Manually create and send raw TCP/IP packets

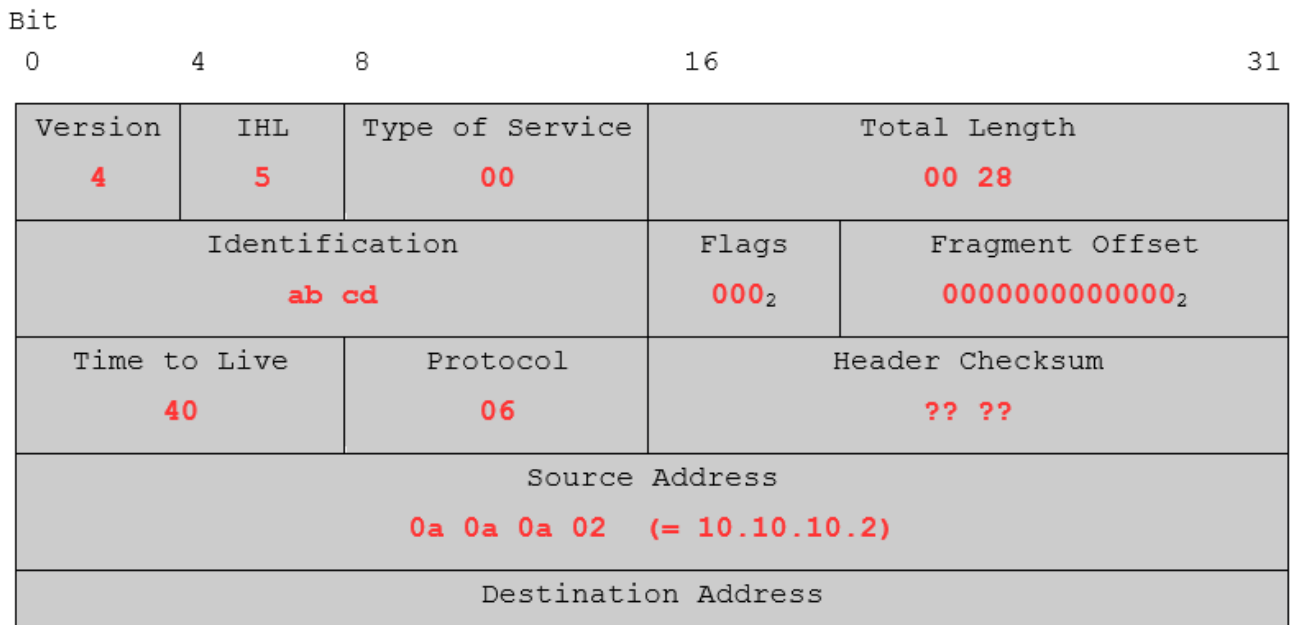
In this chapter we are going to to use our knowledge on packets to manually craft and put them on the wire.

The blueprint

We now want to craft a packet and send it through the network. Let's start by identifying the properties our packet should have:

- IPv4 packet
- TCP SYN packet
- from our client (10.10.10.2) to the server (10.10.10.1)
- from port 12345 to port 80
- fill up the other headers with the necessary data

We will use our packet blueprints from the previous part of this series to aid us in creating the packet:



0a 0a 0a 01 (= 10.10.10.1)										
Source Port 30 39 (= 12345_{10})										Destination Port 00 50 (= 80_{10})
Sequence Number 00 00 00 00										
Acknowledgement Number 00 00 00 00										
Data Offset 0101_2	 000_2	NS 0_2	CWR 0_2	ECE 0_2	URG 0_2	ACK 0_2	PSH 0_2	RST 0_2	SYN 1_2	FIN 0_2	Window Size 71 10
Checksum ?? ??										Urgent Pointer 00 00

Based on our knowledge from the previous parts of this series we filled our blueprint with the relevant values (Identification and Window Size are just random in this case). As a small exercise, try to understand the meaning behind the values in Total Length and Data Offset.

You might have seen, that two values are missing for now, the Header Checksum for the IP segment and the Checksum from the TCP segment.

TCP Checksum & IP Header Checksum

TCP Checksum

Let's start with the TCP checksum. Remembering from the first part of this series we know, that the checksum consists of values of the TCP Header itself, as well as a pseudo-header. For the calculations, all necessary values are used in 16 bit words and added together as shown below. In case the value isn't 16 bit long, it will be prepended with zeros.

Description	**Value**	**Additional Description**
?Protocol	0x0006 +	06
Source Address (IP)	0x0a0a + 0x0a02 +	10.10.10.2
Destination Address (IP)	0x0a0a + 0x0a01 +	10.10.10.1
TCP length (including the data part) in byte (no actual header field, has to be counted!)	0x0014 +	20 bytes (= 14 in hex)
Source + Destination Port	0x3039 + 0x0050 +	1234 and 80
Sequence Number	0x0000 + 0x0000 +	00 00 00 00
Acknowledgement Number	0x0000 + 0x0000 +	00 00 00 00
Data Offset, Reserved, Flags, Window Size	0x5002 + 0x7110 +	0101 000 000000010 and 71 10
Checksum (set to 0x0000 in calculation), Urgent Pointer	0x0000 + 0x0000 =	
Subtotal	**0x119cc**	
Removing the carryover	0x19cc + 0x0001 = 0x19cd	
Negation with 0xffff	0xffff – 0x19cd =	
Checksum	**0xe632**	

IP Header Checksum

The IP header checksum is easy to calculate. It consists out of all values in the IP header, again added in 16 bit words and prepended with zeros in case the value is

too short:

Description	**Value**	**Additional Description**
Version, IHL, Type of Service + Total Length	0x4500 + 0x0028 +	–
Identification + Flags, Fragment Offset	0xabcd + 0x0000 +	–
TTL, Protocol + Header Checksum (0x0000 in calculation)	0x4006 + 0x0000 +	–
Source Address (IP)	0x0a0a + 0x0a02 +	10.10.10.2
Destination Address (IP)	0x0a0a + 0x0a01 =	10.10.10.1
Subtotal	**0x15912**	
Removing the carryover	0x5912 + 0x0001 = 0x5913	
Negation with 0xffff	0xffff – 0x5913 =	
Header Checksum	**0xa6ec**	

As you could see, both checksums follow the same algorithm, just their input values are different.

Now let's put the calculated checksums in our blueprint. For better readability, they are arranged like this:

45	00	00	28
ab	cd	00	00
40	06	a6	ec
0a	0a	0a	02

0a	0a	0a	01
30	39	00	50
00	00	00	00
00	00	00	00
50	02	71	10
e6	32	00	00

Sending a self crafted packet

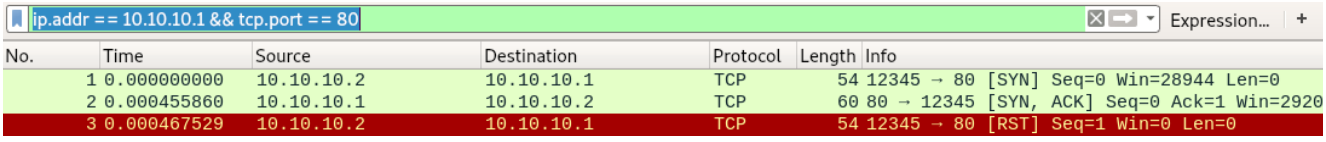
After we have now manually created our first TCP/IP packet, let's put it on the wire. As this is a `[SYN]` packet we are sending to our webserver, we expect him to respond with a `[SYN, ACK]` if everything works as planned. For this example we are going to use Python's built-in socket module:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_TCP)
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

ip_header  = b'\x45\x00\x00\x28'  # Version, IHL, Type of Service | Tota
ip_header += b'\xab\xcd\x00\x00'  # Identification | Flags, Fragment Off
ip_header += b'\x40\x06\xa6\xec'  # TTL, Protocol | Header Checksum
ip_header += b'\x0a\x0a\x0a\x02'  # Source Address
ip_header += b'\x0a\x0a\x0a\x01'  # Destination Address

tcp_header  = b'\x30\x39\x00\x50' # Source Port | Destination Port
tcp_header += b'\x00\x00\x00\x00' # Sequence Number
tcp_header += b'\x00\x00\x00\x00' # Acknowledgement Number
tcp_header += b'\x50\x02\x71\x10' # Data Offset, Reserved, Flags | Windc
tcp_header += b'\xe6\x32\x00\x00' # Checksum | Urgent Pointer

packet = ip_header + tcp_header
s.sendto(packet, ('10.10.10.1', 0))
```

We are creating a socket s out of the Internet Protocol family `AF_INET`, in "raw" mode `SOCK_RAW` which will be sending TCP packets `IPPROTO_TCP`. With the `setsockopt()` we tell the kernel not to generate an IP header, since we are providing it ourselves. For further details on the Python socket module, I recommend the Python documentation on sockets. When working with raw sockets in scripts, most operating system require advanced privileges (e.g. root user) to run them:

```
root@kali:~# python3 send_first_packet.py
```

Utilizing Wireshark, we observe what happens when we send the packet:



As expected, our Python script sends a `[SYN]` packet to port 80 of our webserver. This server replies with a `[SYN, ACK]`, the second step of a typical TCP three-way handshake. The third packet however is a `[RST]` reset sent from our client to the server. This happened because of the value we set as source port of our packet. Despite providing 12345 as source port, there is no application on our side listening on that port to accept the incoming `[SYN, ACK]`. Therefore, a reset packet is sent, and the connection establishment is canceled.

In case you don't get the upper result, check whether your calculated checksums are correct. To verify them in Wireshark, go: Right click (on any packet) > Protocol Preferences > "Validate the TCP checksum if possible".

Improving our crafted packet

Having again a closer look at the first packet we sent, we will see that there are 14 more bytes in front of our IP header (highlighted in blue).

```
0000  00 0c 29 d3 be d6 00 0c  29 e0 c4 af 08 00 45 00   ..).....  ).....E.
0010  00 28 ab cd 00 00 40 06  a6 ec 0a 0a 0a 02 0a 0a   .(....@.  ........
0020  0a 01 30 39 00 50 00 00  00 00 00 00 00 00 50 02   ..09.P..  ......P.
0030  71 10 e6 32 00 00                                 q..2..
```

These bytes are the ethernet layer, the layer below the internet and transport layer:

Destination MAC Address	**Source MAC Address**	**Protocol Type**
00 0c 29 d3 be d6	00 0c 29 e0 c4 af	08 00? (= IPv4)

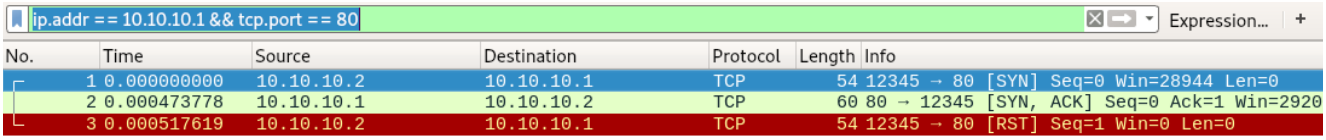
Since we want to create the complete packet by hand, we need to slightly modify our Python script by manually adding the ethernet layer:

```
import socket

s = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
s.bind(("eth0", 0))

ethernet  = b'\x00\x0c\x29\xd3\xbe\xd6' # MAC Address Destination
ethernet += b'\x00\x0c\x29\xe0\xc4\xaf' # MAC Address Source
ethernet += b'\x08\x00'                 # Protocol-Type: IPv4

ip_header  = b'\x45\x00\x00\x28'  # Version, IHL, Type of Service | Tota
ip_header += b'\xab\xcd\x00\x00'  # Identification | Flags, Fragment Off
ip_header += b'\x40\x06\xa6\xec'  # TTL, Protocol | Header Checksum
ip_header += b'\x0a\x0a\x0a\x02'  # Source Address
ip_header += b'\x0a\x0a\x0a\x01'  # Destination Address

tcp_header  = b'\x30\x39\x00\x50' # Source Port | Destination Port
tcp_header += b'\x00\x00\x00\x00' # Sequence Number
tcp_header += b'\x00\x00\x00\x00' # Acknowledgement Number
tcp_header += b'\x50\x02\x71\x10' # Data Offset, Reserved, Flags | Windc
tcp_header += b'\xe6\x32\x00\x00' # Checksum | Urgent Pointer

packet = ethernet + ip_header + tcp_header
s.send(packet)
```

We are again creating a “raw” socket, but this time from the address family “packet”, allowing us to play on a very low protocol level. Then we bind the socket to our network interface `eth0` (might be a different one for you; check that e.g. with “ifconfig” command).

```
root@kali:~# python3 send_first_packet_v2.py
```

After executing the new script we observe the output of Wireshark again:



Everything worked as expected, we have the start of the three-way handshake followed by the reset.

Now we have learned how we can manually create any kind TCP/IP packet we want to. This knowledge has a very large application range, e.g. starting from low level programming to pentesting and fuzzing. An example of what you can do with this will be shown in the next part of the tutorial series where we create a stealth port scanner.

TCP/IP packets

Contact

Twitter: @inc0x0