[Skip to main content](https://lms.alnafi.com/xblock/block-v1:alnafi+DCCS102+2025_DCCS+type@vertical+block@b059608f186d414facc3b296a82f3ea5?exam_access=&recheck_access=1&show_bookmark=0&show_title=0&view=student_view#main)

Reference <https://github.com/robertdavidgraham/masscan>

**MASSCAN: Mass IP port scanner**

This is an Internet-scale port scanner. It can scan the entire Internet in under 5 minutes, transmitting 10 million packets per second, from a single machine.

Its usage (parameters, output) is similar to nmap, the most famous port scanner. When in doubt, try one of those features -- features that support widespread scanning of many machines are supported, while in-depth scanning of single machines aren't.

Internally, it uses asynchronous transmission, similar to port scanners like scanrand, unicornscan, and ZMap. It's more flexible, allowing arbitrary port and address ranges.

NOTE: masscan uses its own **ad hoc TCP/IP stack**. Anything other than simple port scans may cause conflict with the local TCP/IP stack. This means you need to use either the --src-ip option to run from a different IP address, or use --src-port to configure which source ports masscan uses, then also configure the internal firewall (like pf or iptables) to firewall those ports from the rest of the operating system.

This tool is free, but consider contributing money to its development: Bitcoin wallet address: 1MASSCANaHUiyTtR3bJ2sLGuMw5kDBaj4T

**Building**

On Debian/Ubuntu, it goes something like the following. It doesn't really have any dependencies other than a C compiler (such as gcc or clang).

sudo apt-get --assume-yes install git make gcc git clone https://github.com/robertdavidgraham/masscan cd masscan make

This puts the program in the masscan/bin subdirectory. To install it (on Linux) run:

make install

The source consists of a lot of small files, so building goes a lot faster by using the multi-threaded build. This requires more than 2gigs on a Raspberry Pi (and breaks), so you might use a smaller number, like -j4 rather than all possible threads.

make -j

While Linux is the primary target platform, the code runs well on many other systems (Windows, macOS, etc.). Here's some additional build info:

* Windows w/ Visual Studio: use the VS10 project
* Windows w/ MinGW: just type make
* Windows w/ cygwin: won't work
* Mac OS X /w XCode: use the XCode4 project
* Mac OS X /w cmdline: just type make
* FreeBSD: type gmake
* other: try just compiling all the files together, cc src/\*.c -o bin/masscan

On macOS, the x86 binaries seem to work just as fast under ARM emulation.

**Usage**

Usage is similar to nmap. To scan a network segment for some ports:

# masscan -p80,8000-8100 10.0.0.0/8 2603:3001:2d00:da00::/112

This will:

* scan the 10.x.x.x subnet, and 2603:3001:2d00:da00::x subnets
* scans port 80 and the range 8000 to 8100, or 102 ports total, on both subnets
* print output to <stdout> that can be redirected to a file

To see the complete list of options, use the --echo feature. This dumps the current configuration and exits. This output can be used as input back into the program:

# masscan -p80,8000-8100 10.0.0.0/8 2603:3001:2d00:da00::/112 --echo > xxx.conf # masscan -c xxx.conf --rate 1000

**Banner checking**

Masscan can do more than just detect whether ports are open. It can also complete the TCP connection and interaction with the application at that port in order to grab simple "banner" information.

Masscan supports banner checking on the following protocols:

* FTP
* HTTP
* IMAP4
* memcached
* POP3
* SMTP
* SSH
* SSL
* SMBv1
* SMBv2
* Telnet
* RDP
* VNC

The problem with this is that masscan contains its own TCP/IP stack separate from the system you run it on. When the local system receives a SYN-ACK from the probed target, it responds with a RST packet that kills the connection before masscan can grab the banner.

The easiest way to prevent this is to assign masscan a separate IP address. This would look like one of the following examples:

# masscan 10.0.0.0/8 -p80 --banners --source-ip 192.168.1.200 # masscan 2a00:1450:4007:810::/112 -p80 --banners --source-ip 2603:3001:2d00:da00:91d7:b54:b498:859d

The address you choose has to be on the local subnet and not otherwise be used by another system. Masscan will warn you that you've made a mistake, but you might've messed up the other machine's communications for several minutes, so be careful.

In some cases, such as WiFi, this isn't possible. In those cases, you can firewall the port that masscan uses. This prevents the local TCP/IP stack from seeing the packet, but masscan still sees it since it bypasses the local stack. For Linux, this would look like:

# iptables -A INPUT -p tcp --dport 61000 -j DROP # masscan 10.0.0.0/8 -p80 --banners --source-port 61000

You probably want to pick ports that don't conflict with ports Linux might otherwise choose for source-ports. You can see the range Linux uses, and reconfigure that range, by looking in the file:

/proc/sys/net/ipv4/ip\_local\_port\_range

On the latest version of Kali Linux (2018-August), that range is 32768 to 60999, so you should choose ports either below 32768 or 61000 and above.

Setting an iptables rule only lasts until the next reboot. You need to lookup how to save the configuration depending upon your distro, such as using iptables-save and/or iptables-persistent.

On Mac OS X and BSD, there are similar steps. To find out the ranges to avoid, use a command like the following:

# sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last

On FreeBSD and older MacOS, use an ipfw command:

# sudo ipfw add 1 deny tcp from any to any 40000 in # masscan 10.0.0.0/8 -p80 --banners --source-port 40000

On newer MacOS and OpenBSD, use the pf packet-filter utility. Edit the file /etc/pf.conf to add a line like the following:

block in proto tcp from any to any port 40000

Then to enable the firewall, run the command:

# pfctrl -E

If the firewall is already running, then either reboot or reload the rules with the following command:

# pfctl -f /etc/pf.conf

Windows doesn't respond with RST packets, so neither of these techniques are necessary. However, masscan is still designed to work best using its own IP address, so you should run that way when possible, even when it is not strictly necessary.

The same thing is needed for other checks, such as the --heartbleed check, which is just a form of banner checking.

**How to scan the entire Internet**

While useful for smaller, internal networks, the program is really designed with the entire Internet in mind. It might look something like this:

# masscan 0.0.0.0/0 -p0-65535

Scanning the entire Internet is bad. For one thing, parts of the Internet react badly to being scanned. For another thing, some sites track scans and add you to a ban list, which will get you firewalled from useful parts of the Internet. Therefore, you want to exclude a lot of ranges. To blacklist or exclude ranges, you want to use the following syntax:

# masscan 0.0.0.0/0 -p0-65535 --excludefile exclude.txt

This just prints the results to the command-line. You probably want them saved to a file instead. Therefore, you want something like:

# masscan 0.0.0.0/0 -p0-65535 -oX scan.xml

This saves the results in an XML file, allowing you to easily dump the results in a database or something.

But, this only goes at the default rate of 100 packets/second, which will take forever to scan the Internet. You need to speed it up as so:

# masscan 0.0.0.0/0 -p0-65535 --max-rate 100000

This increases the rate to 100,000 packets/second, which will scan the entire Internet (minus excludes) in about 10 hours per port (or 655,360 hours if scanning all ports).

The thing to notice about this command-line is that these are all nmap compatible options. In addition, "invisible" options compatible with nmap are also set for you: -sS -Pn -n --randomize-hosts --send-eth. Likewise, the format of the XML file is inspired by nmap. There are, of course, a lot of differences, because the *asynchronous* nature of the program leads to a fundamentally different approach to the problem.

The above command-line is a bit cumbersome. Instead of putting everything on the command-line, it can be stored in a file instead. The above settings would look like this:

# My Scan rate = 100000.00 output-format = xml output-status = all output-filename = scan.xml ports = 0-65535 range = 0.0.0.0-255.255.255.255 excludefile = exclude.txt

To use this configuration file, use the -c:

# masscan -c myscan.conf

This also makes things easier when you repeat a scan.

By default, masscan first loads the configuration file /etc/masscan/masscan.conf. Any later configuration parameters override what's in this default configuration file. That's where I put my "excludefile" parameter so that I don't ever forget it. It just works automatically.

**Getting output**

By default, masscan produces fairly large text files, but it's easy to convert them into any other format. There are five supported output formats:

1. xml: Just use the parameter -oX <filename>. Or, use the parameters --output-format xml and --output-filename <filename>.
2. binary: This is the masscan builtin format. It produces much smaller files so that when I scan the Internet my disk doesn't fill up. They need to be parsed, though. The command-line option --readscan will read binary scan files. Using --readscan with the -oX option will produce an XML version of the results file.
3. grepable: This is an implementation of the Nmap -oG output that can be easily parsed by command-line tools. Just use the parameter -oG <filename>. Or, use the parameters --output-format grepable and --output-filename <filename>.
4. json: This saves the results in JSON format. Just use the parameter -oJ <filename>. Or, use the parameters --output-format json and --output-filename <filename>.
5. list: This is a simple list with one host and port pair per line. Just use the parameter -oL <filename>. Or, use the parameters --output-format list and --output-filename <filename>. The format is:

<port state> <protocol> <port number> <IP address> <POSIX timestamp> open tcp 80 XXX.XXX.XXX.XXX 1390380064

**Comparison with Nmap**

Where reasonable, every effort has been taken to make the program familiar to nmap users, even though it's fundamentally different. Masscan is tuned for wide range scanning of a lot of machines, whereas nmap is designed for intensive scanning of a single machine or a small range.

Two important differences are:

* no default ports to scan, you must specify -p <ports>
* target hosts are IP addresses or simple ranges, not DNS names, nor the funky subnet ranges nmap can use (like 10.0.0-255.0-255).

You can think of masscan as having the following settings permanently enabled:

* -sS: this does SYN scan only (currently, will change in the future)
* -Pn: doesn't ping hosts first, which is fundamental to the async operation
* -n: no DNS resolution happens
* --randomize-hosts: scan completely randomized, always, you can't change this
* --send-eth: sends using raw libpcap

If you want a list of additional nmap compatible settings, use the following command:

# masscan --nmap

**Transmit rate (IMPORTANT!!)**

This program spews out packets very fast. On Windows, or from VMs, it can do 300,000 packets/second. On Linux (no virtualization) it'll do 1.6 million packets-per-second. That's fast enough to melt most networks.

Note that it'll only melt your own network. It randomizes the target IP addresses so that it shouldn't overwhelm any distant network.

By default, the rate is set to 100 packets/second. To increase the rate to a million use something like --rate 1000000.

When scanning the IPv4 Internet, you'll be scanning lots of subnets, so even though there's a high rate of packets going out, each target subnet will receive a small rate of incoming packets.

However, with IPv6 scanning, you'll tend to focus on a single target subnet with billions of addresses. Thus, your default behavior will overwhelm the target network. Networks often crash under the load that masscan can generate.

**Design**

This section describes the major design issues of the program.

**Code Layout**

The file main.c contains the main() function, as you'd expect. It also contains the transmit\_thread() and receive\_thread() functions. These functions have been deliberately flattened and heavily commented so that you can read the design of the program simply by stepping line-by-line through each of these.

**Asynchronous**

This is an *asynchronous* design. In other words, it is to nmap what the nginx web-server is to Apache. It has separate transmit and receive threads that are largely independent from each other. It's the same sort of design found in scanrand, unicornscan, and ZMap.

Because it's asynchronous, it runs as fast as the underlying packet transmit allows.

**Randomization**

A key difference between Masscan and other scanners is the way it randomizes targets.

The fundamental principle is to have a single index variable that starts at zero and is incremented by one for every probe. In C code, this is expressed as:

for (i = 0; i < range; i++) { scan(i); }

We have to translate the index into an IP address. Let's say that you want to scan all "private" IP addresses. That would be the table of ranges like:

192.168.0.0/16 10.0.0.0/8 172.16.0.0/12

In this example, the first 64k indexes are appended to 192.168.x.x to form the target address. Then, the next 16-million are appended to 10.x.x.x. The remaining indexes in the range are applied to 172.16.x.x.

In this example, we only have three ranges. When scanning the entire Internet, we have in practice more than 100 ranges. That's because you have to blacklist or exclude a lot of sub-ranges. This chops up the desired range into hundreds of smaller ranges.

This leads to one of the slowest parts of the code. We transmit 10 million packets per second and have to convert an index variable to an IP address for each and every probe. We solve this by doing a "binary search" in a small amount of memory. At this packet rate, cache efficiencies start to dominate over algorithm efficiencies. There are a lot of more efficient techniques in theory, but they all require so much memory as to be slower in practice.

We call the function that translates from an index into an IP address the pick() function. In use, it looks like:

for (i = 0; i < range; i++) { ip = pick(addresses, i); scan(ip); }

Masscan supports not only IP address ranges, but also port ranges. This means we need to pick from the index variable both an IP address and a port. This is fairly straightforward:

range = ip\_count \* port\_count; for (i = 0; i < range; i++) { ip = pick(addresses, i / port\_count); port = pick(ports, i % port\_count); scan(ip, port); }

This leads to another expensive part of the code. The division/modulus instructions are around 90 clock cycles, or 30 nanoseconds, on x86 CPUs. When transmitting at a rate of 10 million packets/second, we have only 100 nanoseconds per packet. I see no way to optimize this any better. Luckily, though, two such operations can be executed simultaneously, so doing two of these, as shown above, is no more expensive than doing one.

There are actually some easy optimizations for the above performance problems, but they all rely upon i++, the fact that the index variable increases one by one through the scan. Actually, we need to randomize this variable. We need to randomize the order of IP addresses that we scan or we'll blast the heck out of target networks that aren't built for this level of speed. We need to spread our traffic evenly over the target.

The way we randomize is simply by encrypting the index variable. By definition, encryption is random and creates a 1-to-1 mapping between the original index variable and the output. This means that while we linearly go through the range, the output IP addresses are completely random. In code, this looks like:

range = ip\_count \* port\_count; for (i = 0; i < range; i++) { x = encrypt(i); ip = pick(addresses, x / port\_count); port = pick(ports, x % port\_count); scan(ip, port); }

This also has a major cost. Since the range is an unpredictable size instead of a nice even power of 2, we can't use cheap binary techniques like AND (&) and XOR (^). Instead, we have to use expensive operations like MODULUS (%). In my current benchmarks, it's taking 40 nanoseconds to encrypt the variable.

This architecture allows for lots of cool features. For example, it supports "shards". You can set up 5 machines each doing a fifth of the scan or range / shard\_count. Shards can be multiple machines, or simply multiple network adapters on the same machine, or even (if you want) multiple IP source addresses on the same network adapter.

Or, you can use a 'seed' or 'key' to the encryption function, so that you get a different order each time you scan, like x = encrypt(seed, i).

We can also pause the scan by exiting out of the program, and simply remembering the current value of i, and restart it later. I do that a lot during development. I see something going wrong with my Internet scan, so I hit to stop the scan, then restart it after I've fixed the bug.

Another feature is retransmits/retries. Packets sometimes get dropped on the Internet, so you can send two packets back-to-back. However, something that drops one packet may drop the immediately following packet. Therefore, you want to send the copy about 1 second apart. This is simple. We already have a 'rate' variable, which is the number of packets-per-second rate we are transmitting at, so the retransmit function is simply to use i + rate as the index. One of these days I'm going to do a study of the Internet, and differentiate "back-to-back", "1 second", "10 second", and "1 minute" retransmits this way in order to see if there is any difference in what gets dropped.

**C10 Scalability**

The asynchronous technique is known as a solution to the "c10k problem". Masscan is designed for the next level of scalability, the "C10M problem".

The C10M solution is to bypass the kernel. There are three primary kernel bypasses in Masscan:

* custom network driver
* user-mode TCP stack
* user-mode synchronization

Masscan can use the PF\_RING DNA driver. This driver DMAs packets directly from user-mode memory to the network driver with zero kernel involvement. That allows software, even with a slow CPU, to transmit packets at the maximum rate the hardware allows. If you put 8 10-gbps network cards in a computer, this means it could transmit at 100-million packets/second.

Masscan has its own built-in TCP stack for grabbing banners from TCP connections. This means it can easily support 10 million concurrent TCP connections, assuming of course that the computer has enough memory.

Masscan has no "mutex". Modern mutexes (aka. futexes) are mostly user-mode, but they have two problems. The first problem is that they cause cache-lines to bounce quickly back-and-forth between CPUs. The second is that when there is contention, they'll do a system call into the kernel, which kills performance. A mutex on the fast path of a program severely limits scalability. Instead, Masscan uses "rings" to synchronize things, such as when the user-mode TCP stack in the receive thread needs to transmit a packet without interfering with the transmit thread.

**Portability**

The code runs well on Linux, Windows, and Mac OS X. All the important bits are in standard C (C90). Therefore, it compiles on Visual Studio with Microsoft's compiler, the Clang/LLVM compiler on Mac OS X, and GCC on Linux.

Windows and Macs aren't tuned for packet transmit, and get only about 300,000 packets-per-second, whereas Linux can do 1,500,000 packets/second. That's probably faster than you want anyway.

**Safe code**

A bounty is offered for vulnerabilities, see the VULNINFO.md file for more information.

This project uses safe functions like strcpy\_s() instead of unsafe functions like strcpy().

This project has automated unit regression tests (make regress).

**Compatibility**

A lot of effort has gone into making the input/output look like nmap, which everyone who does port scans is (or should be) familiar with.

**IPv6 and IPv4 coexistence**

Masscan supports IPv6, but there is no special mode, both are supported at the same time. (There is no -6 option -- it's always available).

In any example you see of masscan usage, simply put an IPv6 address where you see an IPv4 address. You can include IPv4 and IPv6 addresses simultaneously in the same scan. Output includes the appropriate address at the same location, with no special marking.

Just remember that IPv6 address space is really big. You probably don't want to scan for big ranges, except maybe the first 64k addresses of a subnet that were assigned via DHCPv6.

Instead, you'll probably want to scan large lists of addresses stored in a file (--include-file filename.txt) that you got from other sources. Like everywhere else, this file can contain lists of both IPv4 and IPv6 addresses. The test file I use contains 8 million addresses. Files of that size need a couple extra seconds to be read on startup (masscan sorts the addresses and removes duplicates before scanning).

Remember that masscan contains its own network stack. Thus, the local machine you run masscan from does not need to be IPv6 enabled -- though the local network needs to be able to route IPv6 packets.

**PF\_RING**

To get beyond 2 million packets/second, you need an Intel 10-gbps Ethernet adapter and a special driver known as ["PF\_RING ZC" from ntop](http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/). Masscan doesn't need to be rebuilt in order to use PF\_RING. To use PF\_RING, you need to build the following components:

* libpfring.so (installed in /usr/lib/libpfring.so)
* pf\_ring.ko (their kernel driver)
* ixgbe.ko (their version of the Intel 10-gbps Ethernet driver)

You don't need to build their version of libpcap.so.

When Masscan detects that an adapter is named something like zc:enp1s0 instead of something like enp1s0, it'll automatically switch to PF\_RING ZC mode.

A more detail discussion can be found in **PoC||GTFO 0x15**.

**Regression testing**

The project contains a built-in unit test:

$ make test bin/masscan --selftest selftest: success!

This tests a lot of tricky bits of the code. You should do this after building.

**Performance testing**

To test performance, run something like the following to a throw-away address, to avoid overloading your local router:

$ bin/masscan 0.0.0.0/4 -p80 --rate 100000000 --router-mac 66-55-44-33-22-11

The bogus --router-mac keeps packets on the local network segments so that they won't go out to the Internet.

You can also test in "offline" mode, which is how fast the program runs without the transmit overhead:

$ bin/masscan 0.0.0.0/4 -p80 --rate 100000000 --offline

This second benchmark shows roughly how fast the program would run if it were using PF\_RING, which has near zero overhead.

By the way, the randomization algorithm makes heavy use of "integer arithmetic", a chronically slow operation on CPUs. Modern CPUs have doubled the speed at which they perform this calculation, making masscan much faster.

**Authors**

This tool created by Robert Graham: email: [robert\_david\_graham@yahoo.com](mailto:robert_david_graham@yahoo.com) twitter: @ErrataRob