5.X
OS details: Linux Kernel 2.4.0 - 2.5.20

Nmap finished: 1 IP address (1 host up) scanned in 42.494
seconds
```

You can see here the way RPC services are treated, with the brute-force RPC scanner being used to determine that port 111 is rpcbind version 2. You may also notice that port 515 gives the service as printer, but that version column is empty. Nmap determined the service name by probing, but was not able to determine anything else. On the other hand, port 953 gives the service as “rndc?”. The question mark tells us that Nmap was not even able to determine the service name through probing. As a fallback, rndc is mentioned because that has port 953 registered in nmap-services. Unfortunately, none of Nmap's probes elicited any sort of response from rndc. If they had, Nmap would have printed a service fingerprint and a submission URL so that it could be recognized in the next version. As it is, Nmap requires a special probe. One might even be available by the time you read this. [the section called “Community Contributions”](#) provides details on writing your own probes.

It is also worth noting that some services provide much more information than just the version number. Examples above include whether X11 permits connections, the SSH protocol number, and the Apache module versions list. Some of the Apache modules even had to be cut from the output to fit on this page.

A few early reviewers questioned the sanity of running services such as SSH and finger over SSL. This was actually just fun with [stunnel](#), in part to ensure that parallel SSL scans actually work.

Technique Described

Nmap version scanning is actually rather straightforward. It was designed to be as simple as possible while still being scalable, fast,

and accurate. The truly nitty-gritty details are best discovered by downloading and reviewing the source code, but a synopsis of the techniques used follows.

Nmap first does a port scan as per your instructions, and then passes all the open or open|filtered TCP and/or UDP ports to the service scanning module. Those ports are then interrogated in parallel, although a single port is described here for simplicity.

1. Nmap checks to see if the port is one of the ports to be excluded, as specified by the Exclude directive in nmap-service-probes. If it is, Nmap will not scan this port for reasons mentioned in [the section called “nmap-service-probes File Format”](#).
2. If the port is TCP, Nmap starts by connecting to it. If the connection succeeds and the port had been in the open|filtered state, it is changed to open. This is rare (for TCP) since people trying to be so stealthy that they use a TCP scan type which produces open|filtered ports (such as FIN scan) generally know better than to blow all of their stealth by performing version detection.
3. Once the TCP connection is made, Nmap listens for roughly five seconds. Many common services, including most FTP, SSH, SMTP, Telnet, POP3, and IMAP servers, identify themselves in an initial welcome banner. Nmap refers to this as the “NULL probe”, because Nmap just listens for responses without sending any probe data. If any data is received, Nmap compares it to hundreds of signature regular expressions in its nmap-service-probes file (described in [the section called “nmap-service-probes File Format”](#)). If the service is fully identified, we are done with that port! The regular expression includes substrings that can be used to pick version numbers out of the response. In some cases, Nmap gets a “soft match” on the service type, but no version info. In that case, Nmap continues but only sends probes that are known to recognize the soft-matched service type.
4. At this point, Nmap UDP probes start, and TCP connections end up here if the NULL probe above fails or soft-matches. Since the reality is that most ports are used by the service they are registered to in nmap-services, every probe has a list of port numbers that are considered to be most effective. For example, the probe called GetRequest that recognizes web servers (among other services) lists 80-85, 8000-8010, and

8080-8085 as probable ports. Nmap sequentially executes the probe(s) that match the port number being scanned.

Each probe includes a probe string (which can be arbitrary ASCII text or \xHH escaped binary), which is sent to the port. Responses that come back are compared to a list of regular expressions of the same type as discussed in the NULL probe description above. As with the NULL probe, these tests can either result in a full match (ends processing for the remote service), a soft match (limits future probes to those which match a certain service), or no match at all. The exact list of regular expressions that Nmap uses to test for a match depends on the probe fallback configuration. For instance, the data returned from the X11Probe is very unlikely to match any regular expressions crafted for the GetRequest probe. On the other hand, it is likely that results returned from a Probe such as RTSPRequest might match a regular expression crafted for GetRequest since the two protocols being tested for are closely related. So the RTSPRequest probe has a fallback to GetRequest matches. For a more comprehensive explanation, see [the section called “Cheats and Fallbacks”](#).

If any response during version detection is ever received from a UDP port which was in the open|filtered state, that state is changed to open. This makes version detection an excellent complement to UDP scan, which is forced to label all scanned UDP ports as open|filtered when some common firewall rules are in effect. While combining UDP scanning with version detection can take many times as long as a plain UDP scan, it is an effective and useful technique. This method is described in [the section called “Disambiguating Open from Filtered UDP Ports”](#).

5. In most cases, the NULL probe or the probable port probe(s) (there is usually only one) described above matches the service. Since the NULL probe shares its connection with the probable port probe, this allows service detection to be done with only one brief connection in most cases. With UDP only one packet is usually required. But should the NULL probe and probable port probe(s) fail, Nmap goes through all of the existing probes sequentially. In the case of TCP, Nmap must make a new connection for each probe to avoid having previous probes corrupt the results. This worst-case scenario can take a bit of time, especially since Nmap must wait about

five seconds for the results from each probe because of slow network connections and otherwise slowly responding services. Fortunately, Nmap utilizes several automatic techniques to speed up scans:

- Nmap makes most probes generic enough to match many services. For example, the GenericLines probe sends two blank lines (“\r\n\r\n”) to the service. This matches daemons of many diverse service types, including FTP, ident, POP3, UUCP, Postgres, and whois. The GetRequest probe matches even more service types. Other examples include “help\r\n” and generic RPC and MS SMB probes.
 - If a service matches a softmatch directive, Nmap only needs to try probes that can potentially match that service.
 - All probes were not created equal! Some match many more services than others. Because of this, Nmap uses the rarity metric to avoid trying probes that are extremely unlikely to match. Experienced Nmap users can force all probes to be tried regardless or limit probe attempts even further than the default by using the --version-intensity, --version-all, and --version-light options discussed in [the section called “Probe Selection and Rarity”](#).
6. One of the probes tests whether the target port is running SSL. If so (and if OpenSSL is available), Nmap connects back via SSL and restarts the service scan to determine what is listening behind the encryption. A special directive allows different probable ports for normal and SSL tunneled connections. For example, Nmap should start against port 443 (HTTPS) with an SSL probe. But after SSL is detected and enabled, Nmap should try the GetRequest probe against port 443 because that port usually has a web server listening behind SSL encryption.
 7. Another generic probe identifies RPC-based services. When these are found, the Nmap RPC grinder (discussed later) is initiated to brute force the RPC program number/name and supported version numbers. Similarly, an SMB post-processor for fingerprinting Windows services may be added eventually.
 8. If at least one of the probes elicits some sort of response, yet Nmap is unable to recognize the service, the response content is printed to the user in the form of a *fingerprint*. If users know what services are actually listening, they are encouraged to submit the fingerprint to Nmap developers for integration into

Nmap, as described in [the section called “Submit Service Fingerprints”](#).

Cheats and Fallbacks

Even though Nmap waits a generous amount of time for services to reply, sometimes an application is slow to respond to the NULL probe. This can occur for a number of reasons, including slow reverse DNS lookups performed by some services. Because of this, Nmap can sometimes match the results from a subsequent probe to a match line designed for the NULL probe.

For example, suppose we scan port 25 (SMTP) on a server to determine what is listening. As soon as we connect, that service may conduct a bunch of DNS blacklist lookups to determine whether we should be treated as spammers and denied service. Before it finishes that, Nmap gives up waiting for a NULL probe response and sends the next probe with port 25 registered, which is “HELP\r\n”. When the service finally completes its anti-spam checks, it prints a greeting banner, reads the Help probe, and responds as shown in [Example 7.4](#).

Example 7.4. NULL probe cheat example output

```
220 hcs.w.org ESMTP Sendmail 8.12.3/8.12.3/Debian-7.1;
Tue, [cut]
214-2.0.0 This is sendmail version 8.12.3
214-2.0.0 Topics:
214-2.0.0      HELO      EHLO      MAIL      RCPT      DATA
214-2.0.0      RSET      NOOP      QUIT      HELP      VRFY
214-2.0.0      EXPN      VERB      ETRN      DSN       AUTH
214-2.0.0      STARTTLS
214-2.0.0 For more info use "HELP <topic>".
214-2.0.0 To report bugs in the implementation send email
to
214-2.0.0      sendmail-bugs@sendmail.org.
214-2.0.0 For local information send email to Postmaster
at your site.
214 2.0.0 End of HELP info
```

Nmap reads this data from the socket and finds that no regular expressions from the Help probe match the data returned. This is

because Nmap normally expects to receive the ESMTP banner during the NULL probe and match it there.

Because this is a relatively common scenario, Nmap “cheats” by trying to match responses to any of the NULL Probe match lines if none of the probe-specific lines match. In this case, a null match line exists which reports that the program is Sendmail, the version is 8.12.3/8.12.3/Debian-7.1, and the hostname is hcs.w.org.

The NULL probe cheat is actually just a specific example of a more general Nmap feature: fallbacks. The fallback directive is described in detail in [the section called “nmap-service-probes File Format”](#). Essentially, any probe that is likely to encounter results that can be matched by regular expressions in other probes has a fallback directive that specifies these other probes.

For example, in some configurations of the popular Apache web server, Apache won't respond to the GetRequest (“GET / HTTP/1.0\r\n\r\n”) probe because no virtual host name has been specified. Nmap is still able to correctly identify these servers because those servers usually respond to the HTTPOptions probe. That probe has a fallback to the GetRequest regular expressions, which are sufficiently general to recognize Apache's responses to the HTTPOptions probes.

Probe Selection and Rarity

In determining what probes to use, Nmap considers their rarity. This is an indication of how likely the probe is to return useful data. If a probe has a high rarity, it is considered less common and is less likely to be tried. Nmap users can specify which probes are tried by changing the intensity level of the version scan, as described below. The precise algorithm Nmap uses when determining which probes to use follows:

1. For TCP, the NULL probe is always tried first.
2. All probes that have the port being scanned listed as a probable port (see [the section called “nmap-service-probes File Format”](#)) are tried in the order they appear in nmap-service-probes.
3. All other probes that have a rarity value less than or equal to the current intensity value of the scan are tried, also in the order they appear in nmap-service-probes.

Once a probe is found to match, the algorithm terminates and results are reported.

Because all of Nmap's probes (other than the NULL probe) have a rarity value associated with them, it is relatively easy to control how many of them are tried when performing a version scan. Simply choose an intensity level appropriate for a scan. The higher an intensity level, the more probes will be tried. So if a very comprehensive scan is desired, a high intensity level is appropriate—even though it may take longer than a scan conducted at a lower intensity level. Nmap's default intensity level is 7 but Nmap provides the following switches for different scanning needs:

`--version-intensity <intensity level between 0 and 9>`

Sets the intensity level of a version scan to the specified value. If 0 is specified, only the NULL probe (for TCP) and probes that list the port as a probable port are tried. Example:
`nmap -sV --version-intensity 3 scanme.nmap.org`

`--version-light`

Sets the intensity level to 2. Example: **`nmap -sV --version-light scanme.nmap.org`**

`--version-all`

Sets the intensity level to 9. Since all probes have a rarity level between 1 and 9, this tries all of the probes. Example:
`nmap -sV --version-all scanme.nmap.org`

Technique Demonstrated

If the English description above is not clear enough, you can see for yourself how it works by adding the `--version-trace` (and usually `-d` (debugging)) options to your Nmap command line. This shows all the connection and data read/write activity of the service scan. An annotated real-world example follows.

```
# nmap -sSV -T4 -F -d --version-trace insecure.org

Starting Nmap ( http://nmap.org )
Host insecure.org (205.217.153.53) appears to be up ...
good.
```

```
Initiating SYN Stealth Scan against insecure.org  
(205.217.153.53) at 19:53  
Initiating service scan against 4 services on 1 host at  
19:53
```

The SYN scan has found 4 open ports—now we are beginning a service scan against each of them in parallel. We start with a TCP connection for the NULL probe:

```
Starting probes against new service: 205.217.153.53:22  
(tcp)  
NSOCK (2.0750s) TCP connection requested to  
205.217.153.53:22 (IOD #1) EID 8  
Starting probes against new service: 205.217.153.53:25  
(tcp)  
NSOCK (2.0770s) TCP connection requested to  
205.217.153.53:25 (IOD #2) EID 16  
Starting probes against new service: 205.217.153.53:53  
(tcp)  
NSOCK (2.0830s) TCP connection requested to  
205.217.153.53:53 (IOD #3) EID 24  
Starting probes against new service: 205.217.153.53:80  
(tcp)  
NSOCK (2.0860s) TCP connection requested to  
205.217.153.53:80 (IOD #4) EID 32  
NSOCK (2.0870s) Callback: CONNECT SUCCESS for EID 32  
[205.217.153.53:80]  
NSOCK (2.0870s) Read request from IOD #4  
[205.217.153.53:80]  
                (timeout: 5000ms) EID 42  
NSOCK (2.0870s) Callback: CONNECT SUCCESS for EID 24  
[205.217.153.53:53]  
NSOCK (2.0870s) Read request from IOD #3  
[205.217.153.53:53]  
                (timeout: 5000ms) EID 50  
NSOCK (2.0870s) Callback: CONNECT SUCCESS for EID 16  
[205.217.153.53:25]  
NSOCK (2.0870s) Read request from IOD #2  
[205.217.153.53:25]  
                (timeout: 5000ms) EID 58  
NSOCK (2.0870s) Callback: CONNECT SUCCESS for EID 8  
[205.217.153.53:22]  
NSOCK (2.0870s) Read request from IOD #1  
[205.217.153.53:22]
```

```
(timeout: 5000ms) EID 66
```

At this point, NULL probe connections have successfully been made to all four services. It starts at 2 seconds because that is how long the ping and SYN scans took.

```
NSOCK (2.0880s) Callback: READ SUCCESS for EID 66
[205.217.153.53:22]
                        (23 bytes): SSH-1.99-
OpenSSH_3.1p1.
Service scan match: 205.217.153.53:22 is ssh.
                        Version: |OpenSSH|3.1p1|protocol
1.99|
```

SSH was nice enough to fully identify itself immediately upon connection as OpenSSH 3.1p1. One down, three to go.

```
NSOCK (2.0880s) Callback: READ SUCCESS for EID 58
[205.217.153.53:25]
                        (27 bytes): 220 core.lnxnet.net
ESMTP..
Service scan soft match: 205.217.153.53:25 is smtp
```

The mail server on port 25 also gave us a useful banner. We do not know what type of mail server it is, but starting with 220 and including the word ESMTP tells us it is a mail (SMTP) server. So Nmap softmatches smtp, meaning that only probes able to match SMTP servers are tried from now on. Note that non-printable characters are represented by dots—so the “..” after ESMTP is really the “\r\n” line termination sequence.

```
NSOCK (2.0880s) Read request from IOD #2
[205.217.153.53:25]
                        (timeout: 4996ms) EID 74
NSOCK (7.0880s) Callback: READ TIMEOUT for EID 74
[205.217.153.53:25]
NSOCK (7.0880s) Write request for 6 bytes to IOD #2 EID
83
                        [205.217.153.53:25]: HELP..
NSOCK (7.0880s) Read request from IOD #2
[205.217.153.53:25]
                        (timeout: 5000ms) EID 90
```

Nmap listens a little longer on the SMTP connection, just in case the server has more to say. The read request times out after five seconds. Nmap then finds the next probe which is registered to port 25 and has SMTP signatures. That probe simply consists of `HELP\r\n`, which Nmap writes into the connection.

```
NSOCK (7.0880s) Callback: READ TIMEOUT for EID 50
[205.217.153.53:53]
NSOCK (7.0880s) Write request for 32 bytes to IOD #3 EID
99
[205.217.153.53:53]:
.....version.bind.....
NSOCK (7.0880s) Read request from IOD #3
[205.217.153.53:53]
(timeout: 5000ms) EID 106
```

The DNS server on port 53 does not return anything at all. The first probe registered to port 53 in `nmap-service-probes` is `DNSVersionBindReq`, which queries a DNS server for its version number. This is sent onto the wire.

```
NSOCK (7.0880s) Callback: READ TIMEOUT for EID 42
[205.217.153.53:80]
NSOCK (7.0880s) Write request for 18 bytes to IOD #4 EID
115
[205.217.153.53:80]: GET / HTTP/1.0....
NSOCK (7.0880s) Read request from IOD #4
[205.217.153.53:80]
(timeout: 5000ms) EID 122
```

The port 80 NULL probe also failed to return any data. An HTTP GET request is sent, since that probe is registered to port 80.

```
NSOCK (7.0920s) Callback: READ SUCCESS for EID 122
[205.217.153.53:80] [EOF](15858 bytes)
Service scan match: insecure.org (205.217.153.53):80 is
http.
Version: |Apache httpd|2.0.39|(Unix)
mod_perl/1.99_07-dev..
```

Apache returned a huge (15KB) response, so it is not printed. That response provided detailed configuration information, which Nmap picks out of the response. There are no other probes registered for port 80. So if this had failed, Nmap would have tried the first TCP

probe in nmap-service-probes. That probe simply sends blank lines (“\r\n\r\n”). A new connection would have been made in case the GET probe confused the service.

```
NSOCK (7.0920s) Callback: READ SUCCESS for EID 106
[205.217.153.53:53]
(50 bytes): .
0.....version.bind.....9.2.1
Service scan match: insecure.org (205.217.153.53):53 is
domain.
Version: |ISC BIND|9.2.1||
```

Port 53 responded to our DNS version request. Most of the response (as with the probe) is binary, but you can clearly see the version 9.2.1 there. If this probe had failed, the next probe registered to port 53 is a DNS server status request (14 bytes: \0\x0C\x00\x10\x00\x00\x00\x00\x00). Having this backup probe helps because many more servers respond to a status request than a version number request.

```
NSOCK (7.0920s) Callback: READ SUCCESS for EID 90
[205.217.153.53:25]
(55 bytes): 214 qmail home page: http...
Service scan match: insecure.org (205.217.153.53):25 is
smtp.
Version: |qmail smtpd|||
```

Port 25 gives a very helpful response to the Help probe. Other SMTP servers such as Postfix, Courier, and Exim can often be identified by this probe as well. If the response did not match, Nmap would have given up on this service because it had already softmatched smtp and there are no more SMTP probes in nmap-service-probes.

```
The service scan took 5 seconds to scan 4 services on 1
host.
```

This service scan run went pretty well. No service required more than one connection. It took five seconds because Qmail and Apache hit the five-second NULL probe timeout before Nmap sent the first real probes. Here is the reward for these efforts:

```
Interesting ports on insecure.org (205.217.153.53):
(The 1212 ports scanned but not shown below are in state:
closed)
```



```
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 3.1p1 (protocol 1.99)
25/tcp    open  smtp      qmail smtpd
53/tcp    open  domain    ISC BIND 9.2.1
80/tcp    open  http      Apache httpd 2.0.39 ((Unix)
mod_perl/1.99_07-dev)

Nmap finished: 1 IP address (1 host up) scanned in 7.104
seconds
```

Post-processors

Nmap is usually finished working on a port once it has deduced the service and version information as demonstrated above. However, there are certain services for which Nmap performs additional work. The post-processors presently available are Nmap Scripting Engine integration, RPC grinding, and SSL tunneling. Windows SMB interrogation is under consideration.

Nmap Scripting Engine Integration

The regular-expression based approach of version detection is powerful, but it cannot recognize everything. Some services cannot be recognized by simply sending a standard probe and matching a pattern to the response. Some services require custom probe strings or a complex multi-step handshaking process. Others require more advanced processing than a regular expression to recognize a response. For example, the Skype v2 service was designed to be difficult to detect due to the risk that incumbent carriers (such as phone companies providing DSL lines) would consider them a competitor and degrade or block the service from their subscribers. The only way we could find to detect this service involved analyzing responses to two different probes. Similarly, we could recognize more SNMP services if we tried a few hundred different community names by brute force. Neither of these tasks are well suited to traditional Nmap version detection, but both are easily accomplished with the Nmap Scripting Language. For these reasons, version detection now calls NSE by default to handle some tricky services, as described in [the section called “Version Detection Using NSE”](#).

RPC Grinding

SunRPC (Sun Remote Procedure Call) is a common Unix protocol used to implement many services including NFS. Nmap ships 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.

If the portmapper (rpcbind) service (UDP or TCP port 111) is available, RPC services can be enumerated with the Unix **rpcinfo** command. [Example 7.5](#) demonstrates this against a default Solaris 9 server.

Example 7.5. Enumerating RPC services with rpcinfo

```
> rpcinfo -p ultra
  program vers  proto   port    service
  100000     4    tcp     111    rpcbind
  100000     4    udp     111    rpcbind
  100232    10    udp    32777    sadmind
  100083     1    tcp    32775    ttdbserverd
  100221     1    tcp    32777    kcms_server
  100068     5    udp    32778    cmsd
  100229     1    tcp    32779    metad
  100230     1    tcp    32781    metamhd
  100242     1    tcp    32783    rpc.metamedd
  100001     4    udp    32780    rstatd
  100002     3    udp    32782    rusersd
  100002     3    tcp    32785    rusersd
  100008     1    udp    32784    walld
  100012     1    udp    32786    sprayd
  100011     1    udp    32788    rquotad
  100024     1    udp    32790    status
  100024     1    tcp    32787    status
  100133     1    udp    32790    nsm_addrand
  100133     1    tcp    32787    nsm_addrand
```

[Dozens of lines cut for brevity]

This example shows that hosts frequently offer many RPC services, which increases the probability that one is exploitable. You should also notice 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.

Because the RPC information is so sensitive, many administrators try to obscure this information by blocking the portmapper port (111). Unfortunately, this does not close the hole. Nmap can determine all of the same 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 RPC program identification probes are done in parallel, and retransmissions are handled for UDP ports. This feature is automatically activated whenever version detection finds any RPC ports. Or it can be performed without version detection by specifying the -sR option. [Example 7.6](#) demonstrates direct RPC scanning done as part of version detection.

Example 7.6. Nmap direct RPC scan

```
# nmap -F -A -sSU ultra

Starting Nmap ( http://nmap.org )
Interesting ports on ultra.nmap.org (192.168.0.50):
(The 2171 ports scanned but not shown below are in state:
closed)
PORT      STATE SERVICE      VERSION
[A whole bunch of ports cut for brevity]
32776/tcp  open  kcms_server  1 (rpc #100221)
```

```

32776/udp open  sadmind          10 (rpc #100232)
32777/tcp open  kcms_server         1 (rpc #100221)
32777/udp open  sadmind          10 (rpc #100232)
32778/tcp open  metad             1 (rpc #100229)
32778/udp open  cmsd             2-5 (rpc #100068)
32779/tcp open  metad             1 (rpc #100229)
32779/udp open  rstatd           2-4 (rpc #100001)
32780/tcp open  metamhd          1 (rpc #100230)
32780/udp open  rstatd           2-4 (rpc #100001)
32786/tcp open  status           1 (rpc #100024)
32786/udp open  sprayd           1 (rpc #100012)
32787/tcp open  status           1 (rpc #100024)
32787/udp open  rquotad          1 (rpc #100011)
Device type: general purpose
Running: Sun Solaris 9
OS details: Sun Solaris 9

Nmap finished: 1 IP address (1 host up) scanned in
252.701 seconds

```

SSL Post-processor Notes

As discussed in the technique section, Nmap has the ability 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 is automatically executed whenever an appropriate (SSL) port is detected. This is demonstrated by [Example 7.7](#).

Example 7.7. Version scanning through SSL

```

nmap -PN -sSV -T4 -F www.amazon.com

Starting Nmap ( http://nmap.org )
Interesting ports on 207-171-184-16.amazon.com
(207.171.184.16):
(The 1214 ports scanned but not shown below are in state:
filtered)
PORT      STATE SERVICE  VERSION
80/tcp    open  http     Apache Stronghold httpd 2.4.2
(based on Apache 1.3.6)

```

```
443/tcp open  ssl/http Apache Stronghold httpd 2.4.2
(based on Apache 1.3.6)

Nmap finished: 1 IP address (1 host up) scanned in 35.038
seconds
```

Note that the version information is the same for each of the two open ports, but the service is http on port 80 and ssl/http on port 443. The common case of HTTPS on port 443 is not hard-coded—Nmap should be able to detect SSL on any port and determine the underlying protocol for any service that Nmap can detect in clear-text. If Nmap had not detected the server listening behind SSL, the service listed would be ssl/unknown. If Nmap had not been built with SSL support, the service listed would have simply been ssl. The version column would be blank in both of these cases.

The SSL support for Nmap depends on the free [OpenSSL library](#). It is not included in the Linux RPM binaries, to avoid breaking systems which lack these libraries. The Nmap source code distribution attempts to detect OpenSSL on a system and link to it when available. See [Chapter 2, Obtaining, Compiling, Installing, and Removing Nmap](#) for details on customizing the build process to include or exclude OpenSSL.

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 Nmap hackers 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. Other lines must contain one of the directives described below. Some readers prefer to peek at the examples in [the section called “Putting It All Together”](#) before tackling the following dissection.

Exclude Directive

Syntax: Exclude *<port specification>*

Examples:

```
Exclude 53,T:9100,U:30000-40000
```

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. The Exclude directive uses the same format as the Nmap -p switch, so ranges and comma separated lists of ports are supported. In the nmap-service-probes included with Nmap the only ports excluded are TCP port 9100 through 9107. These are common ports for printers to listen on and they often print any data sent to them. So a version detection scan can cause them to print many pages full of probes that Nmap sends, such as SunRPC requests, help statements, and X11 probes.

This behavior is often undesirable, especially when a scan is meant to be stealthy. However, Nmap's default behavior of avoiding scanning this port can make it easier for a sneaky user to hide a service: simply run it on an excluded port such as 9100 and it is less likely to be identified by name. The port scan will still show it as open. Users can override the Exclude directive with the --allports option. This causes version detection to interrogate all open ports.

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|
Probe TCP NULL q||
```

The Probe directive tells Nmap what string to send to recognize various services. All of the directives discussed later operate on the most recent Probe statement. 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, \xHH. 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 PureFTPd (\d\S+)/
p/PureFTPd/ v/$1/
match ssh m/^SSH-([\d.]+)-OpenSSH_(\S+)/ p/OpenSSH/ v/$2/
i/protocol $1/
match mysql m/^.\0\0\0\n(4\.[-.\w+)\0...\0/s p/MySQL/ i/
$1/
match chargen m|@ABCDEFGHIJKLMNOPQRSTUVWXYZ|
match uucp m|^login: Password: Login incorrect\.$|
p/SunOS uucpd/ o/SunOS/
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. As a special case, you can prefix the service name with ssl/, as in ssl/vmware-auth. In that case, the service would be stored as vmware-auth tunneled by SSL. This is useful for services which can be fully recognized without the overhead of making an SSL connection.

<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 six 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:

Table 7.1. versioninfo field formats and values

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. 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.

Any of the six fields can be omitted. In fact, all of the fields can be omitted if no further information on the service is available. Any of the version fields can include numbered strings such as \$1 or \$2, which are replaced (in a Perl-like fashion) with the corresponding parenthesized substring in the *<pattern>*.

In rare cases, a *helper function* can be applied to the replacement text before insertion. The \$P() helper function will filter out unprintable characters. This is useful for converting Unicode UTF-16 encoded strings such as W\00\0R\0K\0G\0R\0O\0U\0P\0 into the ASCII approximation WORKGROUP. It can be used in any versioninfo field by passing it the number of the match you want to make printable, like this: i/\$P(3)/.

Another helper function is \$SUBST(). This is used for making substitutions in matches before they are printed. It takes three arguments. The first is the substitution number in the pattern, just as you would use in a normal replacement variable such

as \$1 or \$3. The second and third arguments specify a substring you wish to find and replace, respectively. All instances of the match string found in the substring are replaced, not just the first one. For example, the VanDyke VShell sshd gives its version number in a format such as 2_2_3_578. We use the versioninfo field `v/$SUBST(1,"_",".")/` to convert it to the more conventional form 2.2.3.578.

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. See [the section called “Technique Described”](#) for more details on how this works. Arguments are not defined here because they are the same as for `match` above, except that there is never a `<versioninfo>` argument. Also as with `match`, many `softmatch` statements can exist within a single Probe section.

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. See the examples above. More details on how this works are in [the section called “Technique Described”](#).

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 frequently this probe can be expected to return useful results. The higher the number, the more rare the probe is considered and the less likely it is to be tried against a service. More details can be found in [the section called “Probe Selection and Rarity”](#).

fallback Directive

Syntax: fallback <Comma separated list of probes>

Example:

```
fallback GetRequest,GenericLines
```

This optional directive specifies which probes should be used as fallbacks for if there are no matches in the current Probe section. For more information on fallbacks see [the section called “Cheats and Fallbacks”](#). 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 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.

Putting It All Together

Here are some examples from nmap-service-probes which put this all together (to save space many lines have been skipped). After reading this far into the section, the following should be understood.

```
# The Exclude directive takes a comma separated list of
ports.
# The format is exactly the same as the -p switch.
Exclude T:9100-9107

# This is the NULL probe that just compares any banners
given to us
#####NEXT
PROBE#####
Probe TCP NULL q||
# Wait for at least 5 seconds for data. Otherwise an
Nmap default is used.
totalwaitms 5000
# Windows 2003
match ftp m/^220[ -]Microsoft FTP Service\r\n/
p/Microsoft ftpd/
match ftp m/^220 ProFTPD (\d\S+) Server/ p/ProFTPD/ v/$1/
```

```

softmatch ftp m/^220 [-.\w ]+ftp.*\r\n$/i
match ident m|^flock\(\) on closed filehandle .*midentd|
p/midentd/ i/broken/
match imap m|^.* OK Welcome to Binc IMAP v(\d[-.\w]+) |
p/Binc IMAPd/ v$1/
softmatch imap m/^.* OK [-.\w ]+imap[-.\w ]+\r\n$/i
match lucent-fwadm m|^0001;2$| p/Lucent Secure Management
Server/
match meetingmaker m/^\xc1,$/ p/Meeting Maker
calendar/
# lopster 1.2.0.1 on Linux 1.1
match napster m|^1$| p/Lopster Napster P2P client/

Probe UDP Help q|help\r\n\r\n|
rarity 3
ports 7,13,37
match chargen m|@ABCDEFGHIJKLMNOPQRSTUVWXYZ|
match echo m|^help\r\n\r\n$/i

```

Community Contributions

No matter how technically advanced a service detection framework is, it would be nearly useless without a comprehensive database of services against which to match. This is where the open source nature of Nmap really shines. The Insecure.Org lab is pretty substantial by geek standards, but it can never hope to run more than a tiny percentage of machine types and services that are out there. Fortunately experience with OS detection fingerprints has shown that Nmap users together run all of the common stuff, plus a staggering array of bizarre equipment as well. The Nmap OS fingerprint database contains more than a thousand entries, including all sorts of switches, WAPs, VoIP phones, game consoles, Unix boxes, Windows hosts, printers, routers, PDAs, firewalls, etc. Version detection also supports user submissions. Nmap users have contributed thousands of services. There are three primary ways that the Nmap community helps to make this an exceptional database: submitting service fingerprints, database corrections, and new probes.

Submit Service Fingerprints

If a service responds to one or more of Nmap's probes and yet Nmap is unable to identify that service, Nmap prints a *service fingerprint* like this one:

```
SF-Port21-TCP:V=3.40PVT16%D=9/6%Time=3F5A961C
%r(NULL,3F,"220\x20stage\x20F
SF:TP\x20server\x20\ (Version\x202\.1WU\ (1\)\ +SCO-
2\.6\.1\+-sec\)\x20ready\
SF:.\r\n")
%r(GenericLines,81,"220\x20stage\x20FTP\x20server\x20\
(Version\x
SF:202\.1WU\ (1\)\ +SCO-2\.6\.1\+-
sec\)\x20ready\.\r\n500\x20':\x20command\
SF:x20not\x20understood\.\r\n500\x20':\x20command\x20not
\x20understood\.\
SF:r\n");
```

If you receive such a fingerprint, and are sure you know what daemon version is running on the target host, please submit the fingerprint at the URL Nmap gives you. The whole submission process is anonymous (unless you choose to provide identifying info) and should not take more than a couple minutes. If you are feeling particularly helpful, scan the system again using -d (Nmap sometimes gives longer fingerprints that way) and paste both fingerprints into the fingerprint box on the submission form. Sometimes people read the file format section and submit their own working match lines. This is OK, but please submit the service fingerprint(s) as well because existing scripts make integrating and testing them relatively easy.

For those who care, the information in the fingerprint above is port number (21), protocol (TCP), Nmap version (3.40PVT16), date (September 6), Unix time in hex, and a sequence of probe responses in the form `r({<probename>}, {<responselength>}, "{<responsestring>}")`.

Submit Database Corrections

This is another easy way to help improve the database. When integrating a service fingerprint submitted for “chargen on Windows XP” or “FooBar FTP server 3.9.213”, it is difficult to determine how general the match is. Will it also match chargen on Solaris or FooBar FTP 2.7? Since there is no good way to tell, a very specific name is

used in the hope that people will report when the match needs to be generalized. The only reason the Nmap DB is so comprehensive is that thousands of users have spent a few minutes each to submit new information. If you scan a host and the service fingerprint gives an incorrect OS, version number, application name, or even service type, please let us know as described below:

Upgrade to the latest Nmap (Optional)

Many Linux distributions and other operating systems ship with ancient versions of Nmap. The Nmap version detection database is improved with almost every release, so check your version number by running **nmap -V** and then compare that to the latest available from <http://nmap.org/download.html>. The problem you are seeing may have already been corrected. Installing the newest version takes only a few minutes on most platforms, and is valuable regardless of whether the version detection flaw you are reporting still exists. But even if you don't have time to upgrade right now, submissions from older releases are still valuable.

Be absolutely certain you know what is running

Invalid “corrections” can corrupt the version detection DB. If you aren't certain exactly what is running on the remote machine, please find out before submitting.

Generate a fingerprint

Run the command **nmap -O -PN -sSV -T4 -d --version-trace -p<port> <target>**, where <port> is the port running the misidentified service on the <target> host. If the service is UDP rather than TCP, substitute -sUV for -sSV.

Send us your correction

Now simply submit your correction to us at <http://insecure.org/cgi-bin/submit.cgi?corr-service>. Thanks for contributing to the Nmap community and helping to make version detection even better!

Submit New Probes

Suppose Nmap fails to detect a service. If it received a response to any probes at all, it should provide a fingerprint that can be submitted as described above. But what if there is no response and thus a fingerprint is not available? Create and submit your own probe! These are very welcome. The following steps describe the process.

Steps for creating a new version detection probe

1. Download the latest version of Nmap from <http://nmap.org> and try again. You would feel a bit silly spending time developing a new probe just to find out that it has already been added. Make sure no fingerprint is available, as it is better to recognize services using existing probes if possible than to create too many new ones. If the service does not respond to any of the existing probes, there is no other choice.
2. Decide on a good probe string for recognizing the service. An ideal probe should elicit a response from as many instances of the service as possible, and ideally the responses should be unique enough to differentiate between them. This step is easiest if you understand the protocol very well, so consider reading the relevant RFCs and product documentation. One simple approach is to simply start a client for the given service and watch what initial handshaking is done by sniffing the network with Wireshark or tcpdump, or connecting to a listening Netcat.
3. Once you have decided on the proper string, add the appropriate new Probe line to Nmap (see [the section called "Technique Described"](#) and [the section called "nmap-service-probes File Format"](#)). Do not put in any match lines at first, although a ports directive to make this new test go first against the registered ports is OK. Then scan the service with Nmap a few times. You should get a fingerprint back showing the service's response to your new probe. Send the new probe line and the fingerprints (against different machines if possible, but even a few against the same daemon helps to note differences) to Fyodor at [<fyodor@insecure.org>](mailto:fyodor@insecure.org). It will likely then be integrated into future versions of Nmap. Any details you can provide on the nature of your probe string is helpful as well. For custom services that only appear on your network, it is better to simply add them to your own nmap-service-probes rather than the global Nmap.

SOLUTION: Hack Version Detection to Suit Custom Needs, such as Open Proxy Detection

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

SOLUTION: Find All Servers Running an Insecure or Nonstandard Application Version

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

Chapter 8. Remote OS Detection

Table of Contents

[Introduction](#)

- [Reasons for OS Detection](#)
- [Determining vulnerability of target hosts](#)
- [Tailoring exploits](#)
- [Network inventory and support](#)
- [Detecting unauthorized and dangerous devices](#)
- [Social engineering](#)

[Usage and Examples](#)

[TCP/IP Fingerprinting Methods Supported by Nmap](#)

- [Probes Sent](#)
- [Sequence generation \(SEQ, OPS, WIN, and T1\)](#)
- [ICMP echo \(IE\)](#)
- [TCP explicit congestion notification \(ECN\)](#)
- [TCP \(T2–T7\)](#)
- [UDP \(U1\)](#)

Response Tests

TCP ISN greatest common divisor (GCD)

TCP ISN counter rate (ISR)

TCP ISN sequence predictability index (SP)

IP ID sequence generation algorithm (TI, CI, II)

Shared IP ID sequence Boolean (SS)

TCP timestamp option algorithm (TS)

TCP options (O, O1–O6)

TCP initial window size (W, W1–W6)

Responsiveness (R)

IP don't fragment bit (DF)

Don't fragment (ICMP) (DFI)

IP initial time-to-live (T)

IP initial time-to-live guess (TG)

Explicit congestion notification (CC)

TCP miscellaneous quirks (Q)

TCP sequence number (S)

TCP acknowledgment number (A)

TCP flags (F)

TCP RST data checksum (RD)

IP total length (IPL)

Unused port unreachable field nonzero (UN)

Returned probe IP total length value (RIPL)

Returned probe IP ID value (RID)

Integrity of returned probe IP checksum value (RIPCK)

Integrity of returned probe UDP checksum (RUCK)

Integrity of returned UDP data (RUD)

ICMP response code (CD)

Fingerprinting Methods Avoided by Nmap

Passive Fingerprinting

Exploit Chronology

Retransmission Times

IP Fragmentation

Open Port Patterns

Retired Tests

Understanding an Nmap Fingerprint

Decoding the Subject Fingerprint Format

Decoding the SCAN line of a subject fingerprint

Decoding the Reference Fingerprint Format

Free-form OS description (Fingerprint line)

Device and OS classification (Class lines)

Test expressions

OS Matching Algorithms

Dealing with Misidentified and Unidentified Hosts

[When Nmap Guesses Wrong](#)

[When Nmap Fails to Find a Match and Prints a Fingerprint](#)

[Modifying the nmap-os-db Database Yourself](#)

Introduction

When exploring a network for security auditing or inventory/administration, you usually want to know more than the bare IP addresses of identified machines. Your reaction to discovering a printer may be very different than to finding a router, wireless access point, telephone PBX, game console, Windows desktop, or Unix server. Finer grained detection (such as distinguishing Mac OS X 10.4 from 10.3) is useful for determining vulnerability to specific flaws and for tailoring effective exploits for those vulnerabilities.

In part due to its value to attackers, many systems are tight-lipped about their exact nature and operating system configuration. Fortunately, Nmap includes a huge database of heuristics for identifying thousands of different systems based on how they respond to a selection of TCP/IP probes. Another system (part of version detection) interrogates open TCP or UDP ports to determine device type and OS details. Results of these two systems are reported independently so that you can identify combinations such as a Checkpoint firewall forwarding port 80 to a Windows IIS server.

While Nmap has supported OS detection since 1998, this chapter describes the 2nd generation system released in 2006.

Reasons for OS Detection

While some benefits of discovering the underlying OS and device types on a network are obvious, others are more obscure. This section lists the top reasons I hear for discovering this extra information.

Determining vulnerability of target hosts

It is sometimes very difficult to determine remotely whether an available service is susceptible or patched for a certain vulnerability. Even obtaining the application version number doesn't always help, since OS distributors often back-port security fixes without changing

the version number. The surest way to verify that a vulnerability is real is to exploit it, but that risks crashing the service and can lead to wasted hours or even days of frustrating exploitation efforts if the service turns out to be patched.

OS detection can help reduce these false positives. For example, the Rwho daemon on unpatched Sun Solaris 7 through 9 may be remotely exploitable (Sun alert #57659). Remotely determining vulnerability is difficult, but you can rule it out by finding that a target system is running Solaris 10.

Taking this from the perspective of a systems administrator rather than a pen-tester, imagine you run a large Sun shop when alert #57659 comes out. Scan your whole network with OS detection to find machines which need patching before the bad guys do.

Tailoring exploits

Even after you discover a vulnerability in a target system, OS detection can be helpful in exploiting it. Buffer overflows, format-string exploits, and many other vulnerabilities often require custom-tailored shellcode with offsets and assembly payloads generated to match the target OS and hardware architecture. In some cases, you only get one try because the service crashes if you get the shellcode wrong. Use OS detection first or you may end up sending Linux shellcode to a FreeBSD server.

Network inventory and support

While it isn't as exciting as busting root through a specially crafted format string exploit, there are many administrative reasons to keep track of what is running on your network. Before you renew that IRIX support contract for another year, scan to see if anyone still uses such machines. An inventory can also be useful for IT budgeting and ensuring that all company equipment is accounted for.

Detecting unauthorized and dangerous devices

With the ubiquity of mobile devices and cheap commodity networking equipment, companies are increasingly finding that employees are extending their networks in undesirable ways. They may install a \$20 wireless access point (WAP) in their cubicle without realizing (or caring) that they just opened up the protected corporate network to potential attackers in the parking lot or nearby

buildings. WAPs can be so dangerous that Nmap has a special category for detecting them. Users may also cause sysadmins grief by connecting insecure and/or worm-infected laptops to the corporate network. Regular scanning can detect unauthorized devices for investigation and containment.

Social engineering

Another possible use is social engineering. Lets say that you are scanning a target company and Nmap reports a “Datavoice TxPORT PRISM 3000 T1 CSU/DSU 6.22/2.06”. You could call up the target pretending to be Datavoice support and discuss some issues with their PRISM 3000. Tell them you are about to announce a big security hole, but are first providing the patch to valued customers. Some naive administrators might assume that only an authorized engineer from Datavoice would know so much about their CSU/DSU. Of course the patch you send them is a Trojan horse that gives you remote access to sniff and traipse through their network. Be sure to read the rest of this chapter for detection accuracy and verification advice before trying this. If you guess the target system wrong and they call the police, that will be an embarrassing story to tell your cellmates.

Usage and Examples

The inner workings of OS detection are quite complex, but it is one of the easiest features to use. Simply add `-O` to your scan options. You may want to also increase the verbosity with `-v` for even more OS-related details. This is shown in [Example 8.1](#).

Example 8.1. OS detection with verbosity (`-O -v`)

```
# nmap -O -v scanme.nmap.org

Starting Nmap ( http://nmap.org )
Interesting ports on scanme.nmap.org (64.13.134.52):
Not shown: 994 filtered ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    closed smtp
53/tcp    open  domain
70/tcp    closed gopher
80/tcp    open  http
```

```
113/tcp closed auth
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.20-1 (Fedora Core 5)
Uptime guess: 11.433 days (since Thu Sep 18 13:13:01
2008)
TCP Sequence Prediction: Difficulty=204 (Good luck!)
IP ID Sequence Generation: All zeros

Nmap done: 1 IP address (1 host up) scanned in 6.21
seconds
           Raw packets sent: 2021 (90.526KB) | Rcvd: 23
(1326B)
```

Including the -O -v options caused Nmap to generate the following six extra line items:

Device type

All fingerprints are classified with one or more high-level device types, such as router, printer, firewall, or (as in this case) general purpose. These are further described in [the section called “Device and OS classification \(Class lines\)”](#). Several device types may be shown, in which case they will be separated with the pipe symbol as in “Device Type: router|firewall”.

Running

This field is also related to the OS classification scheme described in [the section called “Device and OS classification \(Class lines\)”](#). It shows the OS Family (Linux in this case) and OS generation (2.6.X) if available. If there are multiple OS families, they are separated by commas. When Nmap can't narrow down OS generations to one specific choice, options are separated by the pipe symbol (|) Examples include OpenBSD 3.X, NetBSD 3.X|4.X and Linux 2.4.X|2.5.X|2.6.X.

If Nmap finds too many OS families to print concisely, it will omit this line. When there are no perfect matches, Nmap changes the field to Running (JUST GUESSING) and adds an accuracy percentage (100% is a perfect match) in

parentheses after each candidate family name. If no fingerprints are close matches, the line is omitted.

OS details

This line gives the detailed description for each fingerprint that matches. While the Device type and Running lines are from predefined enumerated lists that are easy to parse by a computer, the OS details line contains free-form data which is useful to a human reading the report. This can include more exact version numbers, device models, and architectures specific to a given fingerprint. In this example, the only matching fingerprint was Linux 2.6.20-1 (Fedora Core 5). When there are multiple exact matches, they are comma-separated. If there aren't any perfect matches, but some close guesses, the field is renamed Aggressive OS guesses and fingerprints are shown followed by a percentage in parentheses which specifies how close each match was.

Uptime guess

As part of OS detection, Nmap receives several SYN/ACK TCP packets in a row and checks the headers for a timestamp option. Many operating systems use a simple counter for this which starts at zero at boot time then increments at a constant rate such as twice per second. By looking at several responses, Nmap can determine the current values and rate of increase. Simple linear extrapolation determines boot time. The timestamp algorithm is used for OS detection too (see [the section called "TCP timestamp option algorithm \(TS\)"](#)) since the increment rate on different systems varies from 2 Hz to 1,000 Hz.

The uptime guess is labeled a "guess" because various factors can make it completely inaccurate. Some operating systems do not start the timestamp counter at zero, but initialize it with a random value, making extrapolation to zero meaningless. Even on systems using a simple counter starting at zero, the counter eventually overflows and wraps around. With a 1,000 Hz counter increment rate, the counter resets to zero roughly every 50 days. So a host that has been up for 102 days will appear to have been up only two days. Even with these caveats, the uptime guess is accurate much of the time for most operating systems, so it is printed when

available, but only in verbose mode. The uptime guess is omitted if the target gives zeros or no timestamp options in its SYN/ACK packets, or if it does not reply at all. The line is also omitted if Nmap cannot discern the timestamp increment rate or it seems suspicious (like a 30-year uptime).

Network Distance

A side effect of one of the OS detection tests allows Nmap to compute how many routers are between it and a target host. The distance is zero when you are scanning localhost, and one for a machine on the same network segment. Each additional router on the path adds one to the hop count. The Network Distance line is not printed in this example, since Nmap omits the line when it cannot be computed (no reply to the relevant probe).

TCP Sequence Prediction

Systems with poor TCP initial sequence number generation are vulnerable to blind TCP spoofing attacks. In other words, you can make a full connection to those systems and send (but not receive) data while spoofing a different IP address. The target's logs will show the spoofed IP, and you can take advantage of any trust relationship between them. This attack was all the rage in the mid-nineties when people commonly used rlogin to allow logins to their account without any password from trusted IP addresses. Kevin Mitnick is alleged to have used this attack to break into Tsutomu Shimomura's computers in December 1994.

The good news is that hardly anyone uses rlogin anymore, and many operating systems have been fixed to use unpredictable initial sequence numbers as proposed by [RFC 1948](#). For these reasons, this line is only printed in verbose mode. Sadly, many vendors still ship [vulnerable operating systems and devices](#). Even the fixed ones often vary in implementation, which leaves them valuable for OS detection purposes. The class describes the ISN generation algorithm used by the target, and difficulty is a rough estimate of how hard the system makes blind IP spoofing (0 is the easiest). The parenthesized comment is based on the difficulty index and ranges from Trivial joke to Easy, Medium, Formidable, Worthy challenge, and finally Good luck! Further details about sequence tests are

provided in [the section called “TCP ISN greatest common divisor \(GCD\)”](#).

While the rlogin family is mostly a relic of the past, clever attackers can still find effective uses for blind TCP spoofing. For example, it allows for spoofed HTTP requests. You don't see the results, but just the URL (POST or GET request) can have dramatic side effects. The spoofing allows attackers to hide their identity, frame someone else, or exploit IP address restrictions.

IP ID sequence generation

Many systems unwittingly give away sensitive information about their traffic levels based on how they generate the lowly 16-bit ID field in IP packets. This can be abused to spoof a port scan against other systems and for other mischievous purposes discussed in [the section called “TCP Idle Scan \(-sI\)”](#). This field describes the ID generation algorithm that Nmap was able to discern. More information on how it classifies them is available in [the section called “IP ID sequence generation algorithm \(TI, CI, II\)”](#). Note that many systems use a different IP ID space for each host they communicate with. In that case, they may appear vulnerable (such as showing the Incremental class) while still being secure against attacks such as the idle scan. For this reason, and because the issue is rarely critical, the IP ID sequence generation line is only printed in verbose mode. If Nmap does not receive sufficient responses during OS detection, it will omit the whole line. The best way to test whether a host is vulnerable to being an idle scan zombie is to test it with -sI.

While TCP fingerprinting is a powerful method for OS detection, interrogating open ports for clues is another effective approach. Some applications, such as Microsoft IIS, only run on a single platform (thus giving it away), while many other apps divulge their platform in overly verbose banner messages. Adding the -sV option enables Nmap version detection, which is trained to look for these clues (among others). In [Example 8.2](#), Nmap catches the platform details from an FTP server.

Example 8.2. Using version scan to detect the OS

```
# nmap -sV -O -v 129.128.X.XX
```

```

Starting Nmap ( http://nmap.org )
Interesting ports on [hostname] (129.128.X.XX):
Not shown: 994 closed ports
PORT      STATE      SERVICE      VERSION
21/tcp    open      ftp          HP-UX 10.x ftpd 4.1
22/tcp    open      ssh          OpenSSH 3.7.1p1 (protocol
1.99)
111/tcp   open      rpc
445/tcp   filtered microsoft-ds
1526/tcp  open      oracle-tns   Oracle TNS Listener
32775/tcp open      rpc
No exact OS matches for host
TCP Sequence Prediction: Class=truly random
                        Difficulty=9999999 (Good luck!)
IP ID Sequence Generation: Incremental
Service Info: OS: HP-UX

```

In this example, the line “No exact OS matches for host” means that TCP/IP fingerprinting failed to find an exact match. Fortunately, the Service Info field a few lines down discloses that the OS is HP-UX. If several operating systems were detected (which can happen with NAT gateway boxes that redirect ports to several different machines), the field would be OSs and the values would be comma separated. The Service Info line can also contain hostnames and device types found during the version scan. The focus of this chapter is on TCP/IP fingerprinting though, since version detection was covered in [Chapter 7, Service and Application Version Detection](#).

With two effective OS detection methods available, which one should you use? The best answer is usually both. In some cases, such as a proxy firewall forwarding to an application on another host, the answers may legitimately differ. TCP/IP fingerprinting will identify the proxy while version scanning will generally detect the server running the proxied application. Even when no proxying or port forwarding is involved, using both techniques is beneficial. If they come out the same, that makes the results more credible. If they come out wildly different, investigate further to determine what is going on before relying on either. Since OS and version detection go together so well, the -A option enables them both.

OS detection is far more effective if at least one open and one closed TCP port are found. Set the --osscan-limit option and Nmap

will not even try OS detection against hosts which do not meet this criteria. This can save substantial time, particularly on -PN scans against many hosts. You still need to enable OS detection with -O (or -A) for this to have any effect.

Another OS detection option is --osscan-guess. When Nmap is unable to detect a perfect OS match, it sometimes offers up near-matches as possibilities. The match has to be very close for Nmap to do this by default. If you specify this option (or the equivalent --fuzzy option), Nmap will guess more aggressively. Nmap still tells you when an imperfect match is printed and display its confidence level (percentage) for each guess.

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. The --max-os-tries option lets you change this maximum number of OS detection tries. Lowering it (usually to 1) speeds Nmap up, though you miss out on retries which could potentially identify the OS. Alternatively, a high value may be set to allow even more retries when conditions are favorable. This is rarely done, except to generate better fingerprints for submission and integration into the Nmap OS database.

Like just about every other part of Nmap, results ultimately come from the target machine itself. While rare, systems are occasionally configured to confuse or mislead Nmap. Several programs have even been developed specifically to trick Nmap OS detection (see [the section called "OS Spoofing"](#)). Your best bet is to use numerous reconnaissance methods to explore a network, and don't trust any one of them.

TCP/IP fingerprinting requires collecting detailed information about the target's IP stack. The most commonly useful results, such as TTL information, are printed to Nmap output whenever they are obtained. Slightly less pertinent information, such as IP ID sequence generation and TCP sequence prediction difficulty, is only printed in verbose mode. But if you want all of the IP stack details that Nmap collected, you can find it in a compact form called a *subject fingerprint*. Nmap sometimes prints this (for user submission purposes) when it doesn't recognize a host. You can also force Nmap to print it (in normal, interactive, and XML formats) by enabling

debugging with (-d). Then read [the section called “Understanding an Nmap Fingerprint”](#) to interpret it.

TCP/IP Fingerprinting Methods

Supported by Nmap

Nmap OS fingerprinting works by sending up to 16 TCP, UDP, and ICMP probes to known open and closed ports of the target machine. These probes are specially designed to exploit various ambiguities in the standard protocol RFCs. Then Nmap listens for responses. Dozens of attributes in those responses are analyzed and combined to generate a fingerprint. Every probe packet is tracked and resent at least once if there is no response. All of the packets are IPv4 with a random IP ID value. Probes to an open TCP port are skipped if no such port has been found. For closed TCP or UDP ports, Nmap will first check if such a port has been found. If not, Nmap will just pick a port at random and hope for the best.

The following sections are highly technical and reveal the hidden workings of Nmap OS detection. Nmap can be used effectively without understanding this, though the material can help you better understand remote networks and also detect and explain certain anomalies. Plus, some of the techniques are pretty cool. Readers in a hurry may skip to [the section called “Dealing with Misidentified and Unidentified Hosts”](#). But for those of you who are ready for a journey through TCP explicit congestion notification, reserved UDP header bits, initial sequence numbers, bogus flags, and Christmas tree packets: read on!

Even the best of us occasionally forget byte offsets for packet header fields and flags. For quick reference, the IPv4, TCP, UDP, and ICMP header layouts can be found in [the section called “TCP/IP Reference”](#). The layout for ICMP echo request and destination unreachable packets are shown in [Figure 8.1](#) and [Figure 8.2](#).

Figure 8.1. ICMP echo request or reply header layout

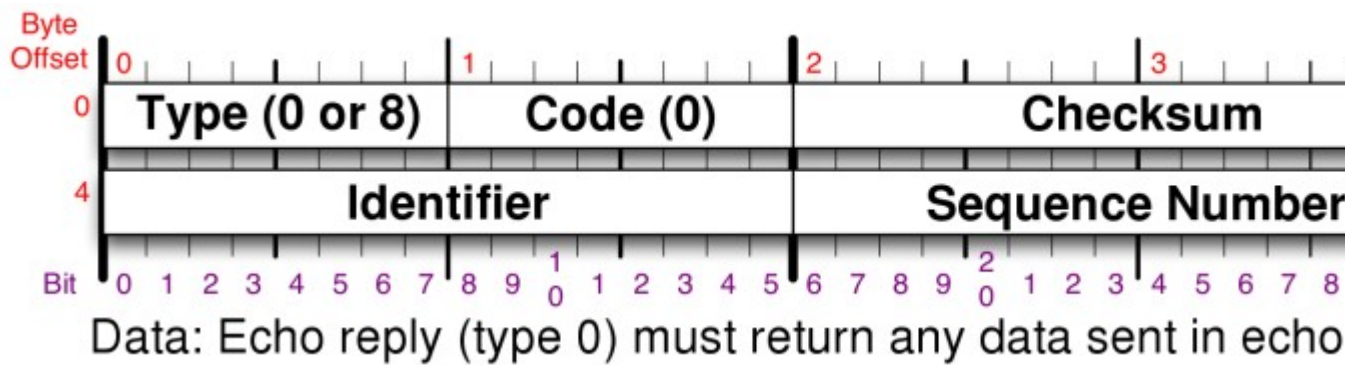
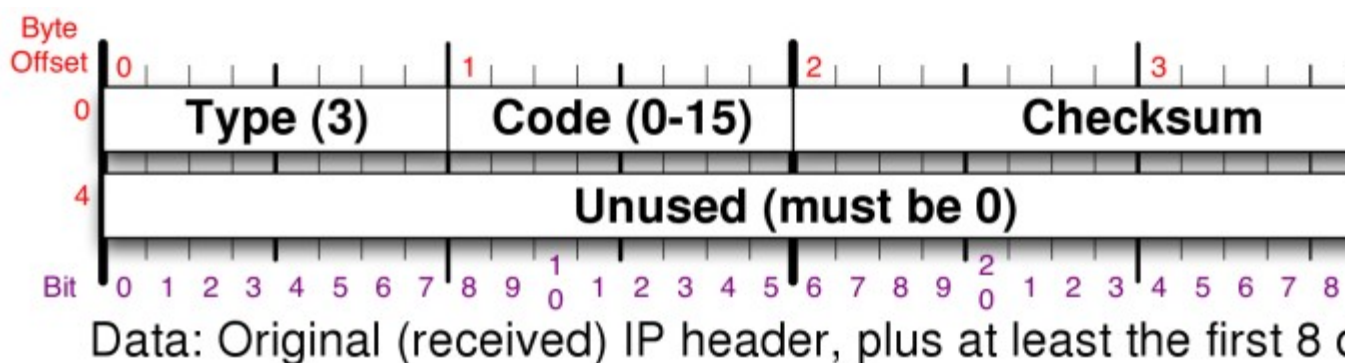


Figure 8.2. ICMP destination unreachable header layout



Probes Sent

This section describes each IP probe sent by Nmap as part of TCP/IP fingerprinting. It refers to Nmap response tests and TCP options which are explained in the following section.

Sequence generation (SEQ, OPS, WIN, and T1)

A series of six TCP probes is sent to generate these four test response lines. The probes are sent exactly 110 milliseconds apart so the total time taken is 550 ms. Exact timing is important as some of the sequence algorithms we detect (initial sequence numbers, IP IDs, and TCP timestamps) are time dependent. This timing value was chosen to take above 500 ms so that we can reliably detect the common 2 Hz TCP timestamp sequences.

Each probe is a TCP SYN packet to a detected open port on the remote machine. The sequence and acknowledgment numbers are random (but saved so Nmap can differentiate responses). Detection

accuracy requires probe consistency, so there is no data payload even if the user requested one with --data-length.

These packets vary in the TCP options they use and the TCP window field value. The following list provides the options and values for all six packets. The listed window field values do not reflect window scaling. EOL is the end-of-options-list option, which many sniffing tools don't show by default.

- **Packet #1:** window scale (10), NOP, MSS (1460), timestamp (TSval: 0xFFFFFFFF; TSecr: 0), SACK permitted. The window field is 1.
- **Packet #2:** MSS (1400), window scale (0), SACK permitted, timestamp (TSval: 0xFFFFFFFF; TSecr: 0), EOL. The window field is 63.
- **Packet #3:** Timestamp (TSval: 0xFFFFFFFF; TSecr: 0), NOP, NOP, window scale (5), NOP, MSS (640). The window field is 4.
- **Packet #4:** SACK permitted, Timestamp (TSval: 0xFFFFFFFF; TSecr: 0), window scale (10), EOL. The window field is 4.
- **Packet #5:** MSS (536), SACK permitted, Timestamp (TSval: 0xFFFFFFFF; TSecr: 0), window scale (10), EOL. The window field is 16.
- **Packet #6:** MSS (265), SACK permitted, Timestamp (TSval: 0xFFFFFFFF; TSecr: 0). The window field is 512.

The results of these tests include four result category lines. The first, SEQ, contains results based on sequence analysis of the probe packets. These test results are GCD, SP, ISR, TI, II, TS, and SS. The next line, OPS contains the TCP options received for each of the probes (the test names are O1 through O6). Similarly, the WIN line contains window sizes for the probe responses (named W1 through W6). The final line related to these probes, T1, contains various test values for packet #1. Those results are for the R, DF, T, TG, W, S, A, F, O, RD, and Q tests. These tests are only reported for the first probe since they are almost always the same for each probe.

ICMP echo (IE)

The IE test involves sending two ICMP echo request packets to the target. The first one has the IP DF bit set, a type-of-service (TOS) byte value of zero, a code of nine (even though it should be zero), the sequence number 295, a random IP ID and ICMP request identifier, and a random character repeated 120 times for the data payload.

The second ping query is similar, except a TOS of four (IP_TOS_RELIABILITY) is used, the code is zero, 150 bytes of data is sent, and the IP ID, request ID, and sequence numbers are incremented by one from the previous query values.

The results of both of these probes are combined into a IE line containing the R, DFI, T, TG, and CD tests. The R value is only true (Y) if both probes elicit responses. The T, and CD values are for the response to the first probe only, since they are highly unlikely to differ. DFI is a custom test for this special dual-probe ICMP case.

These ICMP probes follow immediately after the TCP sequence probes to ensure valid results of the shared IP ID sequence number test (see [the section called “Shared IP ID sequence Boolean \(SS\)”](#)).

TCP explicit congestion notification (ECN)

This probe tests for explicit congestion notification (ECN) support in the target TCP stack. ECN is a method for improving Internet performance by allowing routers to signal congestion problems before they start having to drop packets. It is documented in [RFC 3168](#). Nmap tests this by sending a SYN packet which also has the ECN CWR and ECE congestion control flags set. For an unrelated (to ECN) test, the urgent field value of 0xF7F5 is used even though the urgent flag is not set. The acknowledgment number is zero, sequence number is random, window size field is three, and the reserved bit which immediately precedes the CWR bit is set. TCP options are WScale (10), NOP, MSS (1460), SACK permitted, NOP, NOP. The probe is sent to an open port.

If a response is received, the R, DF, T, TG, W, O, CC, and Q tests are performed and recorded.

TCP (T2-T7)

The six T2 through T7 tests each send one TCP probe packet. With one exception, the TCP options data in each case is (in hex) 03030A0102040109080AFFFFFFFF000000000402. Those 20 bytes correspond to window scale (10), NOP, MSS (265), Timestamp (TSval: 0xFFFFFFFF; TSecr: 0), then SACK permitted. The exception is that T7 uses a Window scale value of 15 rather than 10. The variable characteristics of each probe are described below:

- **T2** sends a TCP null (no flags set) packet with the IP DF bit set and a window field of 128 to an open port.
- **T3** sends a TCP packet with the SYN, FIN, URG, and PSH flags set and a window field of 256 to an open port. The IP DF bit is not set.
- **T4** sends a TCP ACK packet with IP DF and a window field of 1024 to an open port.
- **T5** sends a TCP SYN packet without IP DF and a window field of 31337 to a closed port.
- **T6** sends a TCP ACK packet with IP DF and a window field of 32768 to a closed port.
- **T7** sends a TCP packet with the FIN, PSH, and URG flags set and a window field of 65535 to a closed port. The IP DF bit is not set.

In each of these cases, a line is added to the fingerprint with results for the R, DF, T, TG, W, S, A, F, O, RD, and Q tests.

UDP (U1)

This probe is a UDP packet sent to a closed port. The character 'C' (0x43) is repeated 300 times for the data field. The IP ID value is set to 0x1042 for operating systems which allow us to set this. If the port is truly closed and there is no firewall in place, Nmap expects to receive an ICMP port unreachable message in return. That response is then subjected to the R, DF, T, TG, IPL, UN, RIPL, RID, RIPCK, RUCK, and RUD tests.

Response Tests

The previous section describes probes sent by Nmap, and this one completes the puzzle by describing the barrage of tests performed on responses. The short names (such as DF, R, and RIPCK) are those used in the nmap-os-db fingerprint database to save space. All numerical test values are given in hexadecimal notation, without leading zeros, unless noted otherwise. The tests are documented in roughly the order they appear in fingerprints.

TCP ISN greatest common divisor (GCD)

The SEQ test sends six TCP SYN packets to an open port of the target machine and collects SYN/ACK packets back. Each of these SYN/ACK packets contains a 32-bit initial sequence number (ISN).

This test attempts to determine the smallest number by which the target host increments these values. For example, many hosts (especially old ones) always increment the ISN in multiples of 64,000.

The first step in calculating this is creating an array of differences between probe responses. The first element is the difference between the 1st and 2nd probe response ISNs. The second element is the difference between the 2nd and 3rd responses. There are five elements if Nmap receives responses to all six probes. Since the next couple of sections reference this array, we will call it `diff1`. If an ISN is lower than the previous one, Nmap looks at both the number of values it would have to subtract from the first value to obtain the second, and the number of values it would have to count up (including wrapping the 32-bit counter back to zero). The smaller of those two values is stored in `diff1`. So the difference between 0x20000 followed by 0x15000 is 0xB000. The difference between 0xFFFFFFFF00 and 0xC000 is 0xC0FF. This test value then records the greatest common divisor of all those elements. This GCD is also used for calculating the SP result.

TCP ISN counter rate (ISR)

This value reports the average rate of increase for the returned TCP initial sequence number. Recall that a difference is taken between each two consecutive probe responses and stored in the previously discussed `diff1` array. Those differences are each divided by the amount of time elapsed (in seconds—will generally be about 0.1) between sending the two probes which generated them. The result is an array, which we'll call `seq_rates` containing the rates of ISN counter increases per second. The array has one element for each `diff1` value. An average is taken of the array values. If that average is less than one (e.g. a constant ISN is used), ISR is zero. Otherwise ISR is eight times the binary logarithm (log base-2) of that average value, rounded to the nearest integer.

TCP ISN sequence predictability index (SP)

While the ISR test measures the average rate of initial sequence number increments, this value measures the ISN variability. It roughly estimates how difficult it would be to predict the next ISN from the known sequence of six probe responses. The calculation uses the difference array (`seq_rates`) and GCD values discussed in the previous section.

This test is only performed if at least four responses were seen. If the previously computed GCD value is greater than nine, the elements of the previously computed seq_rates array are divided by that value. We don't do the division for smaller GCD values because those are usually caused by chance. A [standard deviation](#) of the array of the resultant values is then taken. If the result is one or less, SP is zero. Otherwise the binary logarithm of the result is computed, then it is multiplied by eight, rounded to the nearest integer, and stored as SP.

Please keep in mind that this test is only done for OS detection purposes and is not a full-blown audit of the target ISN generator. There are many algorithm weaknesses that lead to easy predictability even with a high SP value.

IP ID sequence generation algorithm (TI, CI, II)

There are three tests that examine the IP header ID field of responses. TI is based on responses to the TCP SEQ probes. CI is from the responses to the three TCP probes sent to a closed port: T5, T6, and T7. II comes from the ICMP responses to the two IE ping probes. For TI, at least three responses must be received for the test to be included; for CI, at least two responses are required; and for II, both ICMP responses must be received.

For each of these tests, the target's IP ID generation algorithm is classified based on the algorithm below. Minor differences between tests are noted. Note that difference values assume that the counter can wrap. So the difference between an IP ID of 65,100 followed by a value of 700 is 1,136. The difference between 2,000 followed by 1,100 is 64,636. Here are the calculation details:

1. If all of the ID numbers are zero, the value of the test is Z.
2. If the IP ID sequence ever increases by at least 20,000, the value is RD (random). This result isn't possible for II because there are not enough samples to support it.
3. If all of the IP IDs are identical, the test is set to that value in hex.
4. If any of the differences between two consecutive IDs exceeds 1,000, and is not evenly divisible by 256, the test's value is RI (random positive increments). If the difference is evenly divisible by 256, it must be at least 256,000 to cause this RI result.

5. If all of the differences are divisible by 256 and no greater than 5,120, the test is set to BI (broken increment). This happens on systems like Microsoft Windows where the IP ID is sent in host byte order rather than network byte order. It works fine and isn't any sort of RFC violation, though it does give away host architecture details which can be useful to attackers.
6. If all of the differences are less than ten, the value is I (incremental). We allow difference up to ten here (rather than requiring sequential ordering) because traffic from other hosts can cause sequence gaps.
7. If none of the previous steps identify the generation algorithm, the test is omitted from the fingerprint.

Shared IP ID sequence Boolean (SS)

This Boolean value records whether the target shares its IP ID sequence between the TCP and ICMP protocols. If our six TCP IP ID values are 117, 118, 119, 120, 121, and 122, then our ICMP results are 123 and 124, it is clear that not only are both sequences incremental, but they are both part of the same sequence. If, on the other hand, the TCP IP ID values are 117–122 but the ICMP values are 32,917 and 32,918, two different sequences are being used.

This test is only included if II is RI, BI, or I and TI is the same. If SS is included, the result is S if the sequence is shared and O (other) if it is not. That determination is made by the following algorithm:

Let avg be the final TCP sequence response IP ID minus the first TCP sequence response IP ID, divided by the difference in probe numbers. If probe #1 returns an IP ID of 10,000 and probe #6 returns 20,000, avg would be $(20,000 - 10,000) / (6 - 1)$, which equals 2,000.

If the first ICMP echo response IP ID is less than the final TCP sequence response IP ID plus three times avg, the SS result is S. Otherwise it is O.

TCP timestamp option algorithm (TS)

TS is another test which attempts to determine target OS characteristics based on how it generates a series of numbers. This one looks at the TCP timestamp option (if any) in responses to the SEQ probes. It examines the TSval (first four bytes of the option)

rather than the echoed TSecr (last four bytes) value. It takes the difference between each consecutive TSval and divides that by the amount of time elapsed between Nmap sending the two probes which generated those responses. The resultant value gives a rate of timestamp increments per second. Nmap computes the average increments per second over all consecutive probes and then calculates the TS as follows:

1. If any of the responses have no timestamp option, TS is set to U (unsupported).
2. If any of the timestamp values are zero, TS is set to 0.
3. If the average increments per second falls within the ranges 0-5.66, 70-150, or 150-350, TS is set to 1, 7, or 8, respectively. These three ranges get special treatment because they correspond to the 2 Hz, 100 Hz, and 200 Hz frequencies used by many hosts.
4. In all other cases, Nmap records the binary logarithm of the average increments per second, rounded to the nearest integer. Since most hosts use 1,000 Hz frequencies, A is a common result.

TCP options (O, O1-O6)

This test records the TCP header options in a packet. It preserves the original ordering and also provides some information about option values. Because [RFC 793](#) doesn't require any particular ordering, implementations often come up with unique orderings. Some platforms don't implement all options (they are, of course, optional). When you combine all of those permutations with the number of different option values that implementations use, this test provides a veritable trove of information. The value for this test is a string of characters representing the options being used. Several options take arguments that come immediately after the character. Supported options and arguments are all shown in [Table 8.1](#).

Table 8.1. O test values

Option Name	Character	Argument (if any)
End of Options List (EOL)	L	
No operation (NOP)	N	
Maximum Segment Size (MSS)	M	The value is appended. Many systems echo the value used in the corresponding probe.

Option Name	Character	Argument (if any)
Window Scale (WS)	W	The actual value is appended.
Timestamp (TS)	T	The T is followed by two binary characters representing the TSval and TSecr values respectively. The characters are 0 if the field is zero and 1 otherwise.
Selective ACK permitted (SACK)	S	

As an example, the string M5B4NW3NNT11 means the packet includes the MSS option (value 0x5B4) followed by a NOP. Next comes a window scale option with a value of three, then two more NOPs. The final option is a timestamp, and neither of its two fields were zero. If there are no TCP options in a response, the test will exist but the value string will be empty. If no probe was returned, the test is omitted.

While this test is generally named O, the six probes sent for sequence generation purposes are a special case. Those are inserted into the special OPS test line and take the names O1 through O6 to distinguish which probe packet they relate to. The “O” stands for “options”. Despite the different names, each test O1 through O6 is processed exactly the same way as the other O tests.

TCP initial window size (W, W1-W6)

This test simply records the 16-bit TCP window size of the received packet. It is quite effective, since there are more than 80 values that at least one OS is known to send. A down side is that some operating systems have more than a dozen possible values by themselves. This leads to false negative results until we collect all of the possible window sizes used by an operating system.

While this test is generally named W, the six probes sent for sequence generation purposes are a special case. Those are inserted into a special WIN test line and take the names W1 through W6. The window size is recorded for all of the sequence number probes because they differ in TCP MSS option values, which causes some operating systems to advertise a different window size. Despite the different names, each test is processed exactly the same way.

Responsiveness (R)

This test simply records whether the target responded to a given probe. Possible values are Y and N. If there is no reply, remaining fields for the test are omitted.

A risk with this test involves probes that are dropped by a firewall. This leads to R=N in the subject fingerprint. Yet the reference fingerprint in nmap-os-db may have R=Y if the target OS usually replies. Thus the firewall could prevent proper OS detection. To reduce this problem, reference fingerprints generally omit the R=Y test from the IE and U1 probes, which are the ones most likely to be dropped. In addition, if Nmap is missing a closed TCP port for a target, it will not set R=N for the T5, T6, or T7 tests even if the port it tries is non-responsive. After all, the lack of a closed port may be because they are all filtered.

IP don't fragment bit (DF)

The IP header contains a single bit which forbids routers from fragmenting a packet. If the packet is too large for routers to handle, they will just have to drop it (and ideally return a “destination unreachable, fragmentation needed” response). This test records Y if the bit is set, and N if it isn't.

Don't fragment (ICMP) (DFI)

This is simply a modified version of the DF test that is used for the special IE probes. It compares results of the don't fragment bit for the two ICMP echo request probes sent. It has four possible values, which are enumerated in [Table 8.2](#).

Table 8.2. DFI test values

Value	Description
N	Neither of the ping responses have the DF bit set.
S	Both responses echo the DF value of the probe.
Y	Both of the response DF bits are set.
O	The one remaining other combination—both responses have the DF bit toggled.

IP initial time-to-live (T)

IP packets contain a field named time-to-live (TTL) which is decremented every time they traverse a router. If the field reaches zero, the packet must be discarded. This prevents packets from

looping endlessly. Because operating systems differ on which TTL they start with, it can be used for OS detection. Nmap determines how many hops away it is from the target by examining the ICMP port unreachable response to the U1 probe. That response includes the original IP packet, including the already-decremented TTL field, received by the target. By subtracting that value from our as-sent TTL, we learn how many hops away the machine is. Nmap then adds that hop distance to the probe response TTL to determine what the initial TTL was when that ICMP probe response packet was sent. That initial TTL value is stored in the fingerprint as the T result.

Even though an eight-bit field like TTL can never hold values greater than 0xFF, this test occasionally results in values of 0x100 or higher. This occurs when a system (could be the source, a target, or a system in between) corrupts or otherwise fails to correctly decrement the TTL. It can also occur due to asymmetric routes.

Nmap can also learn from the system interface and routing tables when the hop distance is zero (localhost scan) or one (on the same network segment). This value is used when Nmap prints the hop distance for the user, but it is not used for T result computation.

IP initial time-to-live guess (TG)

It is not uncommon for Nmap to receive no response to the U1 probe, which prevents Nmap from learning how many hops away a target is. Firewalls and NAT devices love to block unsolicited UDP packets. But since common TTL values are spread well apart and targets are rarely more than 20 hops away, Nmap can make a pretty good guess anyway. Most systems send packets with an initial TTL of 32, 60, 64, 128, or 255. So the TTL value received in the response is rounded up to the next value out of 32, 64, 128, or 255. 60 is not in that list because it cannot be reliably distinguished from 64. It is rarely seen anyway. The resulting guess is stored in the TG field. This TTL guess field is not printed in a subject fingerprint if the actual TTL (T) value was discovered.

Explicit congestion notification (CC)

This test is only used for the ECN probe. That probe is a SYN packet which includes the CWR and ECE congestion control flags. When the response SYN/ACK is received, those flags are examined to set the CC (congestion control) test value as described in [Table 8.3](#).

Table 8.3. CC test values

Value	Description
Y	Only the ECE bit is set (not CWR). This host supports ECN.
N	Neither of these two bits is set. The target does not support ECN.
S	Both bits are set. The target does not support ECN, but it echoes back what it thinks is a reserved bit.
O	The one remaining combination of these two bits (other).

TCP miscellaneous quirks (Q)

This tests for two quirks that a few implementations have in their TCP stack. The first is that the reserved field in the TCP header (right after the header length) is nonzero. This is particularly likely to happen in response to the ECN test as that one sets a reserved bit in the probe. If this is seen in a packet, an “R” is recorded in the Q string.

The other quirk Nmap tests for is a nonzero urgent pointer field value when the URG flag is not set. This is also particularly likely to be seen in response to the ECN probe, which sets a non-zero urgent field. A “U” is appended to the Q string when this is seen.

The Q string must always be generated in alphabetical order. If no quirks are present, the Q test is empty but still shown.

TCP sequence number (S)

This test examines the 32-bit sequence number field in the TCP header. Rather than record the field value as some other tests do, this one examines how it compares to the TCP acknowledgment number from the probe that elicited the response. It then records the appropriate value as shown in [Table 8.4](#).

Table 8.4. S test values

Value	Description
Z	Sequence number is zero.
A	Sequence number is the same as the acknowledgment number in the probe.
A+	Sequence number is the same as the acknowledgment number in the probe plus one.
O	Sequence number is something else (other).

TCP acknowledgment number (A)

This test is the same as S except that it tests how the acknowledgment number in the response compares to the sequence number in the respective probe. The four possible values are given in [Table 8.5](#).

Table 8.5. A test values

Value	Description
Z	Acknowledgment number is zero.
S	Acknowledgment number is the same as the sequence number in the probe.
S+	Acknowledgment number is the same as the sequence number in the probe plus one.
O	Acknowledgment number is something else (other).

TCP flags (F)

This field records the TCP flags in the response. Each letter represents one flag, and they occur in the same order as in a TCP packet (from high-bit on the left, to the low ones). So the value SA represents the SYN and ACK bits set, while the value AS is illegal (wrong order). The possible flags are shown in [Table 8.6](#).

Table 8.6. F test values

Character	Flag name	Flag byte value
E	ECN Echo (ECE)	64
U	Urgent Data (URG)	32
A	Acknowledgment (ACK)	16
P	Push (PSH)	8
R	Reset (RST)	4
S	Synchronize (SYN)	2
F	Final (FIN)	1

TCP RST data checksum (RD)

Some operating systems return ASCII data such as error messages in reset packets. This is explicitly allowed by section 4.2.2.12 of [RFC 1122](#). When Nmap encounters such data, it performs a CRC32 checksum and reports the results. When there is no data, RD is set to zero. Some of the few operating systems that may return data in

their reset packets are HP-UX and versions of Mac OS prior to Mac OS X.

IP total length (IPL)

This test records the total length (in octets) of an IP packet. It is only used for the port unreachable response elicited by the U1 test. That length varies by implementation because they are allowed to choose how much data from the original probe to include, as long as they meet the minimum [RFC 792](#) requirement. That requirement is to include the original IP header and at least eight bytes of data.

Unused port unreachable field nonzero (UN)

An ICMP port unreachable message header is eight bytes long, but only the first four are used. RFC 792 states that the last four bytes must be zero. A few implementations (mostly ethernet switches and some specialized embedded devices) set it anyway. The value of those last four bytes is recorded in this field.

Returned probe IP total length value (RIPL)

ICMP port unreachable messages (as are sent in response to the U1 probe) are required to include the IP header which generated them. This header should be returned just as they received it, but some implementations send back a corrupted version due to changes they made during IP processing. This test simply records the returned IP total length value. If the correct value of 0x148 (328) is returned, the value G (for good) is stored instead of the actual value.

Returned probe IP ID value (RID)

The U1 probe has a static IP ID value of 0x1042. If that value is returned in the port unreachable message, the value G is stored for this test. Otherwise the exact value returned is stored. Some systems, such as Solaris, manipulate IP ID values for raw IP packets that Nmap sends. In such cases, this test is skipped. We have found that some systems, particularly HP and Xerox printers, flip the bytes and return 0x4210 instead.

Integrity of returned probe IP checksum value (RIPCK)

The IP checksum is one value that we *don't* expect to remain the same when returned in a port unreachable message. After all, each network hop during transit changes the checksum as the TTL is

decremented. However, the checksum we receive should match the enclosing IP packet. If it does, the value G (good) is stored for this test. If the returned value is zero, then Z is stored. Otherwise the result is I (invalid).

Integrity of returned probe UDP checksum (RUCK)

The UDP header checksum value should be returned exactly as it was sent. If it is, G is recorded for this test. Otherwise the value actually returned is recorded.

Integrity of returned UDP data (RUD)

This test checks the integrity of the (possibly truncated) returned UDP payload. If all the payload bytes are the expected 'C' (0x43), or if the payload was truncated to zero length, G is recorded; otherwise, I (invalid) is recorded.

ICMP response code (CD)

The code value of an ICMP echo reply (type zero) packet is supposed to be zero. But some implementations wrongly send other values, particularly if the echo request has a nonzero code (as one of the IE tests does). The response code values for the two probes are combined into a CD value as described in [Table 8.7](#).

Table 8.7. CD test values

Value	Description
Z	Both code values are zero.
S	Both code values are the same as in the corresponding probe.
<N N>	When they both use the same non-zero number, it is shown here.
O	Any other combination.

Fingerprinting Methods Avoided by Nmap

Nmap supports many more OS detection techniques than any other program, and we are always interested in hearing about new ideas. Please send them to the Nmap development list (*nmap-dev*) for

discussion. However there are some methods that just aren't a good fit. This section details some of the most interesting ones. While they aren't supported by Nmap, some are useful in combination with Nmap to verify findings or learn further details.

Passive Fingerprinting

Passive fingerprinting uses most of the same techniques as the active fingerprinting performed by Nmap. The difference is that a passive system simply sniffs the network, opportunistically classifying hosts as it observes their traffic. This is more difficult than active fingerprinting, since you have to accept whatever communication happens rather than designing your own custom probes. It is a valuable technique, but doesn't belong in a fundamentally active tool such as Nmap. Fortunately, Michal Zalewski has written the excellent [p0f](#) passive OS fingerprinting tool. He also devised a couple of the current Nmap OS fingerprinting tests. Another option is [SinFP](#) by GomoR, which supports both active and passive fingerprinting.

Exploit Chronology

TCP/IP fingerprinting works well for distinguishing different operating systems, but detecting different versions of the same operating system can be troublesome. The company must change their stack in some way we can differentiate. Fortunately, many OS vendors regularly update their systems to comply with the latest standards. But what about those who don't? Most of them at least get around to fixing exploitable stack bugs eventually. And those fixes are easy to detect remotely. First send the exploit payload, be it a land attack, teardrop, ping of death, SYN flood, or WinNuke. Send one attack at a time, then immediately try to contact the system again. If it is suddenly non-responsive, you have narrowed down the OS to versions which didn't ship with the fix.



Warning

If you use denial of service (DoS) exploits as part of your OS detection suite, remember to perform those tests last.

Retransmission Times

TCP implementations have significant leeway in exactly how long they wait before retransmitting packets. The proof-of-concept tools Ring and Cron-OS are available to exploit this. They send a SYN packet to an open port, then ignore the SYN/ACK they receive rather than acknowledging it with an ACK (to complete the connection) or a RST (to kill it). The target host will resend the SYN/ACK several more times, and these tools track every subsecond of the wait. While some information can indeed be gleaned from this technique, there are several reasons that I haven't incorporated the patch into Nmap:

- It usually requires modifying the source host firewall rules to prevent your system from replying with a RST packet to the SYN/ACK it receives. That is hard to do in a portable way. And even if it was easy, many users don't appreciate applications mucking with their firewall rules.
- It can be very slow. The retransmissions can go on for several minutes. That is a long time to wait for a test that doesn't give all that much information in the first place.
- It can be inaccurate because packet drops and latency (which you have to expect in real-world environments) can lead to bogus results.

I have enumerated these reasons here because they also apply to some other proposed OS detection methods. I would love to add new tests, but they must be quick and require few packets. Messing with host firewall is unacceptable. I try to avoid making full TCP connections for stack fingerprinting, though that is done for OS detection as part of the version scanning system.

IP Fragmentation

IP fragmentation is a complex system and implementations are riddled with bugs and inconsistencies. Possible tests could examine how overlapping fragments are assembled or time the defragmentation timeouts. These tests are avoided for Nmap because many firewalls and other inline devices defragment traffic at gateways. Thus Nmap may end up fingerprinting the firewall rather than the true destination host. In addition, fragments are difficult to send on some operating systems. Linux 2.6 kernels have a tendency to queue the fragments you are trying to send and assemble them itself before transmission.

Open Port Patterns

The target host OS can often be guessed simply by looking at the ports which are open. Microsoft Windows machines often have TCP ports 135 and 139 open. Windows 2000 and newer also listen on port 445. Meanwhile, a machine running services on port 22 (ssh) and 631 (Internet Printing Protocol) is likely running Unix.

While this heuristic is often useful, it just isn't reliable enough for Nmap. Combinations of ports can be obscured by firewall rules, and most mainstream protocols are available on multiple platforms. OpenSSH servers can be [run on Windows](#), and the “Windows SMB” ports can be serviced by [Samba](#) running on a Unix machine. Port forwarding clouds the issue even further. A machine which appears to be running Microsoft IIS might be a Unix firewall simply forwarding port 80 to a Windows machine.

For these reasons, Nmap does not consider open port numbers during TCP/IP stack fingerprinting. However, Nmap can use version detection information (see [Chapter 7, Service and Application Version Detection](#)) to separately discover operating system and device type information. By keeping the OS detection results discovered by OS detection and version detection separate, Nmap can gracefully handle a Checkpoint firewall which uses TCP port forwarding to a Windows web server. The stack fingerprinting results should be “Checkpoint Firewall-1” while version detection should suggest that the OS is Windows. Keep in mind that only a small fraction of version detection signatures include OS and device type information—we can only populate these fields when the application divulges the information or when it only runs on one OS or device type.

Retired Tests

There are some tests that were once performed by Nmap, but which have been retired because they were found not to help in distinguishing operating systems and only took up space in the database. Two tests in the IE line were removed: DLI checked the length of the data payload in returned packets and SI checked the ICMP sequence numbers. They were never found to vary from the values sent. In the U1 line, the RUL test checked the length of the UDP packet returned. It was different from what was sent in only one

case out of more than 1,700. These tests were removed in March 2009.

Other tests were removed because they were *too* discriminating; they caused false differences to be measured that harmed detection accuracy. Both of these had to do with the TOS (type of service) field in responses. TOS did this for the U1 probe and TOSI did it for IE. Although the test values did legitimately differ between operating systems, often false differences were recorded because the TOS was modified by an intermediate host. Better results were had overall when these tests were removed in October 2008.

Understanding an Nmap Fingerprint

When Nmap stores a fingerprint in memory, Nmap uses a tree of attributes and values in data structures that users need not even be aware of. But there is also a special ASCII-encoded version which Nmap can print for users when a machine is unidentified. Thousands of these serialized fingerprints are also read back every time Nmap runs (with OS detection enabled) from the `nmap-os-db` database. The fingerprint format is a compromise between human comprehension and brevity. The format is so terse that it looks like line noise to many inexperienced users, but those who read this document should be able to decipher fingerprints with ease. There are actually two types of fingerprints, though they have the same general structure. The fingerprints of known operating systems that Nmap reads in are called *reference fingerprints*, while the fingerprint Nmap displays after scanning a system is a *subject fingerprint*. The reference fingerprints are a bit more complex since they can be tailored to match a whole class of operating systems by adding leeway to (or omitting) tests that aren't so reliable while allowing only a single possible value for other tests. The reference fingerprints also have OS details and classifications. Since the subject tests are simpler, we describe them first.

Decoding the Subject Fingerprint Format

If Nmap performs OS fingerprinting on a host and doesn't get a perfect OS matches despite promising conditions (such as finding both open and closed ports accessible on the target), Nmap prints a subject fingerprint that shows all of the test results that Nmap deems relevant, then asks the user to submit the data to Nmap.Org.

Tests aren't shown when Nmap has no useful results, such as when the relevant probe responses weren't received. A special line named SCAN gives extra details about the scan (such as Nmap version number) that provide useful context for integrating fingerprint submissions into nmap-os-db. A typical subject fingerprint is shown in [Example 8.3](#).

Example 8.3. A typical subject fingerprint

```
OS:SCAN(V=4.85BETA4%D=3/27%OT=22%CT=1%CU=44663%PV=N
%DS=0%G=Y%TM=49CD5E4B%P=
OS:i686-pc-linux-gnu)SEQ(SP=CB%GCD=1%ISR=CD%TI=Z%CI=Z
%II=I%TS=8)OPS(O1=M400
OS:CST11NW5%O2=M400CST11NW5%O3=M400CNNT11NW5%O4=M400CST11
NW5%O5=M400CST11NW
OS:5%O6=M400CST11)WIN(W1=8000%W2=8000%W3=8000%W4=8000%W5=
8000%W6=8000)ECN(R
OS:=Y%DF=Y%T=40%W=8018%O=M400CNNSNW5%CC=N%Q=)T1(R=Y%DF=Y
%T=40%S=O%A=S+%F=AS
OS:%RD=0%Q=)T2(R=N)T3(R=Y%DF=Y%T=40%W=8000%S=O%A=S+%F=AS
%O=M400CST11NW5%RD=
OS:0%Q=)T4(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=
%RD=0%Q=)T5(R=Y%DF=Y%T=40%W=0%S=
OS:Z%A=S+%F=AR%O=%RD=0%Q=)T6(R=Y%DF=Y%T=40%W=0%S=A%A=Z
%F=R%O=%RD=0%Q=)T7(R=
OS:Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N
%T=40%IPL=164%UN=0%R
OS:IPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N
%T=40%CD=S)
```

Now you may look at this fingerprint and immediately understand what everything means. If so, you can simply skip this section. But I have never seen such a reaction. Many people probably think some sort of buffer overflow or unterminated string error is causing Nmap to spew garbage data at them. This section helps you decode the information so you can immediately tell that blind TCP sequence prediction attacks against this machine are moderately hard, but it may make a good idle scan (-sl) zombie. The first step in understanding this fingerprint is to fix the line wrapping. The tests are all squished together, with each line wrapped at 71 characters. Then OS: is prepended to each line, raising the length to 74 characters. This makes fingerprints easy to cut and paste into the Nmap fingerprint submission form (see [the section called “When](#)

Nmap Fails to Find a Match and Prints a Fingerprint”). Removing the prefix and fixing the word wrapping (each line should end with a right parenthesis) leads to the cleaned-up version in [Example 8.4](#).

Example 8.4. A cleaned-up subject fingerprint

```
SCAN (V=4.85BETA4%D=3/27%OT=22%CT=1%CU=44663%PV=N%DS=0%G=Y
%TM=49CD5E4B%
    P=i686-pc-linux-gnu)
SEQ (SP=CB%GCD=1%ISR=CD%TI=Z%CI=Z%II=I%TS=8)
OPS (O1=M400CST11NW5%O2=M400CST11NW5%O3=M400CNNT11NW5%
    O4=M400CST11NW5%O5=M400CST11NW5%O6=M400CST11)
WIN (W1=8000%W2=8000%W3=8000%W4=8000%W5=8000%W6=8000)
ECN (R=Y%DF=Y%T=40%W=8018%O=M400CNNSNW5%CC=N%Q=)
T1 (R=Y%DF=Y%T=40%S=O%A=S+%F=AS%RD=0%Q=)
T2 (R=N)
T3 (R=Y%DF=Y%T=40%W=8000%S=O%A=S+%F=AS
%O=M400CST11NW5%RD=0%Q=)
T4 (R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T5 (R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6 (R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T7 (R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
U1 (R=Y%DF=N%T=40%IPL=164%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G
%RUD=G)
IE (R=Y%DFI=N%T=40%CD=S)
```

While this still isn't the world's most intuitive format (we had to keep it short), the format is much clearer now. Every line is a category, such as SEQ for the sequence generation tests, T3 for the results from that particular TCP probe, and IE for tests related to the two ICMP echo probes.

Following each test name is a pair of parentheses which enclose results for individual tests. The tests take the format *<testname>=<value>*. All of the possible categories, tests, and values are described in [the section called “TCP/IP Fingerprinting Methods Supported by Nmap”](#). Each pair of tests are separated by a percentage symbol (%). Tests values can be empty, leading to a percentage symbol or category-terminating right-parenthesis immediately following the equal sign. The string “O=%RD=0%Q=)” in T4 of our example shows two of these empty tests. A blank test value must match another blank value, so this empty TCP quirks Q value wouldn't match a fingerprint with Q set to RU.

In some cases, a whole test is missing rather than just its value. For example, T2 of our sample fingerprint has no W (TCP window), S (sequence number), A (acknowledgment number), T (TTL), or TG (TTL guess) tests. This is because the one test and value it does include, R=N, means that no response was returned for the T2 probe. So including a window value or sequence number would make little sense. Similarly, tests which aren't well supported on the system running Nmap are skipped. An example is the RID (IP ID field returned in ICMP packet) test, which doesn't work well on Solaris because that system tends to corrupt the ID field Nmap sends out. Tests which are inconclusive (such as failing to detect the IP ID sequence for the TI, CI, and II tests) are also omitted.

Decoding the SCAN line of a subject fingerprint

The SCAN line is a special case in a subject fingerprint. Rather than describe the target system, these tests describe various conditions of the scan. These help us integrate fingerprints submitted to Nmap.Org. The tests in this line are:

- Nmap version number (V).
- Date of scan (D) in the form month/day.
- Open and closed TCP ports (on target) used for scan (OT and CT). Unlike most tests, these are printed in decimal format. If Nmap was unable to find an open or a closed port, the test is included with an empty value (even when Nmap guesses a possibly closed port and sends a probe there).
- Closed UDP port (CU). This is the same as CT, but for UDP. Since the majority of scans don't include UDP, this test's value is usually empty.
- Private IP space (PV) is Y if the target is on the 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16 private networks ([RFC 1918](#)). Otherwise it is N.
- Network distance (DS) is the network hop distance from the target. It is 0 if the target is localhost, 1 if directly connected on an ethernet network, or the exact distance if discovered by Nmap. If the distance is unknown, this test is omitted.
- Good results (G) is Y if conditions and results seem good enough to submit this fingerprint to Nmap.Org. It is N otherwise. Unless you force them by enabling debugging (-d) or extreme verbosity (-vv), G=N fingerprints aren't printed by Nmap.
- Target MAC prefix (M) is the first six hex digits of the target MAC address, which correspond to the vendor name. Leading

zeros are not included. This field is omitted unless the target is on the same ethernet network (DS=1).

- The OS scan time (TM) is provided in Unix time_t format (in hexadecimal).
- The platform Nmap was compiled for is given in the P field.

Decoding the Reference Fingerprint Format

When Nmap scans a target to create a subject fingerprint, it then tries to match that data against the thousands of *reference fingerprints* in the nmap-os-db database. Reference fingerprints are initially formed from one or more subject fingerprints and thus have much in common. They do have a bit of extra information to facilitate matching and of course to describe the operating systems they represent. For example, the subject fingerprint we just looked at might form the basis for the reference fingerprint in [Example 8.5](#).

Example 8.5. A typical reference fingerprint

```
Fingerprint Sony PlayStation 3 game console
Class Sony | embedded || game console
SEQ(SP=F7-101%GCD=1-6%ISR=FC-106%TI=RD%TS=21)
OPS(O1=M5B4NNSNW1NNT11%O2=M5B4NNSNW1NNT11%O3=M5B4NW1NNT11%O4=M5B4NNSNW1NNT11%O5=M5B4NNSNW1NNT11%O6=M5B4NNSNNT11)
WIN(W1=FFFF%W2=FFFF%W3=FFFF%W4=FFFF%W5=FFFF%W6=FFFF)
ECN(R=Y%DF=N%T=3C-46%TG=40%W=FFFF%O=M5B4NNSNW1%CC=N%Q=)
T1(R=Y%DF=N%T=3C-46%TG=40%S=O%A=S+%F=AS%RD=0%Q=)
T2(R=Y%DF=N%T=3C-46%TG=40%W=0%S=Z%A=O|S%F=AR%O=%RD=0%Q=)
T3(R=Y%DF=N%T=3C-46%TG=40%W=FFFF%S=O%A=S+%F=AS%O=M5B4NNSNW1NNT11%RD=0%Q=)
T4(R=Y%DF=N%T=3C-46%TG=40%W=0%S=A|O%A=Z%F=R%O=%RD=0%Q=)
T5(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=O|S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=N%T=3B-45%TG=40%W=0%S=A|O%A=Z%F=R%O=%RD=0%Q=)
T7(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=O|S%F=AR%O=%RD=0%Q=)
U1(DF=N%T=FA-104%TG=FF%TOS=0%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUL=G%RUD=G)
IE(DFI=N%T=FA-104%TG=FF%TOSI=S%CD=S%SI=S%DLI=S)
```

Some differences are immediately obvious. Line wrapping is not done because that is only important for the submission process. The SCAN line is also removed, since that information describes a specific scan instance rather than general target OS characteristics.

You probably also noticed the two new lines, Fingerprint and Class, which are new to this reference fingerprint. A more subtle change is that some of the individual test results have been removed while others have been enhanced with logical expressions.

Free-form OS description (Fingerprint line)

The Fingerprint line first serves as a token so Nmap knows to start loading a new fingerprint. Each fingerprint only has one such line. Immediately after the Fingerprint token (and a space) comes a textual description of the operating system(s) represented by this fingerprint. These are in free-form English text, designed for human interpretation rather than a machine parser. Nevertheless, Nmap tries to stick with a consistent format including the vendor, product name, and then version number. Version number ranges and comma-separated alternatives discussed previously can be found in this field. Here are some examples:

```
Fingerprint HP LaserJet printer (4050, 4100, 4200, or 8150)
Fingerprint Sun Solaris 9 or 10 (SPARC)
Fingerprint Linux 2.6.22 - 2.6.24
Fingerprint Microsoft Windows Server 2003 SP1
Fingerprint Microsoft Windows XP Professional SP1
Fingerprint Minolta Di550 laser printer
```

In an ideal world, every different OS would correspond to exactly one unique fingerprint. Unfortunately, OS vendors don't make life so easy for us. The same OS release may fingerprint differently based on what network drivers are in use, user-configurable options, patch levels, processor architecture, amount of RAM available, firewall settings, and more. Sometimes the fingerprints differ for no discernible reason. While the reference fingerprint format has an expression syntax for coping with slight variations, creating multiple fingerprints for the same OS is often preferable when major differences are discovered.

Just as multiple fingerprints are often needed for one OS, sometimes a single fingerprint describes several systems. If two systems give the exact same results for every single test, Nmap has little choice but to offer up both as possibilities. This commonly occurs for several reasons. One is that vendors may release a new version of their OS without any significant changes to their IP stack. Maybe they made important changes elsewhere in the system, or perhaps

they did little but want to make a bunch of money selling “upgrades”. In these cases, Nmap often prints a range such as Apple Mac OS X 10.4.8 - 10.4.11 or Sun Solaris 9 or 10.

Another cause of duplicate fingerprints is embedded devices which share a common OS. For example, a printer from one vendor and an ethernet switch from another may actually share an embedded OS from a third vendor. In many cases, subtle differences between the devices still allow them to be distinguished. But sometimes Nmap must simply list a group of possibilities such as Cisco 1200-series WAP, HP ProCurve 2650 switch, or Xerox Phaser 7400N or 8550DT printer.

There are also cases where numerous vendors private label the exact same OEM device with their own brand name and model number. Here again, Nmap must simply list the possibilities. But distinguishing these is less important because they are all fundamentally the same device.



Tip

If the description printed by Nmap (which comes from the Fingerprint line) isn't informative enough for you, more detailed information may be available in comments above the fingerprint itself in nmap-os-db. You can find it installed on your system or look up the latest version at <http://nmap.org/data/nmap-os-db>. Search for the exact OS description that Nmap gives you. Keep in mind that there may be several Fingerprint lines with exactly the same description, so you may have to examine them all. Or use the Nmap XML output, which shows the line number of each match.

Device and OS classification (Class lines)

While the Fingerprint description works great for analysts reading Nmap output directly, many people run Nmap from other scripts and applications. Those applications might use the OS information to check for OS-specific vulnerabilities or just create a pretty graph or report.

A more structured OS classification system exists for these purposes. It is also useful when there are multiple matches. If you only get a partial fingerprint (maybe no open ports were found on the target so many tests had to be skipped), it might match dozens of different fingerprints in the nmap-os-db database. Printing the details for all of those fingerprints would be a mess. But thanks to

OS classification, Nmap can find commonality. If all of the matches are classified as Linux, Nmap will simply print that the target is a Linux box.

Every fingerprint has one or more Class lines. Each contains four well-defined fields: vendor, OS name, OS family, and device type. The fields are separated by the pipe symbol (|).

The device type is a broad classification such as router, printer, or game console and was discussed previously in this chapter. General-purpose operating systems such as Linux and Windows which can be used for just about anything are classified as general purpose.

The vendor is the company which makes an OS or device. Examples are Apple, Cisco, Microsoft, and Linksys. For community projects such as OpenBSD and Linux without a controlling vendor, the OS family name is repeated for the vendor column.

OS family includes products such as Windows, Linux, IOS (for Cisco routers), Solaris, and OpenBSD. There are also hundreds of devices such as switches, broadband routers, and printers which use undisclosed operating systems. When the underlying OS isn't clear, embedded is used.

OS generation is a more granular description of the OS. Generations of Linux include 2.4.X and 2.6.X, while Windows generations include 95, 98, Me, 2000, XP, and Vista. FreeBSD uses generations such as 4.X and 5.X. For obscure operating systems which we haven't subdivided into generations (or whenever the OS is listed simply as embedded), this field is left blank.

Each field may contain just one value. When a fingerprint represents more than one possible combination of these four fields, multiple Class lines are used. [Example 8.6](#) provides some example Fingerprint lines followed by their corresponding classifications.

Example 8.6. Some typical fingerprint descriptions and corresponding classifications

```
Fingerprint D-Link DSL-500G ADSL router
Class D-Link | embedded || broadband router

Fingerprint Linksys WRT54GC or TRENDnet TEW-431BRP WAP
Class Linksys | embedded || WAP
```

```
Class TRENDnet | embedded || WAP

Fingerprint Apple Mac OS X 10.3.9 (Panther) - 10.4.7
(Tiger)
Class Apple | Mac OS X | 10.3.X | general purpose
Class Apple | Mac OS X | 10.4.X | general purpose

Fingerprint Sony PlayStation 3 game console
Class Sony | embedded || game console
```

If these examples aren't enough, a listing of classifications recognized by the latest version of Nmap is maintained at <http://nmap.org/data/os-classes.txt>.

Test expressions

The test expressions don't have to change between a subject and reference fingerprint, but they almost always do. The reference fingerprint often needs to be generalized a little bit to match all instances of a particular OS, rather than just the machine you are scanning. For example, some Windows XP machines return a Window size of F424 to the T1 probe, while others return FAF0. This may be due to the particular ethernet device driver in use, or maybe how much memory is available. In any case, we would like to detect Windows XP no matter which window size is used.

One way to generalize a fingerprint is to simply remove tests that produce inconsistent results. Remove all of the window size tests from a reference fingerprint, and systems will match that print no matter what size they use. The downside is that you can lose a lot of important information this way. If the only Window sizes that a particular system ever sends are F424 and FAF0, you really only want to allow those two values, not all 65,536 possibilities.

While removing tests is overkill in some situations, it is useful in others. The R=Y test value, meaning there was a response, is usually removed from the U1 and IE tests before they are added to nmap-os-db. These probes are often blocked by a firewall, so the lack of a response should not count against the OS match.

When removing tests is undesirable, Nmap offers an expression syntax for allowing a test to match multiple values. For example, W=F424|FAF0 would allow those two Windows XP window values

without allowing any others. [Table 8.8](#) shows the permitted operators in test values.

Table 8.8. Reference fingerprint test expression operators

Op Name	Symbol	Example	Description
Or		O= ME MNNTNW	Matches if the corresponding subject fingerprint test takes the value of any of the clauses. In this example, the initial pipe symbol means that an empty options list will match too.
Range	-	SP=7-A	Matches if the subject fingerprint's corresponding test produces a numeric value which falls within the range specified.
Greater than	>	SP=>8	Matches if the subject fingerprint's corresponding test produces a numeric value which is greater than the one specified.
Less than	<	GCD=<5	Matches if the subject fingerprint's corresponding test produces a numeric value which is less than the one specified.

Expressions can combine operators, as in `GCD=1-6|64|256|>1024`, which matches if the GCD is between one and six, exactly 64, exactly 256, or greater than 1024.

OS Matching Algorithms

Nmap's algorithm for detecting matches is relatively simple. It takes a subject fingerprint and tests it against every single reference fingerprint in `nmap-os-db`.

When testing against a reference fingerprint, Nmap looks at each probe category line from the subject fingerprint (such as `SEQ` or `T1`) in turn. Any probe lines which do *not* exist in the reference fingerprint are skipped. When the reference fingerprint does have a matching line, they are compared.

For a probe line comparison, Nmap examines every individual test (`R`, `DF`, `W`, etc.) from the subject category line in turn. Any tests which do *not* exist in the reference line are skipped. Whenever a matching test is found, Nmap increments the `PossiblePoints` accumulator by the number of points assigned to this test. Then the test values are compared. If the reference test has an empty value, the subject test only matches if its value is empty too. If the reference test is just a plain string or number (no operators), the subject test must match it exactly. If the reference string contains

operators (|, -, >, or <), the subject must match as described in [the section called “Test expressions”](#). If a test matches, the NumMatchPoints accumulator is incremented by the test's point value.

Once all of the probe lines are tested for a fingerprint, Nmap divides NumMatchPoints by PossiblePoints. The result is a confidence factor describing the probability that the subject fingerprint matches that particular reference fingerprint. It is treated as a percentage, so 1.00 is a perfect match while 0.95 is very close.

Test point values are assigned by a special MatchPoints entry (which may only appear once) in nmap-os-db. This entry looks much like a normal fingerprint, but instead of providing results for each test, it provides point values (non-negative integers) for each test. Tests listed in the MatchPoints structure only apply when found in the same test they are listed in. So a value given for the W (Window size) test in T1 doesn't affect the W test in T3. A test can be effectively disabled by assigning it a point value of 0. An example MatchPoints structure is given in [Example 8.7](#).

Example 8.7. The MatchPoints structure

```
MatchPoints
SEQ (SP=25%GCD=75%ISR=25%TI=100%CI=50%II=100%SS=80%TS=100)
OPS (O1=20%O2=20%O3=20%O4=20%O5=20%O6=20)
WIN (W1=15%W2=15%W3=15%W4=15%W5=15%W6=15)
ECN (R=100%DF=20%T=15%TG=15%W=15%O=15%CC=100%Q=20)
T1 (R=100%DF=20%T=15%TG=15%S=20%A=20%F=30%RD=20%Q=20)
T2 (R=80%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
T3 (R=80%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
T4 (R=100%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
T5 (R=100%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
T6 (R=100%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
T7 (R=80%DF=20%T=15%TG=15%W=25%S=20%A=20%F=30%O=10%RD=20%Q=20)
U1 (R=50%DF=20%T=15%TG=15%TOS=0%IPL=100%UN=100%RIPL=100%RID=100%RIPCK=100%RUCK=100%RUL=100%RUD=100)
IE (R=50%DFI=40%T=15%TG=15%TOSI=0%CD=100%SI=100%DLI=100)
```

Once all of the reference fingerprints have been evaluated, Nmap orders them and prints the perfect matches (if there aren't too many). If there are no perfect matches, but some are very close, Nmap may print those. Guesses are more likely to be printed if the `--osscan-guess` option is given.

Dealing with Misidentified and Unidentified Hosts

While Nmap has a huge database, it cannot detect everything. Nmap has no chance to detect most toasters, refrigerators, chairs, or automobiles because they have no IP stack. Yet I wouldn't rule any of these out, given the ever-expanding list of connected devices. The Nmap fingerprint DB includes plenty of game consoles, phones, thermometers, cameras, interactive toys, and media players.

Having an IP address is necessary but not sufficient to guarantee a proper fingerprint. Nmap may still guess wrong or fail to produce any guess at all. Here are some suggestions for improving your results:

Upgrade to the latest Nmap

Many Linux distributions and other operating systems ship with ancient versions of Nmap. The Nmap OS database is improved with almost every release, so check your version number by running **nmap -V** and then compare that to the latest available from <http://nmap.org/download.html>. Installing the newest version takes only a few minutes on most platforms.

Scan all ports

When Nmap detects OS detection problems against a certain host, it will issue warnings. One of the most common is: "Warning: OS detection will be MUCH less reliable because we did not find at least 1 open and 1 closed TCP port". It is possible that such ports really are unavailable on the machine, but retrying your scan with `-p-` to scan all ports may find some

that are responsive for OS detection. Doing a UDP scan (-sU) too can help even more, though it will slow the scan substantially.

Try a more aggressive guess

If Nmap says there are no matches close enough to print, something is probably wrong. Maybe a firewall or NAT box in the way is modifying the probe or response packets. This can cause a hybrid situation where one group of tests look like they are from one OS, while another set look completely different. Adding the --osscan-guess may give more clues as to what is running.

Scan from a different location

The more network hops your packet has to go through to reach its target, the greater the chances that a network device will modify (or drop) the probe or response. NAT gateways, firewalls, and especially port forwarding can confuse OS detection. If you are scanning the IP of a load balancing device which simply redirects packets to a diverse network of servers, it isn't even clear what the "correct" OS detection result would be.

Many ISPs filter traffic to "bad" ports, and others use transparent proxies to redirect certain ports to their own servers. The port 25 or 80 you think are open on your target may actually be spoofed from your ISP to connect to ISP proxy servers. Another behavior which can confuse OS detection is when firewalls spoof TCP reset packets as if they are coming from the destination host. This is particularly common from port 113 (identd). Both the reset spoofing and transparent proxies can often be detected by noticing that every machine on a target network seems to exhibit the behavior—even those which otherwise seem to be down. If you detect any such nonsense, be sure to exclude these ports from your scan so they don't taint your results. You may also want to try from a completely different network location. The closer you are to the target, the more accurate the results will be. In a perfect case, you would always scan the target from the same network segment it resides on.

When Nmap Guesses Wrong

Occasionally Nmap will report an OS guess which you know is wrong. The errors are usually minor (such as reporting a machine running Linux 2.4.16 as “Linux kernel 2.4.8 - 2.4.15”), but there have been reports of Nmap being completely off (such as reporting your web server as an AppleWriter printer). When you encounter such problems (minor or major), please report them so everyone can benefit. The only reason the Nmap DB is so comprehensive is that thousands of users have spent a few minutes each to submit new information. Please follow these instructions:

Have a recent version of Nmap

Run **nmap -V** to determine which version of Nmap you have. You don't need to be running the absolute latest version of Nmap (though that would be ideal), but make sure your version is 4.20 or higher because we only need second generation OS fingerprints, not the old style produced by previous versions. You can determine the latest available version of Nmap by visiting <http://nmap.org/download.html>. If you upgrade, you might find that the identification has already been fixed.

Be absolutely certain you know what is running

Invalid “corrections” can corrupt the OS DB. If you aren't certain exactly what is running on the remote machine, please find out before submitting.

Generate a fingerprint

Run the command **nmap -O -sSU -F -T4 -d <target>**, where **<target>** is the misidentified system in question. Look at the OS detection results to ensure that the misidentification is still present.

If the Nmap output for the host OS results says (JUST GUESSING), it is expected that results may be a little off. Don't submit a correction in this case.

Otherwise, the map command should have produced results including the line OS fingerprint:. Below that is the fingerprint (a series of lines which each start with OS:).

Check that OS detection works against other hosts

Try scanning a couple other hosts on the target network which you know have a different OS. If they aren't detected properly, maybe there is some network obstruction between the systems which is corrupting the packets.

If you have gotten this far and are still able to submit, good for you! Please submit the information at <http://insecure.org/cgi-bin/submit.cgi?corr-os>

When Nmap Fails to Find a Match and Prints a Fingerprint

When Nmap detects that OS detection conditions seem ideal and yet it finds no exact matches, it will print out a message like this:

```
No OS matches for host (If you know what OS is running on
it, see
http://nmap.org/submit/ ).
TCP/IP fingerprint:
OS:SCAN (V=4.85BETA4%D=3/27%OT=22%CT=7%CU=37400%PV=N
%DS=0%G=Y%TM=49CD5F29%P=
OS:i686-pc-linux-gnu) SEQ (SP=CE%GCD=1%ISR=D1%TI=Z%CI=Z
%II=I%TS=8) OPS (O1=M400
OS:CST11NW5%O2=M400CST11NW5%O3=M400CNNT11NW5%O4=M400CST11
NW5%O5=M400CST11NW
OS:5%O6=M400CST11) WIN (W1=8000%W2=8000%W3=8000%W4=8000%W5=
8000%W6=8000) ECN (R
OS:=Y%DF=Y%T=40%W=8018%O=M400CNNSNW5%CC=N%Q=) T1 (R=Y%DF=Y
%T=40%S=O%A=S+%F=AS
OS:%RD=0%Q=) T2 (R=N) T3 (R=Y%DF=Y%T=40%W=8000%S=O%A=S+%F=AS
%O=M400CST11NW5%RD=
OS:0%Q=) T4 (R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=
%RD=0%Q=) T5 (R=Y%DF=Y%T=40%W=0%S=
OS:Z%A=S+%F=AR%O=%RD=0%Q=) T6 (R=Y%DF=Y%T=40%W=0%S=A%A=Z
%F=R%O=%RD=0%Q=) T7 (R=
```

```
OS:Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N
%T=40%IPL=164%UN=0%R
OS:IPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N
%T=40%CD=S)
```

Please consider submitting the fingerprint so that all Nmap users can benefit. It only takes a minute or two and it may mean you don't need to see that ugly message again when you scan the host with the next Nmap version! Simply visit the URL Nmap provides for instructions.

If Nmap finds no matches and yet prints no fingerprint, conditions were not ideal. Even if you obtain the fingerprint through debug mode or XML output, please don't submit it unless Nmap asks you to (as in the previous example).

Modifying the nmap-os-db Database Yourself

People often ask about integrating a fingerprint themselves rather than (or in addition to) submitting it to Nmap.Org. While we don't offer detailed instructions or scripts for this, it is certainly possible after you become intimately familiar with [the section called “Understanding an Nmap Fingerprint”](#). I hope this is useful for your purposes, but there is no need to send your own reference fingerprint creations to us. We can only integrate raw subject fingerprint submissions from the web form.

Chapter 9. Nmap Scripting Engine

Table of Contents

[Introduction](#)

[Usage and Examples](#)

[Script Categories](#)

[Command-line Arguments](#)

[Script Selection](#)

[Arguments to Scripts](#)

[Complete Examples](#)

[Script Format](#)

[description Field](#)

[categories Field](#)

[author Field](#)

- [license Field](#)
 - [runlevel Field](#)
 - [Port and Host Rules](#)
 - [Action](#)
- [Script Language](#)
 - [Lua Base Language](#)
- [NSE Scripts](#)
- [NSE Libraries](#)
 - [List of All Libraries](#)
 - [Adding C Modules to Nselib](#)
- [Nmap API](#)
 - [Information Passed to a Script](#)
 - [Network I/O API](#)
 - [Connect-style network I/O](#)
 - [Raw packet network I/O](#)
 - [Thread Mutexes](#)
 - [Exception Handling](#)
 - [The Registry](#)
- [Script Writing Tutorial](#)
 - [The Head](#)
 - [The Rule](#)
 - [The Mechanism](#)
- [Writing Script Documentation \(NSEDoc\)](#)
 - [NSE Documentation Tags](#)
- [Version Detection Using NSE](#)
- [Example Script: finger.nse](#)
- [Implementation Details](#)
 - [Initialization Phase](#)
 - [Matching Scripts with Targets](#)
 - [Script Execution](#)

Introduction

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. Those scripts are then executed in parallel with the speed and efficiency you expect from Nmap. Users can rely on the growing and diverse set of scripts distributed with Nmap, or write their own to meet custom needs.

We designed NSE to be versatile, with the following tasks in mind:

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.

More sophisticated version detection

The Nmap version detection system ([Chapter 7, Service and Application Version Detection](#)) is able to recognize thousands of different services through its probe and regular expression signature based matching system, but it cannot recognize everything. For example, identifying the Skype v2 service requires two independent probes, which version detection isn't flexible enough to handle. Nmap could also recognize more SNMP services if it tried a few hundred different community names by brute force. Neither of these tasks are well suited to traditional Nmap version detection, but both are easily accomplished with NSE. For these reasons, version detection now calls NSE by default to handle some tricky services. This is described in [the section called "Version Detection Using NSE"](#).

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 we 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. For example, within hours of the MyDoom worm hitting the Internet, Jay Moran posted an Nmap version detection probe and signature so that others could quickly scan their networks

for MyDoom infections. NSE is needed to reliably detect more complex worms and backdoors.

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 we aren't planning to turn Nmap into an exploitation framework such as [Metasploit](#).

These listed items were our initial goals, and we expect Nmap users to come up with even more inventive uses for NSE.

Scripts are written in the embedded [Lua programming language](#). The language itself is well documented in the books [Programming in Lua, Second Edition](#) and [Lua 5.1 Reference Manual](#). The reference manual is also [freely available online](#), as is the [first edition of Programming in Lua](#). Given the availability of these excellent general Lua programming references, this document only covers aspects and extensions specific to Nmap's scripting engine.

NSE is activated with the `-sC` option (or `--script` if you wish to specify a custom set of scripts) and results are integrated into Nmap normal and XML output. Two types of scripts are supported: service and host scripts. Service scripts relate to a certain open port (service) on the target host, and any results they produce are included next to that port in the Nmap output port table. Host scripts, on the other hand, run no more than once against each target IP and produce results below the port table. [Example 9.1](#) shows a typical script scan. Service scripts producing output in this example are `ssh-hostkey`, which provides the system's RSA and DSA SSH keys, and `rpcinfo`, which queries portmapper to enumerate available services. The only host script producing output in this example is `smb-os-discovery`, which collects a variety of information from SMB servers. Nmap discovered all of this information in a third of a second.

Example 9.1. Typical NSE output

```
# nmap -sC -p22,111,139 -T4 localhost

Starting Nmap ( http://nmap.org )
Interesting ports on flog (127.0.0.1):
```

```
PORT      STATE SERVICE
22/tcp    open  ssh
|  ssh-hostkey: 1024
b1:36:0d:3f:50:dc:13:96:b2:6e:34:39:0d:9b:1a:38 (DSA)
|_ 2048 77:d0:20:1c:44:1f:87:a0:30:aa:85:cf:e8:ca:4c:11
(RSA)
111/tcp    open  rpcbind
|  rpcinfo:
|  100000 2,3,4    111/udp  rpcbind
|  100024 1        56454/udp status
|_ 100000 2,3,4    111/tcp  rpcbind
139/tcp    open  netbios-ssn

Host script results:
|  smb-os-discovery: Unix
|  LAN Manager: Samba 3.0.31-0.fc8
|_ Name: WORKGROUP

Nmap done: 1 IP address (1 host up) scanned in 0.33
seconds
```

Usage and Examples

While NSE has a complex implementation for efficiency, it is strikingly easy to use. Simply specify `-sC` to enable the most common scripts. Or specify the `--script` option to choose your own scripts to execute by providing categories, script file names, or the name of directories full of scripts you wish to execute. You can customize some scripts by providing arguments to them via the `--script-args` option. The two remaining options, `--script-trace` and `--script-updatedb`, are generally only used for script debugging and development. Script scanning is also included as part of the `-A` (aggressive scan) option.

Scripts are not run in a sandbox and thus could accidentally or maliciously damage your system or invade your privacy. Never run scripts from third parties unless you trust the authors or have carefully audited the scripts yourself.

Script Categories

NSE scripts define a list of categories they belong to. Currently defined categories are auth, default, discovery, external, intrusive, malware, safe, version, and vuln. Category names are not case sensitive. The following list describes each category.

auth

These scripts try to determine authentication credentials on the target system, often through a brute-force attack. Examples include snmp-brute, http-auth, and ftp-anon.

default

These scripts are the default set and are run when using the -sC or -A options rather than listing scripts with --script. This category can also be specified explicitly like any other using --script=default. Many factors are considered in deciding whether a script should be run by default:

Speed

A default scan must finish quickly, which excludes brute force authentication crackers, web spiders, and any other scripts which can take minutes or hours to scan a single service.

Usefulness

Default scans need to produce valuable and actionable information. If even the script author has trouble explaining why an average networking or security professional would find the output valuable, the script should not run by default. The script may still be worth including in Nmap so that administrators can run for those occasions when they do need the extra information.

Verbosity

Nmap output is used for a wide variety of purposes and needs to be readable and concise. A script which frequently produces pages full of output should not be added to the default category. When there is no important information to report, NSE scripts (particularly default ones) should return nothing. Checking for an obscure vulnerability may be OK by default as long as it only produces output when that vulnerability is discovered.

Reliability

Many scripts use heuristics and fuzzy signature matching to reach conclusions about the target host or service. Examples include sniffer-detect and sql-injection. If the script is often wrong, it doesn't belong in the default category where it may confuse or mislead casual users. Users who specify a script or category directly are generally more advanced and likely know how the script works or at least where to find its documentation.

Intrusiveness

Some scripts are very intrusive because they use significant resources on the remote system, are likely to crash the system or service, or are likely to be perceived as an attack by the remote administrators. The more intrusive a script is, the less suitable it is for the default category.

Privacy

Some scripts, particularly those in the external category described later, divulge information to third parties by their very nature. For example, the whois script must divulge the target IP address to regional whois registries. We have also considered (and decided against) adding scripts which check target SSH and SSL key fingerprints against Internet weak key databases. The more privacy-invasive a script is, the less suitable it is for default category inclusion.

We don't have exact thresholds for each of these criteria, and many of them are subjective. All of these factors are considered together when making a decision whether to promote a script into the default category. A few default scripts are identd-owners (determines the username running remote services using identd), http-auth (obtains authentication scheme and realm of web sites requiring authentication), and ftp-anon (tests whether an FTP server allows anonymous access).

discovery

These scripts try to actively discover more about the network by querying public registries, SNMP-enabled devices, directory

services, and the like. Examples include `html-title` (obtains the title of the root path of web sites), `smb-enum-shares` (enumerates Windows shares), and `snmp-sysdescr` (extracts system details via SNMP).

external

Scripts in this category may send data to a third-party database or other network resource. An example of this is `whois`, which makes a connection to whois servers to learn about the address of the target. There is always the possibility that operators of the third-party database will record anything you send to them, which in many cases will include your IP address and the address of the target. Most scripts involve traffic strictly between the scanning computer and the client; any that do not are placed in this category.

intrusive

These are scripts that cannot be classified in the safe category because the risks are too high that they will crash the target system, use up significant resources on the target host (such as bandwidth or CPU time), or otherwise be perceived as malicious by the target's system administrators. Examples are `http-open-proxy` (which attempts to use the target server as an HTTP proxy) and `snmp-brute` (which tries to guess a device's SNMP community string by sending common values such as `public`, `private`, and `cisco`).

malware

These scripts test whether the target platform is infected by malware or backdoors. Examples include `smtp-strangeport`, which watches for SMTP servers running on unusual port numbers, and `auth-spoof`, which detects `identd` spoofing daemons which provide a fake answer before even receiving a query. Both of these behaviors are commonly associated with malware infections.

safe

Scripts which weren't designed to crash services, use large amounts of network bandwidth or other resources, or exploit security holes are categorized as safe. These are less likely to

offend remote administrators, though (as with all other Nmap features) we cannot guarantee that they won't ever cause adverse reactions. Most of these perform general network discovery. Examples are `ssh-hostkey` (retrieves an SSH host key) and `html-title` (grabs the title from a web page).

version

The scripts in this special category are an extension to the version detection feature and cannot be selected explicitly. They are selected to run only if version detection (`-sV`) was requested. Their output cannot be distinguished from version detection output and they do not produce service or host script results. Examples are `skypev2-version`, `pptp-version`, and `iax2-version`.

vuln

These scripts check for specific known vulnerabilities and generally only report results if they are found. Examples include `realvnc-auth-bypass` and `xampp-default-auth`.

Command-line Arguments

These are the five command line arguments specific to script-scanning:

-sC

Performs a script scan using the default set of scripts. It is equivalent to `--script=default`. Some of the scripts in this default category are considered intrusive and should not be run against a target network without permission.

`--script <filename>|<category>|<directory>|<expression>|all[,...]`

Runs a script scan using the comma-separated list of filenames, script categories, and directories. Each element in the list may also be a Boolean expression describing a more complex set of scripts. Each element is interpreted first as an expression, then as a category, and finally as a file or directory name. The special argument `all` makes every script in Nmap's script database eligible to run.

File and directory names may be relative or absolute. Absolute names are used directly. Relative paths are looked for in the following places until found:

```
--datadir
$NMAPDIR
~/.nmap    (not searched on
Windows)
NMAPDATADIR
the current directory
```

A scripts subdirectory is also tried in each of these.

When a directory name is given, Nmap loads every file in the directory whose name ends with .nse. All other files are ignored and directories are not searched recursively. When a filename is given, it does not have to have the .nse extension; it will be added automatically if necessary.

See [the section called “Script Selection”](#) for examples and a full explanation of the --script option.

Nmap scripts are stored in a scripts subdirectory of the Nmap data directory by default (see [Chapter 14, Understanding and Customizing Nmap Data Files](#)). For efficiency, scripts are indexed in a database stored in scripts/script.db, which lists the category or categories in which each script belongs. Give the argument all to execute all scripts in the Nmap script database.

--script-args <args>

Provides arguments to the scripts. See [the section called “Arguments to Scripts”](#) for a detailed explanation.

--script-trace

This option is similar to --packet-trace, but works at the application level rather than packet by packet. If this option is specified, all incoming and outgoing communication performed by scripts is printed. The displayed information includes the communication protocol, source and target addresses, and the transmitted data. If more than 5% of transmitted data is unprintable, hex dumps are given instead. Specifying --packet-trace enables script tracing too.

`--script-updatedb`

This option updates the script database found in `scripts/script.db` which is used by Nmap to determine the available default scripts and categories. It is only necessary to update the database if you have added or removed NSE scripts from the default scripts directory or if you have changed the categories of any script. This option is used by itself without arguments: **`nmap --script-updatedb`**.

Some other Nmap options have effects on script scans. The most prominent of these is `-sV`. A version scan automatically executes the scripts in the version category. The scripts in this category are slightly different than other scripts because their output blends in with the version scan results and they do not produce any script scan output.

Another option which affects the scripting engine is `-A`. The aggressive Nmap mode implies the `-sC` option.

Script Selection

The `--script` option takes a comma-separated list of categories, filenames, and directory names. Some simple examples of its use:

`nmap --script default,safe`

Loads all scripts in the default and safe categories.

`nmap --script smb-os-discovery`

Loads only the `smb-os-discovery.nse` script. Note that the `.nse` extension is optional.

`nmap --script default,banner,/home/user/customscripts`

Loads the script in the default category, the `banner.nse` script, and all `.nse` files in the directory `/home/user/customscripts`.

`nmap --script all`

Loads every script in `script.db`.

When referring to scripts from script.db by name, you can use a shell-style '*' wildcard.

nmap --script "http-*

Loads all scripts whose name starts with http-, such as http-auth.nse and http-open-proxy.nse. The argument to --script had to be in quotes to protect the wildcard from the shell.

More complicated script selection can be done using the and, or, and not operators to build Boolean expressions. The operators have the same [precedence](#) as in Lua: not is the highest, followed by and and then or. You can alter precedence by using parentheses. Because expressions contain space characters it is necessary to quote them.

nmap --script "not intrusive"

Loads every script except for those in the intrusive category.

nmap --script "default or safe"

This is functionally equivalent to **nmap --script "default,safe"**. It loads all scripts that are in the default category or the safe category or both.

nmap --script "default and safe"

Loads those scripts that are in *both* the default and safe categories.

nmap --script "(default or safe or intrusive) and not http-*

Loads scripts in the default, safe, or intrusive categories, except for those whose names start with http-.

Names in a Boolean expression may be a category, a filename from script.db, or all. A name is any sequence of characters not containing ' ', ',', '(', ')', or ';', except for the sequences and, or, and not, which are operators.

Arguments to Scripts

Arguments may be passed to NSE scripts using the `--script-args` option. The arguments describe a table of key-value pairs and possibly array values. The arguments are provided to scripts as a table in the registry called `nmap.registry.args`.

The syntax for script arguments is similar to Lua's table constructor syntax. Arguments are a comma-separated list of `name=value` pairs. Names and values may be strings not containing whitespace or the characters `'{'`, `'}'`, `'='`, or `'.'`. To include one of these characters in a string, enclose the string in single or double quotes. Within a quoted string, `'\'` escapes a quote. A backslash is only used to escape quotation marks in this special case; in all other cases a backslash is interpreted literally.

Values may also be tables enclosed in `{}`, just as in Lua. A table may contain simple string values, for example a list of proxy hosts; or more name-value pairs, including nested tables. Nested subtables are commonly used to pass arguments specific to one script, in a table named after the script. That is what is happening with the `whois` table in the example below.

Here is a typical Nmap invocation with script arguments:

```
nmap -sC --script-args user=foo,pass=',  
{}=bar',whois={whodb=nofollow+ripe},userdb=C:\Some\Pa  
th\To\File
```

That command results in this Lua table:

```
{user="foo",pass=",  
{}=bar",whois={whodb="nofollow+ripe"},userdb="C:\\Some\\P  
ath\\To\\File"}
```

You could then access the username "foo" inside your script with this statement:

```
local username = nmap.registry.args.user
```

The online NSE Documentation Portal at <http://nmap.org/nsedoc/> lists the arguments that each script accepts.

Complete Examples

```
nmap -sC example.com
```

A simple script scan using the default set of scripts.

```
nmap --script smb-os-discovery --script-trace example.com
```

Execute a specific script with script tracing.

```
nmap --script snmp-sysdescr --script-args snmpcommunity=admin example.com
```

Run an individual script that takes a script argument.

```
nmap --script mycustomscripts,safe example.com
```

Execute all scripts in the mycustomscripts directory as well as all scripts in the safe category.

Script Format

NSE scripts consist of two–five descriptive fields along with either a port or host rule defining when the script should be executed and an action block containing the actual script instructions. Values can be assigned to the descriptive fields just as you would assign any other Lua variables. Their names must be lowercase as shown in this section.

description Field

The description field describes what a script is testing for and any important notes the user should be aware of. Depending on script complexity, the description may vary from a few sentences to a few paragraphs. The first paragraph should be a brief synopsis of the script function suitable for stand-alone presentation to the user. Further paragraphs may provide much more script detail.

categories Field

The categories field defines one or more categories to which a script belongs (see [the section called “Script Categories”](#)). The categories are case-insensitive and may be specified in any order. They are listed in an array-style Lua table as in this example:

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

author Field

The author field contains the script authors' names and contact information. If you are worried about spam, feel free to omit or obscure your email address, or give your home page URL instead. This optional field is not used by NSE, but gives script authors due credit or blame.

license Field

Nmap is a community project and we welcome all sorts of code contributions, including NSE scripts. So if you write a valuable script, don't keep it to yourself! The optional license field helps ensure that we have legal permission to distribute all the scripts which come with Nmap. All of those scripts currently use the standard Nmap license (described in [the section called "Nmap Copyright and Licensing"](#)). They include the following line:

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

The Nmap license is similar to the GNU GPL. Script authors may use a BSD-style license (no advertising clause) instead if they prefer that.

runlevel Field

This optional field determines script execution order. When this section is absent, the run level defaults to 1.0. Scripts with a given runlevel execute after any with a lower runlevel and before any scripts with a higher runlevel against a single target machine. The order of scripts with the same runlevel is undefined and they often run concurrently. One application of run levels is allowing scripts to depend on each other. If script A relies on some information gathered by script B, give B a lower run level than A. Script B can store information in the NSE registry for A to retrieve later. For information on the NSE registry, see [the section called "The Registry"](#).

Port and Host Rules

Nmap uses the script rules to determine whether a script should be run against a target. A script contains either a *port rule*, which governs which ports of a target the scripts may run against, or a *host rule*, which specifies that the script should be run only once against a target IP and only if the given conditions are met. A rule is a Lua function that returns either true or false. The script *action* is only performed if its rule evaluates to true. Host rules accept a host table as their argument and may test, for example, the IP address or hostname of the target. A port rule accepts both host and port tables as arguments for any TCP or UDP port in the open, open|filtered, or unfiltered port states. Port rules generally test factors such as the port number, port state, or listening service name in deciding whether to run against a port. Example rules are shown in [the section called “The Rule”](#).

Action

The action is the heart of an NSE script. It contains all of the instructions to be executed when the script's port or host rule triggers. It is a Lua function which accepts the same arguments as the rule and can return either nil or a string. If a string is returned by a service script, the string and script's filename are printed in the Nmap port table output. A string returned by a host script is printed below the port table. No output is produced if the script returns nil. For an example of an NSE action refer to [the section called “The Mechanism”](#).

Script Language

The core of the Nmap Scripting Engine is an embeddable Lua interpreter. Lua is a lightweight language designed for extensibility. It offers a powerful and well documented API for interfacing with other software such as Nmap.

The second part of the Nmap Scripting Engine is the NSE Library, which connects Lua and Nmap. This layer handles issues such as initialization of the Lua interpreter, scheduling of parallel script execution, script retrieval and more. It is also the heart of the NSE network I/O framework and the exception handling mechanism. It also includes utility libraries to make scripts more powerful and convenient. The utility library modules and extensions are described in [the section called “NSE Libraries”](#).

Lua Base Language

The Nmap scripting language is an embedded [Lua](#) interpreter which was extended with libraries for interfacing with Nmap. The Nmap API is in the Lua namespace `nmap`. This means that all calls to resources provided by Nmap have an `nmap` prefix. `nmap.new_socket()`, for example, returns a new socket wrapper object. The Nmap library layer also takes care of initializing the Lua context, scheduling parallel scripts and collecting the output produced by completed scripts.

During the planning stages, we considered several programming languages as the base for Nmap scripting. Another option was to implement a completely new programming language. Our criteria were strict: NSE had to be easy to use, small in size, compatible with the Nmap license, scalable, fast and parallelizable. Several previous efforts (by other projects) to design their own security auditing language from scratch resulted in awkward solutions, so we decided early not to follow that route. First the Guile Scheme interpreter was considered, but the preference drifted towards the Elk interpreter due to its more favorable license. But parallelizing Elk scripts would have been difficult. In addition, we expect that most Nmap users prefer procedural programming over functional languages such as Scheme. Larger interpreters such as Perl, Python, and Ruby are well-known and loved, but are difficult to embed efficiently. In the end, Lua excelled in all of our criteria. It is small, distributed under the liberal MIT open source license, has coroutines for efficient parallel script execution, was designed with embeddability in mind, has excellent documentation, and is actively developed by a large and committed community. Lua is now even embedded in other popular open source security tools including the Wireshark sniffer and Snort IDS.

NSE Scripts

This section (a long list of NSE scripts with brief summaries) is only provided in the printed edition of this book because we already provide a better online interface to the information at the [NSE Documentation Portal](#).

NSE Libraries

In addition to the significant built-in capabilities of Lua, we have written or integrated many extension libraries which make script writing more powerful and convenient. These libraries (sometimes called modules) are compiled if necessary and installed along with Nmap. They have their own directory, `nselib`, which is installed in the configured Nmap data directory. Scripts need only **require** the default libraries in order to use them.

List of All Libraries

This list is just an overview to give an idea of what libraries are available. Developers will want to consult the complete documentation at <http://nmap.org/nsedoc/>.

base64

Base64 encoding and decoding. Follows RFC 4648.

bin

Pack and unpack binary data.

bit

Bitwise operations on integers.

comm

Common communication functions for network discovery tasks like banner grabbing and data exchange.

datafiles

Read and parse some of Nmap's data files: `nmap-protocols`, `nmap-rpc`, and `nmap-services`.

dns

Simple DNS library supporting packet creation, encoding, decoding, and querying.

http

Client-side HTTP library.

imap

IMAP functions.

ipOps

Utility functions for manipulating and comparing IP addresses.

listop

Functional-style list operations.

match

Buffered network I/O helper functions.

msrpc

By making heavy use of the 'smb' library, this library will call various MSRPC functions. The functions used here can be accessed over TCP ports 445 and 139, with an established session. A NULL session (the default) will work for some functions and operating systems (or configurations), but not for others.

msrpcperformance

This module is designed to parse the PERF_DATA_BLOCK structure, which is stored in the registry under HKEY_PERFORMANCE_DATA. By querying this structure, you can get a whole lot of information about what's going on.

msrpctypes

This module was written to marshall parameters for Microsoft RPC (MSRPC) calls. The values passed in and out are based on structs defined by the protocol, and documented by Samba developers. For detailed breakdowns of the types, take a look at Samba 4.0's .idl files.

nethbios

Creates and parses NetBIOS traffic. The primary use for this is to send NetBIOS name requests.

nmap

Interface with Nmap internals.

nsedebug

Converts an arbitrary data type into a string. Will recursively convert tables. This can be very useful for debugging.

openssl

OpenSSL bindings.

packet

Facilities for manipulating raw packets.

pcre

Perl Compatible Regular Expressions.

pop3

POP3 functions.

shortport

Functions for building short portrules.

smb

Implements functionality related to Server Message Block (SMB, also known as CIFS) traffic, which is a Windows protocol.

smbauth

This module takes care of the authentication used in SMB (LM, NTLM, LMv2, NTLMv2). There is a lot to this functionality, so if you're interested in how it works, read on.

snmp

SNMP functions.

ssh1

Functions for the SSH-1 protocol.

ssh2

Functions for the SSH-2 protocol.

stdnse

Standard Nmap Scripting Engine functions.

strbuf

String buffer facilities.

tab

Arrange output into tables.

unpwdb

Username/password database library.

url

URI parsing, composition, and relative URL resolution.

Adding C Modules to Nselib

A few of the modules included in nselib are written in C or C++ rather than Lua. Two examples are `bit` and `pcre`. We recommend that modules be written in Lua if possible, but C and C++ may be more appropriate if performance is critical or (as with the `pcre` and `openssl` modules) you are linking to an existing C library. This section describes how to write your own compiled extensions to nselib.

The Lua C API is described at length in [Programming in Lua, Second Edition](#), so this is a short summary. C modules consist of functions that follow the protocol of the `lua_CFunction` type. The functions are registered with Lua and assembled into a library by calling the `luaL_register` function. A special initialization function provides the interface between the module and the rest of the NSE code. By convention the initialization function is named in the form `luaopen_<module>`.

The smallest compiled module that comes with NSE is `bit`, and one of the most straightforward is `openssl`. These modules serve as good examples for a beginning module writer. The source code for `bit` is found in `nse_bit.cc` and `nse_bit.h`, while the `openssl` source is in `nse_openssl.cc` and `nse_openssl.h`. Most of the other compiled modules follow this `nse_<module name>.cc` naming convention.

Reviewing the `openssl` module shows that one of the functions in `nse_openssl.cc` is `l_md5`, which calculates an MD5 digest. Its function prototype is:

```
static int l_md5(lua_State *L);
```

The prototype shows that `l_md5` matches the `lua_CFunction` type. The function is static because it does not have to be visible to other compiled code. Only an address is required to register it with Lua. Later in the file, `l_md5` is entered into an array of type `luaL_reg` and associated with the name `md5`:

```
static const struct luaL_reg openssllib[] = {  
    { "md5", l_md5 },  
    { NULL, NULL }  
};
```

This function will now be known as `md5` to NSE. Next the library is registered with a call to `luaL_register` inside the initialization function `luaopen_openssl`, as shown next. Some lines relating to the registration of OpenSSL BIGNUM types have been omitted:

```
LUALIB_API int luaopen_openssl(lua_State *L) {  
    luaL_register(L, OPENSLLIBNAME, openssllib);  
    return 1;  
}
```

The function `luaopen_openssl` is the only function in the file that is exposed in `nse_openssl.h`. `OPENSLLIBNAME` is simply the string `"openssl"`.

After a compiled module is written, it must be added to NSE by including it in the list of standard libraries in `nse_init.cc`. Then the module's source file names must be added to `Makefile.in` in the appropriate places. For both these tasks you can simply follow the example of the other C modules. For the Windows build, the new source files must be added to the `mswin32/nmap.vcproj` project file

using MS Visual Studio (see [the section called “Compile from Source Code”](#)).

Nmap API

NSE scripts have access to several Nmap facilities for writing flexible and elegant scripts. The API provides target host details such as port states and version detection results. It also offers an interface to the Nsock library for efficient network I/O.

Information Passed to a Script

An effective Nmap scripting engine requires more than just a Lua interpreter. Users need easy access to the information Nmap has learned about the target hosts. This data is passed as arguments to the NSE script's action method. The arguments, host and port, are Lua tables which contain information on the target against which the script is executed. If a script matched a hostrule, it gets only the host table, and if it matched a portrule it gets both host and port. The following list describes each variable in these two tables.

host

This table is passed as a parameter to the rule and action functions. It contains information on the operating system run by the host (if the -O switch was supplied), the IP address and the host name of the scanned target.

host.os

The os entry in the host table is an array of strings. The strings (as many as eight) are the names of the operating systems the target is possibly running. Strings are only entered in this array if the target machine is a perfect match for one or more OS database entries. If Nmap was run without the -O option, then host.os is nil.

host.ip

Contains a string representation of the IP address of the target host. If the scan was run against a host name and the reverse DNS query returned more than one IP addresses then the same IP address is used as the one chosen for the scan.

host.name

Contains the reverse DNS entry of the scanned target host represented as a string. If the host has no reverse DNS entry, the value of the field is an empty string.

host.targetname

Contains the name of the host as specified on the command line. If the target given on the command line contains a netmask or is an IP address the value of the field is nil.

host.directly_connected

A Boolean value indicating whether or not the target host is directly connected to (i.e. on the same network segment as) the host running Nmap.

host.mac_addr

MAC address of the destination host (six-byte long binary string) or nil, if the host is not directly connected.

host.mac_addr_src

Our own MAC address, which was used to connect to the host (either our network card's, or (with --spoof-mac) the spoofed address).

host.interface

A string containing the interface name (dnet-style) through which packets to the host are sent.

host.bin_ip

The target host's IPv4 address as a 32-bit binary value.

host.bin_ip_src

Our host's (running Nmap) source IPv4 address as a 32-bit binary value.

port

The port table is passed to an NSE service script (i.e. only those with a portrule rather than a hostrule) in the same fashion as the host table. It contains information about the port against which the script is running. While this table is not passed to host scripts, port states on the target can still be requested from Nmap using the `nmap.get_port_state()` call.

`port.number`

Contains the port number of the target port.

`port.protocol`

Defines the protocol of the target port. Valid values are "tcp" and "udp".

`port.service`

Contains a string representation of the service running on `port.number` as detected by the Nmap service detection. If the `port.version` field is nil, Nmap has guessed the service based on the port number. Otherwise version detection was able to determine the listening service and this field is equal to `port.version.name`.

`port.version`

This entry is a table which contains information retrieved by the Nmap version scanning engine. Some of the values (such as service name, service type confidence, and the RPC-related values) may be retrieved by Nmap even if a version scan was not performed. Values which were not determined default to nil. The meaning of each value is given in the following table:

Table 9.1. port.version values

Name	Description
<code>name</code>	Contains the service name Nmap decided on for the port.
<code>name_confidence</code>	Evaluates how confident Nmap is about the accuracy of <code>name</code> , from 1 (least confident) to 10.
<code>product</code> , <code>version</code> , <code>extrainfo</code> , <code>hostname</code> , <code>ostype</code> , <code>devicetype</code>	These five variables are the same as those described under <code><versioninfo></code> in the section called “match Directive” .
<code>service_tunnel</code>	Contains the string "none" or "ssl" based on whether or

Name	Description
	not Nmap used SSL tunneling to detect the service.
service_fp	The service fingerprint, if any, is provided in this value. This is described in the section called “Community Contributions” .
rpc_status	Contains a string value of good_prog if we were able to determine the program number of an RPC service listening on the port, unknown if the port appears to be RPC but we couldn't determine the program number, not_rpc if the port doesn't appear be RPC, or untested if we haven't checked for RPC status.
rpc_program, rpc_lowver, rpc_highver	The detected RPC program number and the range of version numbers supported by that program. These will be nil if rpc_status is anything other than good_prog .

port.state

Contains information on the state of the port. Service scripts are only run against ports in the open or open|filtered states, so port.state generally contains one of those values. Other values might appear if the port table is a result of the get_port_state function. You can adjust the port state using the nmap.set_port_state() call. This is normally done when an open|filtered port is determined to be open.

Network I/O API

To allow for efficient and parallelizable network I/O, NSE provides an interface to Nsock, the Nmap socket library. The smart callback mechanism Nsock uses is fully transparent to NSE scripts. The main benefit of NSE's sockets is that they never block on I/O operations, allowing many scripts to be run in parallel. The I/O parallelism is fully transparent to authors of NSE scripts. In NSE you can either program as if you were using a single non-blocking socket or you can program as if your connection is blocking. Even blocking I/O calls return once a specified timeout has been exceeded. Two flavors of Network I/O are supported: connect-style and raw packet.

Connect-style network I/O

This part of the network API should be suitable for most classical network uses: Users create a socket, connect it to a remote address, send and receive data and finally close the socket. Everything up to

the Transport layer (which is either TCP, UDP or SSL) is handled by the library.

An NSE socket is created by calling `nmap.new_socket`, which returns a socket object. The socket object supports the usual `connect`, `send`, `receive`, and `close` methods. Additionally the functions `receive_bytes`, `receive_lines`, and `receive_buf` allow greater control over data reception. [Example 9.2](#) shows the use of connect-style network operations. The `try` function is used for error handling, as described in [the section called “Exception Handling”](#).

Example 9.2. Connect-style I/O

```
require("nmap")

local socket = nmap.new_socket()
socket:set_timeout(1000)
try = nmap.new_try(function() socket:close() end)
try(socket:connect(host.ip, port.number))
try(socket:send("login"))
response = try(socket:receive())
socket:close()
```

Raw packet network I/O

For those cases where the connection-oriented approach is too high-level, NSE provides script developers with the option of raw packet network I/O.

Raw packet reception is handled through a Libpcap wrapper inside the Nsock library. The steps are to open a capture device, register listeners with the device, and then process packets as they are received.

The `pcap_open` method creates a handle for raw socket reads from an ordinary socket object. This method takes a callback function, which computes a packet hash from a packet (including its headers). This hash can return any binary string, which is later compared to the strings registered with the `pcap_register` function. The packet hash callback will normally extract some portion of the packet, such as its source address.

The pcap reader is instructed to listen for certain packets using the `pcap_register` function. The function takes a binary string which is compared against the hash value of every packet received. Those packets whose hashes match any registered strings will be returned by the `pcap_receive` method. Register the empty string to receive all packets.

A script receives all packets for which a listener has been registered by calling the `pcap_receive` method. The method blocks until a packet is received or a timeout occurs.

The more general the packet hash computing function is kept, the more scripts may receive the packet and proceed with their execution. To handle packet capture inside your script you first have to create a socket with `nmap.new_socket` and later close the socket with `socket_object.close`—just like with the connection-based network I/O.

Receiving raw packets is important, but sending them is a key feature as well. To accomplish this, NSE can access a wrapper around the `libdnet` library. Raw packet writes do not use a standard socket object like reads do. Instead, call the function `nmap.new_dnet` to create a `dnet` object with ethernet sending methods. Then open an interface with the `ethernet_open` method. Raw ethernet frames can then be sent with `ethernet_send`. When you're done, close the ethernet handle with `ethernet_close`.

Sometimes the easiest ways to understand complex APIs is by example. The `sniffer-detect.nse` script included with Nmap uses raw packet capture and sending in an attempt to detect promiscuous-mode machines on the network (those running sniffers).

Thread Mutexes

Each script execution thread (e.g. `ftp-anon` running against an FTP server on the target host) yields to other scripts whenever it makes a call on network objects (sending or receiving data). Some scripts require finer concurrency control over thread execution. An example is the `whois` script which queries whois servers for each target IP address. Because many concurrent queries often result in getting one's IP banned for abuse, and because a single query may return additional information for targets other threads are running against,

it is useful to have other threads pause while one thread performs a query.

To solve this problem, NSE includes a mutex function which provides a [mutex](#) (mutual exclusion object) usable by scripts. The mutex allows for only one thread to be working on an object. Competing threads waiting to work on this object are put in the waiting queue until they can get a "lock" on the mutex. A solution for the whois problem above is to have each thread block on a mutex using a common string, thus ensuring that only one thread is querying whois servers at once. That thread can store the results in the NSE registry before releasing/unlocking the mutex. The next script in the waiting queue can then run. It will first check the registry and only query whois servers if the previous results were insufficient.

The first step is to create a mutex object using a statement such as:

```
mutexfn = nmap.mutex(object)
```

The mutexfn returned is a function which works as a mutex for the object passed in. This object can be any [Lua data type](#) except nil, booleans, and numbers. The returned function allows you to lock, try to lock, and release the mutex. Its first and only parameter must be one of the following:

"lock"

Make a blocking lock on the mutex. If the mutex is busy (another thread has a lock on it), then the thread will yield and wait. The function returns with the mutex locked.

"trylock"

Makes a non-blocking lock on the mutex. If the mutex is busy then it immediately returns with a return value of false. Otherwise the mutex locks the mutex and returns true.

"done"

Releases the mutex and allows another thread to lock it. If the thread does not have a lock on the mutex, an error will be raised.

"running"

Returns the thread locked on the mutex or nil if the mutex is not locked. This should only be used for debugging as it interferes with garbage collection of finished threads.

A simple example of using the API is provided in [Example 9.3](#). For real-life examples, read the `asn-query.nse` and `whois.nse` scripts in the Nmap distribution.

Example 9.3. Mutex manipulation

```
local mutex = nmap.mutex("My Script's Unique ID");
function action(host, port)
  mutex "lock";
  -- Do critical section work - only one thread at a time
  executes this.
  mutex "done";
  return script_output;
end
```

Exception Handling

NSE provides an exception handling mechanism which is not present in the base Lua language. It is tailored specifically for network I/O operations, and follows a functional programming paradigm rather than an object oriented one. The `nmap.new_try` API method is used to create an exception handler. This method returns a function which takes a variable number of arguments that are assumed to be the return values of another function. If an exception is detected in the return values (the first return value is false), then the script execution is aborted and no output is produced. Optionally, you can pass a function to `new_try` which will be called if an exception is caught. The function would generally perform any required cleanup operations.

[Example 9.4](#) shows cleanup exception handling at work. A new function named `catch` is defined to simply close the newly created socket in case of an error. It is then used to protect connection and communication attempts on that socket. If no `catch` function is specified, execution of the script aborts without further ado—open sockets will remain open until the next run of Lua's garbage collector. If the verbosity level is at least one or if the scan is performed in debugging mode a description of the uncaught error

condition is printed on standard output. Note that it is currently not easily possible to group several statements in one try block.

Example 9.4. Exception handling example

```
local result, socket, try, catch

result = ""
socket = nmap.new_socket()
catch = function()
socket:close()
end
try = nmap.new_try(catch)

try(socket:connect(host.ip, port.number))
result = try(socket:receive_lines(1))
try(socket:send(result))
```

Writing a function which is treated properly by the try/catch mechanism is straightforward. The function should return multiple values. The first value should be a Boolean which is true upon successful completion of the function and false otherwise. If the function completed successfully, the try construct consumes the indicator value and returns the remaining values. If the function failed then the second returned value must be a string describing the error condition. Note that if the value is not nil or false it is treated as true so you can return your value in the normal case and return nil, *<error description>* if an error occurs.

The Registry

The registry is a Lua table (accessible as nmap.registry) with the special property that it is visible by all scripts and retains its state between script executions. The registry is transient—it is not stored between Nmap executions. Every script can read and write to the registry. Scripts commonly use it to save information for other instances of the same script. For example, the whois and asn-query scripts may query one IP address, but receive information which may apply to tens of thousands of IPs on that network. Saving the information in the registry may prevent other script threads from having to repeat the query.

The registry may also be used to hand information to completely different scripts. For example, the `snmp-brute` script saves a discovered community name in the registry where it may be used by other SNMP scripts. Scripts which leave information behind for a second script must have a lower runlevel than that second script, or there is no guarantee that they will run first.

Because every script can write to the registry table, it is important to avoid conflicts by choosing keys wisely (uniquely).

Script Writing Tutorial

Suppose that you are convinced of the power of NSE. How do you go about writing your own script? Let's say that you want to extract information from an identification server to determine the owner of the process listening on a TCP port. This is not really the purpose of `identd` (it is meant for querying the owner of outgoing connections, not listening daemons), but many `identd` servers allow it anyway. Nmap used to have this functionality (called `ident scan`), but it was removed while transitioning to a new scan engine architecture. The protocol `identd` uses is pretty simple, but still too complicated to handle with Nmap's version detection language. First, you connect to the identification server and send a query of the form `<port-on-server>`, `<port-on-client>` and terminated with a newline character. The server should then respond with a string containing the server port, client port, response type, and address information. The address information is omitted if there is an error. More details are available in [RFC 1413](#), but this description is sufficient for our purposes. The protocol cannot be modeled in Nmap's version detection language for two reasons. The first is that you need to know both the local and the remote port of a connection. Version detection does not provide this data. The second, more severe obstacle, is that you need two open connections to the target—one to the identification server and one to the listening port you wish to query. Both obstacles are easily overcome with NSE.

The anatomy of a script is described in [the section called “Script Format”](#). In this section we will show how the described structure is utilized.

The Head

The head of the script is essentially its meta information. This includes the fields: description, categories, runlevel, author, and license as well as initial NSEDoc information such as usage, args, and output tags (see [the section called “Writing Script Documentation \(NSEDoc\)”](#)).

The description field should contain a paragraph or more describing what the script does. If anything about the script results might confuse or mislead users, and you can't eliminate the issue by improving the script or results text, it should be documented in the description. If there are multiple paragraphs, the first is used as a short summary where necessary. Make sure that first paragraph can serve as a stand alone abstract. This description is short because it is such a simple script:

```
description = [[
Attempts to find the owner of an open TCP port by
querying an auth
(identd - port 113) daemon which must also be open on the
target system.
]]
```

Next comes NSEDoc information. This script is missing the common @usage and @args tags since it is so simple, but it does have an NSEDoc @output tag:

```
---
--@output
-- 21/tcp    open      ftp          ProFTPD 1.3.1
-- |_ auth-owners: nobody
-- 22/tcp    open      ssh          OpenSSH 4.3p2 Debian
9etch2 (protocol 2.0)
-- |_ auth-owners: root
-- 25/tcp    open      smtp        Postfix smtpd
-- |_ auth-owners: postfix
-- 80/tcp    open      http         Apache httpd 2.0.61
((Unix) PHP/4.4.7 ...)
-- |_ auth-owners: dhapache
-- 113/tcp   open      auth?
-- |_ auth-owners: nobody
-- 587/tcp   open      submission Postfix smtpd
-- |_ auth-owners: postfix
-- 5666/tcp  open      unknown
-- |_ auth-owners: root
```

Next come the author, license, and categories tags. This script belongs to the safe because we are not using the service for anything it was not intended for. Because this script is one that should run by default it is also in the default category. Here are the variables in context:

```
author = "Diman Todorov <diman.todorov@gmail.com>"

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

categories = {"default", "safe"}
```

The Rule

The rule section is a Lua method which decides whether to skip or execute the script's action method against a particular service or host. This decision is usually based on the host and port information passed to the rule function. In the case of the identification script, it is slightly more complicated than that. To decide whether to run the identification script against a given port we need to know if there is an auth server running on the target machine. In other words, the script should be run only if the currently scanned TCP port is open and TCP port 113 is also open. For now we will rely on the fact that identification servers listen on TCP port 113. Unfortunately NSE only gives us information about the currently scanned port.

To find out if port 113 is open, we use the `nmap.get_port_state` function. If the auth port was not scanned, the `get_port_state` function returns nil. So we check that the table is not nil. We also check that both ports are in the open state. If this is the case, the action is executed, otherwise we skip the action.

```
portrule = function(host, port)
    local auth_port = { number=113, protocol="tcp" }
    local identd = nmap.get_port_state(host,
auth_port)

    if
        identd ~= nil
        and identd.state == "open"
        and port.protocol == "tcp"
        and port.state == "open"
```

```
        then
            return true
        else
            return false
        end
    end
end
```

The Mechanism

At last we implement the actual functionality! The script first connects to the port on which we expect to find the identification server, then it will connect to the port we want information about. Doing so involves first creating two socket options by calling `nmap.new_socket`. Next we define an error-handling catch function which closes those sockets if failure is detected. At this point we can safely use object methods such as `open`, `close`, `send` and `receive` to operate on the network socket. In this case we call `connect` to make the connections. NSE's exception handling mechanism is used to avoid excessive error-handling code. We simply wrap the networking calls in a `try` call which will in turn call our catch function if anything goes wrong.

If the two connections succeed, we construct a query string and parse the response. If we received a satisfactory response, we return the retrieved information.

```
action = function(host, port)
    local owner = ""

    local client_ident = nmap.new_socket()
    local client_service = nmap.new_socket()

    local catch = function()
        client_ident:close()
        client_service:close()
    end

    local try = nmap.new_try(catch)

    try(client_ident:connect(host.ip, 113))
    try(client_service:connect(host.ip, port.number))

    local localip, localport, remoteip, remoteport =
```



```

        try(client_service:get_info())

        local request = port.number .. ", " ..
localport .. "\n"

        try(client_ident:send(request))

        owner = try(client_ident:receive_lines(1))

        if string.match(owner, "ERROR") then
            owner = nil
        else
            owner = string.match(owner, "USERID : .
+ : (.+)\n", 1)
        end

        try(client_ident:close())
        try(client_service:close())

        return owner
    end
end

```

Note that because we know that the remote port is stored in `port.number`, we could have ignored the last two return values of `client_service:get_info()` like this:

```
local localip, localport = try(client_service:get_info())
```

In this example we exit quietly if the service responds with an error. This is done by assigning `nil` to the `owner` variable which will be returned. NSE scripts generally only return messages when they succeed, so they don't flood the user with pointless alerts.

Writing Script Documentation (NSEDoc)

Scripts are used by more than just their authors, so they require good documentation. NSE modules need documentation so developers can use them in their scripts. NSE's documentation system, described in this section, aims to meet both these needs. While reading this section, you may want to browse NSE's online documentation, which is generated using this system. It is at <http://nmap.org/nsedoc/>.

NSE uses a customized version of the [LuaDoc](#) documentation system called NSEDoc. The documentation for scripts and modules is contained in their source code, as comments with a special form. [Example 9.5](#) is an NSEDoc comment taken from the `stdnse.print_debug()` function.

Example 9.5. An NSEDoc comment for a function

```
--- Prints a formatted debug message if the current
verbosity level is greater
-- than or equal to a given level.
--
-- This is a convenience wrapper around
-- nmap.print_debug_unformatted(). The first
optional numeric
-- argument, verbosity, is used as the
verbosity level necessary
-- to print the message (it defaults to 1 if omitted).
All remaining arguments
-- are processed with Lua's string.format()
function.
-- @param level Optional verbosity level.
-- @param fmt Format string.
-- @param ... Arguments to format.
```

Documentation comments start with three dashes: ---. The body of the comment is the description of the following code. The first paragraph of the description should be a brief summary, with the following paragraphs providing more detail. Special tags starting with @ mark off other parts of the documentation. In the above example you see @param, which is used to describe each parameter of a function. A complete list of the documentation tags is found in [the section called “NSE Documentation Tags”](#).

Text enclosed in the HTML-like `<code>` and `</code>` tags will be rendered in a monospace font. This should be used for variable and function names, as well as multi-line code examples. When a sequence of lines start with the characters “*”, they will be rendered as a bulleted list.

It is good practice to document every public function and table in a script or module. Additionally every script and module should have its own file-level documentation. A documentation comment at the

beginning of a file (one that is not followed by a function or table definition) applies to the entire file. File-level documentation can and should be several paragraphs long, with all the high-level information useful to a developer using a module or a user running a script. [Example 9.6](#) shows documentation for the comm module (with a few paragraphs removed to save space).

Example 9.6. An NSEDoc comment for a module

```
--- Common communication functions for network discovery
tasks like
-- banner grabbing and data exchange.
--
-- These functions may be passed a table of options, but
it's not required. The
-- keys for the options table are <code>"bytes"</code>,
<code>"lines"</code>,
-- <code>"proto"</code>, and <code>"timeout"</code>.
<code>"bytes"</code> sets
-- a minimum number of bytes to read.
<code>"lines"</code> does the same for
-- lines. <code>"proto"</code> sets the protocol to
communicate with,
-- defaulting to <code>"tcp"</code> if not provided.
<code>"timeout"</code>
-- sets the socket timeout (see the socket function
<code>set_timeout()</code>
-- for details).
-- @author Kris Katterjohn 04/2008
-- @copyright Same as Nmap--See http://nmap.org/book/man-
legal.html
```

There are some special considerations for documenting scripts rather than functions and modules. In particular, scripts have special variables for some information which would otherwise belongs in @-tag comments (script variables are described in [the section called "Script Format"](#)). In particular, a script's description belongs in the description variable rather than in a documentation comment, and the information that would go in @author and @copyright belong in the variables author and license instead. NSEDoc knows about these variables and will use them in preference to fields in the comments. Scripts should also have an @output tag showing sample output, as well as @args and @usage where appropriate. [Example 9.7](#) shows

proper form for script-level documentation, using a combination of documentation comments and NSE variables.

Example 9.7. An NSEDoc comment for a script

```
description = [[
Maps IP addresses to autonomous system (AS) numbers.

The script works by sending DNS TXT queries to a DNS
server which in
turn queries a third-party service provided by Team Cymru
(team-cymru.org) using an in-addr.arpa style zone set up
especially for
use by Nmap.
]]

---
-- @usage
-- nmap --script asn-query.nse [--script-args dns=<DNS
server>] <target>
-- @args dns The address of a recursive nameserver to use
(optional).
-- @output
-- Host script results:
-- |   AS Numbers:
-- |   BGP: 64.13.128.0/21 | Country: US
-- |       Origin AS: 10565 SVCOLO-AS - Silicon Valley
Colocation, Inc.
-- |       Peer AS: 3561 6461
-- |   BGP: 64.13.128.0/18 | Country: US
-- |       Origin AS: 10565 SVCOLO-AS - Silicon Valley
Colocation, Inc.
-- |_       Peer AS: 174 2914 6461

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

Compiled NSE modules are also documented with NSEDoc, even though they have no Lua source code. Each compiled module has a file `<modulename>.luadoc` that is kept in the `nselib` directory alongside the Lua modules. This file lists and documents the

functions and tables in the compiled module as though they were written in Lua. Only the name of each function is required, not its definition (not even end). You must use the @name and @class tags when documenting a table to assist the documentation parser in identifying it. There are several examples of this method of documentation in the Nmap source distribution (including nmap.luadoc, bit.luadoc, and pcre.luadoc).

NSE Documentation Tags

The following tags are understood by NSEDoc:

@param

Describes a function parameter. The first word following @param is the name of the parameter being described. The tag should appear once for each parameter of a function.

@see

Adds a cross-reference to another function or table.

@return

Describes a return value of a function. @return may be used multiple times for multiple return values.

@usage

Provides a usage example of a function or script. In the case of a function, the example is Lua code; for a script it is an Nmap command line. @usage may be given more than once.

@name

Defines a name for the function or table being documented. This tag is normally not necessary because NSEDoc infers names through code analysis.

@class

Defines the “class” of the object being documented: function, table, or module. Like @name, this is normally inferred automatically.

@field

In the documentation of a table, @field describes the value of a named field.

@args

Describes a script argument, as used with the --script-args option (see [the section called “Arguments to Scripts”](#)). The first word after @args is the name of the argument, and everything following that is the description. This tag is special to script-level comments.

@output

This tag, which is exclusive to script-level comments, shows sample output from a script.

@author

This tag, which may be given multiple times, lists the authors of an NSE module. For scripts, use the author variable instead.

@copyright

This tag describes the copyright status of a module. For scripts, use the license variable instead.

Version Detection Using NSE

The version detection system built into Nmap was designed to efficiently recognize the vast majority of protocols with a simple probe and pattern matching syntax. Some protocols require more complex communication than version detection can handle. A generalized scripting language as provided by NSE is perfect for these tough cases.

NSE's version category contains scripts that enhance standard version detection. Scripts in this category are run whenever you request version detection with -sV; you don't need to use -sC to run these. This cuts the other way too: if you use -sC, you won't get version scripts unless you also use -sV.

One protocol which we were unable to detect with normal version detection is Skype version 2. The protocol was likely designed to frustrate detection out of a fear that telecom-affiliated Internet service providers might consider Skype competition and interfere with the traffic. Yet we did find one way to detect it. If Skype receives an HTTP GET request, it pretends to be a web server and returns a 404 error. But for other requests, it sends back a chunk of random-looking data. Proper identification requires sending two probes and comparing the two responses—an ideal task for NSE. The simple NSE script which accomplishes this is shown in [Example 9.8](#).

Example 9.8. A typical version detection script (Skype version 2 detection)

```
description = [[
Detects the Skype version 2 service.
]]
author = "Brandon Enright <bmenrigh@ucsd.edu>"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"version"}

require "comm"

portrule = function(host, port)
    return (port.number == 80 or port.number == 443
or
            port.service == nil or port.service == ""
or
            port.service == "unknown")
    and port.protocol == "tcp" and port.state
== "open"
    and port.service ~= "http" and
port.service ~= "ssl/http"
end

action = function(host, port)
    local status, result = comm.exchange(host, port,
        "GET / HTTP/1.0\r\n\r\n", {bytes=26,
proto=port.protocol})
    if (not status) then
        return
    end
end
```

```

        if (result ~= "HTTP/1.0 404 Not Found\r\n\r\n")
then
            return
        end
        -- So far so good, now see if we get random data
for another request
        status, result = comm.exchange(host, port,
            "random data\r\n\r\n", {bytes=15,
proto=port.protocol})

        if (not status) then
            return
        end
        if string.match(result, "[^%s!-~].*[^%s!-~].*[^%s!-~]") then
            -- Detected
            port.version.name = "skype2"
            port.version.product = "Skype"
            nmap.set_port_version(host, port,
"hardmatched")
            return
        end
        return
    end
end
end

```

If the script detects Skype, it augments its port table with now-known name and product fields. It then sends this new information to Nmap by calling `nmap.set_port_version`. Several other version fields are available to be set if they are known, but in this case we only have the name and product. For the full list of version fields, refer to the [nmap.set_port_version documentation](#).

Notice that this script does nothing unless it detects the protocol. A script shouldn't produce output (other than debug output) just to say it didn't learn anything.

Example Script: `finger.nse`

The `finger` script (`finger.nse`) is a perfect example of a short and simple NSE script.

First the information fields are assigned. A detailed description of what the script actually does goes in the description field.

```
description = [[
Attempts to get a list of usernames via the finger
service.
]]
author = "Eddie Bell <ejlbell@gmail.com>"
license = "Same as Nmap--See http://nmap.org/book/man-
legal.html"
```

The categories field is a table containing all the categories the script belongs to—These are used for script selection with the --script option:

```
categories = {"default", "discovery"}
```

You can use the facilities provided by the nselib ([the section called “NSE Libraries”](#)) with require. Here we want to use common communication functions and shorter port rules:

```
require "comm"
require "shortport"
```

We want to run the script against the finger service. So we test whether it is using the well-known finger port (79/tcp), or whether the service is named “finger” based on version detection results or in the port number's listing in nmap-services:

```
portrule = shortport.port_or_service(79, "finger")
```

First, the script uses nmap.new_try to create an exception handler that will quit the script in case of an error. Next, it passes control to comm.exchange, which handles the network transaction. Here we have asked to wait in the communication exchange until we receive at least 100 lines, wait at least 5 seconds, or until the remote side closes the connection. Any errors are handled by the try exception handler. The script returns a string if the call to comm.exchange() was successful.

```
action = function(host, port)
    local try = nmap.new_try()

    return try(comm.exchange(host, port, "\r\n",
```

```
        {lines=100, proto=port.protocol,  
timeout=5000}))  
end
```

Implementation Details

Now it is time to explore the NSE implementation details in depth. Understanding how NSE works is useful for designing efficient scripts and libraries. The canonical reference to the NSE implementation is the source code, but this section provides an overview of key details. It should be valuable to folks trying to understand and extend the NSE source code, as well as to script authors who want to better-understand how their scripts are executed.

Initialization Phase

During its initialization stage, Nmap loads the Lua interpreter and its provided libraries. These libraries are fully documented in the [Lua Reference Manual](#). Here is a summary of the libraries, listed alphabetically by their namespace name:

debug

The [debug library](#) provides a low-level API to the Lua interpreter, allowing you to access functions along the execution stack, retrieve function closures and object metatables, and more.

io

The [Input/Output library](#) offers functions such as reading from files or from the output from programs you execute.

math

Numbers in Lua usually correspond to the double C type, so the [math library](#) provides access to rounding functions, trigonometric functions, random number generation, and more.

os

The [Operating System library](#) provides system facilities such as filesystem operations (including file renaming or removal and temporary file creation) and system environment access.

package

Among the functions provided by Lua's [package-lib](#) is `require`, which is used to load nselib modules.

string

The [string library](#) provides functions for manipulating Lua strings, including printf-style string formatting, pattern matching using Lua-style patterns, substring extraction, and more.

table

The [table manipulation library](#) is essential for operating on Lua's central data structure (tables).

In addition to loading the libraries provided by Lua, the `nmap` namespace functions are loaded. The search paths are the same directories that Nmap searches for its data files, except that the `nselib` directory is appended to each. At this stage any provided script arguments are stored inside the registry.

The next phase of NSE initialization is loading the selected scripts, based on the defaults or arguments provided to the `--script` option. The version category scripts are loaded as well if version detection was enabled. NSE first tries to interpret each `--script` argument as a category. This is done with a Lua C function in `nse_init.cc` named `entry` based on data from the `script.db` script categorization database. If the category is found, those scripts are loaded. Otherwise Nmap tries to interpret `--script` arguments as files or directories. If no files or directories with a given name are found in Nmap's search path, an error is raised and the Script Engine aborts.

If a directory is specified, all of the `.nse` files inside it are loaded. Each loaded file is executed by Lua. If a *portrule* is present, it is saved in the *porttests* table with a *portrule* key and file closure value. Otherwise, if the script has a *hostrule*, it is saved in the *hosttests* table in the same manner.

Matching Scripts with Targets

After initialization is finished, the hostrules and portrules are evaluated for each host in the current target group. The rules of every chosen script is tested against every host and (in the case of service scripts) each open and open|filtered port on the hosts. The combination can grow quite large, so portrules should be kept as simple as possible. Save any heavy computation for the script's action.

Next, a Lua thread is created for each of the matching script-target combinations. Each thread is stored with pertinent information such as the runlevel, target, target port (if applicable), host and port tables (passed to the action), and the script type (service or host script). The mainloop function then processes each runlevel grouping of threads in order.

Script Execution

Nmap performs NSE script scanning in parallel by taking advantage of Nmap's Nsock parallel I/O library and the Lua [coroutines](#) language feature. Coroutines offer collaborative multi-threading so that scripts can suspend themselves at defined points and allow other coroutines to execute. Network I/O, particularly waiting for responses from remote hosts, often involves long wait times, so this is when scripts yield to others. Key functions of the Nsock wrapper cause scripts to yield (pause). When Nsock finishes processing such a request, it makes a callback which causes the script to be pushed from the waiting queue back into the running queue so it can resume operations when its turn comes up again.

The mainloop function moves threads between the waiting and running queues as needed. A thread which yields is moved from the running queue into the waiting list. Running threads execute until they either yield, complete, or fail with an error. Threads are made ready to run (placed in the running queue) by calling `process_waiting2running`. This process of scheduling running threads and moving threads between queues continues until no threads exist in either queue.

Chapter 10. Detecting and Subverting Firewalls and Intrusion Detection Systems

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

DNS proxying

MAC Address Spoofing

Source Port Manipulation

**A practical example: bypassing default Snort 2.2.0
rules**

Chapter 11. Defenses Against Nmap

Sorry, but this section or chapter of the Nmap book (Nmap Network Scanning) is not currently available in the free online edition—only in the printed book version ([more book information](#) or [buy on Amazon](#)).

OS Spoofing

Chapter 12. Zenmap GUI Users' Guide

Table of Contents

[Introduction](#)

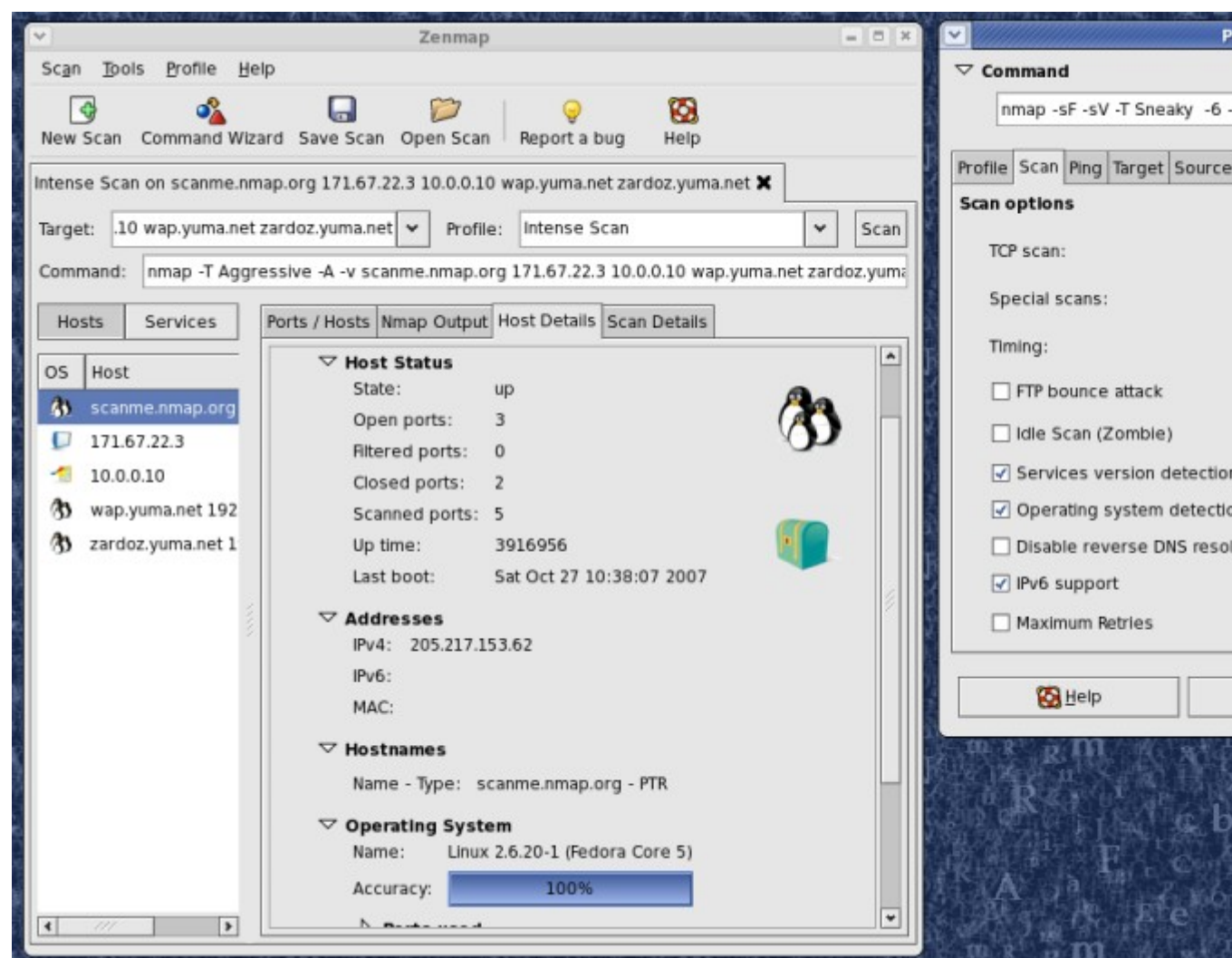
[The Purpose of a Graphical Frontend for Nmap
Scanning](#)

- Profiles
- Scan Aggregation
- Interpreting Scan Results
 - Scan Results Tabs
 - The Nmap Output tab
 - The Ports / Hosts tab
 - The Topology tab
 - The Host Details tab
 - The Scans tab
 - Sorting by Host
 - Sorting by Service
- Saving and Loading Scan Results
 - The Recent Scans Database
- Surfing the Network Topology
 - An Overview of the Topology Tab
 - Legend
 - Controls
 - Action controls
 - Interpolation controls
 - Layout controls
 - View controls
 - Fisheye controls
 - Keyboard Shortcuts
 - The Hosts Viewer
- The Profile Editor
 - Editing a Command
 - Creating a New Profile
 - Editing or Deleting a Profile
- Searching Saved Results
- Comparing Results
- Zenmap in Your Language
 - Creating a new translation
- Files Used by Zenmap
 - The nmap Executable
 - System Configuration Files
 - Per-user Configuration Files
 - Output Files
- Description of zenmap.conf
 - Sections of zenmap.conf
- Command-line Options
 - Synopsis
 - Options Summary
 - Error Output
- History

Introduction

Zenmap is the official graphical user interface (GUI) for the Nmap Security Scanner. It is a multi-platform, free and open-source application designed to make Nmap easy for beginners to use while providing advanced features for experienced Nmap users. Frequently used scans can be saved as profiles to make them easy to run repeatedly. A command creator allows interactive creation of Nmap command lines. Scan results can be saved and viewed later. Saved scans can be compared with one another to see how they differ. The results of recent scans are stored in a searchable database. A typical Zenmap screen shot is shown in [Figure 12.1](#). See the [official Zenmap web page](#) for more screen shots.

Figure 12.1. Typical Zenmap screen shot



This guide is meant to make Nmap and Zenmap easy to use together, even if you haven't used either before. For the parts of this guide that deal specifically with Nmap (command-line options and such), refer to [Chapter 15, Nmap Reference Guide](#).

The Purpose of a Graphical Frontend for Nmap

No frontend can replace good old command-line Nmap. The nature of a frontend is that it depends on another tool to do its job. Therefore the purpose of Zenmap is not to replace Nmap, but to make Nmap *more useful*. Here are some of the advantages Zenmap offers over plain Nmap.

Interactive and graphical results viewing

In addition to showing Nmap's normal output, Zenmap can arrange its display to show all ports on a host or all hosts running a particular service. It summarizes details about a single host or a complete scan in a convenient display. Zenmap can even draw a topology map of discovered networks. The results of several scans may be combined together and viewed at once.

Comparison

Zenmap has the ability to show the differences between two scans. You can see what changed between the same scan run on different days, between scans of two different hosts, between scans of the same hosts with different options, or any other combination. This allows administrators to easily track new hosts or services appearing on their networks, or existing ones going down.

Convenience

Zenmap keeps track of your scan results until you choose to throw them away. That means you can run a scan, see the results, and then decide whether to save them to a file. There is no need to think of a file name in advance.

Repeatability

Zenmap's command profiles make it easy to run the exact same scan more than once. There's no need to set up a shell script to do a common scan.

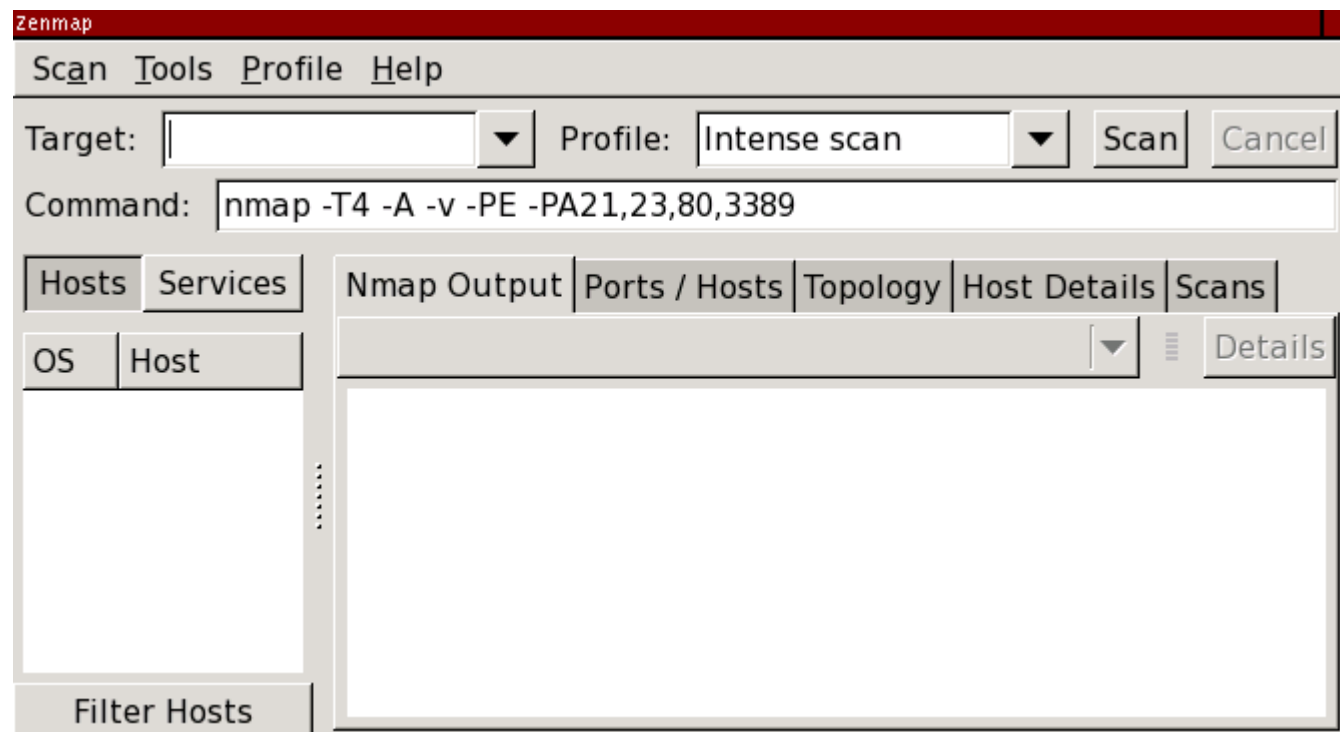
Discoverability

Nmap has literally hundreds of options, which can be daunting for beginners. Zenmap's interface is designed to always show the command that will be run, whether it comes from a profile or was built up by choosing options from a menu. This helps beginners learn and understand what they are doing. It also helps experts double-check exactly what will be run before they press “Scan”.

Scanning

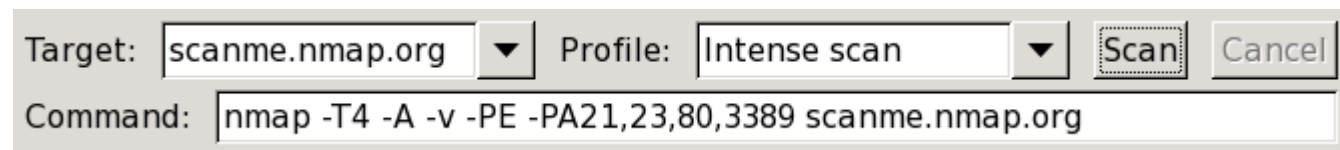
Begin Zenmap by typing **zenmap** in a terminal or by clicking the Zenmap icon in the desktop environment. The main window, as shown in [Figure 12.2](#), is displayed.

Figure 12.2. Zenmap's main window



One of Zenmap's goals is to make security scanning easy for beginners and for experts. Running a scan is as simple as typing the target in the “Target” field, selecting the “Intense scan” profile, and clicking the “Scan” button. This is shown in [Figure 12.3](#).

Figure 12.3. Target and profile selection

A screenshot of the Zenmap application's main window. The 'Target' field contains 'scanme.nmap.org'. The 'Profile' dropdown menu is set to 'Intense scan'. The 'Command' field displays the generated Nmap command: 'nmap -T4 -A -v -PE -PA21,23,80,3389 scanme.nmap.org'. There are 'Scan' and 'Cancel' buttons to the right of the profile dropdown.

While a scan is running (and after it completes), the output of the Nmap command is shown on the screen.

Any number of targets, separated by spaces, may be entered in the target field. All the target specifications supported by Nmap are also supported by Zenmap, so targets such as 192.168.0.0/24 and 10.0.0-5.* work. Zenmap remembers the targets scanned most recently. To re-scan a host, select the host from the combo box attached to the “Target” text field.

Profiles

The “Intense scan” is just one of several scan profiles that come with Zenmap. Choose a profile by selecting it from the “Profile” combo box. Profiles exist for several common scans. After selecting a profile the Nmap command line associated with it is displayed on the screen. Of course, it is possible to edit these profiles or create new ones. This is covered in [the section called “The Profile Editor”](#).

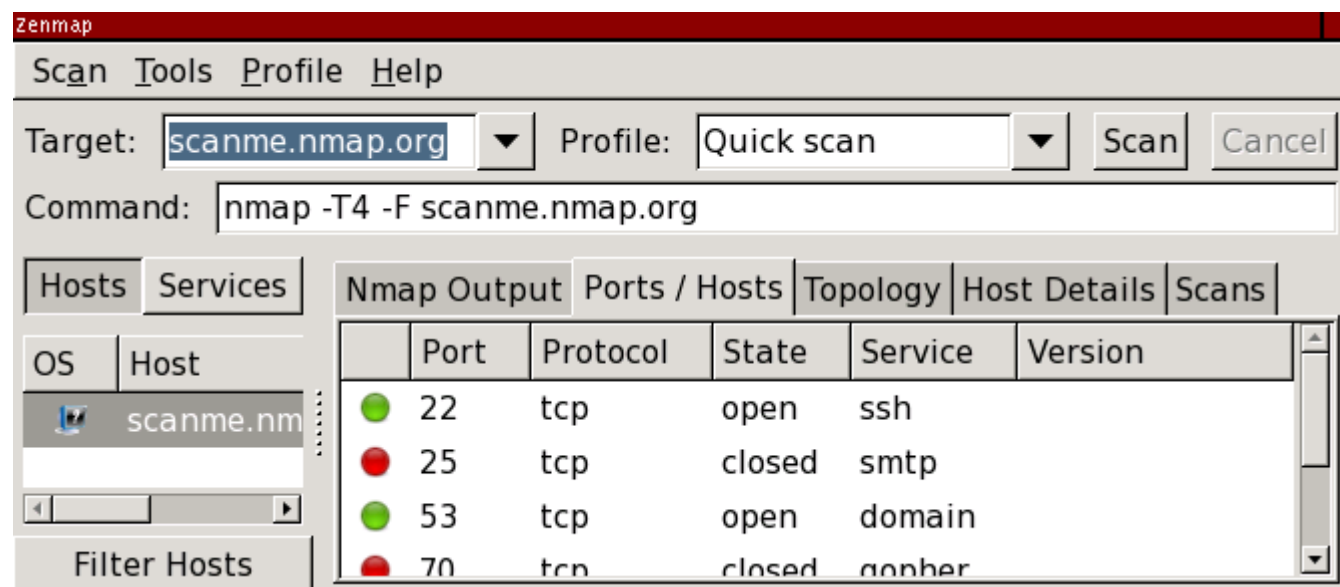
It is also possible to type in an Nmap command and have it executed without using a profile. Just type in the command and press return or click “Scan”. When you do this the “Profile” entry becomes blank to indicate that the scan is not using any profile—it comes directly from the command field.

Scan Aggregation

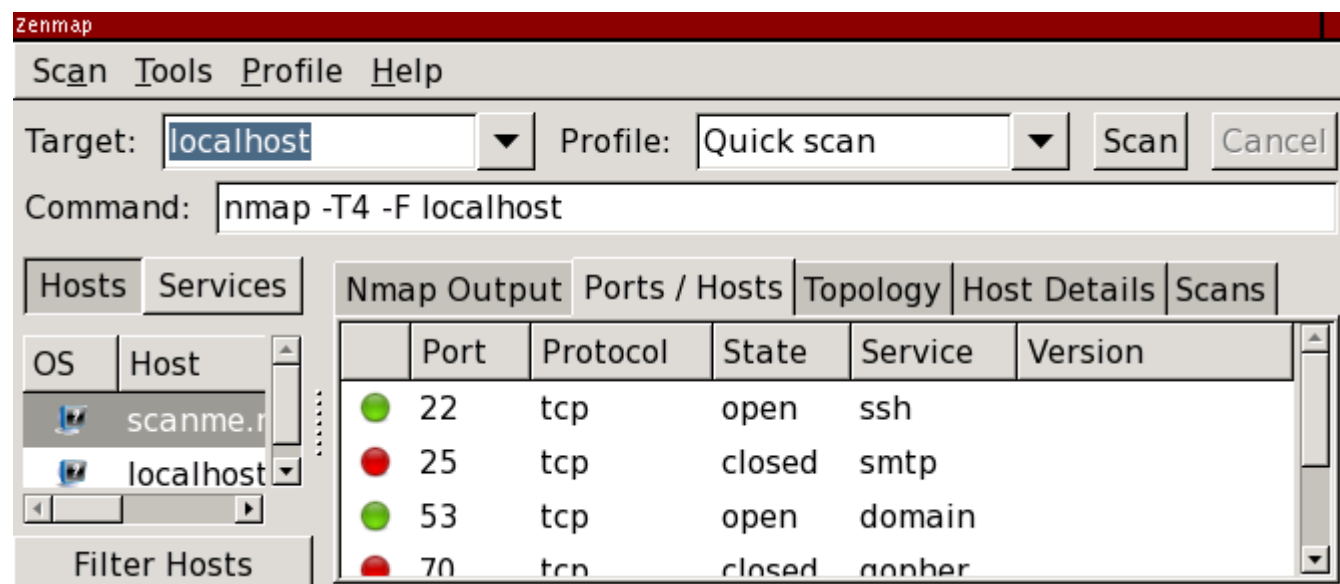
Zenmap has the ability to combine the results of many Nmap scans into one view, a feature known as *scan aggregation*. When one scan

is finished, you may start another in the same window. When the second scan is finished, its results are merged with those from the first. The collection of scans that make up an aggregated view is called a *network inventory*.

An example of aggregation will make the concept clearer. Let's run a quick scan against scanme.nmap.org.

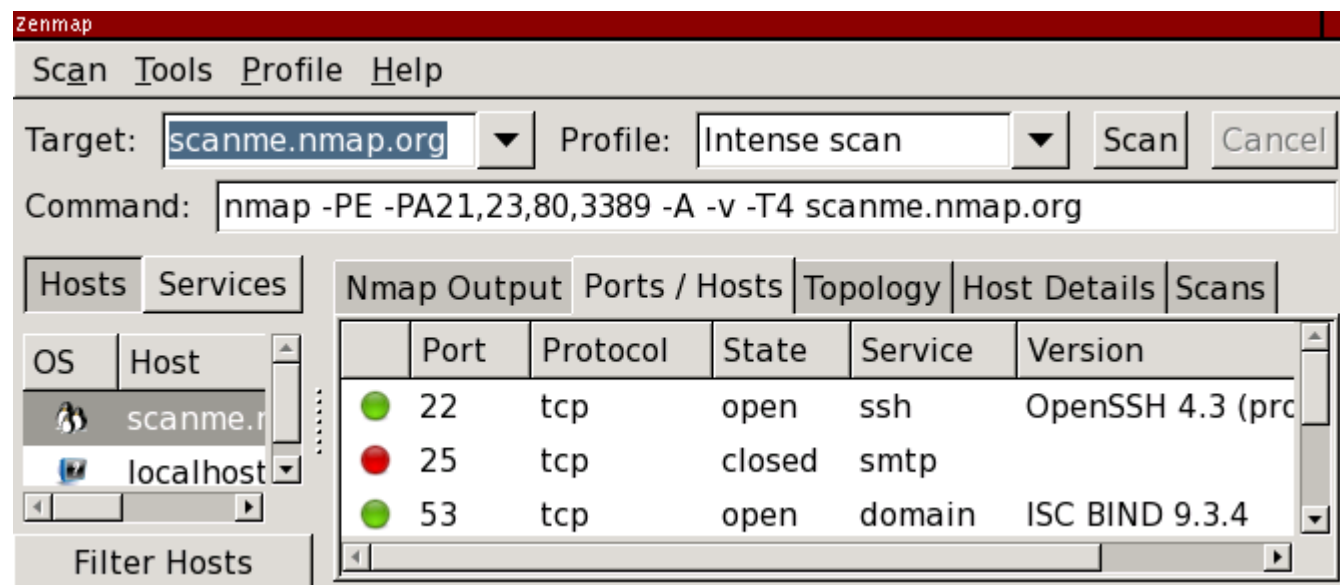


Now do the same against localhost:



Now results for both scanme and localhost are shown. This is something you could have done with one Nmap scan, giving both targets, although it's convenient not to have to think of all the

targets in advance. Now suppose we want some more information about scanme, so we launch an intense scan on it.



Now scanme has a little penguin icon showing that its operating system has been detected as Linux. Additionally some of its services have been identified. Now we're doing something you can't do with a single Nmap scan, because you can't single out a host for more intense scanning like we did. The results for localhost are still present, though we won't know more about it than we did before unless we decide to do a more in-depth scan.

It is not necessary to wait for one scan to finish before starting another. Several scans may run concurrently. As each one finishes its results are added to the inventory. Any number of scans may make up an inventory; the collection of scans is managed in the "Scans" scan results tab, as fully described in [the section called "The Scans tab"](#).

It is possible to have more than one inventory open at the same time. Zenmap uses the convention that one window represents one network inventory. To start a new inventory, select "New Window" from the "Scan" menu or use the **ctrl+N** keyboard shortcut. Starting a scan with the "Scan" button will append the scan to the inventory in the current window. To put it in a different inventory open up a separate window and run the scan from there. Loading scan results from a file or directory will start a new inventory, unless you use the "Open Scan in This Window" menu item. For more on saving and loading network inventories and individual scans see [the section called "Saving and Loading Scan Results"](#).

To close a window choose “Close Window” from the “Scan” menu or press **ctrl+W**. When all open windows are closed the application will terminate. To close all open windows select “Quit” or press **ctrl+Q**.

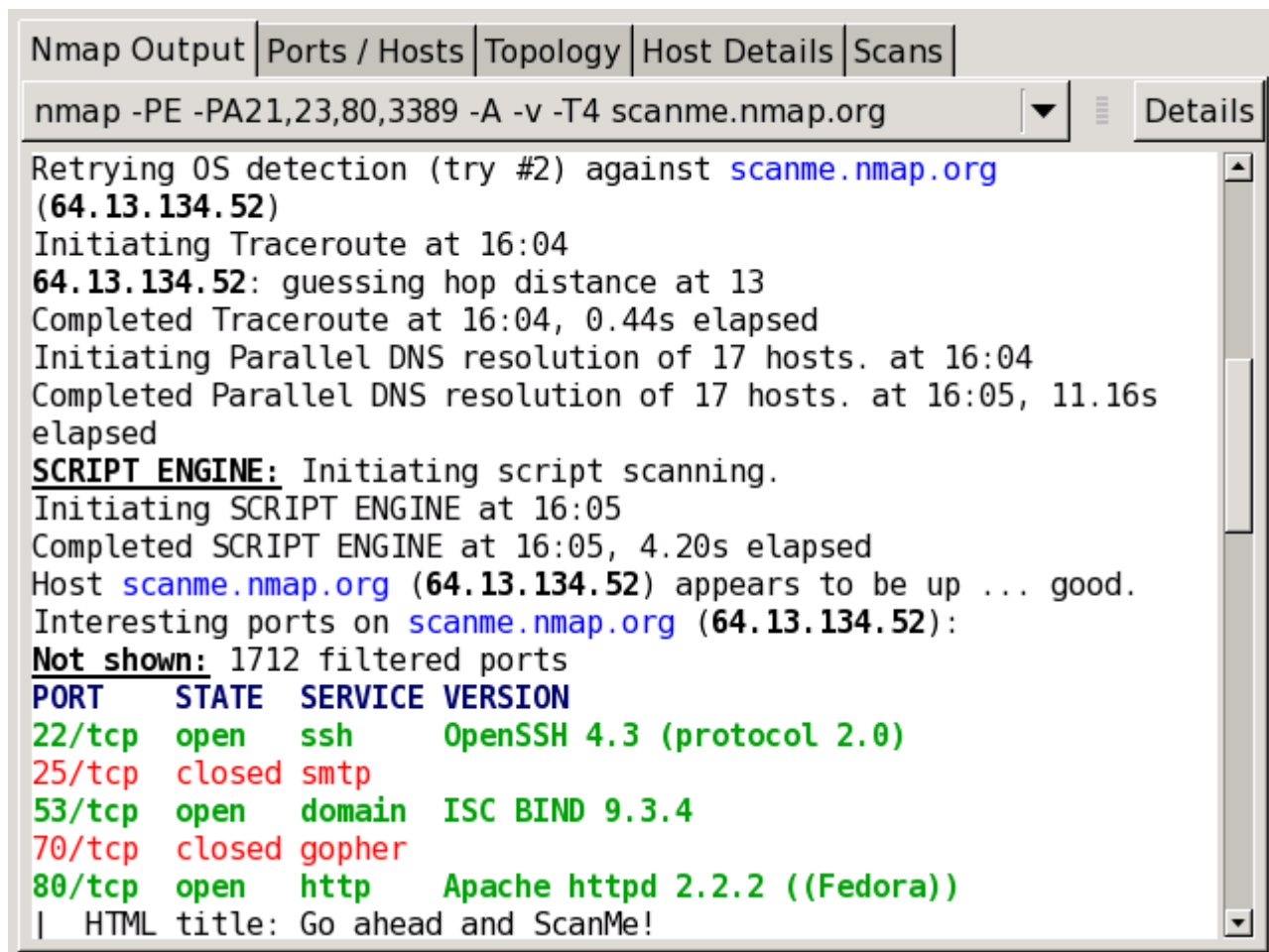
Interpreting Scan Results

Nmap's output is displayed during and after a scan. This output will be familiar to Nmap users. Except for Zenmap's color highlighting, this doesn't offer any visualization advantages over running Nmap in a terminal. However, other parts of Zenmap's interface interpret and aggregate the terminal output in a way that makes scan results easier to understand and use.

Scan Results Tabs

Each scan window contains five tabs which each display different aspects of the scan results. They are: “Nmap Output”, “Ports / Hosts”, “Topology”, “Host Details”, and “Scans”. Each of these are discussed in this section.

The “Nmap Output” tab



The “Nmap Output” tab is displayed by default when a scan is run. It shows the familiar Nmap terminal output. The display highlights parts of the output according to their meaning; for example, open and closed ports are displayed in different colors. Custom highlights can be configured in `zenmap.conf` (see [the section called “Description of zenmap.conf”](#)).

Recall that the results of more than one scan may be shown in a window (see [the section called “Scan Aggregation”](#)). The drop-down combo box at the top of the tab allows you to select the scan to display. The “Details” button brings up a window showing miscellaneous information about the scan, such as timestamps, command-line options, and the Nmap version number used.

The “Ports / Hosts” tab

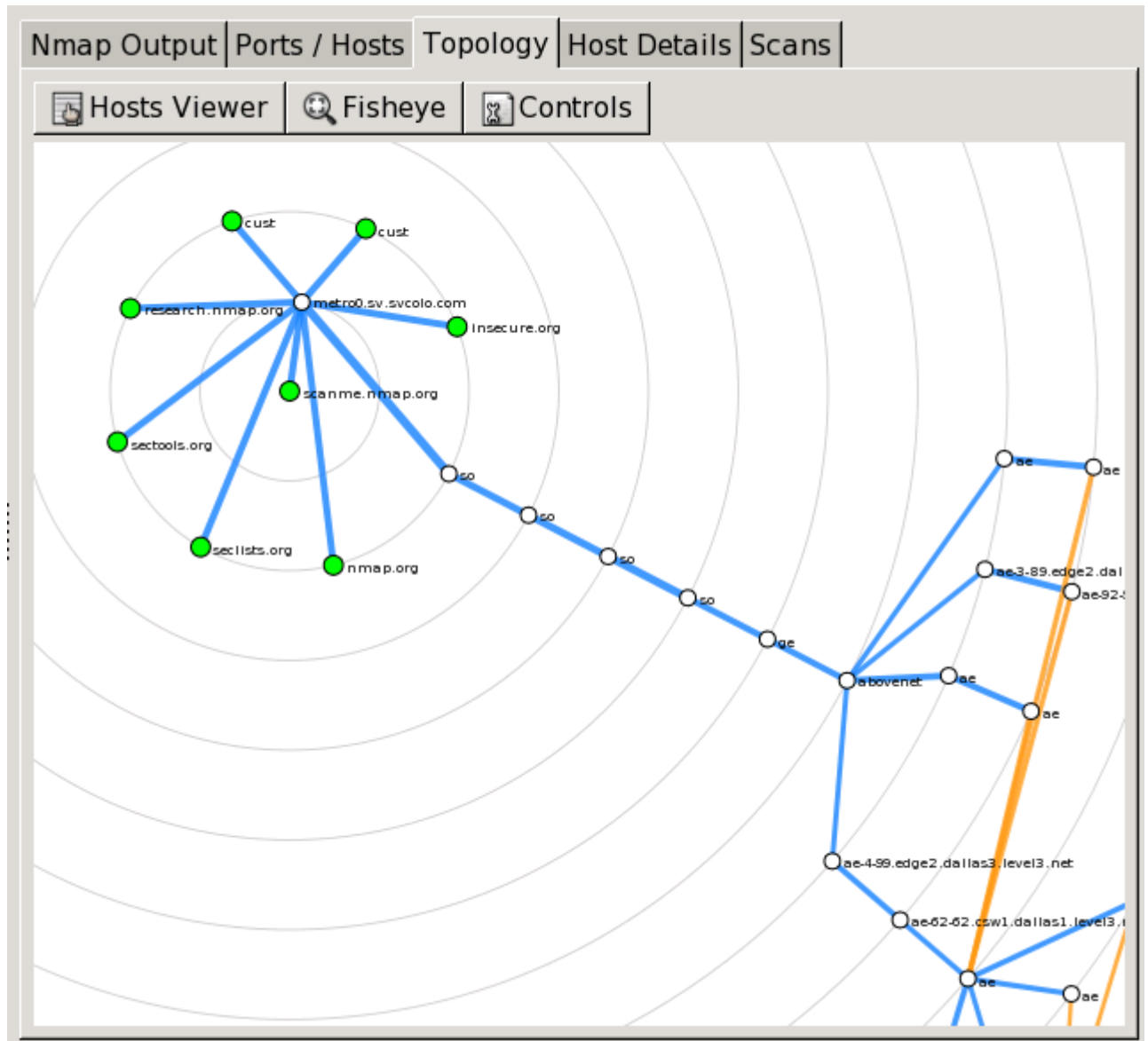
Nmap Output					
Ports / Hosts		Topology	Host Details		Scans
	Port	Protocol	State	Service	Version
●	22	tcp	open	ssh	OpenSSH 4.3 (protocol 2.0)
●	25	tcp	closed	smtp	
●	53	tcp	open	domain	ISC BIND 9.3.4
●	70	tcp	closed	gopher	
●	80	tcp	open	http	Apache httpd 2.2.2 ((Fedora))
●	113	tcp	closed	auth	

The “Ports / Hosts” tab's display differs depending on whether a host or a service is currently selected. When a host is selected, it shows all the interesting ports on that host, along with version information when available. Host selection is further described in [the section called “Sorting by Host”](#).

Nmap Output					
Ports / Hosts		Topology	Host Details		Scans
	Hostname	Port	Protocol	State	Version
●	home.domain.actds/tmp (192.168.0.1)	80	tcp	open	Vonage
●	scanme.nmap.org (64.13.134.52)	80	tcp	open	Apache

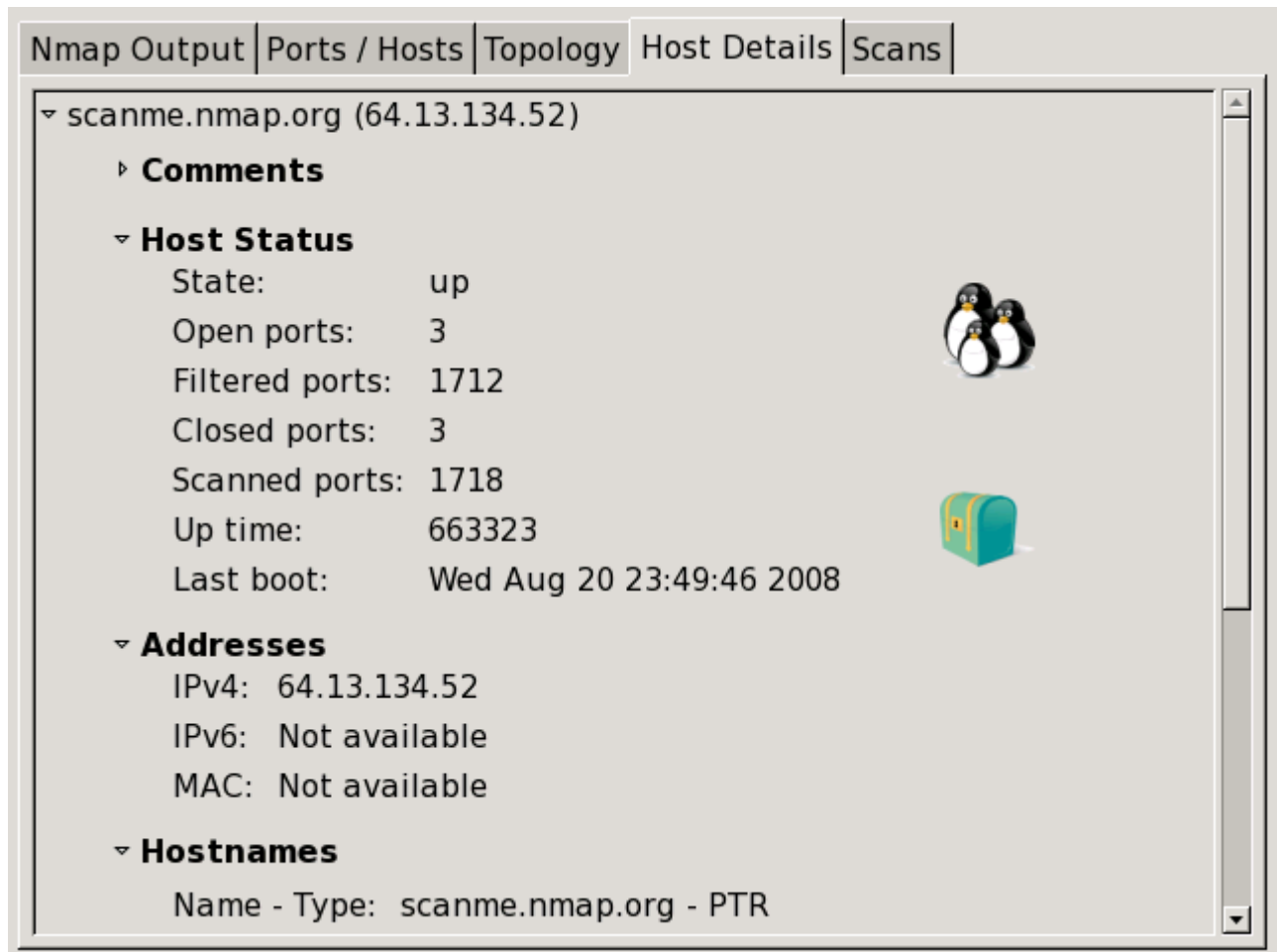
When a service is selected, the “Ports / Hosts” tab shows all the hosts which have that port open or filtered. This is a good way to quickly answer the question “What computers are running HTTP?” Service selection is further described in [the section called “Sorting by Service”](#).

The “Topology” tab



The “Topology” tab is an interactive view of the connections between hosts in a network. Hosts are arranged in concentric rings. Each ring represents an additional network hop from the center node. Clicking on a node brings it to the center. Because it shows a representation of the network paths between hosts, the “Topology” tab benefits from the use of the `--traceroute` option. Topology view is discussed in more detail in [the section called “Surfing the Network Topology”](#).

The “Host Details” tab



The “Host Details” tab breaks all the information about a single host into a hierarchical display. Shown are the host's names and addresses, its state (up or down), and the number and status of scanned ports. The host's uptime, operating system, OS icon (see [Figure 12.5, “OS icons”](#)), and other associated details are shown when available. When no exact OS match is found, the closest matches are displayed. There is also a collapsible text field for storing a comment about the host which will be saved when the scan is saved to a file (see [the section called “Saving and Loading Scan Results”](#)).

Each host has an icon that provides a very rough “vulnerability” estimate, which is based solely on the number of open ports. The icons and the numbers of open ports they correspond to are



0–2 open ports,



3–4 open ports,



5–6 open ports,



7–8 open ports, and



9 or more open ports.

The “Scans” tab

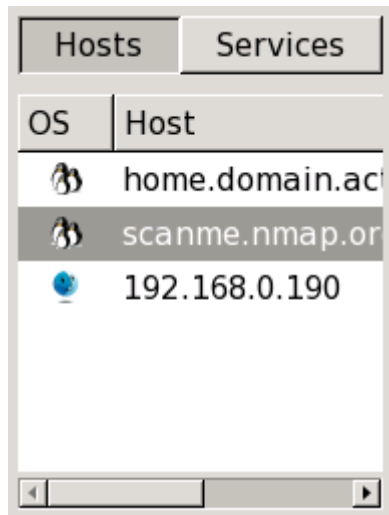
Nmap Output Ports / Hosts Topology Host Details Scans	
Status	Command
	nmap -PE -PA21,23,80,3389 -A -v -T4 scanme.nmap.org
	nmap -PE -PA21,23,80,3389 -A -v -T4 192.168.0.190
	nmap -PE -PA21,23,80,3389 -A -v -T4 192.168.0.1
<div><div> Append Scan</div><div> Remove Scan</div><div> Cancel Scan</div></div>	

The “Scans” tab shows all the scans that are aggregated to make up the network inventory. From this tab you can add scans (from a file or directory) and remove scans.

While a scan is executing and not yet complete, its status is “Running”. You may cancel a running scan by clicking the “Cancel Scan” button.

Sorting by Host

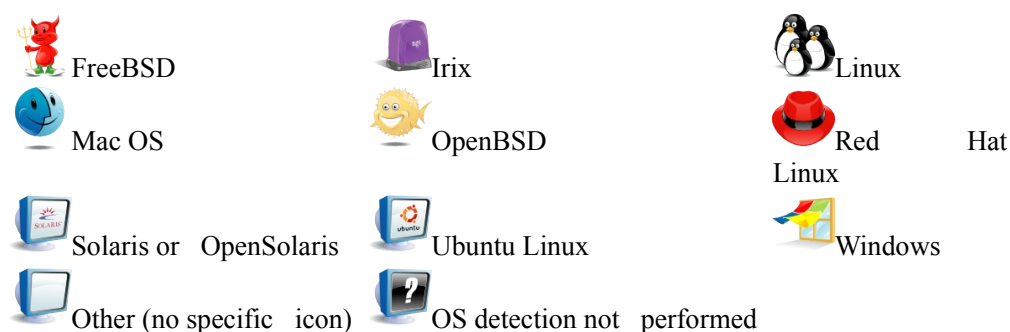
Figure 12.4. Host selection



On the left side of Zenmap's main window is a column headed by two buttons labeled “Hosts” and “Services”. Clicking the “Hosts” button will bring up a list of all hosts that were scanned, as in [Figure 12.4](#). Commonly this contains just a single host, but it can contain thousands in a large scan. The host list can be sorted by OS or host name/IP address by clicking the headers at the top of the list. Selecting a host will cause the “Ports / Hosts” tab to display the interesting ports on that host.

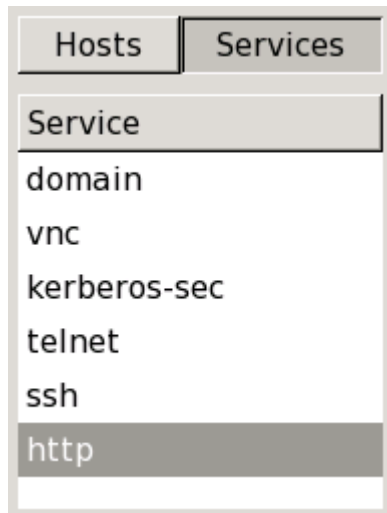
Each host is labeled with its host name or IP address and has an icon indicating the operating system that was detected for that host. The icon is meaningful only if OS detection (-O) was performed. Otherwise, the icon will be a default one indicating that the OS is unknown. [Figure 12.5](#) shows all possible icons. Note that Nmap's OS detection cannot always provide the level of specificity implied by the icons; for example a Red Hat Linux host will often be displayed with the generic Linux icon.

Figure 12.5. OS icons



Sorting by Service

Figure 12.6. Service selection



Above the same list that contains all the scanned hosts is a button labeled “Services”. Clicking that will change the list into a list of all ports that are open, filtered, or open|filtered on any of the targets, as shown in [Figure 12.6](#). (Ports that were not listed explicitly in Nmap output are not included.) The ports are identified by service name (http, ftp, etc.). The list can be sorted by clicking the header of the list.

Selecting a host will cause the “Ports / Hosts” tab to display all the hosts that have that service open or filtered.

Saving and Loading Scan Results

To save an individual scan to a file, choose “Save Scan” from the “Scan” menu (or use the keyboard shortcut **ctrl+S**). If there is more than one scan into the inventory you will be asked which one you want to save. Results are saved in Nmap XML format, which is discussed in [the section called “XML Output \(-oX\)”](#).

You can save every scan in an inventory with “Save All Scans to Directory” under the “Scan” menu (**ctrl+alt+S**). When saving an inventory for the first time, you will commonly create a new directory using the “Create Folder” button in the save dialog. In

subsequent saves you can continue saving to the same directory. To reduce the chance of overwriting unrelated scan files, the save-to-directory function will refuse to continue if the chosen directory contains a file that doesn't belong to the inventory. If you are sure you want to save to that directory, delete any offending files and then save again.

Saved results are loaded by choosing “Open Scan” from the “Scan” menu, or by typing the **ctrl+O** keyboard shortcut. In the file selector, the “Open” button opens a single scan, while the “Open Directory” button opens every file in the chosen directory (perhaps created using “Save All Scans to Directory”).

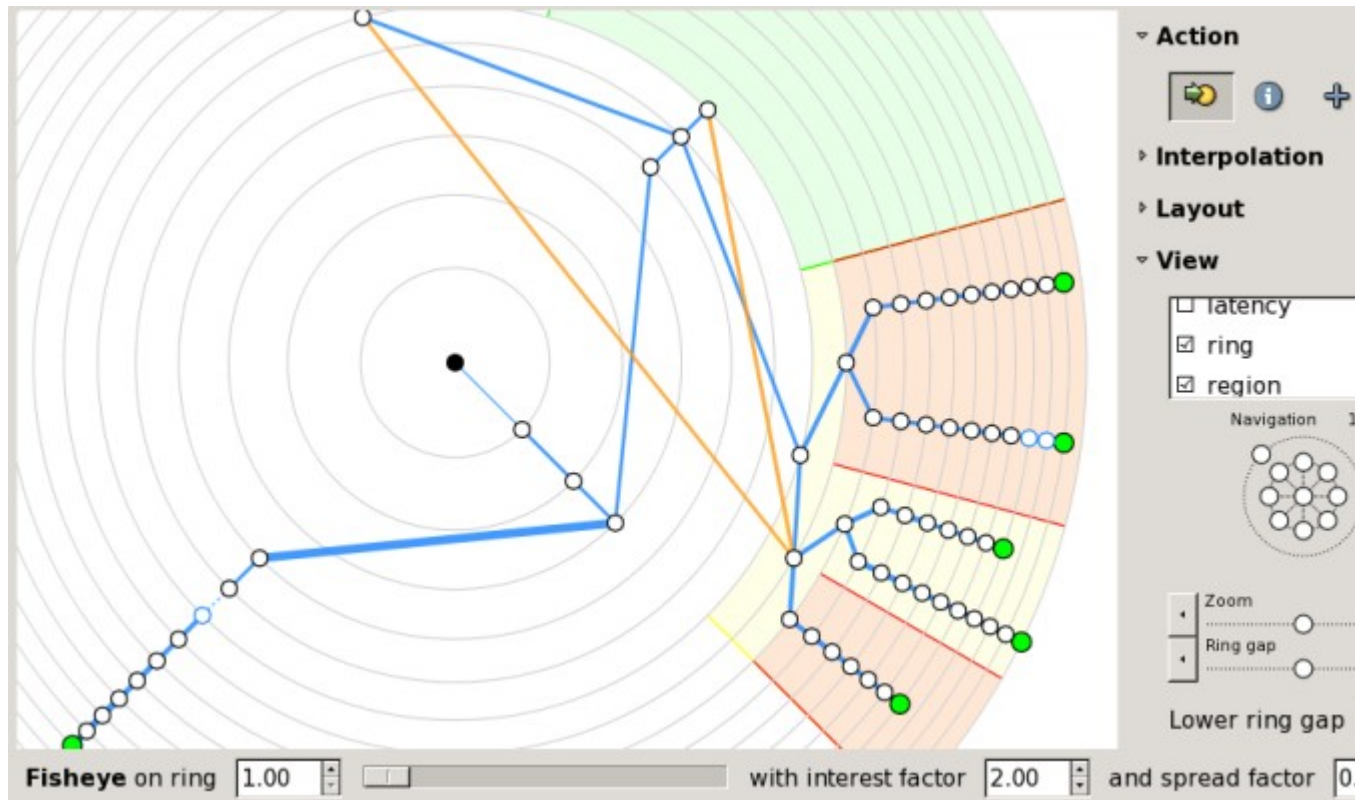
“Open Scan” opens loaded scans in a new window, thereby creating a new inventory. To merge loaded scans into the current inventory instead, use “Open Scan in This Window”.

The Recent Scans Database

Scan results that are not saved to a file are automatically stored in a database. Scan results that are loaded from a file, and are then modified (such as by the addition of a host comment) but not re-saved, are also stored in the database. The database is stored in a file called zenmap.db and its location is platform-dependent (see [the section called “Files Used by Zenmap”](#)). By default, scans are kept in the database for 60 days and then removed. This time interval can be changed by modifying the value of the save_time variable in the [search] section of zenmap.conf (see [the section called “Description of zenmap.conf”](#)).

Zenmap's search interface, because it searches the contents of the recent scans database by default, doubles as a database viewer. On opening the search window every scan in the database is shown. The list of scans may then be filtered by a search string. See [the section called “Searching Saved Results”](#).

Surfing the Network Topology



An Overview of the “Topology” Tab

Zenmap's “Topology” tab provides an interactive, animated visualization of the connections between hosts on a network. Hosts are shown as nodes on a graph that extends radially from the center. Click and drag to pan the display, and use the controls provided to zoom in and out. Click on a host and it becomes the new center. The graph rearranges itself in a smooth animation to reflect the new view of the network. Run a new scan and every new host and network path will be added to the topology automatically.








The topology view is most useful when combined with Nmap's `--traceroute` option, because that's the option that discovers the network path to a host. You can view a network inventory that doesn't have traceroute information in the topology, but network paths will not be visible. Remember, though, that you can add traceroute information to a network inventory just by running another scan thanks to Zenmap's scan aggregation.


Initially the topology is shown from the point of view of localhost, with you at the center. Click on a host to move it to the center and see what the network looks like from its point of view.

The topology view is an adaptation of the RadialNet program by João Paulo S. Medeiros.

Legend


The topology view uses many symbols and color conventions. This section explains what they mean.

-  Each regular host in the network is represented by a little circle. The color and size of the circle is determined by the
-  number of open ports on the host. The more open ports, the
-  larger the circle. A white circle represents an intermediate host
-  in a network path that was not port scanned. If a host has fewer than three open ports, it will be green; between three and six open ports, yellow; more than six open ports, red.
-  If a host is a router, switch, or wireless access point, it is drawn with a square rather than a circle.
- 
- 



-  Network distance is shown as concentric gray rings. Each additional ring signifies one more network hop from the center host.




Connections between hosts are shown with colored lines. Primary traceroute connections are shown with blue lines. Alternate paths (paths between two hosts where a different path already exists) are drawn in orange. Which path is primary and which paths are alternates is arbitrary and controlled by the order in which paths were recorded. The thickness of a line is proportional to its round-trip time; hosts with a higher RTT have a thicker line. Hosts with no traceroute information are clustered around localhost, connected with a dashed black line.



-  If there is no RTT for a hop (a missing traceroute entry), the connection is shown with a blue dashed line and the unknown host that makes the connection is shown with a blue outline.

Some special-purpose hosts may carry one or more icons describing what type of host they are:

-  A router.
-  A switch.

-  A wireless access point.
-  A firewall.
-  A host with some ports filtered.

Controls

The controls appear in a column when the “Controls” button is clicked. The controls are divided into sections.

Action controls



The controls in the “Action” section control what happens when you click on a host. The buttons in this section are, from left to right, “Change focus”, “Show information”, “Group children”, and “Fill region”. When the mode is “Change focus”, clicking on a host rearranges the display to put the selected host at the center. When the mode is “Show information”, clicking on a host brings up a window with information about it.

When the mode is “Group children”, clicking a host collapses into it all of its children—those nodes that are farther from the center.


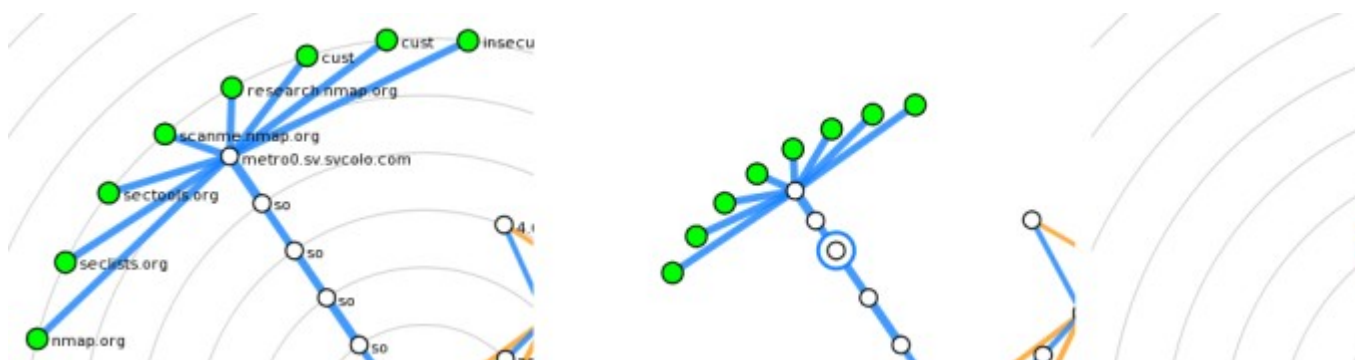
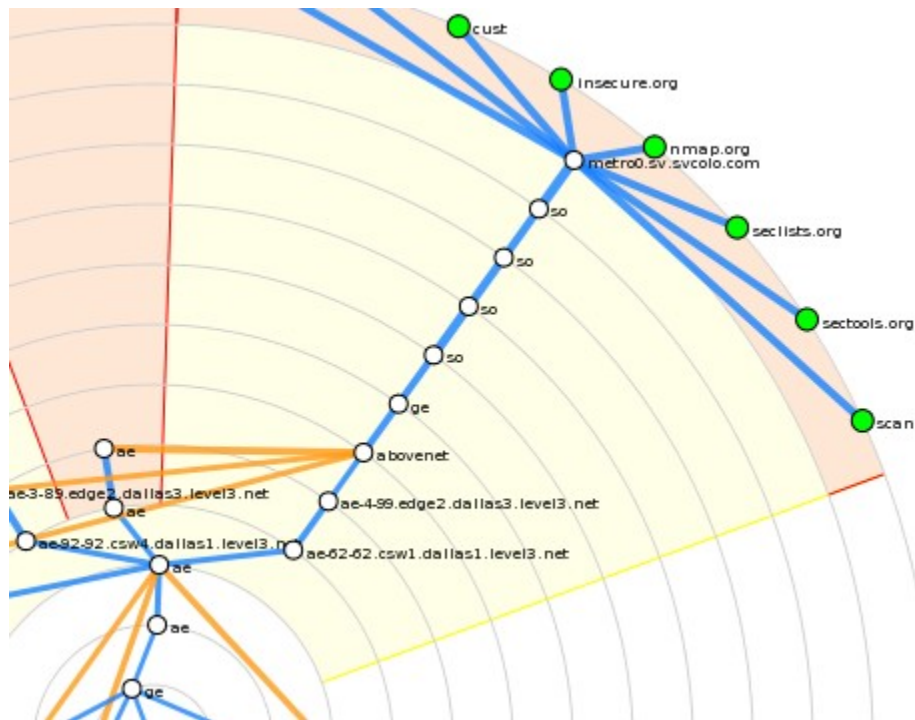
When a host is grouped it appears thus: . Clicking on a grouped node ungroups it again. This diagram shows the process of grouping.

Figure 12.7. Grouping a host's children



When the mode is “Fill region”, clicking a host highlights the region of the display occupied by the host and its children. The highlighted hosts are exactly the same as those that would be grouped in “Group children” mode. You can choose different colors to highlight different regions. This diagram shows an example of several regions highlighted in different colors.

Figure 12.8. Highlighting regions of the topology

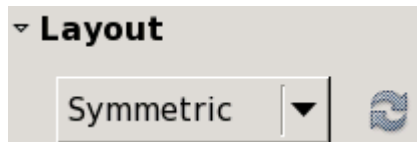


Interpolation controls



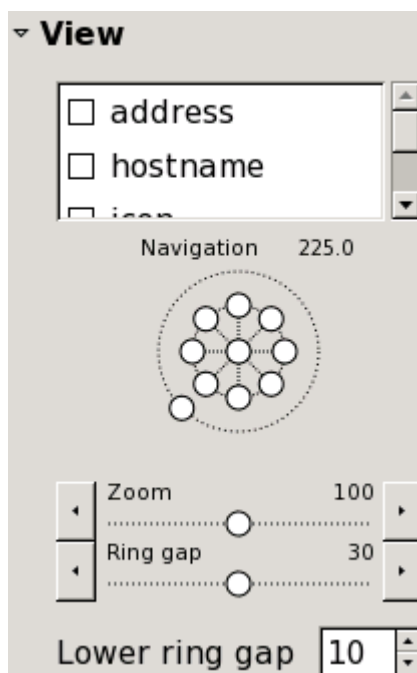
The controls in the “Interpolation” section control how quickly the animation proceeds when part of the graph changes.

Layout controls



There are two options for the automatic layout of nodes. Symmetric mode gives each subtree of a host an equal-sized slice of the graph. It shows the network hierarchy well but hosts far from the center can be squeezed close together. Weighted mode gives hosts with more children a larger piece of the graph.

View controls



The checkboxes in the “View” section enable and disable parts of the display. For example, disable “hostname” to show only an IP address for each host, or disable “address” to use no labels at all. The “latency” option enables and disables the display of the round-trip times to each host, as determined by Nmap's --traceroute option. If “slow in/out” is checked, the animation will not be linear, but will go faster in the middle of the animation and slower at the beginning and end.

The compass-like widget pans the screen in eight directions. Click the center to return to the center host. The ring around the outside controls the rotation of the entire graph.

“Zoom” and “Ring gap” both control the overall size of the graph. “Zoom” changes the size of everything—hosts, labels, connecting lines. “Ring gap” just increases the spacing between the concentric rings, keeping everything else the same size. “Lower ring gap” gives a minimum spacing for the rings, useful mainly when fisheye is enabled.

Fisheye controls



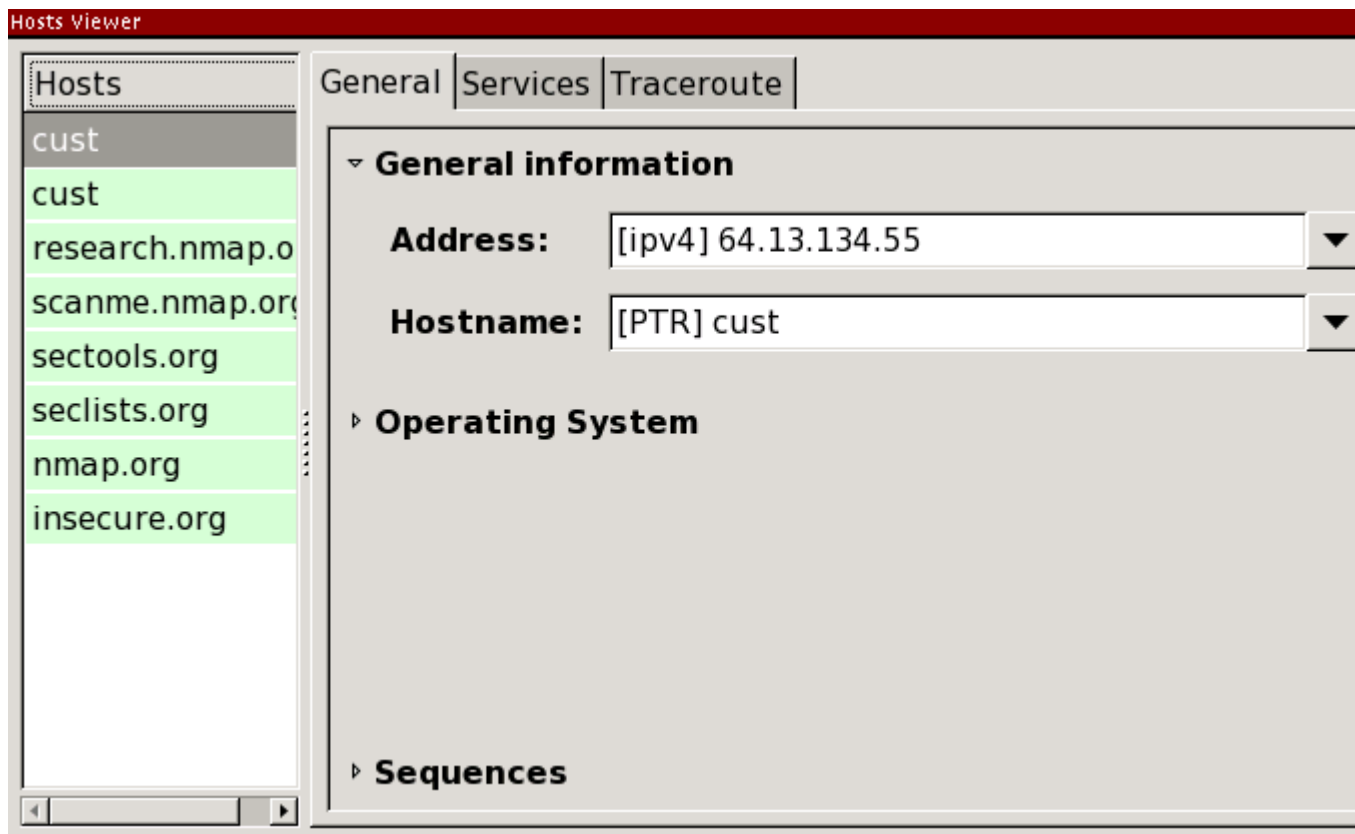
The fisheye controls give more space to a selected ring, compressing all the others. The slider controls which ring gets the most attention. The “interest factor” is how many times greater the ring spacing is for the chosen ring than it would be with no fisheye. The “spread factor” ranges from -1 to 1 . It controls how many adjacent rings are expanded around the selected ring, with higher numbers meaning more spread.

Keyboard Shortcuts

The topology display recognizes these keyboard shortcuts:

Key	Function
c	Return the display to the center host.
a	Show or hide host addresses.
h	Show or hide hostnames.
i	Show or hide host icons.
l	Show or hide latency.
r	Show or hide the rings.

The Hosts Viewer

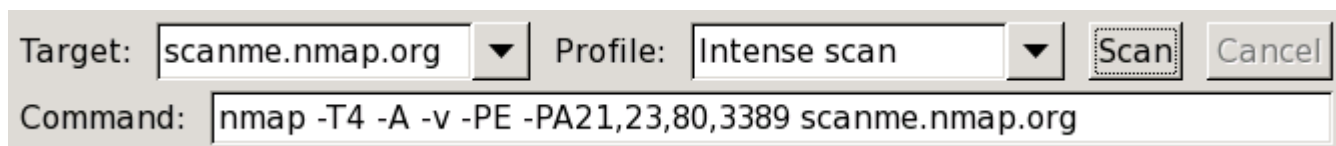


The host viewer is an alternative way to get details about hosts. Activate the viewer by clicking the “Hosts Viewer” button. All the hosts in the inventory are presented in a list. Select any host to get details about it.

The Profile Editor

It is common with Nmap to want to run the same scan repeatedly. For example, a system administrator may run a scan of an entire network once a month to keep track of things. Zenmap's mechanism for facilitating this is called profiles.

Figure 12.9. Choosing a profile



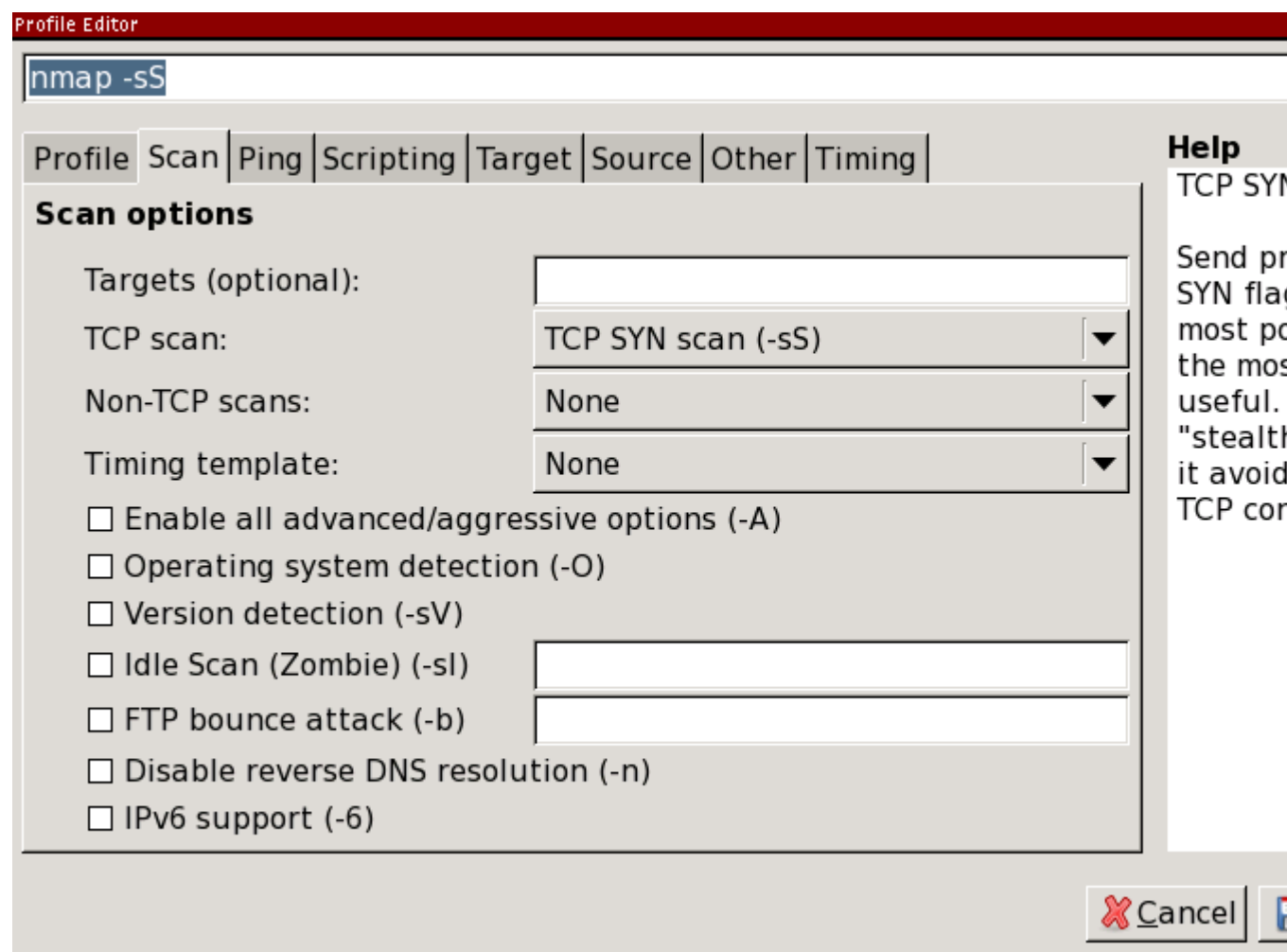
Each window contains a combo box labeled “Profile”. Opening it shows what profiles are available. Selecting a profile will cause the

“Command” field to display the command line that will be executed. The profiles that come with Zenmap are sufficient for many scanning purposes, but sooner or later you will want to create your own.

Editing a Command

The profile editor can be used as a handy interactive Nmap command editor. Select “New Profile or Command” from under the “Profile” menu or use the **ctrl+P** keyboard shortcut. The profile editor will appear, displaying whatever command was shown in the main window.

Figure 12.10. The profile editor



The text entry at the top shows the command being edited. You can type directly in this field if you know the options you want to use. The controls in the middle let you choose options by checking boxes or selecting from menus. There is a two-way relationship between the command string and the controls: when you change one of the controls it causes an immediate change in the command string, and when you edit the command string the controls update themselves to match. Hover the mouse pointer over an option to see a description of what it does and what kind of input it expects.

To run the new command line, click the “Scan” button. This will copy the command to the main window, dismiss the profile editor, and start running the scan. To make further changes to the command, just select “New Profile or Command” again, remembering that it will use whatever command is shown on the screen.

Creating a New Profile

The procedure for creating a new profile is almost the same as for editing a command. Select “New Profile or Command” from the “Profile” menu and edit the command as you wish. Then, instead of clicking “Scan”, go to the “Profile” tab and give a name to the profile. Then click “Save Changes” to save the new profile.

A profile may or may not include scan targets. If you often run the same scan against the same set of targets, you will find it convenient to list the targets within the profile. If you plan to run the same scan against different targets, leave the “Targets” field blank, and fill in the targets later, when you run the scan.

Editing or Deleting a Profile

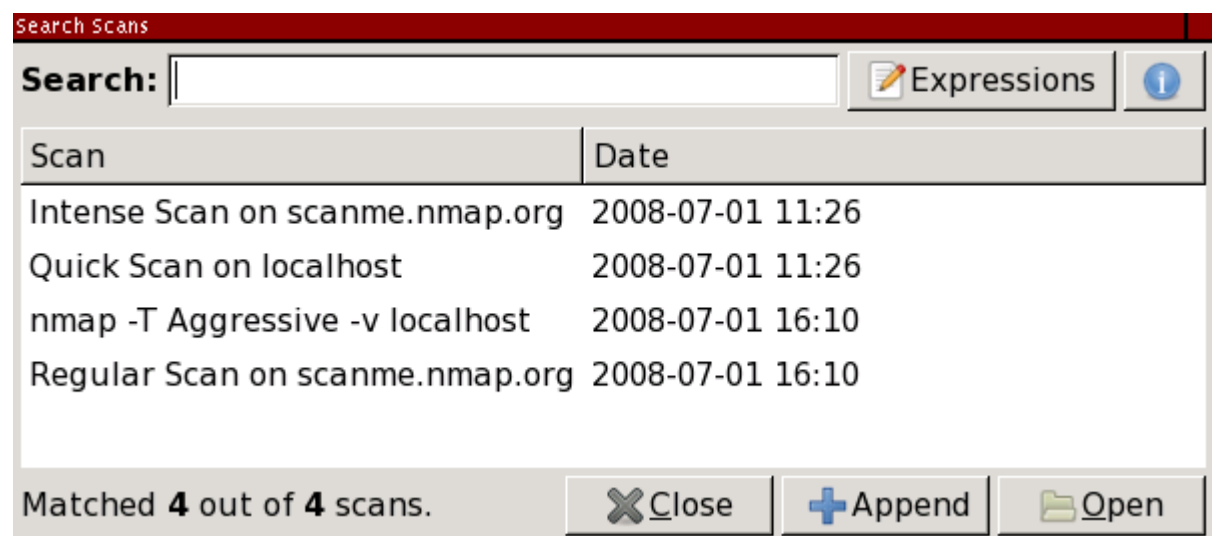
To edit a profile, select the profile you want to edit, then choose “Edit Selected Profile” from the “Profile” menu or use the **ctrl+E** keyboard shortcut. The profile editor will open, this time with the name and description filled from the profile selected. Click “Save Changes” to save any changes or “Cancel” to leave without saving.

When you open the profile editor using “Edit Selected Profile”, an additional “Delete” button will be present at the bottom. Zenmap will present a warning before deleting the profile.

Searching Saved Results

Zenmap allows you to search saved scan results files and the database of recent scans. To begin searching, select “Search Scan Results” from the “Tools” menu or use the **ctrl+F** keyboard shortcut. The search dialog appears as shown in [Figure 12.11](#).

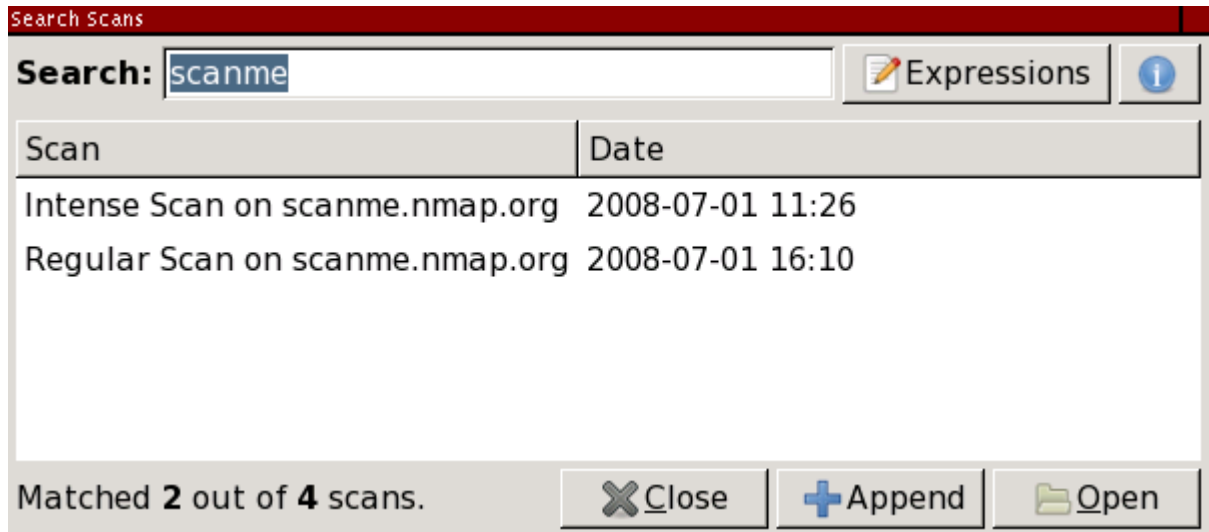
Figure 12.11. The search dialog



The search interface initially shows all the scans in the recent scans database (for which see [the section called “The Recent Scans Database”](#)). The reason all the scans are shown is simple—no restrictions have yet been placed on the search, so every possible result is returned.

Searches may be given in terms of several search criteria, however the simplest search is just a keyword search. Just type a word like scanme in the “Search” field to find all scans that have that word as part of their output, whether as a host name, operating system name, profile, or anything else. An example of this is shown in [Figure 12.12](#).

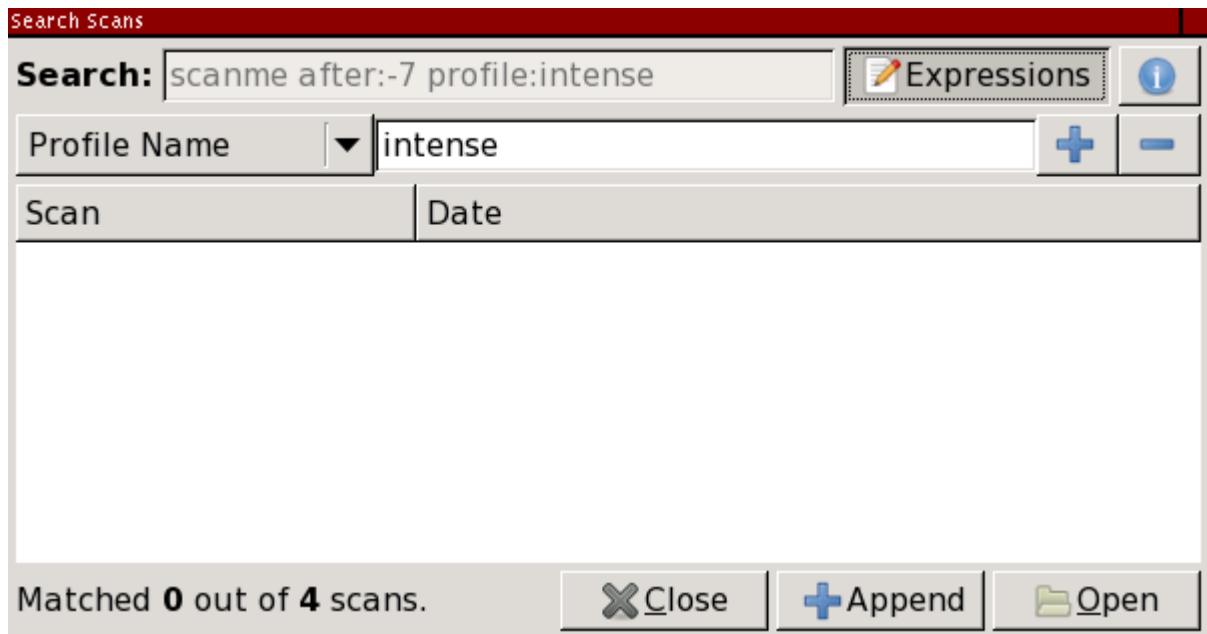
Figure 12.12. Keyword search



Searches happen live, as you type. When you have found the scan you want click the “Open” button or double-click on the scan name.

More complicated searches can be built up using the “Expressions” interface. Click the “Expressions” button and graphical representation of the current search will appear. Modify the search by selecting from the combo boxes displayed. Click “+” to add a criterion and “-” to remove one. Click the “Expressions” button again to hide the criteria (they are still present in the search string). Editing of the search text is disabled while the expressions are shown. An example of a more complicated search is shown in [Figure 12.13](#).

Figure 12.13. Expressions search



Search Scans

Search: scanme after:-7 profile:intense

Expressions

Profile Name intense

Scan	Date
------	------

Matched 0 out of 4 scans.

Close Append Open

Searches are and-based, meaning that all the criteria must be true for a scan to match and appear in the results list. Most searches are case-insensitive. (The only case-sensitive criterion is option:.) By default only the scans in the recent scans database are searched. To recursively search files in a directory, use the “Include Directory” expression.

You will have noticed that whenever you choose a search expression a text representation of it appears in the search entry. The string in the “Search” field is what really controls the search; the “Expressions” interface is just a convenient way to set it. When you have learned what search strings correspond to what expressions, you may skip the expressions interface and just type in a search string directly.

The following is a list of all the textual search criteria recognized by the search interface. Most criteria have a short form: d:-5 is the same as date:-5 and op:80 is the same as open:80. The short form of each criterion is given in the list below.

<keyword>

An unadorned word matches anything in a scan. For example, apache will match all Apache servers and linux will match all Linux hosts. There is a chance of false positives when using

the keyword search, like if a host happens to be named apache or linux.

Port states

Every possible port state is also a search criterion. They are

open:<ports>(op: for short)
closed:<ports>(cp: for short)
filtered:<ports>(fp: for short)
unfiltered:<ports>(ufp: for short)
open|filtered:<ports>(ofp: for short)
closed|filtered:<ports>(cfp: for short)

Use open:80 to match scans that have a host with port 80 open. The <ports> argument may also be a comma-separated list.

Additionally the scanned:<ports>(sp: for short) criterion matches scans in which the given ports were scanned, whatever their final state.

date:<YYYY-MM-DD> or date:-<n>(d: for short)

Matches scans that occurred on the given date in <YYYY-MM-DD> format. Or use date:-<n> to match scans that occurred any on the day <n> days ago. Use date:-1 to find scans performed yesterday.

When using the <YYYY-MM-DD> format, the date may be followed by one or more ~, each of which widens the range of dates matched by one day on both sides. date:2007-12-23 matches scans that occurred between 00:00 and 24:00 on December 23, 2007. date:2007-12-23~ matches scans that took place between 00:00 on December 22 and 24:00 on December 24. This “fuzzy” date matching is useful when you can't remember exactly when you ran a scan.

after:<YYYY-MM-DD> or after:-<n>(a: for short)

Matches scans that occurred on or after the given date in <YYYY-MM-DD> format. Or use after:-<n> to match scans that occurred within the last <n> days. For example, after:-7 matches scans that happened in the last week.

before:<YYYY-MM-DD> or before:-<n>(b: for short)

Matches scans that occurred on or before the given date in <YYYY-MM-DD> format. Or use before:-<n> to match scans that occurred any time before <n> days ago.

target:<name>(t: for short)

Matches scans of any hosts with the given name. The name may be either the name specified in the scan or the reverse-DNS name of any host.

option:<option>(o: for short)

Matches scans that used the given command-line option. Omit any leading - or --: option:A matches scans that used the -A option.

This criterion matches only literally. option:O will not match scans that used -A, even though -A implies -O. Similarly option:sU will not match scans that used -sSU. Option matching is case-sensitive.

os:<string>

Matches scans of hosts with the given string in any part of their OS description. os:windows will return scans of Microsoft Windows hosts broadly.

service:<string>(s: for short)

Matches scans of hosts with the given string in any part of the service description of any of their ports. service:ssh will return scans of hosts running any type of SSH.

profile:<name>(pr: for short)

Matches scans that used the named profile, for example profile:"intense scan".

inroute:<host>(ir: for short)

Matches scans where the given host appears as an intermediate router in --traceroute output.

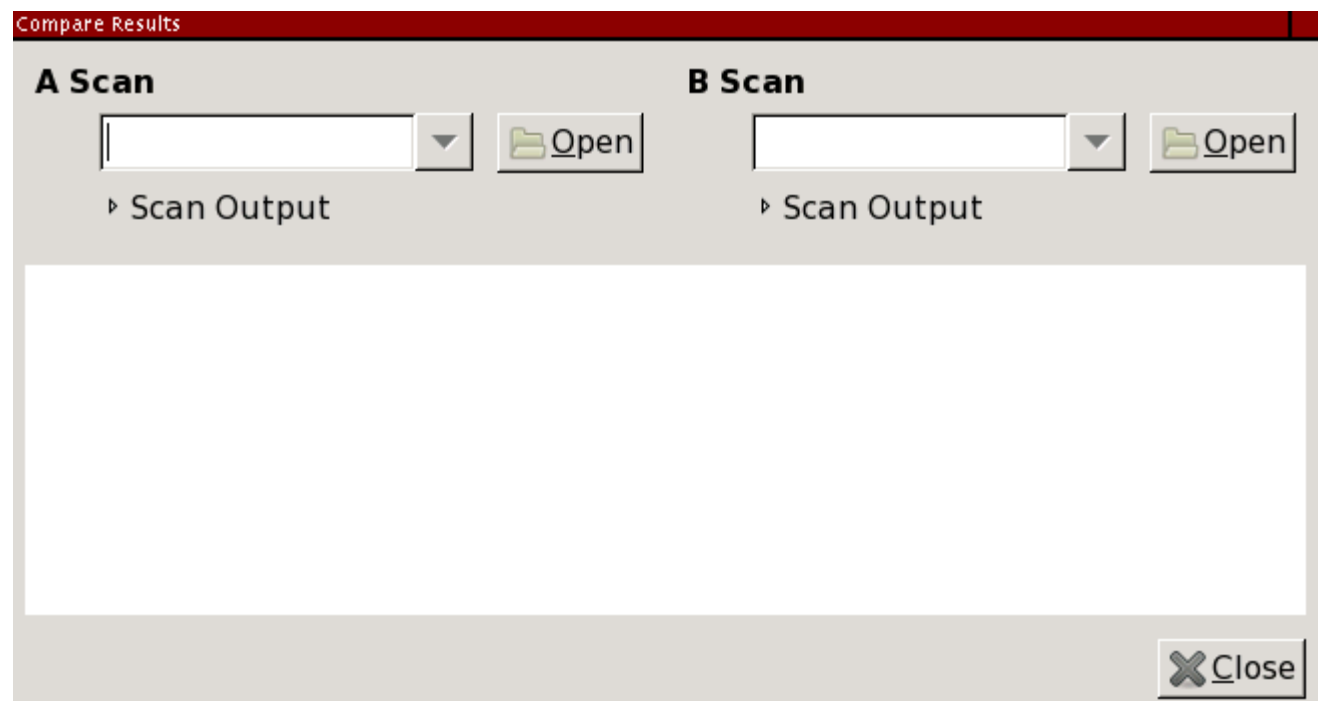
dir:<directory>

dir: is not really a search criterion. Rather it is the way to search a directory in the filesystem in addition to those in the recent scans database. Directories are searched recursively for files ending with certain extensions, xml only by default. To match more file names modify the file_extension variable of the [search] section of zenmap.conf according to the instructions in [the section called “Sections of zenmap.conf”](#).

Comparing Results

It is a common desire to run the same scan twice at different times, or run two slightly different scans at the same time, and see how they differ. Zenmap provides an interface for comparing scan results, shown in [Figure 12.14](#). Open the comparison tool by selecting “Compare Results” from the “Tools” menu or by using the **ctrl+D** (think “diff”) keyboard shortcut. Zenmap supports comparing two scan results at a time.

Figure 12.14. Comparison tool

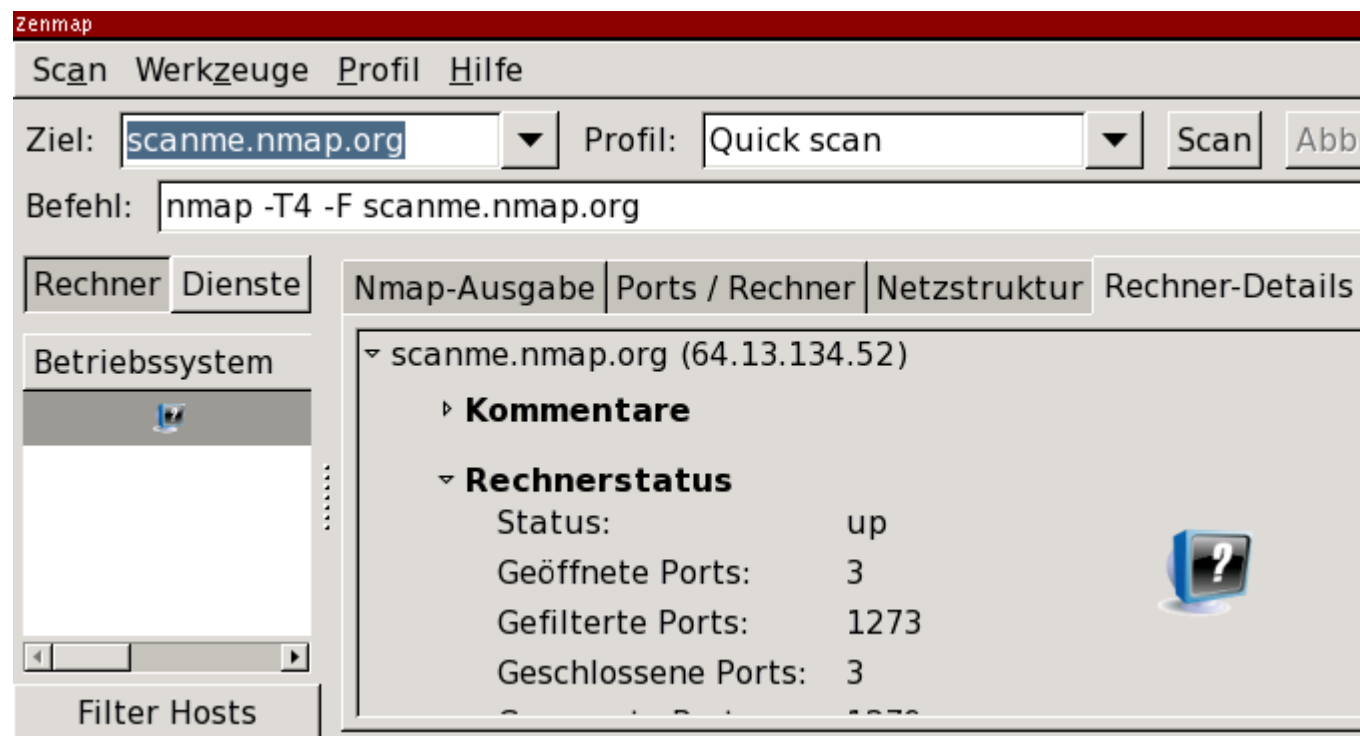


The first step in performing a comparison is selecting two scans to compare, which are called the “A scan” and the “B scan”. The

Zenmap in Your Language

Zenmap has been translated into a few languages other than English. [Figure 12.16](#) shows what Zenmap looks like in German. This section shows how to use Zenmap's translations.

Figure 12.16. Zenmap in German



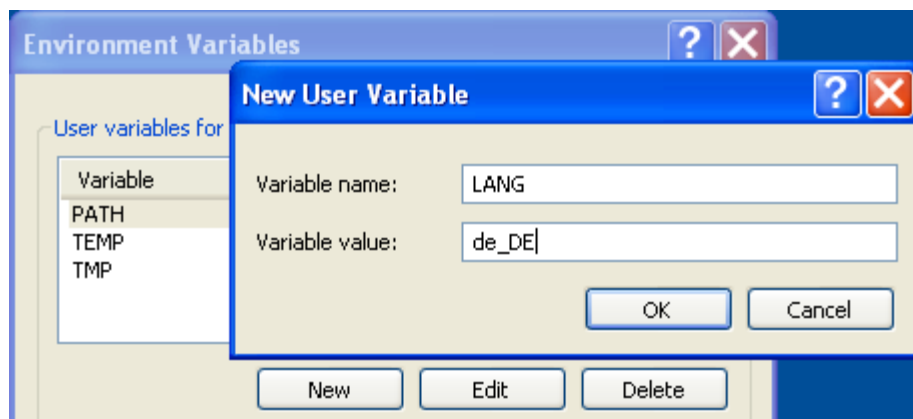
On Unix-like systems, you select your preferred language by setting the LANG environment variable. Even on other operating systems with different language selection facilities, setting LANG is the most foolproof way to get translations when other methods don't work. Your Unix-like operating system may set LANG as a side effect of its language configuration. If it does not, add a line like the following to your .login or .profile, replacing de with your locale name:

```
export LANG=de
```

A locale name is a language code optionally followed by a country code and sometimes other information. Language codes are from [ISO 639](#) and country codes are from [ISO 3166](#). Here “de” means German; another example is “pt_BR” for Brazilian Portuguese.

To set LANG on Windows XP, follow these steps. Open the Control Panel and choose the “System” item. Click the “Advanced” tab and then the “Environment Variables” button. A new display will open up; under “User variables” click “New”. In the form that appears enter “LANG” for the variable name and your locale name for the value. The process is illustrated in [Figure 12.17](#).

Figure 12.17. Setting the LANG environment variable on Windows XP



On Mac OS X, setting LANG in a shell startup file as described above has an effect only when Zenmap is started from a terminal. The graphical Finder interface keeps its environment variables in a separate file, `.MacOSX/environment.plist`. To create it, open the TextEdit application and enter the following, replacing `de` with your locale name:

```
{ LANG=de; }
```

Then from the “Format” menu choose “Make Plain Text”. Open the “Save” dialog, select your home directory and click “New Folder”. Create a folder called `.MacOSX` and click past the warning that appears. Save the file with the name `environment.plist`, and insist on the extension `.plist` in the next warning that appears. Finally, log out and back in to make the change take effect. A portion of this process is shown in [Figure 12.18](#).

Figure 12.18. Setting the LANG environment variable on Mac OS X



Creating a new translation

Creating a new translation for Zenmap, or updating an existing one, is not hard technically, though of course it requires some know-how and knowledge of at least one language other than English. Zenmap's translations are handled by the GNU gettext system, which is documented fully at <http://www.gnu.org/software/gettext/manual/>. This section is a summary of the manual translation process. There also exist specialized translation editing programs that do these steps automatically.

Let's say you are going to make a translation into Spanish, which has the language code "es". Within the Zenmap source tree there is a plain-text file `share/zenmap/locale/zenmap.pot` containing all the translatable strings in the application. You create a new *portable object* (.po) file by running **`msginit -l es.po -i zenmap.pot`**. The new `es.po` file contains all the application's English strings followed by blank strings where you fill in the appropriate translations.

To test the portable object file you must turn it into a *machine object* (.mo) file. Create the directory `es/LC_MESSAGES` and then run **`msgfmt es.po -o es/LC_MESSAGES/zenmap.mo`**. With this file in place Zenmap will use your translation when `LANG` is set properly. To update the portable object file when `zenmap.pot` changes, run **`msgmerge -U es.po zenmap.pot`**. This will add new strings and mark any obsolete strings so they can be removed.

When you start a new translation, announce your intention to the *nmap-dev* mailing list. The mailing list is also a good place to get translation advice. Then when you are finished, send in the .po file.

Files Used by Zenmap

Zenmap uses a number of configuration and control files, and of course requires Nmap to be installed. Where the files are stored

depends on the platform and how Zenmap was configured. The configuration files are divided into two categories: system files and per-user files.

The nmap Executable

Zenmap depends on the nmap command-line executable being installed. The program is first searched for in all of the directories specified in the PATH environment variable.

On some platforms the nmap command isn't commonly installed in any of the directories in PATH. As a convenience for those platforms, the following additional directories will be searched if the command is not found in the PATH:

- On Mac OS X, the directory /usr/local/bin is searched.
- On Windows, the directory containing the Zenmap executable is searched.

To use an absolute path to the executable, or if the executable is installed under a name other than nmap, modify the nmap_command_path variable in the [paths] section of zenmap.conf. For example, if you have installed nmap in /opt/bin, use

```
[paths]
nmap_command_path = /opt/bin/nmap
```

Or if you have a custom-compiled version of Nmap called nmap-custom, use

```
[paths]
nmap_command_path = nmap-custom
```

See [the section called “Description of zenmap.conf”](#).

System Configuration Files

These files affect the installation of Zenmap across an entire installation. On Unix and Mac OS X, they are in `<prefix>/share/zenmap`, where `<prefix>` is the filesystem prefix Zenmap was compiled with. The prefix is likely /usr or /usr/local, so Zenmap's file are probably in /usr/share/zenmap or

/usr/local/share/zenmap. On Windows, the location also depends on where Zenmap was installed. They are probably in C:\Program Files\Nmap\share\zenmap. The Zenmap system configuration directory contains the following:

config/

The files under config are copied to per-user configuration directories. See [the section called “Per-user Configuration Files”](#).

docs/

The files in the docs subdirectory are Zenmap's documentation files.

locale/

The files in the locale/ subdirectory contain translations of the text used by Zenmap into other languages.

misc/profile_editor.xml

This file defines what options are presented by the profile editor (see [the section called “The Profile Editor”](#)). It can be edited with care to alter the profile editor system-wide.

Per-user Configuration Files

These files affect only one user of Zenmap. Some of them are copied from the config subdirectory of the system files when Zenmap is run for the first time. Per-user files are in <HOME>/.zenmap on Unix and Mac OS X, where <HOME> is the current user's home directory. They are in C:\Users\<USER>\.zenmap on Windows Vista and C:\Documents and Settings\<USER>\.zenmap on previous versions of Windows, where <USER> is the name of the current user.

recent_scans.txt

This contains a list of file names of recently saved scans. These scans are shown under the “Scan” menu. Scans must have been saved to a file to appear here. See [the section](#)

called [“Saving and Loading Scan Results”](#). If this file doesn't exist it is created when Zenmap is run.

scan_profile.usp

This file contains descriptions of scan profiles, including the defaults and user-created profiles. I recommend using the profile editor (see [the section called “The Profile Editor”](#)) to make changes to this file. This file is copied from the system configuration directory the first time Zenmap is run.

target_list.txt

This file contains a list of recently scanned targets. If it doesn't exist it is created when Zenmap is run.

zenmap.conf

This is Zenmap's main configuration file. It holds the settings for a particular user's copy of Zenmap and is discussed in more detail in [the section called “Description of zenmap.conf”](#).

zenmap.db

This is the database of recent scans, as described in [the section called “The Recent Scans Database”](#). It is created if it doesn't already exist.

zenmap_version

This file contains the version of Zenmap that was used to create this per-user configuration directory. It may be helpful to compare the version number in this file with the file of the same name in the system configuration directory if you suspect a version conflict. It is simply copied from the system configuration the first time Zenmap is run.

Output Files

Whenever a scan is run, Zenmap instructs Nmap to put XML output in a temporary file so that Zenmap can parse it. Normally the XML output file is deleted when the scan is finished. However, if the command line in Zenmap contains an -oX or -oA option, XML output is written to the named file instead, and that file isn't deleted when

the scan completes. In other words, -oX and -oA work the way you would expect. -oG, -oN, and -oS work too, even though Zenmap doesn't use the output files produced by those options.

There is one important thing to note in Zenmap's handling of these filenames. Percent characters (%) are escaped to keep them from being interpreted as strftime-like format specifiers (see [the section called “Controlling Output Type”](#)). This is because Zenmap must know exactly what name Nmap will use for its output file. If in Zenmap you type -oX scan-%T-%D.xml, the output file will be saved in the file scan-%T-%D.xml, not scan-144840-121307.xml or whatever it would have been based on the current time and date if you were executing Nmap directly.

Description of zenmap.conf

zenmap.conf is the user-specific configuration file for Zenmap. It is a plain text file located in the per-user configuration directory (see [the section called “Per-user Configuration Files”](#)). The syntax is that recognized by the Python [ConfigParser](#) module, which is similar to that of Windows INI files. Sections are delimited by titles in square brackets. Within sections are lines containing <name>=<value> or <name>: <value> pairs. An excerpt from a zenmap.conf is shown.

```
[output_highlight]
enable_highlight = True

[paths]
nmap_command_path = nmap
ndiff_command_path = ndiff

[search]
search_db = 1
file_extension = xml
store_results = 1
directory =
save_time = 60;days
```

Some of these settings can be controlled from within Zenmap without editing the configuration file directly.

Sections of zenmap.conf

Boolean values are normalized from True, true, or 1 to true or anything else to false.

[paths]

The [paths] section defines important paths used by Zenmap.

nmap_command_path

The path to the Nmap executable. Whatever the first word is in a command line executed by Zenmap will be replaced by the value of this variable. Its default value of nmap is appropriate for most systems. See [the section called “The nmap Executable”](#) for examples.

ndiff_command_path

The path to the Ndiff scan comparison utility. Zenmap uses Ndiff to do scan comparisons; see [the section called “Comparing Results”](#).

[search]

The [search] section defines how the search tool (see [the section called “Searching Saved Results”](#)) behaves. The names in this section correspond to the options in the “Search options” tab of the search dialog. It has the following names defined.

directory

The directory to search for saved scan results files.

file_extension

A semicolon-separated list of file name extensions to search.

search_db

A Boolean controlling whether to search the recent scans database.

store_results

A Boolean controlling whether to store scan results in the recent scans database. See [the section called “The Recent Scans Database”](#).

save_time

How long to keep scan results in the recent scans database. Results older than this are deleted when Zenmap is closed. The format is a number and a time interval separated by semicolons, for example 60;days or 1;years.

[diff]

The [diff] section defines how the comparison tool (see [the section called “Comparing Results”](#)) behaves. It has the following names defined.

diff_mode

Controls whether comparisons are shown by default in graphical or text mode. Must be either compare for graphical mode or text.

colored_diff

A Boolean that controls if comparisons use color.

[diff_colors]

The [diff_colors] section defines the colors used by the comparison tool. It has the following names defined: unchanged, added, not_present, and modified, the meanings of which are defined in [the section called “Comparing Results”](#). The value of each of these is a list of three integers in the range 0–65535 representing red, green, and blue in the format [*<red>*, *<green>*, *<blue>*]. For example, [65535, 0, 0] specifies red.

[output_highlight]

The [output_highlight] section contains a single Boolean variable enable_highlight, which enables output highlighting when True and disables it if False.

[date_highlight],

[hostname_highlight],

[ip_highlight],

[port_list_highlight], [open_port_highlight], [closed_port_highlight], [filtered_port_highlight], [details_highlight]

These sections all define the nature of Nmap output highlighting, which is discussed in [the section called “The Nmap Output tab”](#). These are best edited from within Zenmap. Within each of these sections, the following names are defined.

regex

The regular expression that matches the relevant part of the output.

bold

A Boolean controlling whether to make this highlight bold.

italic

A Boolean controlling whether to make this highlight italic.

underline

A Boolean controlling whether to underline this highlight.

text

The color of the text in this highlight. The syntax is a list of three integers in the range 0–65535 representing red, green, and blue in the format [*<red>*, *<green>*, *<blue>*]. For example, [65535, 0, 0] for a red highlight.

highlight

The color of the background in this highlight. The syntax is the same as for text.

Command-line Options

Being a graphical application, most of Zenmap's functionality is exposed through its graphical interface. Zenmap's command-line options are given here for completeness and because they are sometimes useful. In particular, it's good to know that the command

zenmap *<results file>* starts Zenmap with the results in *<results file>* already open.

Synopsis

zenmap [*<options>*] [*<results file>*]

Options Summary

-f, --file *<results file>*

Open the given results file for viewing. The results file may be an Nmap XML output file (.xml, as produced by **nmap -oX**), or a file previously saved by Zenmap.

-h, --help

Show a help message and exit.

-n, --nmap *<Nmap command line>*

Run the given Nmap command within the Zenmap interface. After -n or --nmap, every remaining command line argument is read as the command line to execute. This means that -n or --nmap must be given last, after any other options. Note that the command line must include the **nmap** executable name: **zenmap -n nmap -sS target**.

-p, --profile *<profile>*

Start with the given profile selected. The profile name is just a string: "Regular scan". If combined with -t, begin a scan with the given profile against the specified target.

-t, --target *<target>*

Start with the given target. If combined with -p, begin a scan with the given profile against the specified target.

-v, --verbose

Increase verbosity (of Zenmap, not Nmap). This option may be given multiple times for even more verbosity printed to the console window used to start Zenmap.

Error Output

If Zenmap happens to crash, it normally helps you send a bug report with a stack trace. Set the environment variable `ZENMAP_DEVELOPMENT` (the value doesn't matter) to disable automatic crash reporting and have errors printed to the console. Try the Bash shell command **`ZENMAP_DEVELOPMENT=1 zenmap -v -v -v`** to get a useful debugging output.

On Windows, standard error is redirected to the file `zenmap.exe.log` in the same directory as `zenmap.exe` rather than being printed to the console.

History

Zenmap was originally derived from [Umit](#), an Nmap GUI created during the Google-sponsored Nmap Summer of Code in 2005 and 2006. The primary author of Umit was Adriano Monteiro Marques. When Umit was modified and integrated into Nmap in 2007, it was renamed Zenmap.

Chapter 13. Nmap Output Formats

Table of Contents

[Introduction](#)

[Command-line Flags](#)

[Controlling Output Type](#)

[Controlling Verbosity of Output](#)

[Enabling Debugging Output](#)

[Handling Error and Warning Messages](#)

[Enabling Packet Tracing](#)

[Resuming Aborted Scans](#)

[Interactive Output](#)

[Normal Output \(-oN\)](#)

[Script Output \(-oS\)](#)

[XML Output \(-oX\)](#)

- Using XML Output
- Manipulating XML Output with Perl
- Output to a Database
- Creating HTML Reports
 - Saving a Permanent HTML Report
- Grepable Output (-oG)
 - Grepable Output Fields
 - Host field
 - Ports field
 - Protocols field
 - Ignored State field
 - OS field
 - Seq Index field
 - IP ID Seq field
 - Status field
 - Parsing Grepable Output on the Command Line

Introduction

A common problem with open-source security tools is confusing and disorganized output. They often spew out many lines of irrelevant debugging information, forcing users to dig through pages of output trying to discern important results from the noise. Program authors often devote little effort to organizing and presenting results effectively. The output messages can be difficult to understand and poorly documented. This shouldn't be too surprising—writing clever code to exploit some TCP/IP weakness is usually more gratifying than documentation or UI work. Since open source authors are rarely paid, they do what they enjoy.

At the risk of offending my friend Dan Kaminsky, I'll name his [Scanrand](#) port scanner as an example of a program that was clearly developed with far more emphasis on neat technical tricks than a user friendly UI. The sample output in [Example 13.1](#) is from the Scanrand documentation page.

Example 13.1. Scanrand output against a local network

```
bash-2.05a# scanrand 10.0.1.1-254:quick
UP:      10.0.1.38:80      [01]    0.003s
UP:      10.0.1.110:443    [01]    0.017s
UP:      10.0.1.254:443    [01]    0.021s
UP:      10.0.1.57:445     [01]    0.024s
```

```

UP:      10.0.1.59:445    [01]    0.024s
UP:      10.0.1.38:22     [01]    0.047s
UP:      10.0.1.110:22    [01]    0.058s
UP:      10.0.1.110:23    [01]    0.058s
UP:      10.0.1.254:22    [01]    0.077s
UP:      10.0.1.254:23    [01]    0.077s
UP:      10.0.1.25:135    [01]    0.088s
UP:      10.0.1.57:135    [01]    0.089s
UP:      10.0.1.59:135    [01]    0.090s
UP:      10.0.1.25:139    [01]    0.097s
UP:      10.0.1.27:139    [01]    0.098s
UP:      10.0.1.57:139    [01]    0.099s
UP:      10.0.1.59:139    [01]    0.099s
UP:      10.0.1.38:111    [01]    0.127s
UP:      10.0.1.57:1025   [01]    0.147s
UP:      10.0.1.59:1025   [01]    0.147s
UP:      10.0.1.57:5000    [01]    0.156s
UP:      10.0.1.59:5000    [01]    0.157s
UP:      10.0.1.53:111    [01]    0.182s
bash-2.05a#

```

While this does get the job done, it is difficult to interpret. Output is printed based on when the response was received, without any option for sorting the port numbers or even grouping all open ports on a target host together. A bunch of space is wasted near the beginning of each line and no summary of results is provided.

Nmap's output is also far from perfect, though I do try pretty hard to make it readable, well-organized, and flexible. Given the number of ways Nmap is used by people and other software, no single format can please everyone. So Nmap offers several formats, including the interactive mode for humans to read directly and XML for easy parsing by software.

In addition to offering different output formats, Nmap provides options for controlling the verbosity of output as well as debugging messages. Output types may be sent to standard output or to named files, which Nmap can append to or clobber. Output files may also be used to resume aborted scans. This chapter includes full details on these options and every output format.

Command-line Flags

As with almost all other Nmap capabilities, output behavior is controlled by command-line flags. These flags are grouped by category and described in the following sections.

Controlling Output Type

The most fundamental output control is designating the format(s) of output you would like. Nmap offers five types, as summarized in the following list and fully described in later sections.

Output formats supported by Nmap

Interactive output

This is the output that Nmap sends to the standard output stream (stdout) by default. So it has no special command-line option. Interactive mode caters to human users reading the results directly and it is characterized by a table of interesting ports that is shown in dozens of examples throughout this book.

Normal output (-oN)

This is very similar to interactive output, and is sent to the file you choose. It does differ from interactive output in several ways, which derive from the expectation that this output will be analyzed after the scan completes rather than interactively. So interactive output includes messages (depending on verbosity level specified with -v) such as scan completion time estimates and open port alerts. Normal output omits those as unnecessary once the scan completes and the final interesting ports table is printed. This output type prints the nmap command-line used and execution time and date on its first line.

XML output (-oX)

XML offers a stable format that is easily parsed by software. Free XML parsers are available for all major computer languages, including C/C++, Perl, Python, and Java. In almost all cases that a non-trivial application interfaces with Nmap, XML is the preferred format. This chapter also discusses how

XML results can be transformed into other formats, such as HTML reports and database tables.

Grepable output (-oG)

This simple format is easy to manipulate on the command line with simple Unix tools such as `grep`, `awk`, `cut`, and `diff`. Each host is listed on one line, with the tab, slash, and comma characters used to delimit output fields. While this can be handy for quickly grokking results, the XML format is preferred for more significant tasks as it is more stable and contains more information.

sCRiPt KiDDi3 OutPU+ (-oS)

This format is provided for the l33t haXXorZ!

While interactive output is the default and has no associated command-line options, the other four format options use the same syntax. They take one argument, which is the filename that results should be stored in. Multiple formats may be specified, but each format may only be specified once. For example, you may wish to save normal output for your own review while saving XML of the same scan for programmatic analysis. You might do this with the options `-oX myscan.xml -oN myscan.nmap`. While this chapter uses the simple names like `myscan.xml` for brevity, more descriptive names are generally recommended. The names chosen are a matter of personal preference, though I use long ones that incorporate the scan date and a word or two describing the scan, placed in a directory named after the company I'm scanning. As a convenience, you may specify `-oA <basename>` to store scan results in normal, XML, and grepable formats at once. They are stored in `<basename>.nmap`, `<basename>.xml`, and `<basename>.gnmap`, respectively. As with most programs, you can prefix the filenames with a directory path, such as `~/nmaplogs/foocorp/` on Unix or `c:\hacking\sco` on Windows.

While these options save results to files, Nmap still prints interactive output to stdout as usual. For example, the command **`nmap -oX myscan.xml target`** prints XML to `myscan.xml` and fills standard output with the same interactive results it would have printed if `-oX` wasn't specified at all. You can change this by passing a hyphen character as the argument to one of the format types. This causes Nmap to deactivate interactive output, and instead print results in

the format you specified to the standard output stream. So the command `nmap -oX - target` will send only XML output to stdout. Serious errors may still be printed to the normal error stream, stderr.

When you specify a filename to an output format flag such as `-oN`, that file is overwritten by default. If you prefer to keep the existing content of the file and append the new results, specify the `--append-output` option. All output filenames specified in that Nmap execution will then be appended to rather than clobbered. This doesn't work well for XML (`-oX`) scan data as the resultant file generally won't parse properly until you fix it up by hand.

Unlike some Nmap arguments, the space between the logfile option flag (such as `-oX`) and the filename or hyphen is mandatory. If you omit the flags and give arguments such as `-oG-` or `-oXscan.xml`, a backwards compatibility feature of Nmap will cause the creation of *normal format* output files named `G-` and `Xscan.xml` respectively.

All of these arguments support strftime-like conversions in the filename. `%H`, `%M`, `%S`, `%m`, `%d`, `%y`, and `%Y` are all exactly the same as in strftime. `%T` is the same as `%H%M%S`, `%R` is the same as `%H%M`, and `%D` is the same as `%m%d%y`. A `%` followed by any other character just yields that character (`%%` gives you a percent symbol). So `-oX 'scan-%T-%D.xml'` will use an XML file in the form of `scan-144840-121307.xml`.

Controlling Verbosity of Output

After deciding which format(s) you wish results to be saved in, you can decide how detailed those results should be. The first `-v` option enables verbosity with a level of one. Specify `-v` twice for a slightly greater effect. Verbosity levels greater than two aren't useful. Most changes only effect interactive output, and some also affect normal and script kiddie output. The other output types are meant to be processed by machines, so Nmap can give substantial detail by default in those formats without fatiguing a human user. However, there are a few changes in other modes where output size can be reduced substantially by omitting some detail. For example, a comment line in the greppable output that provides a list of all ports scanned is only printed in verbose mode because it can be quite long. The following list describes the major changes you get with at least one `-v` option.

Scan completion time estimates

On scans that take more than a minute or two, you will see occasional updates like this in interactive output mode:

SYN Stealth Scan Timing: About 30.01% done; ETC: 16:04 (0:01:09 remaining)

New updates are given if the estimates change significantly. All port scanning techniques except for idle scan and FTP bounce scan support completion time estimation, and so does version scanning.

Open ports reported when discovered

When verbosity is enabled, open ports are printed in interactive mode as they are discovered. They are still reported in the final interesting ports table as well. This allows users to begin investigating open ports before Nmap even completes. Open port alerts look like this:

Discovered open port 53/tcp on 64.13.134.52

Additional warnings

Nmap always prints warnings about obvious mistakes and critical problems. That standard is lowered when verbosity is enabled, allowing more warnings to be printed. There are dozens of these warnings, covering topics from targets experiencing excessive drops or extraordinarily long latency, to ports which respond to probes in unexpected ways. Rate limiting prevents these warnings from flooding the screen.

Additional notes

Nmap prints many extra informational notes when in verbose mode. For example, it prints out the time when each port scan is started along with the number of hosts and ports scanned. It later prints out a concluding line disclosing how long the scan took and briefly summarizing the results.

Extra OS detection information

With verbosity, results of the TCP ISN and IP ID sequence number predictability tests are shown. These are done as a


```
--
nmap_rpc.cc: if (o.debugging || o.verbose)
nmap_rpc.cc-   gh_perror("recvfrom in get_rpc_results");
--
ossan.cc: if (o.verbose && openport != (unsigned long)
-1)
ossan.cc-   log_write(LOG_STDOUT, "For OSScan assuming
port %d is open, %d..."
--
output.cc: if (o.verbose)
output.cc-   log_write(LOG_NORMAL|LOG_SKID|LOG_STDOUT,
               "IP ID Sequence Generation:
%s\n", ...
```

The following two examples put all of this together. [Example 13.3](#) shows the output of a normal scan without the -v option.

Example 13.3. Interactive output without verbosity enabled

```
# nmap -T4 -A -p- scanme.nmap.org

Starting Nmap ( http://nmap.org )
Interesting ports on scanme.nmap.org (64.13.134.52):
Not shown: 65529 filtered ports
PORT      STATE  SERVICE VERSION
22/tcp    open   ssh      OpenSSH 4.3 (protocol 2.0)
25/tcp    closed smtp
53/tcp    open   domain   ISC BIND 9.3.4
70/tcp    closed gopher
80/tcp    open   http      Apache httpd 2.2.2 ((Fedora))
|_ HTML title: Go ahead and ScanMe!
113/tcp   closed auth
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.17 - 2.6.21, Linux 2.6.23

TRACEROUTE (using port 22/tcp)
HOP RTT    ADDRESS
1    16.92  node-m-sfc-vl245-act-security-gw-1-113.ucsd.edu
(132.239.1.113)
[... nine similar lines cut ...]
11   21.97  scanme.nmap.org (64.13.134.52)
```

```
OS and Service detection performed. Please report any
incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 168.10
seconds
```

[Example 13.4](#) is the output of the same scan with verbosity enabled. Features such as the extra OS identification data, completion time estimates, open port alerts, and extra informational messages are easily identified in the latter output. This extra info is often helpful during interactive scanning, so I always specify `-v` when scanning a single machine unless I have a good reason not to.

Example 13.4. Interactive output with verbosity enabled

```
# nmap -v -T4 -A -p- scanme.nmap.org
Starting Nmap ( http://nmap.org )
Initiating Ping Scan at 00:12
Completed Ping Scan at 00:12, 0.02s elapsed (1 total
hosts)
Initiating SYN Stealth Scan at 00:12
Scanning scanme.nmap.org (64.13.134.52) [65535 ports]
Discovered open port 80/tcp on 64.13.134.52
Discovered open port 53/tcp on 64.13.134.52
Discovered open port 22/tcp on 64.13.134.52
SYN Stealth Scan Timing: About 16.66% done; ETC: 00:15
(0:02:30 remaining)
Completed SYN Stealth Scan at 00:14, 125.13s elapsed
(65535 total ports)
Scanning 3 services on scanme.nmap.org (64.13.134.52)
Completed Service scan at 00:14, 6.05s elapsed (3
services on 1 host)
Initiating OS detection (try #1) against scanme.nmap.org
(64.13.134.52)
[Removed some verbose traceroute and parallel DNS related
messages]
Initiating SCRIPT ENGINE at 00:14
Completed SCRIPT ENGINE at 00:14, 4.09s elapsed
Host scanme.nmap.org (64.13.134.52) appears to be up ...
good.
Interesting ports on scanme.nmap.org (64.13.134.52):
Not shown: 65529 filtered ports
PORT      STATE  SERVICE VERSION
22/tcp    open   ssh      OpenSSH 4.3 (protocol 2.0)
```

```
25/tcp  closed smtp
53/tcp  open   domain  ISC BIND 9.3.4
70/tcp  closed gopher
80/tcp  open   http    Apache httpd 2.2.2 ((Fedora))
|_ HTML title: Go ahead and ScanMe!
113/tcp closed auth
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.17 - 2.6.21, Linux 2.6.23
Uptime guess: 12.476 days (since Wed Jul  2 12:48:56
2008)
TCP Sequence Prediction: Difficulty=198 (Good luck!)
IP ID Sequence Generation: All zeros

TRACEROUTE (using port 22/tcp)
HOP RTT    ADDRESS
1   0.25    nodem-msfc-vl245-act-security-gw-1-113.ucsd.edu
(132.239.1.113)
[... nine similar lines cut ...]
11  20.67    scanme.nmap.org (64.13.134.52)

OS and Service detection performed. Please report any
incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 147.462
seconds

Raw packets sent: 131128 (5.771MB) | Rcvd:
283637 (12.515MB)
```

Enabling Debugging Output

When even verbose mode doesn't provide sufficient data for you, debugging is available to flood you with much more! As with the verbosity option (-v), debugging is enabled with a command-line flag (-d) and the debug level can be increased by specifying it multiple times. Alternatively, you can set a debug level by giving an argument to -d. For example, -d9 sets level nine. That is the highest effective level and will produce thousands of lines unless you run a very simple scan with very few ports and targets.

Debugging output is useful when a bug is suspected in Nmap, or if you are simply confused as to what Nmap is doing and why. As this feature is mostly intended for developers, debug lines aren't always

self-explanatory. If you don't understand a line, your only recourses are to ignore it, look it up in the source code, or request help from the development list (*nmap-dev*). Some lines are self explanatory, but messages become more obscure as the debug level is increased. [Example 13.5](#) shows a few different debugging lines that resulted from a -d5 scan of Scanme.

Example 13.5. Some representative debugging lines

```
Timeout vals: srtt: 27495 rttvar: 27495 to: 137475 delta
-2753
          ==> srtt: 27150 rttvar: 21309 to: 112386
RCVD (15.3330s) TCP 64.13.134.52:25 > 132.239.1.115:50122
RA ttl=52
          id=0 iplen=40 seq=0 win=0 ack=4222318673
**TIMING STATS** (15.3350s): IP, probes
active/freshportsleft/retry_stack/
                                outstanding/retra
nwait/onbench,
                                cwnd/ccthresh/delay,
timeout/srtt/rttvar/
  Groupstats (1/1 incomplete): 83/*/*/*/*/* 82.80/75/*
100000/25254/4606
  64.13.134.52: 83/60836/0/777/316/4295 82.80/75/0
100000/26200/4223
Current sending rates: 711.88 packets / s, 31322.57 bytes
/ s.
Overall sending rates: 618.24 packets / s, 27202.62 bytes
/ s.
Discovered filtered port 10752/tcp on 64.13.134.52
Packet capture filter (device eth0): dst host
132.239.1.115 and
                                (icmp or ((tcp or
udp) and
                                (src host
64.13.134.52)))
SCRIPT ENGINE: TCP 132.239.1.115:59045 > 64.13.134.52:53
| CLOSE
```

No full example is given here because debug logs are so long. A scan against Scanme used 32 lines of text without verbosity ([Example 13.3, “Interactive output without verbosity enabled”](#)), and 61 with it ([Example 13.4, “Interactive output with verbosity](#)

enabled"). The same scan with -d instead of -v took 113 lines. With -d2 it ballooned to 65,731 lines, and -d5 output 396,879 lines! The debug option implicitly enables verbosity, so there is no need to specify them both.

Determining the best output level for a certain debug task is a matter of trial and error. I try a low level first to understand what is going on, then increase it as necessary. As I learn more, I may be able to better isolate the problem or question. I then try to simplify the command in order to offset some increased verbiage of the higher debug level.

Just as **grep** can be useful to identify the changes and levels associated with verbosity, it also helps with investigating debug output. I recommend running this command from the nmap-<VERSION> directory in the Nmap source tarball:

```
grep -A1 o.debugging *.cc
```

Handling Error and Warning Messages

Warnings and errors printed by Nmap usually go only to the screen (interactive output), leaving any normal-format output files (usually specified with -oN) uncluttered. When you do want to see those messages in the normal output file you specified, use the --log-errors option. It is useful when you aren't watching the interactive output or when you want to record errors while debugging a problem. The error and warning messages will still appear in interactive mode too. This won't work for most errors related to bad command-line arguments because Nmap may not have initialized its output files yet. In addition, some Nmap error and warning messages use a different system which does not yet support this option.

An alternative to --log-errors is redirecting interactive output (including the standard error stream) to a file. Most Unix shells make this approach easy. For example, tcsh uses the format **nmap <options> >& alloutput.nmap**. Bash uses a slightly different syntax: **nmap <options> &> alloutput.nmap**. The Windows cmd.exe syntax for doing this is so convoluted that --log-errors is recommended instead. For example, you can run **nmap --log-errors -oN alloutput.nmap <options>**.

Enabling Packet Tracing

The `--packet-trace` option causes Nmap to print a summary of every packet it sends and receives. This can be extremely useful for debugging or understanding Nmap's behavior, as examples throughout this book demonstrate. [Example 13.6](#) shows a simple ping scan of Scanme with packet tracing enabled.

Example 13.6. Using `--packet-trace` to detail a ping scan of Scanme

```
# nmap --packet-trace -n -sP scanme.nmap.org

Starting Nmap ( http://nmap.org )
SENT (0.0230s) ICMP 132.239.1.115 > 64.13.134.52 echo
request
                (type=8/code=0) ttl=38 id=5420 iplen=28
SENT (0.0230s) TCP 132.239.1.115:43743 > 64.13.134.52:80
A ttl=57
                id=29415 iplen=40  seq=2799605278 win=2048
ack=2120834905
RCVD (0.0380s) TCP 64.13.134.52:80 > 132.239.1.115:43743
R ttl=52
                id=0 iplen=40  seq=2120834905 win=0
Host 64.13.134.52 appears to be up.
Nmap done: 1 IP address (1 host up) scanned in 0.04
seconds
```

This Nmap execution shows three extra lines caused by packet tracing (each have been wrapped for readability). Each line contains several fields. The first is whether a packet is sent or received by Nmap, as abbreviated to SENT and RCVD. The next field is a time counter, providing the elapsed time since Nmap started. The time is in seconds, and in this case Nmap only required a tiny fraction of one. The next field is the protocol: TCP, UDP, or ICMP. Next comes the source and destination IP addresses, separated with a directional arrow. For TCP or UDP packets, each IP is followed by a colon and the source or destination port number.

The remainder of each line is protocol specific. As you can see, ICMP provides a human-readable type if available (echo request in this case) followed by the ICMP type and code values. The ICMP packet

logs end with the IP TTL, ID, and packet length field. TCP packets use a slightly different format after the destination IP and port number. First comes a list of characters representing the set TCP flags. The flag characters are SAFRPUEC, which stand for SYN, ACK, FIN, RST, PSH, URG, ECE, and CWR, respectively. The latter two flags are part of TCP explicit congestion notification, described in [RFC 3168](#).

Because packet tracing can lead to thousands of output lines, it helps to limit scan intensity to the minimum that still serves your purpose. A scan of a single port on a single machine won't bury you in data, while the output of a `--packet-trace` scan of a whole network can be overwhelming. Packet tracing is automatically enabled when the debug level (`-d`) is at least three.

Sometimes `--packet-trace` provides specialized data that Nmap never shows otherwise. For example, [Example 13.6, “Using `--packet-trace` to detail a ping scan of Scanme”](#) shows ICMP and TCP ping packets sent to the target host. The target responds to the ICMP echo request, which can be valuable information that Nmap doesn't otherwise show. It is possible that the target host replied to the TCP packet as well—Nmap stops listening once it receives one response to a ping scan since that is all it takes to determine that a host is online.

Resuming Aborted Scans

Some extensive Nmap runs take a very long time—on the order of days. Such scans don't always run to completion. Restrictions may prevent Nmap from being run during working hours, the network could go down, the machine Nmap is running on might suffer a planned or unplanned reboot, or Nmap itself could crash. The administrator running Nmap could cancel it for any other reason as well, by pressing **ctrl-C**. Restarting the whole scan from the beginning may be undesirable. Fortunately, if normal (`-oN`) or grepable (`-oG`) logs were kept, the user can ask Nmap to resume scanning with the target it was working on when execution ceased. Specify the `--resume` option and pass the normal/grepable output file as its argument. No other arguments are permitted, as Nmap parses the output file to use the same ones specified previously. Simply call Nmap as **nmap --resume <logfile>**. Nmap will append new results to the data files specified in the previous execution.

This feature does have some limitations. Resumption does not support the XML output format because combining the two runs into one valid XML file would be difficult. This feature only skips hosts for which all scanning was completed. If a scan was in progress against a certain target when Nmap was stopped, the `--resume` will restart scanning of that host from the beginning.

Interactive Output

Interactive output is what Nmap prints to the stdout stream, which usually appears on the terminal window you executed Nmap from. In other circumstances, you might have redirected stdout to a file or another application such as Nessus or an Nmap GUI may be reading the results. If a larger application is interpreting the results rather than printing Nmap output directly to the user, then using the XML output discussed in [the section called “XML Output \(-oX\)”](#) would be more appropriate.

This format has but one goal: to present results that will be valuable to a human reading over them. No effort is made to make these easily machine parsable or to maintain a stable format between Nmap versions. Better formats exist for these things. The toughest challenge is deciding which information is valuable enough to print. Omitting data that a user wants is a shame, though flooding the user with pages of mostly irrelevant output can be even worse. The verbosity, debugging, and packet tracing flags are available to shift this balance based on individual users' preferences.

This output format needs no extensive description here, as most Nmap examples in this book already show it. To understand Nmap's interactive output for a certain feature, see the section of this book dedicated to that feature. Typical examples of interactive output are given in [Example 13.3, “Interactive output without verbosity enabled”](#) and [Example 13.4, “Interactive output with verbosity enabled”](#).

Normal Output (-oN)

Normal output is printed to a file when the `-oN` option is specified with a filename argument. It is similar to interactive output, except that notes which lose relevance once a scan completes are removed. It is assumed that the file will be read after Nmap

completes, so estimated completion times and new open port alerts are redundant to the actual completion time and the ordered port table. Since output may be saved a long while and reviewed among many other logs, Nmap prints the execution time, command-line arguments, and Nmap version number on the first line. A similar line at the end of a scan divulges final timing and a host count. Those two lines begin with a pound character to identify them as comments. If your application must parse normal output rather than XML/grepable formats, ensure that it ignores comments that it doesn't recognize rather than treating them as an error and aborting. [Example 13.7](#) is a typical example of normal output. Note that `-oN` was used to prevent interactive output and send normal output straight to stdout.

Example 13.7. A typical example of normal output

```
# nmap -T4 -A -p- -oN - scanme.nmap.org

# Nmap 4.68 scan initiated Tue Jul 15 07:27:26 2008 as:
nmap -T4 -A -p- -oN - scanme.nmap.org
Interesting ports on scanme.nmap.org (64.13.134.52):
Not shown: 65529 filtered ports
PORT      STATE  SERVICE VERSION
22/tcp    open   ssh      OpenSSH 4.3 (protocol 2.0)
25/tcp    closed smtp
53/tcp    open   domain   ISC BIND 9.3.4
70/tcp    closed gopher
80/tcp    open   http      Apache httpd 2.2.2 ((Fedora))
|_ HTML title: Go ahead and ScanMe!
113/tcp   closed auth
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.17 - 2.6.21, Linux 2.6.23

TRACEROUTE (using port 22/tcp)
HOP RTT    ADDRESS
1    2.98    nodem-msfc-vl245-act-security-gw-1-113.ucsd.edu
(132.239.1.113)
[... nine similar lines cut ...]
11   13.34   scanme.nmap.org (64.13.134.52)

OS and Service detection performed. Please report any
incorrect results at http://nmap.org/submit/ .
```

```
# Nmap done at Tue Jul 15 07:29:45 2008 -- 1 IP address
(1 host up) scanned in 138.938 seconds
```

\$scripT klddI3 OuTPut (-os)

Script kiddie output is like interactive output, except that it is post-processed to better suit the 'l33t HaXXorZ! They previously looked down on Nmap due to its consistent capitalization and spelling. It is best understood by example, as given in [Example 13.8](#).

Example 13.8. A typical example of \$scripT KiDDi3 OutPut

```
# nmap -T4 -A -oS - scanme.nmap.org

StaRtIng NMap ( http://nmap.org )
Int3restIng p0rtz On $CAnme.nmap.org (64.13.134.52):
NOt ShOwn: 65529 FilterEd p0rt$
PORT      $TATE  $ERVIC3 V3R$IoN
22/tcP    0p3n    s$h      0pen$$H 4.3 (pr0tOcol 2.0)
25/TcP    closEd $mtp
53/tcp    op3n    dOma!n  I$c BIND 9.3.4
70/tcp    clo$ed G0ph3r
80/tcp    0p3n    htTP    4pach3 httpd 2.2.2 ((F3d0ra))
|_ HTML tITl3: gO aheAD And $canM3!
113/tcp   cl0$ed auTh
DeviCe type: g3NeraL purp0$3
RUnnIng: L1Nux 2.6.X
oS detAIlz: LinUx 2.6.17 - 2.6.21, L1nux 2.6.23
[Many lines cut for brevity]
NmAp doNe: 1 ip addre$z (1 H0$t up) $canned in 138.94
$ec0NdS
```

Some humor-impaired people take this option far too seriously, and scold me for catering to script kiddies. It is simply a joke *making fun* of the script kiddies—they don't actually use this mode (I hope).

XML Output (-ox)

XML, the *extensible markup language*, has its share of critics as well as plenty of zealous proponents. I was long in the former group, and only grudgingly incorporated XML into Nmap after volunteers

performed most of the work. Since then, I have learned to appreciate the power and flexibility that XML offers, and even wrote this book in the DocBook XML format. I strongly recommend that programmers interact with Nmap through the XML interface rather than trying to parse the normal, interactive, or grepable output. That format includes more information than the others and is extensible enough that new features can be added without breaking existing programs that use it. It can be parsed by standard XML parsers, which are available for all popular programming languages, usually for free. Editors, validators, transformation systems, and many other applications already know how to handle the format. Normal and interactive output, on the other hand, are custom to Nmap and subject to regular changes as I strive for a clearer presentation to end users. Grepable output is also Nmap-specific and tougher to extend than XML. It is considered deprecated, and many Nmap features such as MAC address detection are not presented in this output format.

An example of Nmap XML output is shown in [Example 13.9](#). Whitespace has been adjusted for readability. In this case, XML was sent to stdout thanks to the `-oX` construct. Some programs executing Nmap opt to read the output that way, while others specify that output be sent to a filename and then they read that file after Nmap completes.

Example 13.9. An example of Nmap XML output

```
# nmap -T4 -A -p- -oX - scanme.nmap.org
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="/usr/share/nmap/nmap.xsl"
type="text/xsl"?>
<!-- Nmap 4.68 scan initiated Tue Jul 15 07:27:26 2008
as:
      nmap -T4 -A -p- -oX - scanme.nmap.org -->
<nmaprun scanner="nmap" args="nmap -T4 -A -p- -oX -
scanme.nmap.org"
  start="1216106846" startstr="Tue Jul 15 07:27:26
2008"
    version="4.68" xmloutputversion="1.02">
  <scaninfo type="syn" protocol="tcp" numservices="65535"
services="1-65535" />
  <verbose level="0" /> <debugging level="0" />
    <host starttime="1216106846" endtime="1216106985">
      <status state="up" reason="reset" />
```

```

    <address addr="64.13.134.52" addrtype="ipv4" />
    <hostnames><hostname name="scanme.nmap.org"
type="PTR" /></hostnames>
    <ports><extraports state="filtered" count="65529">
        <extrareasons reason="no-responses" count="65529"
/></extraports>
        <port protocol="tcp" portid="22">
            <state state="open" reason="syn-ack"
reason_ttl="52" />
            <service name="ssh" product="OpenSSH"
version="4.3"
                extrainfo="protocol 2.0" method="probed"
conf="10" /> </port>
            <!-- Several port elements removed for brevity -->
            <port protocol="tcp" portid="80">
                <state state="open" reason="syn-ack"
reason_ttl="52" />
                <service name="http" product="Apache httpd"
version="2.2.2"
                    extrainfo="(Fedora)" method="probed"
conf="10" />
                <script id="HTML title" output="Go ahead and
ScanMe!" /> </port>
            <port protocol="tcp" portid="113">
                <state state="closed" reason="reset"
reason_ttl="52" />
                <service name="auth" method="table" conf="3" />
</port> </ports>
    <os>
        <portused state="open" proto="tcp" portid="22" />
        <portused state="closed" proto="tcp" portid="25" />
        <osclass type="general purpose" vendor="Linux"
osfamily="Linux"
            osgen="2.6.X" accuracy="100" />
        <osmatch name="Linux 2.6.17 - 2.6.21"
accuracy="100" line="11886" />
        <osmatch name="Linux 2.6.23" accuracy="100"
line="13895" /> </os>
    <uptime seconds="1104050" lastboot="Wed Jul  2
12:48:55 2008" />
    <tcpsequence index="203" difficulty="Good luck!"
        values="31F88BFB,327D2AA6,329B817C,329D4191,321A
15D3,32B3D917" />

```

```

    <ipidsequence class="All zeros"
values="0,0,0,0,0,0,0" />
    <tcptssequence class="1000HZ"
    values="41CE58DD,41CE5941,41CE59A5,41CE5A09,41CE
5A6D,41CE5AD5" />
    <trace port="22" proto="tcp">
    <hop ttl="1" rtt="2.98" ipaddr="132.239.1.113"
    host="nodem-msfc-vl245-act-security-gw-1-
113.ucsd.edu" />
    <!-- Several hop elements removed for brevity -->
    <hop ttl="11" rtt="13.34" ipaddr="64.13.134.52"
    host="scanme.nmap.org" /> </trace>
    <times srtt="14359" rttvar="1215" to="100000" />
</host>
    <runstats><finished time="1216106985" timestr="Tue Jul
15 07:29:45 2008" />
    <hosts up="1" down="0" total="1" />
    <!-- Nmap done at Tue Jul 15 07:29:45 2008;
    1 IP address (1 host up) scanned in 138.938
seconds -->
    </runstats>
</nmaprun>

```

Another advantage of XML is that its verbose nature makes it easier to read and understand than other formats. Readers familiar with Nmap in general can likely understand most of the XML output in [Example 13.9, “An example of Nmap XML output”](#) without further documentation. The grepable output format, on the other hand, is tough to decipher without its own reference guide.

There are a few aspects of the example XML output which may not be self-explanatory. For example, look at the two port elements in [Example 13.10](#)

Example 13.10. Nmap XML port elements

```

    <port protocol="tcp" portid="80">
    <state state="open" reason="syn-ack"
reason_ttl="52" />
    <service name="http" product="Apache httpd"
version="2.2.2"
    extrainfo="(Fedora)" method="probed"
conf="10" />

```

```
<script id="HTML title" output="Go ahead and
ScanMe!" />
</port>
<port protocol="tcp" portid="113">
  <state state="closed" reason="reset"
reason_ttl="52" />
  <service name="auth" method="table" conf="3" />
</port>
```

The port protocol, ID (port number), state, and service name are the same as would be shown in the interactive output port table. The service product, version, and extrainfo attributes come from version detection and are combined together into one field of the interactive output port table. The method and conf attributes aren't present in any other output types. The method can be table, meaning the service name was simply looked up in nmap-services based on the port number and protocol, or it can be probed, meaning that it was determined through the version detection system. The conf attribute measures the confidence Nmap has that the service name is correct. The values range from one (least confident) to ten. Nmap only has a confidence level of three for ports determined by table lookup, while it is highly confident (level 10) that port 80 of [Example 13.10, "Nmap XML port elements"](#) is Apache httpd, because Nmap connected to the port and found a server exhibiting the HTTP protocol with Apache banners.

One other aspect that some users find confusing is that the attributes /nmaprun/@start and /nmaprun/runstats/finished/@time hold timestamps given in Unix time, the number of seconds since January 1, 1970. This is often easier for programs to handle. For the convenience of human readers, versions 3.78 and newer include the equivalent calendar time written out in the attributes /nmaprun/@startstr and /nmaprun/runstats/finished/@endstr.

Nmap includes a document type definition (DTD) which allows XML parsers to validate Nmap XML output. While it is primarily intended for programmatic use, it can also help humans interpret Nmap XML output. The DTD defines the legal elements of the format, and often enumerates the attributes and values they can take on. It is reproduced in [Appendix A, Nmap XML Output DTD](#).

Using XML Output

The Nmap XML format can be used in many powerful ways, though few users actually take any advantage of it. I believe this is due to inexperience of many users with XML, combined with a lack of practical, solution-oriented documentation on using the Nmap XML format. This chapter provides several practical examples, including [the section called “Manipulating XML Output with Perl”](#), [the section called “Output to a Database”](#), and [the section called “Creating HTML Reports”](#).

A key advantage of XML is that you do not need to write your own parser as you do for specialized Nmap output types such as grepat and interactive output. Any general XML parser should do.

The XML parser that people are most familiar with is the one in your web browser. Both IE and Mozilla/Firefox include capable parsers that can be used to view Nmap XML data. Using them is as simple as typing the XML filename or URL into the address bar. [Figure 13.1](#) shows an example of XML output rendered by a web browser. How this automatic rendering works and how to save a permanent copy of an HTML report is covered in [the section called “Creating HTML Reports”](#).

Figure 13.1. XML output in a web browser

file:///home/fyodor/nmap/logs/xml-sample.xml

64.13.134.52 / scanme.nmap.org

ping results

- reset

address

- 64.13.134.52 (ipv4)

hostnames

- scanme.nmap.org (PTR)

ports

The 1709 ports scanned but not shown below are in state: **filtered**

- 1709 ports replied with: **no-responses**

Port	State	Service	Reason	Product	Version	Extra info	
22	tcp	open	ssh	syn-ack	OpenSSH	4.3	protocol 2.0
25	tcp	closed	smtp	reset			
53	tcp	open	domain	syn-ack	ISC BIND	9.3.4	
70	tcp	closed	gopher	reset			
80	tcp	open	http	syn-ack	Apache httpd	2.2.2	(Fedora)
113	tcp	closed	auth	reset			

Done

Nmap XML output can of course be viewed in any text editor or XML editor. Some spreadsheet programs, including Microsoft Excel, are able to import Nmap XML data directly for viewing. These general-purpose XML processors share the limitation that they treat Nmap XML generically, just like any other XML file. They don't understand the relative importance of elements, nor how to organize the data for a more useful presentation. The use of specialized XML processors that make sense of Nmap XML output is the subject of the following sections.

Manipulating XML Output with Perl

Generic XML parsers are available for all popular programming languages, often for free. Examples are the libxml C library and the Apache Xerces parser for Java and C++ (with Perl and COM bindings). While these parsers are sufficient for handling Nmap XML output, developers have created custom modules for several languages which can make the task of interoperating with Nmap XML even easier.

The language with the best custom Nmap XML support is Perl. Max Schubert (affectionately known as Perldork) has created a module named [Nmap::Scanner](#) while Anthony Persaud created [Nmap::Parser](#). These two modules have many similarities: they can execute Nmap themselves or read from an output file, are well documented, come with numerous example scripts, are part of the Comprehensive Perl Archive Network (CPAN), and are popular with users. They each offer both a callback based parser for interpreting data as Nmap runs as well as an all-at-once parser for obtaining a fully parsed document once Nmap finishes executing. Their APIs are a bit different—Nmap::Scanner relies on type-safe classes while Nmap::Parser relies on lighter-weight native Perl arrays. I recommend looking at each to decide which best meets your needs and preferences.

[Example 13.11](#) is a simple demonstration of Nmap::Parser. It comes from the module's documentation (which contains many other examples as well). It performs a quick scan, then prints overall scan statistics as well as information on each available target host. Notice how readable it is compared to scripts using other Nmap output formats that are dominated by parsing logic and regular expressions. Even people with poor Perl skills could use this as a starting point to create simple programs to automate their Nmap scanning needs.

Example 13.11. Nmap::Parser sample code

```
use Nmap::Parser;

    #PARSING
my $np = new Nmap::Parser;

$nmmap_exe = '/usr/bin/nmap';
$np->parsescan($nmmap_exe, '-sT -p1-1023', @ips);

#or
```

```

$np->parsefile('nmap_output.xml'); #using filenames

    #GETTING SCAN INFORMATION

print "Scan Information:\n";
$si = $np->get_scaninfo();
#get scan information by calling methods
print
'Number of services scanned: ' . $si->
num_of_services() . "\n",
'Start Time: ' . $si->start_time() . "\n",
'Scan Types: ', (join ' ', $si->scan_types()) . "\n";

    #GETTING HOST INFORMATION

print "Hosts scanned:\n";
for my $host_obj ($np->get_host_objects()){
    print
    'Hostname   : ' . $host_obj->hostname() . "\n",
    'Address    : ' . $host_obj->ipv4_addr() . "\n",
    'OS match   : ' . $host_obj->os_match() . "\n",
    'Open Ports: ' . (join ' ', $host_obj->
tcp_ports('open')) . "\n";
        #... you get the idea...
}

#frees memory--helpful when dealing with memory intensive
scripts
$np->clean();

```

For comparison, [Example 13.12](#) is a sample Perl script using Nmap::Scanner, copied from its documentation. This one uses an event-driven callback approach, registering the functions scan_started and port_found to print real-time alerts when a host is found up and when each open port is discovered on the host.

Example 13.12. Nmap::Scanner sample code

```

my $scanner = new Nmap::Scanner;
$scanner->register_scan_started_event(\&scan_started);
$scanner->register_port_found_event(\&port_found);

```

```

$scanner->scan('-sS -p 1-1024 -O --max-rtt-timeout 200
somehost.org.net.it');

sub scan_started {
    my $self      = shift;
    my $host      = shift;

    my $hostname = $host->name();
    my $addresses = join(', ', map {$_->address()} $host-
>addresses());
    my $status = $host->status();

    print "$hostname ($addresses) is $status\n";
}

sub port_found {
    my $self      = shift;
    my $host      = shift;
    my $port      = shift;

    my $name = $host->name();
    my $addresses = join(', ', map {$_->addr()} $host-
>addresses());

    print "On host $name ($addresses), found ",
        $port->state(), " port ",
        join('/', $port->protocol(), $port-
>portid()), "\n";
}

```

Output to a Database

A common desire is to output Nmap results to a database for easier queries and tracking. This allows users from an individual penetration tester to an international enterprise to store all of their scan results and easily compare them. The enterprise might run large scans daily and schedule queries to mail administrators of newly open ports or available machines. The penetration tester might learn of a new vulnerability and search all of his old scan results for the affected application so that he can warn the relevant clients. Researchers may scan millions of IP addresses and keep the results in a database for easy real-time queries.

While these goals are laudable, Nmap offers no direct database output functionality. Not only are there too many different database types for me to support them all, but users' needs vary so dramatically that no single database schema is suitable. The needs of the enterprise, pen-tester, and researcher all call for different table structures.

For projects large enough to require a database, I recommend deciding on an optimal DB schema first, then writing a simple program or script to import Nmap XML data appropriately. Such scripts often take only minutes, thanks to the wide availability of XML parsers and database access modules. Perl often makes a good choice, as it offers a powerful database abstraction layer and also custom Nmap XML support. [the section called “Manipulating XML Output with Perl”](#) shows how easily Perl scripts can make use of Nmap XML data.

Another option is to use a custom Nmap database support patch. One example is [nmap-sql](#), which adds MySQL logging functionality into Nmap itself. The downsides are that it currently only supports the MySQL database and it must be frequently ported to new Nmap versions. An XML-based approach, on the other hand, is less likely to break when new Nmap versions are released.

Another option is [PBNJ](#), a suite of tools for monitoring changes to a network over time. It stores scan data such as online hosts and open ports to a database (SQLite, MySQL or Postgres). It offers a flexible querying and alerting system for accessing that data or displaying changes.

Creating HTML Reports

Nmap does not have an option for saving scan results in HTML, however it is easy to get an HTML view of Nmap XML output just by opening the XML file in a web browser. An example is shown in [Figure 13.1, “XML output in a web browser”](#).

How does the web browser know how to convert XML to HTML? An Nmap XML output file usually contains a reference to an [XSL](#) stylesheet called `nmap.xsl` that describes how the transformation takes place.

The XML processing instruction that says where the stylesheet can be found will look something like

```
<?xml-stylesheet href="/usr/share/nmap/nmap.xsl"
type="text/xsl"?>
```

The exact location may be different depending on the platform and how Nmap was configured.

Such a stylesheet reference will work fine when viewing scan results on the same machine that initiated the scan, but it will not work if the XML file is transferred to another machine where the nmap.xsl file is in a different place or absent entirely. To make the XML styling portable, give the --webxml option to Nmap. This will change the processing instruction to read

```
<?xml-stylesheet href="http://nmap.org/data/nmap.xsl"
type="text/xsl"?>
```

The resultant XML output file will render as HTML on any web-connected machine. Using the network location in this fashion is often more useful, but the local copy of nmap.xsl is used by default for privacy reasons.

To use a different stylesheet, use the --stylesheet *<file>* option. Note that --webxml is an alias for --stylesheet <http://nmap.org/data/nmap.xsl>.

To omit the stylesheet entirely, use the option --no-stylesheet. This will cause web browsers to show the output as a plain, uninterpreted XML tree.

Saving a Permanent HTML Report

While web browsers can display an HTML view of Nmap XML, they don't usually make it easy to save the generated HTML to a file. For that a standalone XSLT processor is required. Here are commands that turn an Nmap XML output file into an HTML file using common XSLT processors.

[xsltproc](#)

```
xsltproc <nmap-output.xml> -o <nmap-output.html>
```

Saxon

Saxon 9: **java -jar saxon9.jar -s:<nmap-output.xml> -o:<nmap-output.html>**

Previous Saxon releases: **java -jar saxon.jar -a <nmap-output.xml> -o <nmap-output.html>**

Xalan

Using Xalan C++: **Xalan -a <nmap-output.xml> -o <nmap-output.html>**

Using Xalan Java: **java -jar xalan.jar -IN <nmap-output.xml> -OUT <nmap-output.html>**

Grepable Output (-oG)

This output format is covered last because it is deprecated. The XML output format is far more powerful, and is nearly as convenient for experienced users. XML is a standard for which dozens of excellent parsers are available, while grepable output is my own simple hack. XML is extensible to support new Nmap features as they are released, while I often must omit those features from grepable output for lack of a place to put them.

Nevertheless, grepable output is still quite popular. It is a simple format that lists each host on one line and can be trivially searched and parsed with standard Unix tools such as `grep`, `awk`, `cut`, `sed`, `diff`, and `Perl`. Even I usually use it for one-off tests done at the command line. Finding all the hosts with the SSH port open or that are running Solaris takes only a simple `grep` to identify the hosts, piped to an `awk` or `cut` command to print the desired fields. One grepable output aficionado is Lee “MadHat” Heath, who contributed to this section.

[Example 13.13](#) shows a typical example of grepable output. Normally each host takes only one line, but I split this entry into seven lines to fit on the page. There are also three lines starting with a hash prompt (not counting the Nmap command line). Those are comments describing when Nmap started, the command line options used, and completion time and statistics. One of the comment lines enumerates the port numbers that were scanned. I

shortened it to avoid wasting dozens of lines. That particular comment is only printed in verbose (-v) mode. Increasing the verbosity level beyond one -v will not further change the grepable output. The times and dates have been replaced with [time] to reduce line length.

Example 13.13. A typical example of grepable output

```
# nmap -oG - -T4 -A -v scanme.nmap.org
# Nmap 4.68 scan initiated [time] as: nmap -oG - -T4 -A
-v scanme.nmap.org
# Ports scanned: TCP(1715;1-1027,1029-1033,...,65301)
UDP(0;) PROTOCOLS(0;)
Host: 64.13.134.52 (scanme.nmap.org)    Ports:
22/open/tcp//ssh//OpenSSH 4.3(protocol 2.0)/,
25/closed/tcp//smtp///, 53/open/tcp//domain//ISC BIND
9.3.4/,70/closed/tcp//gopher///,
80/open/tcp//http//Apache httpd 2.2.2
((Fedora))/,113/closed/tcp//auth/// Ignored State:
filtered (1709)  OS: Linux 2.6.20-1 (Fedora Core 5) Seq
Index: 203  IP ID Seq: All zeros
# Nmap done at [time] -- 1 IP address (1 host up) scanned
in 34.96 seconds
```

The command-line here requested that grepable output be sent to standard output with the - argument to -oG. Aggressive timing (-T4) as well as OS and version detection (-A) were requested. The comment lines are self-explanatory, leaving the meat of grepable output in the Host line. Had I scanned more hosts, each of the available ones would have its own Host line.

Grepable Output Fields

The host line is split into fields, each of which consist of a field name followed by a colon and space, then the field content. The fields are separated by tab characters (ASCII number nine, '\t').

[Example 13.13, “A typical example of grepable output”](#) shows six fields: Host, Ports, Ignored State, OS, Seq Index, and IP ID. A Status section is included in list (-sL) and ping (-sP) scans, and a Protocols section is included in IP protocol (-sO) scans. The exact fields given depend on Nmap options used. For example, OS detection triggers

the OS, Seq Index, and IP ID fields. Because they are tab delimited, you might split up the fields with a Perl line such as:

```
@fields = split("\t", $host_line);
```

In the case of [Example 13.13, “A typical example of grepable output”](#), the array @fields would contain six members. \$fields[0] would contain “Host: 64.13.134.52 (scanme.nmap.org)”, and \$fields[1] would contain the long Ports field. Scripts that parse grepable output should ignore fields they don't recognize, as new fields may be added to support Nmap enhancements.

The eight possible fields are described in the following sections.

Host field

Example: Host: 64.13.134.52 (scanme.nmap.org)

The Host field always comes first and is included no matter what Nmap options are chosen. The contents are the IP address (an IPv6 address if -6 was specified), a space, and then the reverse DNS name in parentheses. If no reverse name is available, the parentheses will be empty.

Ports field

Example: Ports: 111/open/tcp//rpcbind (rpcbind V2)/
(rpcbind:100000*2-2)/2 (rpc #100000)/, 113/closed/tcp//auth///

The Ports field is by far the most complex, as can be seen in [Example 13.13, “A typical example of grepable output”](#). It includes entries for every interesting port (the ones which would be included in the port table in normal Nmap output). The port entries are separated with a comma and a space character. Each port entry consists of seven subfields, separated by a forward slash (/). The subfields are: port number, state, protocol, owner, service, SunRPC info, and version info. Some subfields may be empty, particularly for basic port scans without OS or version detection. The consecutive slashes in [Example 13.13, “A typical example of grepable output”](#) reveal empty subfields. In Perl, you might split them up as so:

```
($port, $state, $protocol, $owner, $service, $rpc_info,  
$version) =  
    split('/', $ports);
```


Alternatively, you could grab the information from the command line using commands such as these:

```
cut -d/ -f<fieldnumbers>  
awk -F/ '{print $<fieldnumber>}'
```

Certain subfields can contain a slash in other output modes. For example, an SSL-enabled web server would show up as ssl/http and the version info might contain strings such as mod_ssl/2.8.12. Since a slash is the subfield delimiter, this would screw up parsing. To avoid this problem, slashes are changed into the pipe character (|) when they would appear anywhere in the Port field.

Parsers should be written to allow more than seven slash-delimited subfields and to simply ignore the extras because future Nmap enhancements may call for new ones. The following list describes each of the seven currently defined Port subfields.

Port number

This is simply the numeric TCP or UDP port number.

State

The same port state which would appear in the normal output port table is shown here.

Protocol

This is tcp or udp.

Owner

This used to specify the username that the remote server is running under based on results from querying an identd (auth) server of the target host. The ident scan (-I) is no longer available with Nmap, so this field is always empty. Ident data can still be obtained using the identd-owners.nse NSE script, though results are not placed in this field.

Service

The service name, as obtained from an nmap-services lookup, or (more reliably) through version detection (-sV) if it was requested and succeeded. With version detection enabled,

compound entries such as `ssl|http` and entries with a trailing question mark may be seen. The meaning is the same as for normal output, as discussed in [Chapter 7, Service and Application Version Detection](#).

SunRPC info

If version detection (`-sV`) or RPC scan (`-sR`) were requested and the port was found to use the SunRPC protocol, the RPC program number and accepted version numbers are included here. A typical example is `"(rpcbind:100000*2-2)"`. The data is always returned inside parentheses. It starts with the program name, then a colon and the program number, then an asterisk followed by the low and high supported version numbers separated by a hyphen. So in this example, `rpcbind` (program number 100,000) is listening on the port for `rpcbind` version 2 requests.

Version info

If version detection is requested and succeeds, the results are provided here in the same format used in interactive output. For SunRPC ports, the RPC data is printed here too. The format for RPC results in this column is `<low version number>-<high version number> (rpc #<rpc program number>)`. When only one version number is supported, it is printed by itself rather than as a range. A port which shows `(rpcbind:100000*2-2)` in the SunRPC info subfield would show `2 (rpc #100000)` in the version info subfield.

Protocols field

Example: `Protocols: 1/open/icmp/, 2/open|filtered/igmp/`

The IP protocol scan (`-sO`) has a Protocols field rather than Ports. Its contents are quite similar to the Ports field, but it has only three subfields rather than seven. They are delimited with slashes, just as with the Ports field. Any slashes that would appear in a subfield are changed into pipes (`|`), also as done in the Ports field. The subfields are protocol number, state, and protocol name. These correspond to the three fields shown in interactive output for a protocol scan. An example of IP protocol scan greppable output is shown in [Example 13.14](#). The Host line, which would normally be all one line, is here wrapped for readability.

Example 13.14. Greppable output for IP protocol scan

```
# nmap -v -oG - -s0 localhost
# Nmap 4.68 scan initiated [time] as: nmap -v -oG - -s0
localhost
# Ports scanned: TCP(0;) UDP(0;) PROTOCOLS(256;0-255)
Host: 127.0.0.1 (localhost)
      Protocols: 1/open/icmp/, 2/open|filtered/igmp/,
6/open/tcp/,
                  17/open/udp/, 136/open|
filtered/udplite/, 255/open|filtered//
      Ignored State: closed (250)
# Nmap done at [time] -- 1 IP address (1 host up) scanned
in 2.345 seconds
```

Ignored State field

Example: Ignored State: filtered (1658)

To save space, Nmap may omit ports in one non-open state from the list in the Ports field. Nmap does this in interactive output too. Regular Nmap users are familiar with the lines such as “The 1658 ports scanned but not shown below are in state: filtered”. For greppable mode, that state is given in the Ignored State field. Following the state name is a space, then in parentheses is the number of ports found in that state.

OS field

Example: OS: Linux 2.4.0 - 2.5.20

Any perfect OS matches are listed here. If there are multiple matches, they are separated by a pipe character as shown in [Example 13.13, “A typical example of greppable output”](#). Only the free-text descriptions are provided. Greppable mode does not provide the vendor, OS family, and device type classification shown in other output modes.

Seq Index field

Example: Seq Index: 3004446

This number is an estimate of the difficulty of performing TCP initial sequence number prediction attacks against the remote host. These

are also known as blind spoofing attacks, and they allow an attacker to forge a full TCP connection to a remote host as if it was coming from some other IP address. This can always help an attacker hide his or her tracks, and it can lead to privilege escalation against services such as rlogin that commonly grant extra privileges to trusted IP addresses. The Seq Index value is only available when OS detection (-O) is requested and succeeds in probing for this. It is reported in interactive output when verbosity (-v) is requested. More details on the computation and meaning of this value are provided in [Chapter 8, Remote OS Detection](#).

IP ID Seq field

Example: IP ID Seq: All zeros

This simply describes the remote host's IP ID generation algorithm. It is only available when OS detection (-O) is requested and succeeds in probing for it. Interactive mode reports this as well, and it is discussed in [Chapter 8, Remote OS Detection](#).

Status field

Example: Status: Up

Ping and list scans contain only two fields in grepable mode: Host and Status. Status describes the target host as either Up, Down, or Unknown. List scan always categorizes targets as Unknown because it does not perform any tests. Ping scan lists a host as up if it responds to at least one ping probe, and down if no responses are received. It used to also report Smurf if ping probes sent to the target resulted in one or more responses from other hosts, but that is no longer done. Down hosts are only shown when verbosity is enabled with -v. [Example 13.15](#) demonstrates a ping scan of 100 random hosts, while [Example 13.16](#) demonstrates a list scan of five hosts.

Example 13.15. Ping scan grepable output

```
# nmap -sP -oG - -iR 100
# nmap [version] scan initiated [time] as: nmap -sP -oG -
-iR 100
Host: 67.101.77.102 (h-67-101-77-102.nycmny83.covad.net)
Status: Up
Host: 219.93.164.197 () Status: Up
```

```
Host: 222.113.158.200 () Status: Up
Host: 66.130.155.190 (modemcable190.155-130-
66.mc.videotron.ca) Status: Up
# Nmap done at [time] -- 100 IP addresses (4 hosts up)
scanned in 13.22 seconds
```

Example 13.16. List scan grepable output

```
# nmap -sL -oG - -iR 5
# nmap [version] scan initiated [time] as: nmap -sL -oG -
-iR 5
Host: 199.223.2.1 () Status: Unknown
Host: 191.222.112.87 () Status: Unknown
Host: 62.23.21.157 (host.157.21.23.62.rev.coltfrance.com)
Status: Unknown
Host: 138.217.47.127 (CPE-138-217-47-
127.vic.bigpond.net.au) Status: Unknown
Host: 8.118.0.91 () Status: Unknown
# Nmap done at [time] -- 5 IP addresses (0 hosts up)
scanned in 1.797 seconds
```

Parsing Grepable Output on the Command Line

Grepable output really shines when you want to gather information quickly without the overhead of writing a script to parse XML output. [Example 13.17](#) shows a typical example of this. The goal is to find all hosts on a class C sized network with port 80 open. Nmap is told to scan just that port of each host (skipping the ping stage) and to output a grepable report to stdout. The results are piped to a trivial awk command which finds lines containing /open/ and outputs fields two and three for each matching line. Those fields are the IP address and hostname (or empty parentheses if the hostname is unavailable).

Example 13.17. Parsing grepable output on the command line

```
> nmap -p80 -PN -oG - 10.1.1.0/24 | awk '/open/{print $2
" " $3}'
10.1.1.72 (userA.corp.foocompany.biz)
10.1.1.73 (userB.corp.foocompany.biz)
```

```
10.1.1.75 (userC.corp.foocompany.biz)
10.1.1.149 (admin.corp.foocompany.biz)
10.1.1.152 (printer.corp.foocompany.biz)
10.1.1.160 (10-1-1-160.foocompany.biz)
10.1.1.161 (10-1-1-161.foocompany.biz)
10.1.1.201 (10-1-1-201.foocompany.biz)
10.1.1.254 (10-1-1-254.foocompany.biz)
```