A Taste of Scapy

Judy Novak

A Taste of Scapy

By Judy Novak

Introduction

Scapy, a Python packet crafting tool, has become my new BFF. Have you ever envisioned that there may be an easy way to craft a TCP session beginning with the TCP three-way handshake so that you can emulate a client side of a TCP connection?

I learned early on to avoid using any programming language that requires strictly typed variables, since I seemed most adept at dumping core and generating segment faults rather than creating any useable code. Yet, until Scapy arrived, crafting something as sophisticated as a TCP session was possible only using less forgiving languages such as C. With a little knowledge and a handful of short lines of code, Scapy is able to easily craft either the client or server side of a TCP session.

Having worked for five years at Sourcefire, the commercial company associated with Snort, my interest then and now has been emulating the client side of a TCP connection to examine a particular destination host operating system's response to some "unique" stimulus. One of the features that Snort has that no other product I've tested has is something known as target-based knowledge. Simply stated, this means that Snort can be configured with a specific TCP stream reassembly policy that is most appropriate for a given destination host's or CIDR block's operating system. Different operating systems may react uniquely to a given ambiguous or ill-defined aspect of behavior. When configured properly, this allows Snort to reassemble a TCP session identically as the destination host operating system, thus avoiding TCP evasions.

Take for instance the unusual behavior from a current Linux operating system running kernel 2.4 and higher. Most operating systems adhere to the specifications of RFC 793 "Transmission Control Protocol" that offers guidance for TCP implementations. The RFC specifies that all TCP segments after the initial client SYN should have the acknowledgement flag set. However, current versions of Linux do not require this and actually accept and acknowledge a segment in an established session where no TCP flags are set and where the segment has payload. No other well-known current operating system accepts this same segment.

Why is this useful knowledge? This offers valuable reconnaissance if you are attempting to fingerprint a remote operating system to discover if it is running Linux. There is a tool called p0f that performs operating system fingerprinting, but it examines and makes its determination from field values in the IP and TCP headers of a SYN segment originating from the host. For example, it uses the TCP window size value to aid in its assessment and matches it against expected TCP window sizes. If a savvy administrator is aware of this and wants to thwart accurate p0f fingerprinting, she need only alter the TCP window size. This may have some unintended adverse consequences, but there are other lesser impact fields that may be altered to evade detection. As well, some applications such as web server software advise changing the TCP window size to maximize efficiency. This unintentionally foils p0f identification.

Yet a characteristic such as Linux acknowledging a segment with no TCP flags sent in the middle of a TCP session is not easily changed unless you alter the source code. Therefore, using this behavior as an operating system identification method is practically foolproof. How would you implement such an application? With a little background knowledge, this is fairly simple to achieve using Scapy.
Let's recap the requirements for the session we would need to code this:

- ♦ We need to create the three-way handshake
- ♦ We need to be able to control the TCP flag value supplied in the TCP header
- ♦ We need to assign the data payload

Oh, did I mention that creating the three-way handshake requires the crafter to listen for the server's SYN/ACK response to extract its TCP sequence number, increment the value by one, and place the new value in the acknowledgement number field? This is a tall order; but Scapy provides the tool to make all these requirements very manageable.

Crafting the Three-way Handshake

Crafting a run-of-the-mill type of packet where we do not care about the response is actually fairly trivial and is easily accomplished using any of a number of command line tools such as hping3, sendip, and nemesis, to name a few. However, these tools are inappropriate if you need to listen for a response and extract a particular field from the response as required by the client to acknowledge the server's Initial Sequence Number (ISN). Scapy has a command that allows you to craft a SYN request and match the corresponding returned SYN/ACK segment. The SYN/ACK sequence number is easily extracted and incremented for use in the client's acknowledgement value. First, let's examine what we need to do using pseudo-code:

- Send the client's SYN to a listening server
  - o Craft an IP header containing the source and destination IP addresses
  - o Craft a TCP header where we generate the TCP source port, assign the destination port that the server listens on, set the TCP flags to turn the SYN bit on, and generate the client's ISN

- Listen for the server's response
  - o Save the server's response
  - o Extract the server's TCP sequence number and increment the value by one

- Craft the client's acknowledgement of the server's response
  - o Craft an IP header containing the same source and destination IP addresses on the SYN
  - o Craft a TCP header where with the same SYN segment TCP source and destination ports, set the TCP flags to turn the ACK bit on, increment the client's ISN by one since the SYN consumes one sequence number, set the acknowledgement value to the incremented server's sequence number value

Let's suppose this is an abbreviated tcpdump-like display of the three-way handshake:

```
192.168.1.103 1024 > 192.168.1.104 80   flags=SYN seq=12345
192.168.1.104 80      > 192.168.1.103 1024 flags=SYN, ACK seq=9998 ack=12346
192.168.1.103 1024 > 192.168.1.104 80   flags=ACK seq=12346 ack=9999
```

Assuming that there are default values for other IP and TCP headers, we need to accomplish the following:

- Send the client's SYN to a listening server
  - o Craft an IP header containing the source IP 192.168.1.103 and destination IP 192.168.1.104

  - Craft a TCP header where we generate the TCP source port 1024, assign the destination port 80, set the TCP flags to turn the SYN bit on, and generate the client's ISN of 12345

- Listen for the server's response
  - Save the server's response
  - Extract the server's TCP sequence number 9998, and add 1 to get 9999

- Craft the client's acknowledgement of the server's response
  - Craft an IP header containing the source IP 192.168.1.103 and destination IP 192.168.1.104
  - Craft a TCP header where with the same TCP source port 1024, assign the destination port 80, set the TCP flags to turn the ACK bit on, increment the client's ISN by one to 12346, set the acknowledgement value to the incremented server's sequence number value – 9999

Ultimately, we would create a Python program to accomplish this because we do not want to perform this interactively. But, we'll use Scapy's command line interface to demonstrate the code. You need to be root or super user to craft Scapy packets because they are sent directly to the network card driver using the PF_PACKET socket family protocol on our Linux host.

```
user@desktop: sudo –s
user@desktop: scapy

Welcome to Scapy (2.0.1)

>>>ip=IP(src="192.168.1.103", dst="192.168.1.104")
>>>SYN=TCP(sport=1024, dport=80, flags="S", seq=12345)
>>>packet=ip/SYN
```

Let's review what the above code does. First, we create an instance of an IP header called *ip*. Scapy is case-sensitive so *ip* is different than *IP*. We define an IP header using Scapy's *IP* and supply all the fields and values between the left and right parentheses. We assign only the source IP address *src="192.168.1.103"* and destination IP address using *dst="192.168.1.104"*. Scapy uses default values if you don't define a given field. You can define any of the fields that are in the IP header; you can discover what these are by executing the Scapy command **ls(IP)**.

Now we define an instance of the TCP header called *SYN*. We use the same format of referencing Scapy's TCP header *TCP* and defining all the fields we want to change – the source and destination port, the flags, and the sequence number. Again, if you wanted to examine the field names in the TCP header, you could issue the command **ls(TCP)**. Finally, we assemble an instance of a packet appropriately called *packet* that is our IP layer *ip* layered with (indicated using the forward slash) our instance of the TCP layer *SYN*.

We want to send this and capture the server's response so that we can extract the server's TCP sequence number and acknowledge it.

```
>>>SYNACK=sr1(packet)
>>>my_ack=SYNACK.seq + 1
```

The previous block of code sends one packet and matches the first response using Scapy's Layer3 command **sr1**. Specifically, we send the packet we crafted and store the response from the server in an instance of the packet called *SYNACK*. Next, we use the notation *SYNACK.seq* to extract the TCP sequence number from the server and increment it by 1 and store the value in the variable *my_ack*.

```
>>>ACK= TCP(sport=1024, dport=80, flags="A", seq=12346, ack=my_ack)
>>>send(ip/ACK)
```

Above, we create a new instance of the TCP header and call it *ACK*. It is very similar to the initial *SYN* header we crafted for the SYN, but we change the flags field to have an acknowledgement *flags="A"*, we increment the sequence number by 1 to 12346 since the SYN consumes a sequence number, and finally we place the acknowledgement value for the server's sequence number in *ack=my_ack*. We send it using the **send** command – a Layer 3 send that does not listen for a response. If we have done everything correctly, we have just created the three-way handshake. All we have to do now is create the segment with no TCP flags and payload and send it.

```
>>>PUSH=( sport=1024, dport=80, flags="", seq=12346, ack=my_ack)
>>>data="SEND THIS!"
>>>send(ip/PUSH/data)
```

We create a new instance of a TCP header called *PUSH* and set the TCP flags field with no flags *flags=""*. All other field values remain the same. We assign some data payload and then send the new packet layering our IP header *ip*, TCP header *PUSH*, and payload *data*. We should see a TCP acknowledgement of this segment if the destination host is a Linux server.

Complications

There is an impediment to crafting TCP sessions via Scapy because it circumvents the native TCP/IP stack. What this means is that the host is unaware that Scapy is sending packets. This has an unpleasant side effect because the native host will be confused when the server responds with the SYN/ACK. As far as the native host's TCP/IP stack is concerned, it never sent a SYN and does not expect a SYN/ACK in return. It's as if the native host just received a rogue unsolicited SYN/ACK that is not associated with any open session/socket it knows about. Therefore, the host resets the connection when it receives the SYN/ACK. And, that isn't what we want at all. That's "game over" right then and there.

The resolution for this is to use the host's firewall, such as iptables, to block the outbound resets. For the above session we'd issue the following on the command line (outside of Scapy):

```
root@desktop: iptables –A OUTPUT –p tcp –d 192.168.1.104 –s 192.168.1.103 - -dport 80 - -tcp-flags RST
RST –j DROP
```

This drops all outbound packets that are TCP and destined for IP address 192.168.1.104 from source IP 192.168.1.103 to destination port 80 where the flags field reset bit should be examined, and if it is set, drop the packet. This doesn't prevent the originating host from generating a reset each time it receives a packet from this session, but it blocks it from leaving the host. This "silences" the reset and you and Scapy are able to craft the rest of the session.

Conclusion

While this was a whirlwind introduction to Scapy, it is obvious how useful it can be once you understand its benefits and side effects. It takes a little time and some hands-on experience to become comfortable with Scapy. But it's an amazingly powerful tool once you do.

If you're interested in learning more about Scapy, SANS offers a new one-day course, SEC567 "Power Packet Crafting Using Scapy".

http://www.sans.org/security-training/power-packet-crafting-with-scapy-1382-mid

This course is jam-packed with hands-on exercises where you learn to craft this session and many others.

Last Updated: April 18th, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**