How do those hacking tools work? The Art of Port Scanning

0x00sec.org/t/how-do-those-hacking-tools-work-the-art-of-port-scanning/619

19 June 2016

The Rules of Port Scanning

Despite of how sophisticated these scans may look, all of them follow some very basic rules and a few exception to those rules… There are always exceptions. So, the Rules of the Port Scanning are:

- **Rule 1**. A packet sent to a machine that does not exist will produce as response an `ICMP` packet indicating that situation
- **Rule 2**. A packet sent to a closed port will produce as response an `ICMP` packet indicating that (`UDP`) or a `RST` packet (`TCP`)
- **Rule 3**. A malformed/illegal packet will produce as response a `RST` packet
- **Rule 4**. A filtered port will produce as response an `ICMP` packet or no response at all (Usually the latest)

Yes, this is it. We will go through the main `nmap` scanning modes and we will check how all of them resolve to these rules (plus some exceptions to the rules :).

Note that, port scanning only makes sense on transport protocol… protocols with ports. So you can port scan a machine for `TCP` or `UDP` services… Therefore, to follow the rest of this paper we will have to take a look to the `TCP` protocol. In principle, it is not necessary to fiddle with `IP` , except for the `idle` scan. Discussing `TCP` scans will be long enough so I think we will leave `UDP` for later… or even better for you, dear reader, to research.

Furthermore, `nmap` provides quite some types of `TCP` scans. I'm not going to describe all of them, but I will try to provide you with the basic background to further explore those scans on your own.

A TCP Packet

The first thing you need to know, is that, the TCP protocol is described by RFC 793 (http://www.rfc-editor.org/rfc/rfc793.txt 31). Open the link and go to section 3.1 to see how a `TCP` packet looks like… OK, you lazy reader, here you go:

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           |U|A|P|R|S|F|                               |
   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
   |       |           |G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             data                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This is it. Most of the scanning techniques will fiddle with the flags in the fourth row: `URG`, `ACK`, `PSH`, `RST`, `SYN` and `FIN`.

If you already know the basics about `TCP` internals, you can move on, otherwise, here are a couple of concepts that you need to know.

Three-way handshake

This is how the `TCP` connection process is usually named (check section 3.4 in the RFC). It requires the transmission of three packets (that's the three-way in the name…). So, when you try to connect to a `TCP` server (and this is what the port scanner will be doing all the time), your program will call the `connect` syscall, and that will fire this process inside the kernel… actually in the TCP/IP stack implementation of the kernel.

1. The machine that starts the connection will send a `SYN` packet. This is a packet with the `SYN` flag (find it in the 4th row in the figure above) set to 1, a random sequence number and proper values for the port fields…
2. The target machine, supposing that there is a program that had run the `listen` and `accept` system calls, will receive the packet and send a `SYN/ACK` packet back (this is done by the kernel). This packet has the flags `SYN` and `ACK` set to 1, a new random sequence number and the sequence number received in the `SYN` packet copied in the field `ACK` number and increased by 1.
3. To complete the process the machine that started the connection has to send another `ACK` packet to the remote machine with updated sequence numbers.

Let's duplicate here the nice diagram with real numbers from the RFC

TCP A   TCP B

1. CLOSED   LISTEN
2. SYN-SENT   --> <SEQ=100><CTL=SYN>   --> SYN-RECEIVED
3. ESTABLISHED <-- <SEQ=300><ACK=101><CTL=SYN,ACK> <-- SYN-RECEIVED
4. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK>   --> ESTABLISHED
5. ESTABLISHED --> <SEQ=101><ACK=301><CTL=ACK><DATA> --> ESTABLISHED

Talking, Hanging up and Responding to the Unexpected

Once the connection is established, whenever the programs in both ends execute a `write` / `send` or `read` / `recv` system call, the TCP/IP stack in the kernel will put your data in the `Data block` (the last row in our `TCP` packet diagram), and update the sequence numbers accordingly.

The packet will be stored in the kernel (after sending it out), until the other end sends an `ACK` packet with an acknowledge number higher that the packet sequence number. In other words, the packet will be stored until the other ends indicates that it has received it. Well, this, together with other features (sliding windows algorithm, congestion control, etc…) is what give to TCP its 'reliability' feature.

Connection are finished using the `FIN` flag and here, again there are quite some different cases. You can read the RFC for further details, for our current discussion we just need to know that the `FIN` flag is used to close a connection.

This is how it works in the nominal case. However, if a strange or unexpected packet is received, the TCP/IP stack will drop that packet and, in general, send back a `RST` (reset) packet to let the other end know that its packet was discarded.

This is roughly it. Let's go now into the scanning process.

Understanding ICMP

The other protocol involved in the port scanning process is ICMP, the *Internet Control Message Protocol*. This is a network protocol that sits besides IP intended to interchange control messages between machines. It is specified in RFC 792 (https://tools.ietf.org/html/rfc792 [5]). The format is shown below:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |     Type      |     Code      |          Checksum             |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                             unused                            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |      Internet Header + 64 bits of Original Data Datagram      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ICMP can send messages of a specified `Type` and, for each type of message, you can also specify a code. There are many different types of messages, but the ones we are interested on are the `Type` 3: *Destination Unreachable*.

This message can contain the following codes:

```
0 = net unreachable;
1 = host unreachable;
2 = protocol unreachable;
3 = port unreachable;
4 = fragmentation needed and DF set;
5 = source route failed.
```

You can see many interesting codes in this list. These messages are generated at different points in the network. Some examples

- If you computer does not know how to send data to a given host or net (your network configuration is broken or your network is isolated), it will produce the ICMP packet.
- If you host believe it can send the packet, but your router/gateway finds out it cannot, it may also generate a ICMP message for you. Our packet have not even leave our local network
- If our make make it up to the remote network, but then the remote router/gateway/firewall determines that the machine does not exist (or is not up) it will produce the ICMP packet
- If our packet make it up to the final host through the whole Internet, but the **UDP** port we are trying to reach does not have any program listening, the remote host will generate the ICMP packet. (For a TCP we will get a `RST` packet instead)

So, as you can see, the ICMP protocol really lives at the network level. Those messages can be generated at any point in the network. From the point of view of the port scanner, it does not really matter who has generated the message, it just needs to capture it and properly interpret it.

Now that we know the low level details, let's see what happens every time you run `nmap`.

TCP Connect Scanning

The TCP connect scanning is the simplest one. The scanner just tries to establish a normal connection, calling `connect`. If `connect` succeeds (the three-way handshake is completed) the port is open, otherwise the port is closed. The advantage of this scanning technique is that you do not need root privileges to perform it. We are trying to establish normal connection and that does not require special privileges.

The downside is that it is very noisy. Any decent firewall out there will log the connection attempt and, if many connections attempts happens in a short period of time, it will also know that the machine is being scanned.

TCP SYN Scan

If we look to the three-way handshake process, we will notice that it is possible to know if the port is open before the handshake is completed. We just need to send a `SYN` packet (that gives the name to this scan) and check what we get back:

- If we get back a normal `SYN/ACK` packet, then the port is open and we are done
- If we get back a `RST` packet, that means the port is closed (Rule 2)
- If no response or a `ICMP` unreachable (type 3) port is received then the port is filtered. A firewall is dropping our `SYN` packet and we are not getting any response back at all (Rule 1, Rule 4)

This scan is also easily detected but usually not by default. On top of that, this is a lot faster that a connect scan as only one packet has to be sent.

This scan does not help when the target is behind a firewall. Usually firewalls block any connection attempt (`SYN` packets) from outside (unless it is against one of the publicly provided services), that means that a scan against a machine behind a firewall will result in a quite slow scan reporting all ports as filtered. This means that no reply whatsoever was received for the sent packet and our scanner time-out expired for each connection try.

NULL, FIN and Xmas Scans

All these scans are basically the same, the only difference between them is the set of TCP flags in the packet being sent by the scanner. A NULL scan, sets all flags to 0, a FIN scan just sets the `FIN` flag, and a Xmas (tree) scan set all the TCP flags. `nmap` actually allows you to provide your own flag pattern if you want to.

All those scans work based on a TCP behaviour described in the RFC. According to the RFC, if a packet reaches a closed port and the packet does not contain a `RST`, a `RST` response has to be sent back. In the same section, it is also explained that any packet without the `SYN`, `RST` or `ACK` flags set, send to an open port will force the machine to drop the packet.

So, these scans are intended to differentiate between open and close ports.

This is a very stealthy scan (no connection attempt) but, as usual, modern IDS/IPS can be configured to detect them. The main drawback is that some systems are not compliant with the TCP RFC (RFC 793) and therefore, the scan does not work (the `RST` packet is not sent back).

These scans are useful to get information on services behind some firewall or packet filter in the target network. This scan can be used to determine which ones of the filtered ports (reported by, for instance, a SYN scan) are actually closed and which ones are open.

ACK Scan

The ACK scan is the latest we are going to describe. This one is special as it is not intended to find open ports but to map firewall rules.

Firewalls are quite complex nowadays and they can work at different levels. A traditional classification of firewalls is stateful and non-stateful. A stateful firewall keeps track of the connections taking place through it. When a connection is established, the firewall note that down and, from that point on, it will drop any packet that does not belong to one of those connections.

This means that, if we send an `ACK` packet to a port behind a stateful firewall, the packet will be dropped or an `ICMP` packet type 3 will be generated, because that packet does not belong to any on-going connection. For non-stateful firewalls, the packet will pass through and the remote machine will respond with an `RST` as the `ACK` packet will not contain a valid sequence number and it will be identified as not belonging to an existing connection.

So, an ACK scan is intended to figure out if the firewall protecting the victim machine is behind a stateful firewall or not. Well, it can be used for that.

Other Scans

`nmap` provides other scan types, namely TCP Window scan, Maimon scan, Idle scan, ... You can check the man page for `nmap` to check the complete list and a very comprehensive explanation of how they work. I hope that with our discussion so far you have now the tools to figure out how those scans works. Otherwise, do not hesitate to ask in the comments.

By now, you may know that the scanning process is not that straight forward. It is not just about running a program that will tell you which ports are open, sometimes it becomes more sophisticated and you have to do some thinking on what is going on.

To finish with this paper, I will just describe the general programming elements you may need to build your own *Port Scanner*. As I said at the beginning if you have a chance to use `nmap` it is not wise to use your own tool, but, you never know when you may need to write some small tool for a very specific case for which `nmap` is not an option… (maybe running a scan from a small router??)

Programming a Scanner

Programming a scanner is a very interesting topic because it implies hooking deep into the TCP/IP stack and learn a lot on how the stack really works. Basically you need to do two things:

1. Craft special packets. You cannot generate this packets using a normal socket
2. Capture the returned traffic to determine the outcome of the sent packet.

Both tasks requires `root` privileges, because both tasks requires the use of sockets RAW (`man 7 raw`). Actually, it is a nice exercise to write your port scanner using sockets RAW directly as that is the lower level you can reach at user space… Then, only the kernel is left. However, tools like `nmap` does not do that, they use well-known libraries as `libpcap` for capturing packets and `libdnet` for crafting packets.

Are you curious why those libraries are used?. Grab the code and take a look. You will see a lot of conditional blocks for different system. Each operating systems, when working at this level, has its own peculiarities and it is quite a work to write code that works on all of them. When you target a specific platform you can write shorter code for it but such a code will not be the most portable in the world. On top of that, a lot of very smart people have worked on these libraries for years and, let's face it, you think you can do better?..

The `libdnet` library provides a convenient interface to build any kind of packet. You can *easily* build any of the packets we had discussed in previous sections, setting flags, ports and sequence numbers.

The `libpcap` library provides a convenient interface to capture packets. The scanner will have to capture TCP/UDP packets as response from the remote system (`RST` packets mainly) but it will also have to capture the `ICMP` messages that will come as response to those packets and that contains valuable information about the state of the port.