Detecting and deceiving network scans

Jan Engelhardt

July 14, 2007
(Last updated: May 13, 2008)

Inversion of the "robustness principle" makes us more robust actually.
"*Be conservative in what you accept and be liberal in what you do.*"

Contents

1 Introduction **2**

2 TCP Stealth Scan detection **2**

3 TCP SYN Scan detection **3**

4 TCP Connect Scan detection **4**

5 TCP Grab Scan detection **5**

6 portscan match **8**

7 CHAOS target **9**

8 DELUDE target **11**

9 Results **11**

10 Basic filters **15**

11 Pitfalls, security considerations **17**

1 Introduction

What Chaostables is, is not, can, cannot, does, does not

Chaostables is the software package that contains this document ("Detecting and deceiving network scans") and the Netfilter, Xtables and iptables extensions (plugins). It is not a Netfilter chain like INPUT and neither a table like mangle.

Chaostables does not disguise the OS type — at least not intentionally. It may happen however, that nmap's OS detection gets confused as a welcomed side effect.

The Chaostables codebase was merged into the Xtables-addons package in April 2008.

2 TCP Stealth Scan detection

In the ideal case, a user should make a connection, do what needs to be done, and close it. Standardized TCP connections begin with a SYN packet [RFC793, p.23]. Everything else can be considered anomalous. To match a scan, the inner workings of the scan program are needed. Sometimes, it also suffices to see what traffic is coming in, because that is what can be matched. As said, normal TCP connections always begin with a SYN, anything else must be forged, and therefore is easy to match. The following rule will match most anomalies, including TCP NULL, TCP FIN, TCP XMAS, and possibly other strange combinations:

```
-p tcp ! --syn -m conntrack --ctstate INVALID
```

It does *not* match TCP ACK scans, because a "spurious" ACK may very well be part of an already-existing connection where our machine just does not know about its state (e.g. after a reboot or conntrack flush).

An extra rule is required to be able to continue to receive "Connection refused" message ourselves — e.g. if you run '`telnet somehost someClosedPort`' — the returned RST and/or RST-ACK packets are not associated with any connection in Netfilter. Hence, we need an exclusion rule to the above:

```
-p tcp --tcp-flags SYN,FIN,RST RST
```

According to [RFC793], p.65, an RST will not be replied to, hence no information leak will occur as a result of accepting it. (RST-ACK is required as a response to SYN to a closed port, see [RFC793], p.37 par.3, so we will explicitly take up ACK into the `--tcp-flags` argument below.)

You can incorporate this into your rule set as follows. A user-defined chain is handy, but you can have it any way:

```
-N tcp_inval;
-A tcp_inval -p tcp --tcp-flags SYN,FIN,RST,ACK RST,ACK -j RETURN;
-A tcp_inval -j LOG --log-prefix "[STEALTH] ";
-A tcp_inval -j DROP;
-A INPUT -p tcp ! --syn -m conntrack --ctstate INVALID -j tcp_inval;
```

This allows the use of more targets, such as LOG (shown here), without repeatedly matching non-SYN, so this is the preferred way. This rule set can also be used in the FORWARD chain without fear to kill already-open connections. Active connections that have not yet been seen by Netfilter will become NEW and ESTABLISHED after the next two packets, respectively, without making the connection INVALID.

3 TCP SYN Scan detection

A SYN scan half-opens a TCP connection and terminates the handshake in the middle. In other words, if a SYN is received, we send the obligatory SYN-ACK and then the scanner immediately sends an RST. (The exact implementation may differ from scanner to scanner and OS. For example, nmap sends a SYN using a raw socket, but when the Linux *kernel* receives the return SYN-ACK, it [the kernel] does not know anything about the connection and responds with RST.) Using a state machine (automaton) inside iptables, it is easy to match the third packet:

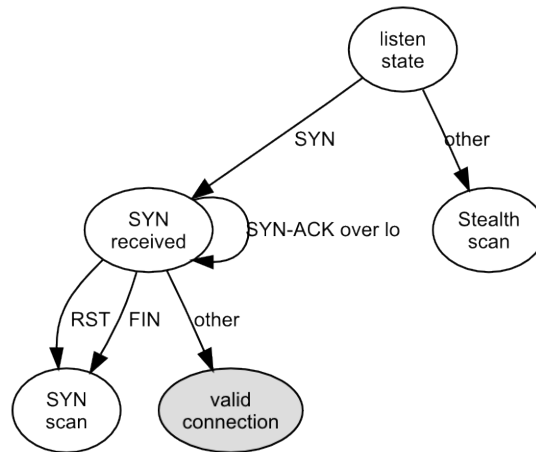


Figure 1: State graph for SYN Scan detection

When connecting to localhost, special attention needs to be given since we receive our own packets. When the socket is open, the server side sends its SYN-ACK. Under normal circumstances, this packet is only seen in the OUTPUT chain and hence is not of relevance for the state graph, which is modeled upon incoming packets. However, when the loopback interface is involved, we will see our own SYN-ACK packet again in the INPUT chain, so it is to be ignored. The rule set can modeled as follows with iptables:

```
SYN=401;
CLOSED=402;
SYNSCAN=403;
ESTAB=404;

-N mark_closed;
-A mark_closed -j CONNMARK --set-mark $CLOSED;
```

```
-N mark_estab;
-A mark_estab -j CONNMARK --set-mark $ESTAB;
-N tcp_new1;
-A tcp_new1 -i lo -p tcp --tcp-flags ALL SYN,ACK -j RETURN;
-A tcp_new1 -i lo -p tcp --tcp-flags ALL RST,ACK -g mark_closed;
-A tcp_new1 -p tcp --tcp-flags ALL ACK -g mark_estab;
-A tcp_new1 -j CONNMARK --set-mark $SYNSCAN;
-A INPUT -m connmark --mark $SYN -j tcp_new1;
-A INPUT -p tcp --syn -m conntrack --ctstate NEW -j CONNMARK --set-mark $SYN;
```

When a SYN packet in a new connection arrives, it does not have any mark set (assuming you did not set one), hence will only match the second rule in the INPUT chain (as shown here). The connection will then be marked with some integer that we define as the "SYN received" state. When the client then gives the third packet in the TCP handshake, the first rule in INPUT triggers and the second does not, because the connection is marked with `$SYN` and already has state ESTABLISHED. Note that the order of the rules is important.

If the third packet is an ACK, a goto (`-g`, note that this is different from `-j`) to the mark_estab chain is executed, the connection will be marked with `$ESTAB` and control is returned to the INPUT chain. Also note the special case for SYN-ACK, which is ignored by returning from the tcp_new1 chain and leaving the mark as-is, and the case for RST(-ACK), which will trigger the connection to be marked with `$CLOSED` so that it does not inadvertently match any other rule of the detection logic.

Blocking SYN scans is impossible, because you cannot tell in advance whether a SYN sent by the remote side is intended to be a real connection or a scan attempt. However, you could, for example, block all further requests for a while from the host which already *did* a SYN scan, using the "recent" module. Assuming handle_evil is a user-defined chain doing that, you have two ways for implementation, varying in the position of the jump to handle_evil in your own rule set (in one of two ways):

```
-A tcp_new1 -j handle_evil;
-A INPUT -m connmark --mark $SYNSCAN -j handle_evil;
```

4 TCP Connect Scan detection

SYN scans require a raw socket, which is not available to unprivileged users. Instead, such users have to use the regular interface involving the `connect` system call, where the kernel does a standards-conforming three-packet TCP handshaking. Connect scans then immediately terminate the connection using appropriate syscalls, like `shutdown` or `close`. I have noticed that nmap manages to send an RST even though `close` normally makes the kernel do the FIN sequence. Either way, it does not really matter if we get a RST or a FIN. The extended state graph is shown below, as are the iptables rules to model it.

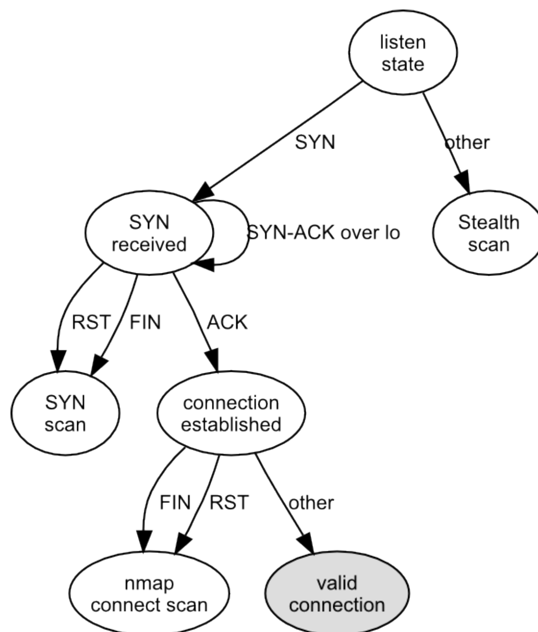


Figure 2: State graph enhanced with Connect Scan detection

```
CNSCAN=406;
VALID=408;
-N mark_cnscan;
-A mark_cnscan -j CONNMARK --set-mark $CNSCAN;
-N tcp_new3;
-A tcp_new3 -p tcp --tcp-flags SYN,FIN,RST RST -g mark_cnscan;
-A tcp_new3 -p tcp --tcp-flags SYN,FIN,RST FIN -g mark_cnscan;
-A tcp_new3 -j CONNMARK --set-mark $VALID;
-A INPUT -m connmark --mark $ESTAB -j tcp_new3;
```

Note that last rule in this code snippet must come *before* the rule to jump to tcp_new1, i.e.:

```
-A INPUT -m connmark --mark $ESTAB -j tcp_new3;
-A INPUT -m connmark --mark $SYN -j tcp_new1;
```

5 TCP Grab Scan detection

There is yet another type of scan, the banner grab scan, where a client connects to solely read bytes and then disconnect. Note that such an action may very well be part of a non-malicious action (FTP DATA connections, for example), so connections should be handled with care. Some services, such as SSH, always are bidirectional from a layer-7 point of view, so it is safe to apply Grab Scan Detection on it. Speaking of SSH, it is a prominent service that presents its data before the client takes any action, i.e. it shows the banner “SSH-2.0-OpenSSH_5.0” voluntarily. I would like to show a way how to match such “grab scans”. As already mentioned, a grab scan is where the client sends no data itself, hence its TCP packets have no payload.

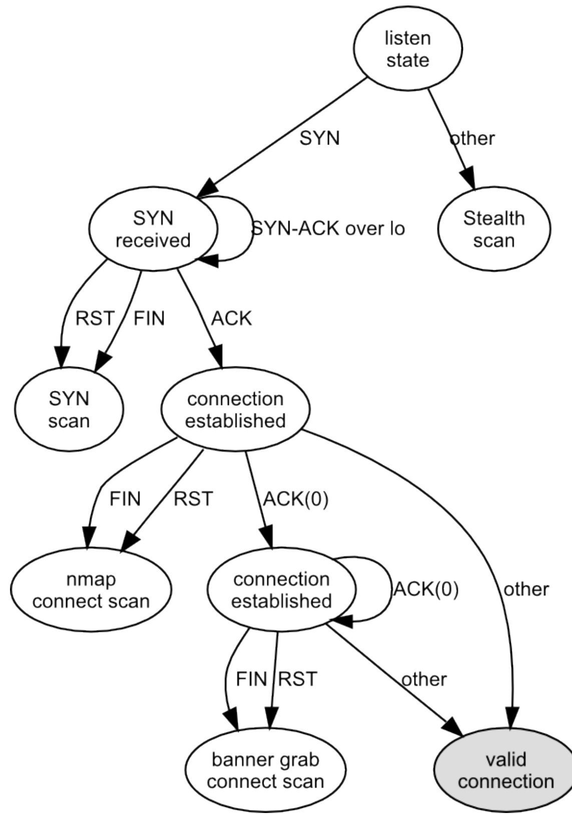


Figure 3: State graph enhanced by Grab Scan detection

Packets (even with a known amount of payload) can vary in size since the IP and TCP header allow for variable-sized option fields. It is therefore impossible to unambiguously match Grab Scans with the matches as existing of iptables version 1.4.0. A typical empty Linux TCP packet is 52 octets, that includes 20 octets for the IP header (no options), 20 octets for the TCP header and 12 octets for the TCP_LINGER2 option Linux sends with packets. However, 52 octets could also be composed of 20 IP header octets, 12 IP option octets and 20 TCP header octets. Or 20 IP header octets, 20 TCP header octets and 12 TCP payload octets. Matching with `-m length --length 52` only looks at the total layer-3 packet length, hence its use on layer-4 and above will be ambiguous. However, if one were to use it, this is how it would be done with iptables rules:

```
ESTAB2=405;
CNSCAN=406;
GRSCAN=407;
VALID=408;

-N mark_grscan;
-A mark_grscan -j CONNMARK --set-mark $GRSCAN

-N tcp_new3;
-A tcp_new3 -p tcp --tcp-flags SYN,FIN,RST RST -g mark_cnscan;
-A tcp_new3 -p tcp --tcp-flags SYN,FIN,RST FIN -g mark_cnscan;
```

```
-A tcp_new3 ! -i lo -p tcp --tcp-flags SYN,FIN,RST,ACK ACK
        -m length --length 52 -g mark_estab2;
-A tcp_new3 -j CONNMARK --set-mark $VALID;

-N tcp_new4;
-A tcp_new4 -p tcp --tcp-flags SYN,FIN,RST,ACK ACK
        -m length --length 52 -j RETURN;
-A tcp_new4 -p tcp --tcp-flags SYN,FIN,RST RST -g mark_grscan;
-A tcp_new4 -p tcp --tcp-flags SYN,FIN,RST FIN -g mark_grscan;
-A tcp_new4 -j CONNMARK --set-mark $VALID;

-A INPUT -m connmark --mark $ESTAB2 -j tcp_new4;
-A INPUT -m connmark --mark $ESTAB -j tcp_new3;
```

Note that the loopback interface is excluded again, because seeing our own packets makes it trigger early. There are possibly ways around this, but that is beyond the scope of this document. A full example iptables rule set for use with iptables-restore can be found in the source distribution. If you load it, running, for example, the Grab Scan will look like this (Netfilter log messages are from the host "master"):

```
master# iptables-restore scan_detect.ipt
master# ssh vm6402
vm6402# telnet master 22
[ESTAB1] IN=vmnet2 OUT= MAC= SRC=192.168.64.2 DST=192.168.64.1 LEN=52
TOS=0x10 PREC=0x00 TTL=64 ID=31040 DF PROTO=TCP SPT=3180
DPT=22 WINDOW=1460 RES=0x00 ACK URGP=0
Trying 192.168.64.1...
Connected to 192.168.64.1.
Escape character is '^]'.
[ESTAB2] IN=vmnet2 OUT= MAC= SRC=192.168.64.2 DST=192.168.64.1 LEN=52
TOS=0x10 PREC=0x00 TTL=64 ID=31041 DF PROTO=TCP SPT=3180
DPT=22 WINDOW=1460 RES=0x00 ACK URGP=0
SSH-2.0-OpenSSH_4.7
^]
telnet> exit
[GRSCAN] IN=vmnet2 OUT= MAC= SRC=192.168.64.2 DST=192.168.64.1 LEN=52
TOS=0x10 PREC=0x00 TTL=64 ID=31042 DF PROTO=TCP SPT=3180
DPT=22 WINDOW=1460 RES=0x00 ACK FIN URGP=0
Connection closed.
```

The portscan kernel module will correctly match a TCP packet with empty payload since it is able to inspect the IP and TCP headers more closely than the length module.

(Addendum 2008-02-15: There is a pending upgrade of the length match to do more fine-grained matching as would be required.)

6 portscan match

As the number of rules for all this portscan logic grows, it becomes a little hard to keep track of it. By putting it all into one kernel module, it can be nicely wrapped up into a single match that is listed in your iptables chains. Processing speed will also improve since the Netfilter stack is run through less often. This is a simple example for logging SYN scans:

```
-A INPUT -p tcp -m portscan --synscan -j LOG --log-prefix "[SYNSCAN] ";
```

portscan marks the connection with different values while the connection is active, reflecting the current state as per above's state graph. You *must* make sure it is configured so that it will not interfere with mark values that you use for the rest of your firewall. The mark values portscan uses can be configured at module load time or by use of the module sysfs interface in `/sys/module/-xt_portscan/parameters/`. You will find `connmark_mask`, `packet_mask` and a number of `mark_*` files in this directory. `connmark_mask` can be used to limit portscan to using a specific set of bits of the mark value. For more details, see the iptables manual for the CONNMARK target.

When portscan is matched on a packet, the packet will be specially marked so that matching the same packet with portscan in another rule will not re-trigger the detection logic and inadvertent state transitions. Like with the connection mark, you must also make sure this is configured properly. The module parameters (and sysfs attributes) `packet_mask` and `mask_seen` are used for a single packet.

Because matching on portscan will act like if you matched the connmark value, the following two rule sets are equivalent:

```
-A INPUT -m portscan --synscan -j LOG --log-prefix "[SYNSCAN] ";
-A INPUT -m portscan --cnscan -j LOG --log-prefix "[CNSCAN] ";
-A INPUT -m portscan --grscan -j LOG --log-prefix "[GRSCAN] ";

-A INPUT -m portscan;
-A INPUT -m connmark --mark $SYNSCAN -j LOG --log-prefix "[SYNSCAN] ";
-A INPUT -m connmark --mark $CNSCAN -j LOG --log-prefix "[CNSCAN] ";
```

However, the first approach is preferred because you do not need to specify the mark values (possibly numeric, depending on your scripts). libxt_portscan (the iptables code part) knows the following four options that match their corresponding state as explained before: `--stealth`, `--synscan`, `--cnscan` and `--grscan`.

6.1 UDP

The portscan module will not match on UDP. Due to its connectionless nature, you cannot apply the state machinery as described earlier.

If you are concerned about nmap scans, you can use a heuristic like "tapping on N closed ports means this must be a portscan".

6.2 SCTP/DCCP

SCTP and DCCP do have a handshake procedure which we could inspect like we do for TCP. At the moment however, the portscan code does not inspect SCTP/DCCP. (Volunteers and patches welcome.)

7 CHAOS target

Network scanners such as nmap have extra measures for operating systems that rate-limit the number of return ICMP and/or RST packets comprising net-unreachable, host-unreachable, port-unreachable (port closed) and other control messages. Those operating systems do this to not flood the network more than already is by a scan, rather than being an active stopping power to scans. When the rate limit kicks in, nmap throttles its scan timing to accommodate for this to not lose scan result accuracy. Even though Linux is not one of those operating systems exhibiting this rate-limiting behavior naturally, it can be reproduced with iptables using something as simple as:

```
-A status -m hashlimit --hashlimit-name st_limit --hashlimit-mode srcip
--hashlimit-above 2/sec -j DROP
```

For kernels before 2.6.25 and iptables (userspace) before 1.4.0.77 where hashlimit does not support inversion (`--hashlimit-above`; it used to match "below" only), the following workaround is needed:

```
-N status;
-A status -m hashlimit --hashlimit-name st_limit --hashlimit-mode srcip
        --hashlimit 2/sec --hashlimit-burst 2 -j RETURN;
-A status -j DROP;
-A OUTPUT -p icmp -j status;
-A OUTPUT -p tcp --tcp-flags SYN,FIN,RST RST -j status;
```

This limits outgoing ICMP and RST packets to two per second. Note that if the loopback interface is used, the actual number of ICMP packets you can send is halved, since you send both an ICMP echo and an echo reply through OUTPUT, therefore reaching the limit earlier.

In a better setup, one would possibly use the "srcip-dstip" hashlimit mode, or even limit based upon actual incoming traffic. For example, rate-limiting replies to TCP FNX (`-sF`, `-sN`, `-sX`) scans could be done with:

```
-A INPUT -p tcp ! --syn -m conntrack --ctstate INVALID -m hashlimit
--hashlimit-name rstlimit --hashlimit-mode srcip --hashlimit-above 1/sec
--hashlimit-burst 1 -j RETURN;
```

or for pre-2.6.25:

```
-N tcp_inval;
-A tcp_inval -m hashlimit --hashlimit-name rstlimit --hashlimit-mode
        srcip --hashlimit 1/sec --hashlimit-burst 1 -j RETURN;
-A tcp_inval -j DROP;
-A INPUT -p tcp ! --syn -m conntrack --ctstate INVALID -j tcp_inval;
```

To make things even more interesting, we can also be evil by using the nth or random matches (called statistic in Linux 2.6.18 and above).

```
-N xlimit;
-A xlimit -m statistic --mode nth --every 10 -j RETURN;
-A xlimit -j DROP;
-A INPUT -p tcp ! --syn -m conntrack --ctstate INVALID -j xlimit;
-A INPUT -p tcp ! --syn -m conntrack --ctstate INVALID -m statistic
        --mode random --probability 0.90 -j DROP;
```

The first code snippet will make the TCP stack reply to every 10th packet only, the second one drops packets with a 90% chance, hence giving 10% of the time a RST in response to closed ports. Note that these rules also work with UDP, since closed UDP ports will return a ICMP port-unreachable message. (You are encouraged to experiment yourself a bit around.) If all of this knowledge is combined, we get the CHAOS target, which, if represented with iptables rules, is equivalent to:

```
-N chaos;
-A chaos -m statistic --mode random --probability 0.01
        -j REJECT --reject-with host-unreach;
-A chaos -p tcp -m statistic --mode random --probability 0.0101
        -j DELUDE;
-A chaos -j DROP;
```

CHAOS sends host-unreachable messages at a probability of 1% (reason for this rate next subsection), otherwise sends the TCP connection (if applies) to either TARPIT or DELUDE target, or DROPs it (UDP and others). Using DELUDE/TARPIT has the extra bonus that ports will get listed as "open" in nmap's results even though there is nothing to see there. What's more, the use of random provides back non-deterministic information. Rerunning nmap multiple times on the same port or port range will yield different results. The end result is that there are much more ports listed open than there really are so the scanner is none the wiser as to which ports are "really" open without using more intrusive and detectable methods.

> "Interesting this use of random. I'll have to play with it when I get that rare bit of spare time for testing and fooling about with things not in prod or requirening immediate attention to fix! Which tend to be even more rare these days in our understaffed env. But, your reports of this random further confusing the scanner and slowing it down are extremely interesting..."
> —R. DuFresne on netfilter-devel [DuFresne]

Even in its "Insane timing" (as fast as possible) mode, nmap (version 3.81) reduces itself down to at most two TCP ports per second if it recognizes an ICMP rate limit, and even less ports per second on UDP. For the record, the "Insane Timing" mode is also a knock-often, i.e. nmap sends multiple packets per port. Anyway, random matches "every once in a while", as do nth and hashlimit so it is basically personal preference of what match to pick. It is yet to be shown which of the three has best effect, or if they all yield approximately the same slowdown.

8 DELUDE target

Due to the way TARPIT works, ports will be listed as "open" in nmap's output, though there is nothing interesting besides the tar pit there. Because the TARPIT target leaves connections open, it will fill up the conntrack tables of your machine if you happen to use connection tracking — which is most likely the case. Even if you use the NOTRACK target to deactivate connection tracking for packets (and thus possibly whole connections), routers before you may still use connection tracking whose tables can fill up and as a result evict connections from their tables. Many home routers only have support for about 1000 concurrent connections, and on embedded systems that run Linux/Netfilter, having only 32 MB RAM will also set the conntrack list size to 1024 entries. In consequence, the connection, when picked up again by the conntrack code, may get reset, or hangs, depending on firewall rules — consider the following ruleset:

```
-A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT;
-A FORWARD -m conntrack --ctstate NEW -p tcp --syn -j ACCEPT;
-A FORWARD -p icmp -j ACCEPT;
-A FORWARD -p udp -j ACCEPT;
-P FORWARD DROP;
```

With this ruleset, packets from a connection that got dropped from conntrack will not be able to pass.

To make a port look open, all we need is to send a SYN-ACK packet in the TCP handshake. A more light-weight variant of TARPIT is DELUDE, which only sends this SYN-ACK in response to SYN, and otherwise behaves like REJECT, sending RSTs out. Of course it lacks the feature to keep the client in a busy loop (which would be handy against any spammer who actually wants to send out data), but should not keep unnecessary entries in the conntrack tables.

To nmap, it looks like the port is open. Programs running through the full handshake will end up with "Connection reset by peer" instead (in 2.6.18), giving a bit more useful info than just to hang the connection like TARPIT does. This may be a feature, or something to watch out for.

9 Results

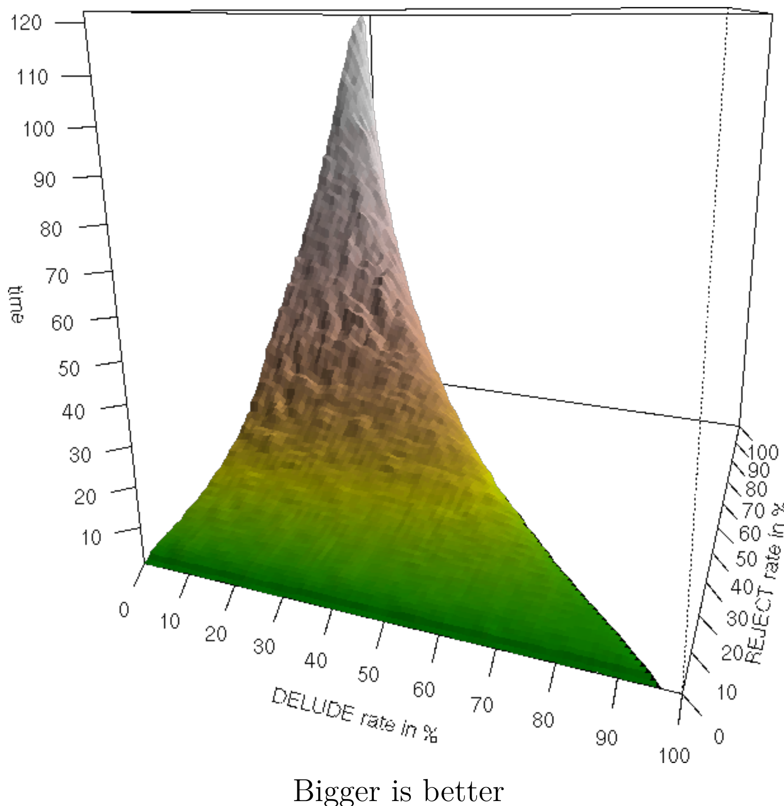
I found out that reducing the RST/ICMP reply rate ("REJECT" rate) slows nmap more. Best results are around 4%–1%. The big picture is that TCP scans are slowed down for up to 10,000% (with nmap's `-T5` "Insane timing") and UDP about 50,000%, compared to a "minimal" firewall that drops all unwanted packets. The exact details can be found in the figures below.

What is shown here are the best (fastest) scanning times by scanning localhost (which has the best "connection". You can expect to not being able to run `-T5` or `-T4` ("Aggressive timing") over the Internet, or only with losing accuracy, so results are likely to be much "worse" (= "better" for the firewall) than shown here.

9.1 SYN/Connect scan

The SYN scan is the default mode of nmap when run as root, or Connect Scan when that privilege is not given. While I have not timed Connect Scan, the results are expected to be the same — nmap should be using a timeout — or even above.

Figure 4 shows the run time of nmap scans for various combinations of REJECT and DELUDE percentages against an iptables rule that sends all packets to the CHAOS target. (Remember that the percentages are control values to the CHAOS target.) The reason this is triangle-shaped is that when the chosen DELUDE rate is for example at 90%, at most 10% of the time REJECT can be used. The DROP rate is then 100% – DELUDE – REJECT, and the height gives the nmap time for that (DELUDE rate, TARPIT rate, DROP rate) tuple.



Bigger is better

Figure 4: nmap 4.20 -T5 SYN (smoothed) time graph for CHAOS, with DELUDE as a reply engine

9.2 DELUDE vs. TARPIT

One can choose between TARPIT and DELUDE (see section 8). Figure 5 depicts the delta between the two "triangles" that the DELUDE (figure 4) and TARPIT[1] targets give.

Red blocks mean that trapping nmap scans with DELUDE is more effective than with TARPIT, blue being the opposite, and white meaning neutral. The tendencies for each are split up into a few levels of five seconds difference each. As can be seen, there is a fair mix of red-tendency and blue-tendency blocks in the figure, from which we can deduce that DELUDE and TARPIT yield about equal timings. The variance is within ±25 seconds, and is most likely due to process scheduling, nmap port scheduling, and cosmic rays.

[1] Omitted in this document, since it looks the same. Details on difference shown in figure 5 instead.

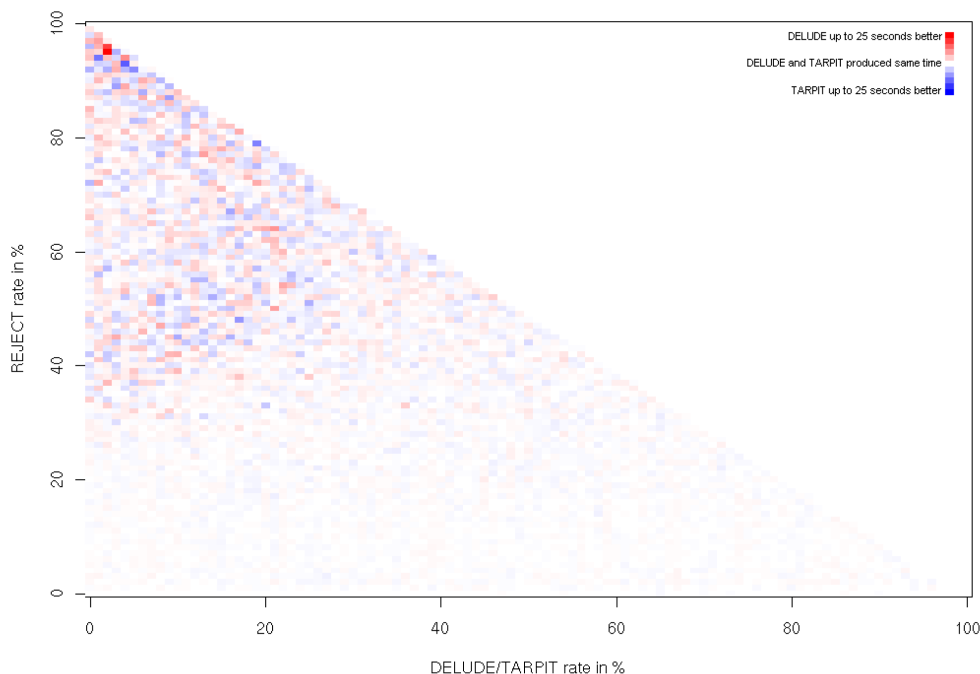


Figure 5: nmap slowdown: DELUDE vs. TARPIT

9.3 Exotic scans

The other five scanning modes, TCP FIN, TCP NULL, TCP XMAS — collectively referred to as FNX in this document — as well as ACK and UDP scan modes, can also be delayed with success. I have additionally provided the line for the `-T3` "Normal" nmap timing parameter as a comparison. Each line is the average of nine runs — three FIN, three NULL and three XMAS runs. Please note that all graphs have a *logarithmic* time scale![2]

The lines for `-T4` and `-T5` are mostly constantly increasing, only the `-T3` lines have a sudden time bump at about 70% — this seems to be an nmap heuristic between the two extremes (a) general packet drop that can happen on the Internet (b) explicit packet drop by the target system. The timing quickly raises in figure 6, most likely because FNX scans exploit operating system-dependent behavior and are thus given more time for a reply.

In figure 7, only the `-T3` line raises very quickly, while `-T4` and `-T5` remain at a "low" resulting time (but still remembering this is logarithmic scale), only raising sharply at the end, at about 7%. We can probably say this is all internal nmap magic[3].

However, it should also be noted that there is a drop in effectiveness in the 2%–0.2% REJECT rate window for most scans.

[2]Yes, it took an awful long time to acquire all the empirical results.

[3]I have not had a look at the nmap code what exactly it does.

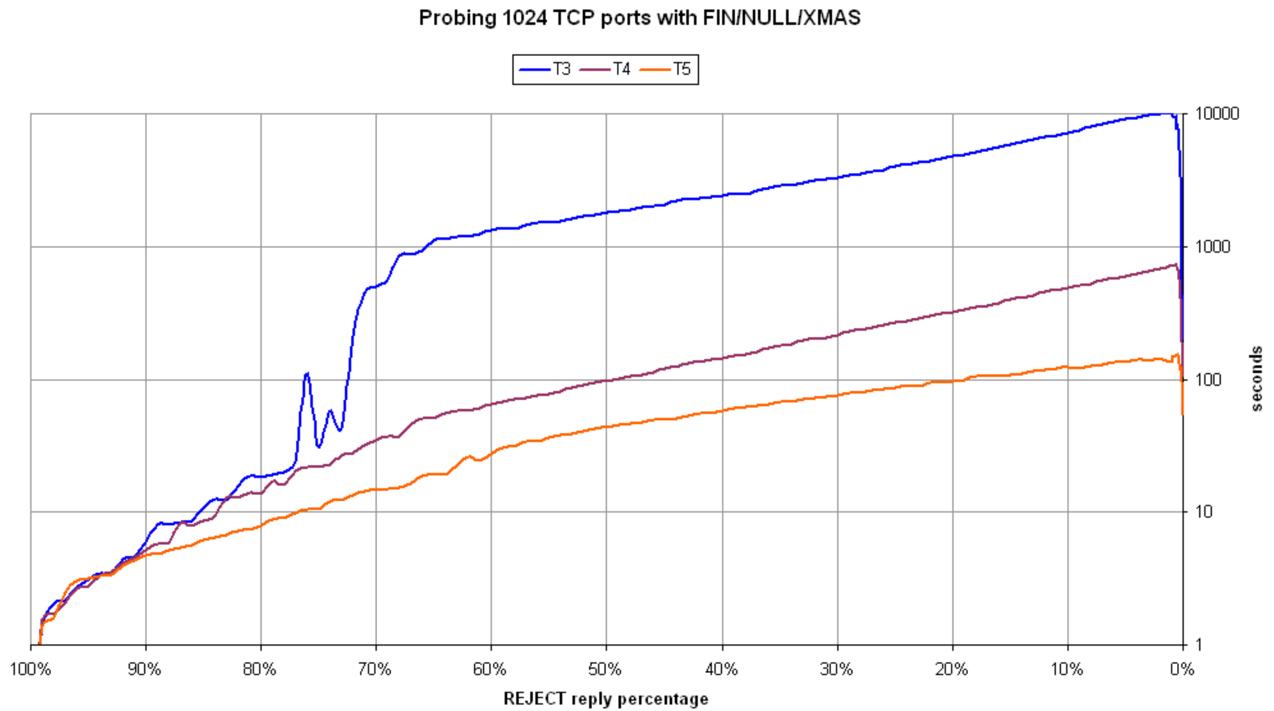


Figure 6: nmap 4.20 FIN/NULL/XMAS time graph (average over all three) for `-T3`, `-T4` and `-T5` timings

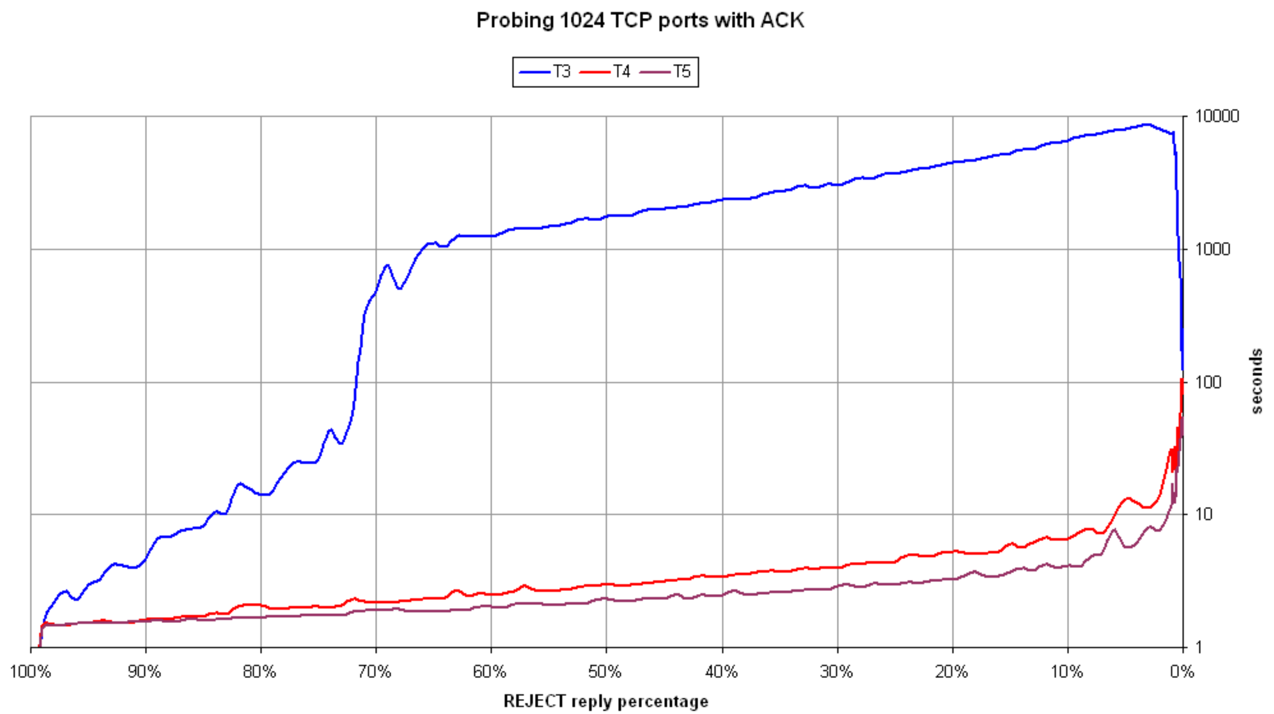


Figure 7: nmap 4.20 ACK time graph

9.4 UDP scan

UDP scans have a *very* high variance between 90% and 60%, which is probably again nmap's implementation. I believe it is not the randomness that CHAOS delivers, since that same randomness makes for a smooth graph in the TCP scans. Again, these are empirical results; to get a theoretical line, the data points between 90% and 60% should probably be undertaken a monotonically-increasing interpolation.

It can be observed that in `-T5` mode, nmap imposes an upper limit on itself for scanning ports.

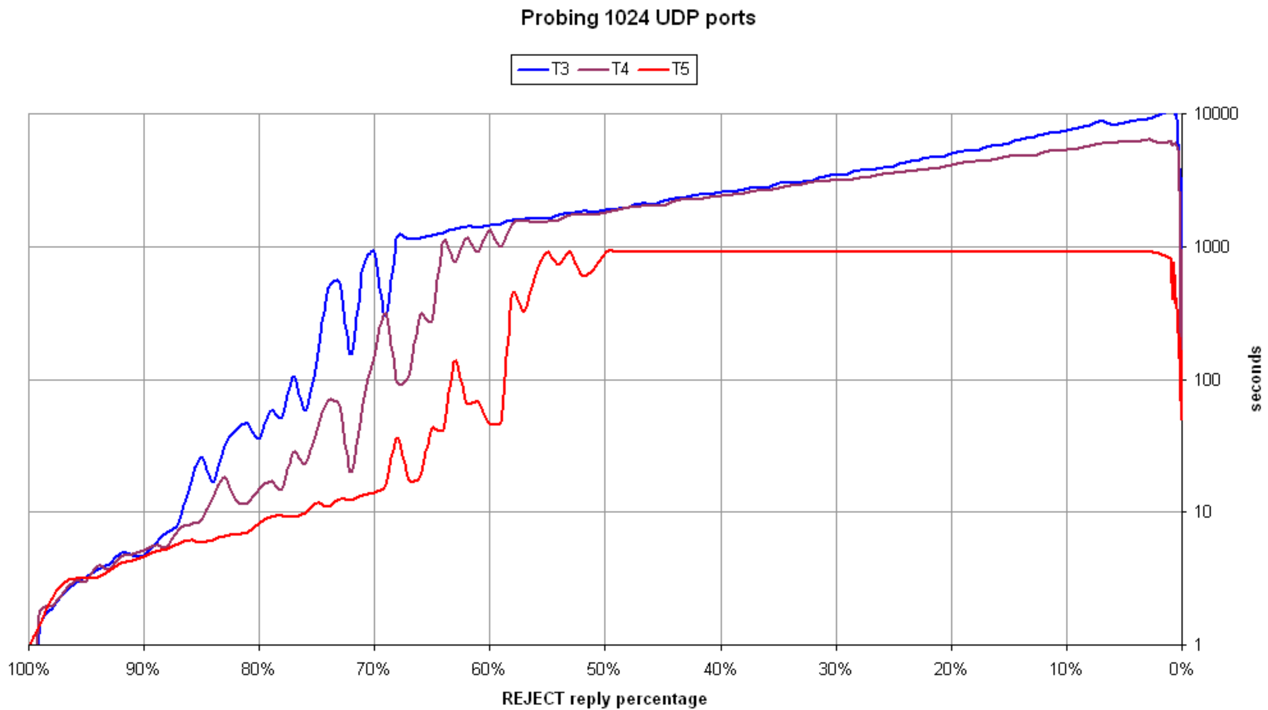


Figure 8: nmap 4.20 UDP time graph

9.5 Conclusion

The CHAOS target comes preconfigured with 1% REJECT reply rate[4] and 1% DELUDE reply rate. My personal opinion is to use the DELUDE target on most ports, and TARPIT only on a selected number of ports, for example 25/tcp for SMTP.

10 Basic filters

This section contains a collection of other thoughts regarding firewalling. Even if not used in a ruleset, add it to your pool of knowledge.

[4] I should probably set the default to 2% or 3%.

10.1 Kernel policies

Other parts in the Linux kernel besides Netfilter may also have switches to control packet flow. The routing layer has some of these, which can be changed in `/proc/sys/net/ipv4/conf/*/`. The important ones are `accept_redirects`, `accept_source_route`, `rp_filter`, and, secondarily, `send_redirects`. The first two define whether the routing code should consider these kinds of ICMP messages (Redirect, Router Solicitation) for its execution flow. `rp_filter` checks if the packet can legitimately come from the interface it was received on [LXRfib].

10.2 RFC1256 Router Discovery packets

Windows 98 likes to spew out multicast router-solicitation packets every now and then in LAN networks. Most ISPs should filter multicast, Linux's routing code is not configured to accept these by default, multicast packets do not pass through ip_tables and ip6_tables (two kernel modules providing the table-like firewalling as we know it), and above all, multicast packets typically behave like having a TTL of zero. Paranoid sysadmins may anyway block ICMP Redirect [RFC792], Router Advertisement and Router Solicitation [RFC1256] in FORWARD anyway, in case someone crafts malicious unicast packets.

[RFC792]divides ICMP packet types hierarchically into so-called types and codes. iptables may either match all codes of a specific type or one code of one specific type, or any type.

```
-N icmp_drop;
-A icmp_drop -p icmp --icmp-type redirect -j DROP;
-A icmp_drop -p icmp --icmp-type router-advertisement -j DROP;
-A icmp_drop -p icmp --icmp-type router-solicitation -j DROP;
-A INPUT -j icmp_drop;
-A FORWARD -j icmp_drop;
```

The redirect type includes four codes, hence blocks both network-redirect and host-redirect, plus the two TOS variants thereof. router-advertisement and router-solicitation are types without any subcodes.

10.3 Traceroute filtering

A remote host can send an ICMP Echo packet with the "IP Traceroute" [RFC1393] flag. This cannot reliably be blocked, since the ipv4options module does not have code to look at it. However, we can block the return packets passing the machine using:

```
-A FORWARD -p icmp --icmp-type 30 -j DROP;
```

Type 30 must be specified numerically (as shown above), since iptables does not know the names for such extensions that do not seem to be widely deployed. Linux does not seem to support IP Traceroute as of this writing, so it is not necessary to block it in the OUTPUT chain.

Filtering UDP Traceroute and TCP Traceroute is quite impossible without deranging other traffic. A very low TTL may indicate an ongoing traceroute scan, but any TTL is legitimate.

11 Pitfalls, security considerations

Thanks to Doug Hoyte for bringing cases 11.2-11.4 to my attention. Quoting him:

> "However, keep in mind that sometimes implementing measures to limit data gathered by scanners can actually become a security risk itself. I suppose it's a tradeoff between just how paranoid you want to be and how many DoS vectors you open. In particular, is it ever possible for an attacker to rate limit or close connections to degrade service, or spoof scan attempts in log files?"
> —D.Hoyte via private mail

11.1 Empty connections

Sometimes, "good" hosts send packets that get classified as a SYN scan or Connect Scan attempt. Windows XP is one such candidate (it also sets the URG/PSH flag in the TCP handshake in a SMB connections). It is therefore advised to use the CHAOS target cautiously in conjunction with its `--tarpit` option in internal networks. `--delude` should be fine, but confusing to the end user.

11.2 TCP SYN from spoofed IP address or unwanted TCP flags

If a zombie host sends TCP SYN, the host kernel will answer with SYN ACK, but the decoy (the IP address the zombie sent) responds with TCP RST since it has no knowledge about the connection tuple. Now, if the user combines `-m portscan --synscan` with a blacklist (e.g. `-m recent` with `-j REJECT`, as suggested above), the decoy IP address gets locked out for the time period the user set with `-m recent`. However, it is up to the user to combine [or not] `-m portscan` with a blacklist mechanism.

11.3 Accessing closed TCP/UDP ports

The user may implement a blacklist mechanism at the end of an iptables chain to catch and block clients trying to access random ports that are either closed anyway or forbidden. This falls into the user domain again, not Chaostables. Two sub-cases come up:

1. If the address is spoofed, the decoy address gets blacklisted.
2. The IP address gets blocked/rejected. If that address is a NAT gateway, tough luck for all its users. You could say that a user who runs filesharing programs (which often do random connects — e.g. edonkey, gnutella) should not try access to your machine. It might also be the next Internet virus. Chaostables should be used cautiously on gateways. (You could combine it with the xt_layer7 or ipt_ipp2p matches to explicitly accept known filesharing protocols.) If however, e.g. filesharing is forbidden in your local network, blacklisting hosts that do random things helps isolating which host runs it (by means of user claiming the network does not work).

11.4 Terminating TCP connections

Chaostables, specifically the DELUDE target, only works on addresses/ports/etc. that you specify in your iptables rules. The rest is basic TCP — if a zombie sends a spoofed SYN RST, with the correct ACK value and TCP window, etc., the connection might terminate the same way as without Chaostables.

References

[DuFresne] R. DuFresne on the Netfilter mailing list – response to the idea
`http://lists.netfilter.org/pipermail/netfilter/2005-June/061328.html`

[LXRfib] `linux/net/ipv4/fib_frontend.c` from kernel 2.6.18
`http://lxr.linux.no/source/net/ipv4/fib_frontend.c?v=2.6.18#L162`

[PacketFlow] Packet flow in the Linux kernel
`http://jengelh.hopto.org/images/nf-packet-flow.png`

[RFC792] RFC 792: Internet Control Message Protocol (ICMP)
`http://rfc.net/rfc792.html`

[RFC793] RFC 793: Transmission Control Protocol (TCP)
`http://rfc.net/rfc793.html`

[RFC1256] RFC 1256: ICMP Router Discovery Messages
`http://rfc.net/rfc1256.html`

[RFC1393] RFC 1393: Traceroute using an IP option
`http://rfc.net/rfc1393.html`