The Art of Port Scanning

by Fyodor <fyodor@insecure.org>

(Last significant update: Sat Sep 6 03:24:53 GMT 1997)

Warning, the interface to nmap has changed a bit and so not all the flags and options mentioned in this paper are still accurate. The authoritative documentation is now the man page (html version). This article still contains a lot of information on port scanning though and so I recommend that nmap users read it.

Abstract

This paper details many of the techniques used to determine what ports (or similar protocol abstraction) of a host are listening for connections. These ports represent potential communication channels. Mapping their existence facilitates the exchange of information with the host, and thus it is quite useful for anyone wishing to explore their networked environment, including hackers. Despite what you have heard from the media, the Internet is NOT all about TCP port 80. Anyone who relies exclusively on the WWW for information gathering is likely to gain the same level of proficiency as your average AOLer, who does the same. This paper is also meant to serve as an introduction to and ancillary documentation for a coding project I have been working on. It is a full featured, robust port scanner which (I hope) solves some of the problems I have encountered when dealing with other scanners and when working to scan massive networks. The tool, nmap, supports the following:

- Vanilla TCP connect() scanning,
- TCP SYN (half open) scanning,
- TCP FIN (stealth) scanning,
- TCP ftp proxy (bounce attack) scanning,
- SYN/FIN scanning using IP fragments (bypasses packet filters),
- UDP recvfrom() scanning,
- UDP raw ICMP port unreachable scanning,
- ICMP scanning (ping-sweep), and
- Reverse-ident scanning.

The freely distributable source code is available at http://www.insecure.org/nmap/

Introduction

Scanning, as a method for discovering exploitable communication channels, has been around for ages. The idea is to probe as many listeners as possible, and keep track of the ones that are receptive or useful to your particular need. Much of the field of advertising is based on this paradigm, and the "to current resident" brute force style of bulk mail is an almost perfect parallel to what we will discuss. Just stick a message in every mailbox and wait for the responses to trickle back.

Scanning entered the h/p world along with the phone systems. Here we have this tremendous global telecommunications network, all reachable through codes on our telephone. Millions of numbers are reachable locally, yet we may only be interested in 0.5% of these numbers, perhaps those that answer with a carrier.

The logical solution to finding those numbers that interest us is to try them all. Thus the field of "wardialing" arose. Excellent programs like Toneloc were developed to facilitate the probing of entire exchanges and more. The basic idea is simple. If you dial a number and your modem gives you a CONNECT, you record it. Otherwise the computer hangs up and tirelessly dials the next one.

While wardialing is still useful, we are now finding that many of the computers we wish to communicate with are connected through networks such as the Internet rather than analog phone dialups. Scanning these machines involves the same brute force technique. We send a blizzard of packets for various protocols, and we deduce which services are listening from the responses we receive (or don't receive).

Techniques

Over time, a number of techniques have been developed for surveying the protocols and ports on which a target machine is listening. They all offer different benefits and problems. Here is a line up of the most common:

- TCP connect() scanning : This is the most basic form of TCP scanning. The connect() system call provided by your operating system is used to open a connection to every interesting port on the machine. If the port is listening, connect() will succeed, otherwise the port isn't reachable. One strong advantage to this technique is that you don't need any special privileges. Any user on most UNIX boxes is free to use this call. Another advantage is speed. While making a separate connect() call for every targeted port in a linear fashion would take ages over a slow connection, you can hasten the scan by using many sockets in parallel. Using non-blocking I/O allows you to set a low time-out period and watch all the sockets at once. This is the fastest scanning method supported by nmap, and is available with the -t (TCP) option. The big downside is that this sort of scan is easily detectable and filterable. The target hosts logs will show a bunch of connection and error messages for the services which take the connection and then have it immediately shutdown.

- TCP SYN scanning : This technique is often referred to as "half-open" scanning, because you don't open a full TCP connection. You send a SYN packet, as if you are going to open a real connection and wait for a response. A SYN|ACK indicates the port is listening. A RST is indicative of a non- listener. If a SYN|ACK is received, you immediately send a RST to tear down the connection (actually the kernel does this for us). The primary advantage to this scanning technique is that fewer sites will log it. Unfortunately you need root privileges to build these custom SYN packets. SYN scanning is the -s option of nmap.

- TCP FIN scanning : There are times when even SYN scanning isn't clandestine enough. Some firewalls and packet filters watch for SYNs to restricted ports, and programs like synlogger and Courtney are available to detect these scans. FIN packets, on the other hand, may be able

to pass through unmolested. This scanning technique was featured in detail by Uriel Maimon in Phrack 49, article 15. The idea is that closed ports tend to reply to your FIN packet with the proper RST. Open ports, on the other hand, tend to ignore the packet in question. As Alan Cox has pointed out, this is required TCP behavior. However, some systems (notably Micro$oft boxes), are broken in this regard. They send RST's regardless of the port state, and thus they aren't vulnerable to this type of scan. It works well on most other systems I've tried. Actually, it is often useful to discriminate between a *NIX and NT box, and this can be used to do that. FIN scanning is the -U (Uriel) option of nmap.

- Fragmentation scanning : This is not a new scanning method in and of itself, but a modification of other techniques. Instead of just sending the probe packet, you break it into a couple of small IP fragments. You are splitting up the TCP header over several packets to make it harder for packet filters and so forth to detect what you are doing. Be careful with this! Some programs have trouble handling these tiny packets. My favorite sniffer segmentation faulted immediately upon receiving the first 36-byte fragment. After that comes a 24 byte one! While this method won't get by packet filters and firewalls that queue all IP fragments (like the CONFIG_IP_ALWAYS_DEFRAG option in Linux), a lot of networks can't afford the performance hit this causes. This feature is rather unique to scanners (at least I haven't seen any others that do this). Thanks to daemon9 for suggesting it. The -f instructs the specified SYN or FIN scan to use tiny fragmented packets.

- TCP reverse ident scanning : As noted by Dave Goldsmith in a 1996 Bugtraq post, the ident protocol (rfc1413) allows for the disclosure of the username of the owner of any process connected via TCP, even if that process didn't initiate the connection. So you can, for example, connect to the http port and then use identd to find out whether the server is running as root. This can only be done with a full TCP connection to the target port (i.e. the -t option). nmap's -i option queries identd for the owner of all listen()ing ports.

- FTP bounce attack : An interesting "feature" of the ftp protocol (RFC 959) is support for "proxy" ftp connections. In other words, I should be able to connect from evil.com to the FTP server-PI (protocol interpreter) of target.com to establish the control communication connection. Then I should be able to request that the server-PI initiate an active server-DTP (data transfer process) to send a file ANYWHERE on the internet! Presumably to a User-DTP, although the RFC specifically states that asking one server to send a file to another is OK. Now this may have worked well in 1985 when the RFC was just written. But nowadays, we can't have people hijacking ftp servers and requesting that data be spit out to arbitrary points on the internet. As *Hobbit* wrote back in 1995, this protocol flaw "can be used to post virtually untraceable mail and news, hammer on servers at various sites, fill up disks, try to hop firewalls, and generally be annoying and hard to track down at the same time." What we will exploit this for is to (surprise, surprise) scan TCP ports from a "proxy" ftp server. Thus you could connect to an ftp server behind a firewall, and then scan ports that are more likely to be blocked (139 is a good one). If the ftp server allows reading from and writing to a directory (such as /incoming), you can send arbitrary data to ports that you do find open.

  For port scanning, our technique is to use the PORT command to declare that our passive "User-DTP" is listening on the target box at a certain port number. Then we try to LIST the current directory, and the result is sent over the Server-DTP channel. If our target host is

listening on the specified port, the transfer will be successful (generating a 150 and a 226 response). Otherwise we will get "425 Can't build data connection: Connection refused." Then we issue another PORT command to try the next port on the target host. The advantages to this approach are obvious (harder to trace, potential to bypass firewalls). The main disadvantages are that it is slow, and that some FTP servers have finally got a clue and disabled the proxy "feature". For what it is worth, here is a list of banners from sites where it does/doesn't work:

Bounce attacks worked:

```
220 xxxxxxx.com FTP server (Version wu-2.4(3) Wed Dec 14 ...) ready.
220 xxx.xxx.xxx.edu FTP server ready.
220 xx.Telcom.xxxx.EDU FTP server (Version wu-2.4(3) Tue Jun 11 ...) ready.
220 lem FTP server (SunOS 4.1) ready.
220 xxx.xxx.es FTP server (Version wu-2.4(11) Sat Apr 27 ...) ready.
220 elios FTP server (SunOS 4.1) ready
```

Bounce attack failed:

```
220 wcarchive.cdrom.com FTP server (Version DG-2.0.39 Sun May 4 ...) ready.
220 xxx.xx.xxxxx.EDU Version wu-2.4.2-academ[BETA-12](1) Fri Feb 7
220 ftp Microsoft FTP Service (Version 3.0).
220 xxx FTP server (Version wu-2.4.2-academ[BETA-11](1) Tue Sep 3 ...)
ready.
220 xxx.unc.edu FTP server (Version wu-2.4.2-academ[BETA-13](6) ...) ready.
```

The 'x's are partly there to protect those guilty of running a flawed server, but mostly just to make the lines fit in 80 columns. Same thing with the ellipse points. The bounce attack is available with the -b option of nmap. proxy_server can be specified in standard URL format, username:password@server:port , with everything but server being optional.

- UDP ICMP port unreachable scanning : This scanning method varies from the above in that we are using the UDP protocol instead of TCP. While this protocol is simpler, scanning it is actually significantly more difficult. This is because open ports don't have to send an acknowledgement in response to our probe, and closed ports aren't even required to send an error packet. Fortunately, most hosts do send an ICMP_PORT_UNREACH error when you send a packet to a closed UDP port. Thus you can find out if a port is NOT open, and by exclusion determine which ports which are. Neither UDP packets, nor the ICMP errors are guaranteed to arrive, so UDP scanners of this sort must also implement retransmission of packets that appear to be lost (or you will get a bunch of false positives). Also, this scanning technique is slow because of compensation for machines that took RFC 1812 section 4.3.2.8 to heart and limit ICMP error message rate. For example, the Linux kernel (in net/ipv4/icmp.h) limits destination unreachable message generation to 80 per 4 seconds, with a 1/4 second penalty if that is exceeded. At some point I will add a better algorithm to nmap for detecting this. Also, you will need to be root for access to the raw ICMP socket necessary

for reading the port unreachable. The -u (UDP) option of nmap implements this scanning method for root users.

Some people think UDP scanning is lame and pointless. I usually remind them of the recent Solaris rcpbind hole. Rpcbind can be found hiding on an undocumented UDP port somewhere above 32770. So it doesn't matter that 111 is blocked by the firewall. But can you find which of the more than 30,000 high ports it is listening on? With a UDP scanner you can!

- UDP recvfrom() and write() scanning : While non-root users can't read port unreachable errors directly, Linux is cool enough to inform the user indirectly when they have been received. For example a second write() call to a closed port will usually fail. A lot of scanners such as netcat and Pluvius' pscan.c does this. I have also noticed that recvfrom() on non-blocking UDP sockets usually return EAGAIN ("Try Again", errno 13) if the ICMP error hasn't been received, and ECONNREFUSED ("Connection refused", errno 111) if it has. This is the technique used for determining open ports when non-root users use -u (UDP). Root users can also use the -l (lamer UDP scan) options to force this, but it is a really dumb idea.

- ICMP echo scanning : This isn't really port scanning, since ICMP doesn't have a port abstraction. But it is sometimes useful to determine what hosts in a network are up by pinging them all. the -P option does this. ICMP scanning is now in parallel, so it can be quite fast. To speed things up even more, you can increase the number of pings in parallel with the '-L ' option. It can also be helpful to tweek the ping timeout value with '-T '. nmap supports a host/bitmask notation to make this sort of thing easier. For example 'nmap -P cert.org/24 152.148.0.0/16' would scan CERT's class C network and whatever class B entity 152.148.* represents. Host/26 is useful for 6-bit subnets within an organization. Nmap now also offers a more powerful form. You can now do things like '150.12,17,71-79.7.*' and it will do what you expect. For each of the four values, you can either put a single number, a range (with '-'), a comma-separated list of numbers and ranges, or a '*' which is just a short cut for 0-255. By default, likely network/broadcast addresses like .0 and .255 are not scanned, but the '-A' option allows you to do this if you wish.

Features

Prior to writing nmap, I spent a lot of time with other scanners exploring the Internet and various private networks (note the avoidance of the "intranet" buzzword). I have used many of the top scanners available today, including strobe by Julian Assange, netcat by *Hobbit*, stcp by Uriel Maimon, pscan by Pluvius, ident-scan by Dave Goldsmith, and the SATAN tcp/udp scanners by Wietse Venema. These are all excellent scanners! In fact, I ended up hacking most of them to support the best features of the others. Finally I decided to write a whole new scanner, rather than rely on hacked versions of a dozen different scanners in my /usr/local/sbin. While I wrote all the code, nmap uses a lot of good ideas from its predecessors. I also incorporated some new stuff like fragmentation scanning and options that were on my "wish list" for other scanners. Here are some of the (IMHO) useful features of nmap:

- dynamic delay time calculations: Some scanners require that you supply a delay time between sending packets. Well how should I know what to use? Sure, I can ping them, but that is a pain, and plus the response time of many hosts changes dramatically when they are being flooded with requests. nmap tries to determine the best delay time for you. It also tries to keep track of packet retransmissions, etc. so that it can modify this delay time during the course of the scan. For root users, the primary technique for finding an initial delay is to time the internal "ping" function. For non-root users, it times an attempted connect() to a closed port on the target. It can also pick a reasonable default value. Again, people who want to specify a delay themselves can do so with -w (wait), but you shouldn't have to.

- retransmission: Some scanners just send out all the query packets, and collect the responses. But this can lead to false positives or negatives in the case where packets are dropped. This is especially important for "negative" style scans like UDP and FIN, where what you are looking for is a port that does NOT respond. In most cases, nmap implements a configurable number of retransmissions for ports that don't respond.

- parallel port scanning: Some scanners simply scan ports linearly, one at a time, until they do all 65535. This actually works for TCP on a very fast local network, but the speed of this is not at all acceptable on a wide area network like the Internet. nmap uses non-blocking i/o and parallel scanning in all TCP and UDP modes. The number of scans in parallel is configurable with the -M (Max sockets) option. On a very fast network you will actually decrease performance if you do more than 18 or so. On slow networks, high values increase performance dramatically.

- Flexible port specification: I don't always want to just scan all 65535 ports. Also, the scanners which only allow you to scan ports 1 - N sometimes fall short of my need. The -p option allows you to specify an arbitrary number of ports and ranges for scanning. For example, '-p 21-25,80,113, 60000-' does what you would expect (a trailing hyphen means up to 65536, a leading hyphen means 1 through). You can also use the -F (fast) option, which scans all the ports registered in your /etc/services (a la strobe).

- Flexible target specification: I often want to scan more then one host, and I certainly don't want to list every single host on a large network to scan. Everything that isn't an option (or option argument) in nmap is treated as a target host. As mentioned before, you can optionally append /mask to a hostname or IP address in order to scan all hosts with the same initial bits of the 32 bit IP address. You can use the same powerful syntax as the port specifications to specify targets like '150.12.17.71-79.7.*'. '*' is just a shortcut for 0-255, remember to escape it from your shell if used.

- detection of down hosts: Some scanners allow you to scan large networks, but they waste a huge amount of time scanning 65535 ports of a dead host! By default, nmap pings each host to make sure it is up before wasting time on it. It also does thin in parallel, to speed things up. You can change the parrallel ping lookahead with '-L' and the ping timeout with '-T'. You can turn pinging off completely with the '-D' command line option. This is useful for scanning networks like microsoft.com where ICMP echo requests can't get through. Nmap is also capable of bailing on hosts that seem down based on strange port scanning errors. It is also

meant to be tolerant of people who accidentally scan network addresses, broadcast addresses, etc.

- detection of your IP address: For some reason, a lot of scanners ask you to type in your IP address as one of the parameters. Jeez, I don't want to have to 'ifconfig' and figure out my current address every time I scan. Of course, this is better then the scanners I've seen which require recompilation every time you change your address! nmap first tries to detect your address during the ping stage. It uses the address that the echo response is received on, as that is the interface it should almost always be routed through. If it can't do this (like if you don't have host pinging enabled), nmap tries to detect your primary interface and uses that address. You can also use -S to specify it directly, but you shouldn't have to (unless you want to make it look like someone ELSE is SYN or FIN scanning a host.

Some other, more minor options:

```
 -v (verbose): This is highly recommended for interactive use.  Among other
useful messages, you will see ports come up as they are found, rather than
having to wait for the sorted summary list.

 -r (randomize): This will randomize the order in which the target host's
ports are scanned.

 -q (quash argv): This changes argv[0] to FAKE_ARGV ("pine" by default).
It also eliminates all other arguments, so you won't look too suspicious in
'w' or 'ps' listings.

 -h for an options summary.

 -R show and resolve all hosts, even down ones.
```

Also look for http://www.insecure.org/nmap/, which is the web site I plan to put future versions and more information on. In fact, you would be well advised to check there right now. (If that isn't where you are reading this).

Example Usage

To launch a stealth scan of the entire class 'B' networks 166.66.0.0 and 166.67.0.0 for the popularly exploitable imapd daemon:

```
# nmap -Up 143 166.66.0.0/16 166.67.0.0/16
```

To do a standard tcp scan on the reserved ports of host <target>:

```
> nmap target
```

To check the class 'C' network on which warez.com sits for popular services (via fragmented SIN scan):

```
# nmap -fsp 21,22,23,25,80,110 warez.com/24
```

To scan the same network for all the services in your /etc/services via (very fast) tcp scan:

```
> nmap -F warez.com/24
```

To scan secret.pathetic.net using the ftp bounce attack off of ftp.pathetic.net:

```
> nmap -Db ftp.pathetic.net secret.pathetic.net
```

To find hosts that are up in the the adjacent class C's 193.14.12, .13, .14, .15, ... , .30:

```
> nmap -P '193.14.[12-30].*'
```

If you don't want to have to quote it to avoid shell interpretation, this does the same thing:

```
> nmap -P 193.14.12-30.0-255
```