[Skip to main content](https://lms.alnafi.com/xblock/block-v1:alnafi+DCCS102+2025_DCCS+type@vertical+block@46ff3266fffd4ff09be15625e30527c7?exam_access=&recheck_access=1&show_bookmark=0&show_title=0&view=student_view#main)

**Firewall Detection**

The first step toward bypassing firewall rules is to understand them. Where possible, Nmap distinguishes between ports that are reachable but closed and those that are actively filtered. An effective technique is to start with a normal SYN port scan, then move on to more exotic techniques such as ACK scan and IP ID sequencing to gain a better understanding of the network.

**Standard SYN Scan**

One helpful feature of the TCP protocol is that systems are required by [RFC 793](http://www.rfc-editor.org/rfc/rfc793.txt) to send a negative response to unexpected connection requests in the form of a TCP RST (reset) packet. The RST packet makes closed ports easy for Nmap to recognize. Filtering devices such as firewalls, on the other hand, tend to drop packets destined for disallowed ports. In some cases, they send ICMP error messages (usually port unreachable) instead. Because dropped packets and ICMP errors are easily distinguishable from RST packets, Nmap can reliably detect filtered TCP ports from open or closed ones, and it does so automatically. This is shown in [Example 10.1](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-ids-standardsyn).

Example 10.1. Detection of closed and filtered TCP ports

# nmap -sS -T4 scanme.nmap.org

Starting Nmap ( https://nmap.org )

Nmap scan report for 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

Nmap done: 1 IP address (1 host up) scanned in 5.40 seconds

One of the most important lines in [Example 10.1](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-ids-standardsyn) is the note “Not shown: 994 filtered ports”. In other words, this host has a proper deny-by-default firewall policy. Only those ports the administrator explicitly allowed are reachable, while the default action is to deny (filter) them. Three of the enumerated ports are in the open state (22, 53, and 80), and another three are closed (25, 70, and 113). The remaining 994 tested ports are unreachable by this standard scan (filtered).

Sneaky firewalls that return RST

While the Nmap distinction between closed TCP ports (which return a RST packet) and filtered ports (returning nothing or an ICMP error) is usually accurate, many firewall devices are now capable of forging RST packets as though they are coming from the destination host and claiming that the port is closed. One example of this capability is the Linux iptables system, which offers many methods for rejecting undesired packets. The iptables man page documents this feature as follows:

--reject-with type

The type given can be icmp-net-unreachable, icmp-host-unreachable, icmp-port-unreachable, icmp-proto-unreachable, icmp-net-prohibited or icmp-host-prohibited, which return the appropriate ICMP error message (port-unreachable is the default). The option tcp-reset can be used on rules which only match the TCP protocol: this causes a TCP RST packet to be sent back. This is mainly useful for blocking ident (113/tcp) probes which frequently occur when sending mail to broken mail hosts (which won't accept your mail otherwise).

Forging RST packets by firewalls and IDS/IPS is not particularly common outside of port 113, as it can be confusing to legitimate network operators and it also allows scanners to move on to the next port immediately without waiting on the timeout caused by dropped packets. Nevertheless, it does happen. Such forgery can usually be detected by careful analysis of the RST packet in comparison with other packets sent by the machine. [the section called “Detecting Packet Forgery by Firewall and Intrusion Detection Systems”](https://nmap.org/book/firewall-ids-packet-forgery.html) describes effective techniques for doing so.

**ACK Scan**

As described in depth in [the section called “TCP ACK Scan (-sA)”](https://nmap.org/book/scan-methods-ack-scan.html), the ACK scan sends TCP packets with only the ACK bit set. Whether ports are open or closed, the target is required by [RFC 793](http://www.rfc-editor.org/rfc/rfc793.txt) to respond with a RST packet. Firewalls that block the probe, on the other hand, usually make no response or send back an ICMP destination unreachable error. This distinction allows Nmap to report whether the ACK packets are being filtered. The set of filtered ports reported by an Nmap ACK scan is often smaller than for a SYN scan against the same machine because ACK scans are more difficult to filter. Many networks allow nearly unrestricted outbound connections, but wish to block Internet hosts from initiating connections back to them. Blocking incoming SYN packets (without the ACK bit set) is an easy way to do this, but it still allows any ACK packets through. Blocking those ACK packets is more difficult, because they do not tell which side started the connection. To block unsolicited ACK packets (as sent by the Nmap ACK scan), while allowing ACK packets belonging to legitimate connections, firewalls must statefully watch every established connection to determine whether a given ACK is appropriate. These stateful firewalls are usually more secure because they can be more restrictive. Blocking ACK scans is one extra available restriction. The downsides are that they require more resources to function, and a stateful firewall reboot can cause a device to lose state and terminate all established connections passing through it.

While stateful firewalls are widespread and rising in popularity, the stateless approach is still quite common. For example, the Linux Netfilter/iptables system supports the --syn convenience option to make the stateless approach described above easy to implement.

In the previous section, a SYN scan showed that all but six of 1,000 common ports on scanme.nmap.org were in the filtered state. [Example 10.2](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-ids-ackscan-scanme) demonstrates an ACK scan against the same host to determine whether it is using a stateful firewall.

Example 10.2. ACK scan against Scanme

# nmap -sA -T4 scanme.nmap.org

Starting Nmap ( https://nmap.org )

Nmap scan report for scanme.nmap.org (64.13.134.52)

Not shown: 994 filtered ports

PORT    STATE      SERVICE

22/tcp  unfiltered ssh

25/tcp  unfiltered smtp

53/tcp  unfiltered domain

70/tcp  unfiltered gopher

80/tcp  unfiltered http

113/tcp unfiltered auth

Nmap done: 1 IP address (1 host up) scanned in 5.96 seconds

The same six ports displayed in the SYN scan are shown here. The other 994 are still filtered. This is because Scanme is protected by this stateful iptables directive: iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT. This only accepts packets that are part of or related to an established connection. Unsolicited ACK packets sent by Nmap are dropped, except to the six special ports shown. Special rules allow all packets to the ports 22, 25, 53, 70, and 80, as well as sending a RST packet in response to port 113 probes. Note that the six shown ports are in the unfiltered state, since the ACK scan cannot further divide them into open (22, 53, and 80) or closed (25, 70, 113).

Now let us look at another example. A Linux host named Para on my local network uses the following (simplified to save space) firewall script:

#!/bin/sh

#

# A simple, stateless, host-based firewall script.

# First of all, flush & delete any existing tables

iptables -F

iptables -X

# Deny by default (input/forward)

iptables --policy INPUT DROP

iptables --policy OUTPUT ACCEPT

iptables --policy FORWARD DROP

# I want to make ssh and www accessible from outside

iptables -A INPUT -m multiport -p tcp --destination-port 22,80 -j ACCEPT

# Allow responses to outgoing TCP requests

iptables -A INPUT --proto tcp ! --syn -j ACCEPT

This firewall is stateless, as there is no sign of the --state option or the -m state module request. [Example 10.3](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-ids-scans-para) shows SYN and ACK scans against this host.

Example 10.3. Contrasting SYN and ACK scans against Para

# nmap -sS -p1-100 -T4 para

Starting Nmap ( https://nmap.org )

Nmap scan report for para (192.168.10.191)

Not shown: 98 filtered ports

PORT   STATE  SERVICE

22/tcp open   ssh

80/tcp closed http

MAC Address: 00:60:1D:38:32:90 (Lucent Technologies)

Nmap done: 1 IP address (1 host up) scanned in 3.81 seconds

# nmap -sA -p1-100 -T4 para

Starting Nmap ( https://nmap.org )

All 100 scanned ports on para (192.168.10.191) are: unfiltered

MAC Address: 00:60:1D:38:32:90 (Lucent Technologies)

Nmap done: 1 IP address (1 host up) scanned in 0.70 seconds

In the SYN scan, 98 of 100 ports are filtered. Yet the ACK scan shows every scanned port being unfiltered. In other words, all of the ACK packets are sneaking through unhindered and eliciting RST responses. These responses also make the scan more than five times as fast, since it does not have to wait on timeouts.

Now we know how to distinguish between stateful and stateless firewalls, but what good is that? The ACK scan of Para shows that some packets are probably reaching the destination host. I say probably because firewall forgery is always possible. While you may not be able to establish TCP connections to those ports, they can be useful for determining which IP addresses are in use, OS detection tests, certain IP ID shenanigans, and as a channel for tunneling commands to rootkits installed on those machines. Other scan types, such as FIN scan, may even be able to determine which ports are open and thus infer the purpose of the hosts. Such hosts may be useful as zombies for an IP ID idle scan.

This pair of scans also demonstrates that what we are calling a port state is not solely a property of the port itself. Here, the same port number is considered filtered by one scan type and unfiltered by another. What IP address you scan from, the rules of any filtering devices along the way, and which interface of the target machine you access can all affect how Nmap sees the ports. The port table only reflects what Nmap saw when running from a particular machine, with a defined set of options, at the given time.

**IP ID Tricks**

The humble identification field within IP headers can divulge a surprising amount of information. Later in this chapter it will be used for port scanning (idle scan technique) and to detect when firewall and intrusion detection systems are forging RST packets as though they come from protected hosts. Another neat trick is to discern what source addresses make it through the firewall. There is no point spending hours on a blind spoofing attack “from” 192.168.0.1 if some firewall along the way drops all such packets.

I usually test this condition with Nping, the free network probing tool that comes with Nmap. This is a rather complex technique, but it can be valuable sometimes. Here are the steps I take:

1. Find at least one accessible (open or closed) port of one machine on the internal network. Routers, printers, and Windows boxes often work well. Recent releases of Linux, Solaris, and OpenBSD have largely resolved the issue of predictable IP ID sequence numbers and will not work. The machine chosen should have little network traffic to avoid confusing results.
2. Verify that the machine has predictable IP ID sequences. The following command tests a Windows XP machine named Playground. The Nping options request that five SYN packets be sent to port 80, one second apart.
3. # nping -c 5 --delay 1 -p 80 --tcp playground
4. Starting Nping ( https://nmap.org/nping )
5. SENT (0.0210s) TCP 192.168.0.21:42091 > 192.168.0.40:80 S ttl=64 id=48089iplen=40  seq=136013019 win=1480
6. RCVD (0.0210s) TCP 192.168.0.40:80 > 192.168.0.21:42091 RA ttl=128 id=4900iplen=40  seq=0 win=0
7. SENT (1.0220s) TCP 192.168.0.21:42091 > 192.168.0.40:80 S ttl=64 id=41250 iplen=40  seq=136013019 win=1480
8. RCVD (1.0220s) TCP 192.168.0.40:80 > 192.168.0.21:42091 RA ttl=128 id=4901iplen=40  seq=0 win=0
9. SENT (2.0240s) TCP 192.168.0.21:42091 > 192.168.0.40:80 S ttl=64 id=10588 iplen=40  seq=136013019 win=1480
10. RCVD (2.0250s) TCP 192.168.0.40:80 > 192.168.0.21:42091 RA ttl=128 id=4902iplen=40  seq=0 win=0
11. SENT (3.0270s) TCP 192.168.0.21:42091 > 192.168.0.40:80 S ttl=64 id=55928 iplen=40  seq=136013019 win=1480
12. RCVD (3.0280s) TCP 192.168.0.40:80 > 192.168.0.21:42091 RA ttl=128 id=4903iplen=40  seq=0 win=0
13. SENT (4.0300s) TCP 192.168.0.21:42091 > 192.168.0.40:80 S ttl=64 id=3309 iplen=40  seq=136013019 win=1480
14. RCVD (4.0300s) TCP 192.168.0.40:80 > 192.168.0.21:42091 RA ttl=128 id=4904iplen=40  seq=0 win=0
15. Max rtt: 0.329ms | Min rtt: 0.288ms | Avg rtt: 0.300ms
16. Raw packets sent: 5 (200B) | Rcvd: 5 (230B) | Lost: 0 (0.00%)
17. Tx time: 4.00962s | Tx bytes/s: 49.88 | Tx pkts/s: 1.25
18. Rx time: 5.01215s | Rx bytes/s: 45.89 | Rx pkts/s: 1.00
19. Nping done: 1 IP address pinged in 5.03 seconds

Since the IP ID fields are perfectly sequential, we can move on to the next test. If they were random or very far apart, we would have to find a new accessible host.

1. Start a flood of probes to the target from a host near your own (just about any host will do). An example command is nping -S scanme.nmap.org --rate 10 -p 80 -c 10000 --tcp playground. Replace scanme.nmap.org with some other host of your choice, and playground with your target host. Getting replies back is not necessary, because the goal is simply to increment the IP ID sequences. Do not use the real address of the machine you are running Nping from. Using a machine nearby on the network is advised to reduce the probability that your own ISP will block the packets.

While this is going on, redo the test from the previous step against your target machine.

# nping -c 5 --delay 1 -p 80 --tcp playground

Starting Nping ( https://nmap.org/nping )

SENT (0.0210s) TCP 192.168.0.21:1781 > 192.168.0.40:80 S ttl=64 id=61263iplen=40  seq=292367194 win=1480

RCVD (0.0220s) TCP 192.168.0.40:80 > 192.168.0.21:1781 RA ttl=128 id=5755iplen=40  seq=0 win=0

SENT (1.0220s) TCP 192.168.0.21:1781 > 192.168.0.40:80 S ttl=64 id=30096iplen=40  seq=292367194 win=1480

RCVD (1.0220s) TCP 192.168.0.40:80 > 192.168.0.21:1781 RA ttl=128 id=5766iplen=40  seq=0 win=0

SENT (2.0240s) TCP 192.168.0.21:1781 > 192.168.0.40:80 S ttl=64 id=26815iplen=40  seq=292367194 win=1480

RCVD (2.0240s) TCP 192.168.0.40:80 > 192.168.0.21:1781 RA ttl=128 id=5777iplen=40  seq=0 win=0

SENT (3.0260s) TCP 192.168.0.21:1781 > 192.168.0.40:80 S ttl=64 id=49116iplen=40  seq=292367194 win=1480

RCVD (3.0270s) TCP 192.168.0.40:80 > 192.168.0.21:1781 RA ttl=128 id=5788iplen=40  seq=0 win=0

SENT (4.0290s) TCP 192.168.0.21:1781 > 192.168.0.40:80 S ttl=64 id=2916iplen=40  seq=292367194 win=1480

RCVD (4.0300s) TCP 192.168.0.40:80 > 192.168.0.21:1781 RA ttl=128 id=5799iplen=40  seq=0 win=0

Max rtt: 0.342ms | Min rtt: 0.242ms | Avg rtt: 0.272ms

Raw packets sent: 5 (200B) | Rcvd: 5 (230B) | Lost: 0 (0.00%)

Tx time: 4.00853s | Tx bytes/s: 49.89 | Tx pkts/s: 1.25

Rx time: 5.01106s | Rx bytes/s: 45.90 | Rx pkts/s: 1.00

Nping done: 1 IP address pinged in 5.03 seconds

This time, the IP IDs are increasing by roughly 11 per second instead of one. The target is receiving our 10 forged packets per second, and responding to each of them. Each response increments the IP ID. Some hosts use a unique IP ID sequence for each IP address they communicate with. If that had been the case, we would not have seen the IP ID leaping like this and we would have to look for a different target host on the network.

1. Repeat step 3 using spoofed addresses that you suspect may be allowed through the firewall or trusted. Try addresses behind their firewall, as well as the [RFC 1918](http://www.rfc-editor.org/rfc/rfc1918.txt) private networks such as 10.0.0.0/8, 192.168.0.0/16, and 172.16.0.0/12. Also try localhost (127.0.0.1) and maybe another address from 127.0.0.0/8 to detect cases where 127.0.0.1 is hard coded in. There have been many security holes related to spoofed localhost packets, including the infamous Land denial of service attack. Misconfigured systems sometimes trust these addresses without checking whether they came from the loopback interface. If a source address gets through to the end host, the IP ID will jump as seen in step 3. If it continues to increment slowly as in step 2, the packets were likely dropped by a firewall or router.

The end result of this technique is a list of source address netblocks that are permitted through the firewall, and those that are blocked. This information is valuable for several reasons. The IP addresses a company chooses to block or allow may give clues as to what addresses are used internally or trusted. For example, machines on a company's production network might trust IP addresses on the corporate network, or trust a system administrator's personal machine. Machines on the same production network also sometimes trust each other, or trust localhost. Common IP-based trust relationships are seen in NFS exports, host firewall rules, TCP wrappers, custom applications, rlogin, etc. Another example is SNMP, where a spoofed request to a Cisco router could cause the router to transfer (TFTP) its configuration data back to the attacker. Before spending substantial time trying to find and exploit these problems, use the test described here to determine whether the spoofed packets even get through.

A concrete example of this trusted-source-address problem is that I once found that a company's custom UDP service allowed users to skip authentication if they came from special netblocks entered into a configuration file. These netblocks corresponded to different corporate locations, and the feature was meant to ease administration and debugging. Their Internet-facing firewall smartly tried to block those addresses, as actual employees could access production from a private link instead. But by using the techniques described in this section, I found that the firewall was not perfectly synced with the config file. There were a few addresses from which I could successfully forge the UDP control messages and take over their application.

This technique of mapping out the firewall rules does not use Nmap, but the results are valuable for future runs. For example, this test can show whether to use certain decoys (-D). The best decoys will make it all the way to the target system. In addition, forged packets must get through for the IP ID idle scan (discussed later) to work. Testing potential source IPs with this technique is usually easier than finding and testing every potential idle proxy machine on a network. Potential idle proxies need only be tested if they pass step number two, above.

**UDP Version Scanning**

The previous sections have all focused on the prevalent TCP protocol. Working with UDP is often more difficult because the protocol does not provide acknowledgment of open ports like TCP does. Many UDP applications will simply ignore unexpected packets, leaving Nmap unsure whether the port is open or filtered. So Nmap places these ambiguous ports in the open|filtered state, as shown in [Example 10.4](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-udp-scan).

Example 10.4. UDP scan against firewalled host

# nmap -sU -p50-59 scanme.nmap.org

Starting Nmap ( https://nmap.org )

Nmap scan report for scanme.nmap.org (64.13.134.52)

PORT   STATE         SERVICE

50/udp open|filtered re-mail-ck

51/udp open|filtered la-maint

52/udp open|filtered xns-time

53/udp open|filtered domain

54/udp open|filtered xns-ch

55/udp open|filtered isi-gl

56/udp open|filtered xns-auth

57/udp open|filtered priv-term

58/udp open|filtered xns-mail

59/udp open|filtered priv-file

Nmap done: 1 IP address (1 host up) scanned in 1.38 seconds

This 10-port scan was not very helpful. No port responded to the probe packets, and so they are all listed as open or filtered. One way to better understand which ports are actually open is to send a whole bunch of UDP probes for dozens of different known UDP services in the hope of eliciting a response from any open ports. Nmap version detection ([Chapter 7, Service and Application Version Detection](https://nmap.org/book/vscan.html)) does exactly that. [Example 10.5](https://nmap.org/book/determining-firewall-rules.html#defeating-firewalls-udp-version-scan) shows the same scan with the addition of version detection (-sV).

Example 10.5. UDP version scan against firewalled host

# nmap -sV -sU -p50-59 scanme.nmap.org

Starting Nmap ( https://nmap.org )

Nmap scan report for scanme.nmap.org (64.13.134.52)

PORT   STATE         SERVICE    VERSION

50/udp open|filtered re-mail-ck

51/udp open|filtered la-maint

52/udp open|filtered xns-time

53/udp open          domain     ISC BIND 9.3.4

54/udp open|filtered xns-ch

55/udp open|filtered isi-gl

56/udp open|filtered xns-auth

57/udp open|filtered priv-term

58/udp open|filtered xns-mail

59/udp open|filtered priv-file

Nmap done: 1 IP address (1 host up) scanned in 56.59 seconds

Version detection shows beyond a doubt that port 53 (domain) is open, and even what it is running. The other ports are still open|filtered because they did not respond to any of the probes. They are probably filtered, though this is not guaranteed. They could be running a service such as SNMP which only responds to packets with the correct community string. Or they could be running an obscure or custom UDP service for which no Nmap version detection probe exists. Also note that this scan took more than 40 times as long as the previous scan. Sending all of those probes to each port is a relatively slow process. Adding the --version-intensity 0 option would reduce scan time significantly by only sending the probes most likely to elicit a response from services at a given port number.