EE379K Enterprise Network Security Lab 1 Report

Student: Sean Wang, szw87
Professor: Mohit Tiwari, Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

September 13, 2019

Part 1 - Networking and Denial of Service

Step 1 - Client and Server in C

For step 1, the client and server in C were implemented to closely match the Python versions. For simplicity, the client sends the same hard coded message each time, similar to the Python client. The C client and server were tested with the Python client and server to ensure cross-functionality and that the client implementation in both languages worked almost identically. The only difference between the C client and Python client was the output of the Python client showing

```
From Server: b'INPUT LOWERCASE SENTENCE:'
```

and the C client showing

```
From Server: INPUT LOWERCASE SENTENCE:
```

To build the server, compile it with:

```
$ gcc -o server server.c
```

Similarly for the client, compile it with:

```
$ gcc -o client client.c
```

To run them, simply execute either:

```
$ ./server
$ ./client
```

Step 2 - DOS Attack

For step 2, the DOS attack was implemented using a command line tool called `hping3` using the options

```
$ sudo hping3
    --count 15000 \          # number of packets
    --destport 12000 \       # destination port
    --rand-source \          # randomize source IP
    --flood \                # send as fast as possible
    --syn \                  # send SYN packets
    127.0.0.2                # destination IP
```

These flags specify to stop sending packets to `127.0.0.2:12000` after sending/receiving 15000 SYN packets, using randomized IP addresses to disguise the actual source and prevent the server's SYN-ACK packets from reaching the actual source. Additionally, the `--flood` option just says to send packets as fast as possible.

As a result, the server receives many requests for establishing a connection, but because the SYN-ACK sent from the server never reaches the actual sender of the initial SYN packet, the 3-way handshake is never completed and the server is left waiting on a response from what it sees as many clients. This can be seen in Figure 1, with some packet details cut out to ensure legibility.

No.	Time	Source	Destination	Protoco	Length	Info
5	0.000339	229.220.168.249	127.0.0.2	TCP	56	2945 → 12000 [SYN] Seq=0 Win=512 Len=0
6	0.000364	127.0.0.2	229.220.168.249	TCP	60	12000 → 2945 [SYN, ACK] Seq=0 Ack=1 Win=65495 Len=0 MSS=65495

Figure 1: An incomplete three-way handshake

Since the server is now swamped with connection requests, the real client (at IP `127.0.0.1`) cannot have its connection request processed by the server and times out, as shown in Figure 2, also with some packet details cut out to ensure legibility.

No.	Time	Source	Destination	Protoco	Length	Info
171837	1.475633	127.0.0.1	127.0.0.2	TCP	76	37564 → 12000 [SYN] Seq=0 Win=65495 Len=0 MSS=
334856	2.490957	127.0.0.1	127.0.0.2	TCP	76	[TCP Retransmission] 37564 → 12000 [SYN] Seq=0
531548	4.511392	127.0.0.1	127.0.0.2	TCP	76	[TCP Retransmission] 37564 → 12000 [SYN] Seq=0

Figure 2: Client's sent SYN packet and client timeout

The rest of the tcpdump record of the DOS attack is in `output.pcap` and shows the flood of SYN packets sent to the server at IP `127.0.0.2:12000`.

Part 2 - zmap scan

For this part, a zmap scan was run for about 2 hours and 47 minutes (10000 seconds) with the following options:

```
$ sudo zmap \
    --blacklist-file=blacklist.conf \ # subnets to exclude
    --target-port=80 \                # port number to scan
    --output-file=zmap_result.txt \   # output file
    --max-runtime=10000               # max runtime (seconds)
```

The results of the zmap scan were:

```
1261303 machines probed
1534 machines responded
0.12% hitrate
```

Further information on each IP in the result list can be found using the `whois` command, such as the network the IP belongs in. This is specified by the Classless Inter-Domain Routing (CIDR) info. In order to group all the IPs that belong in the same network, a Python and Bash script (`network_whois.py` and `whois_grep.sh`) were used to systematically run `whois` on each IP and capture the CIDR or inetnum to get the range of IPs that belong in the same network. Then, using another Python script, `analyze.py`, the IPs and other network information were grouped accordingly into the following JSON files:

```
- large_nets.json: list of networks that have multiple subnets
      (represented by multiple CIDRs)
- net_ips.json: dictionary mapping a network's CIDR to a list
      of probed IPs in the network
- net_sizes.json: dictionary mapping a network's CIDR to the
      number of probed IPs in the network
- subnets.json: list of networks and their subnets
```

Some notable observations were that there were some networks with a large range of IPs, so the `whois` command would show several CIDRs to show the full span of the network. However, sometimes the command would also return several fields of CIDRs, indicating that the probed IP belonged to a subnetwork of a larger network, or even more nested than just two layers of networks.

Part 3 - Internet Traffic On Different Connections

For this part, a script (`auto_collect.sh`) was used to automate accessing 10 different websites 10 times each and recording network traffic with `tcpdump`. This script was run once per connection type (VPN, TOR, Firefox). Afterwards, another script (`auto_summarize.sh`) was used to automate summarizing the resulting `.pcap` files for packets sent per access and the average packet size per access. The collected data regarding the average number of packets is summarized in Figure 3 and the data regarding the average size of packets is summarized in Figure 4.

Using just a regular browser to access the websites, a passive device on the network can see all kinds of information, including which websites were visited and the sizes of the packets. With a VPN, `tcpdump` captured information showing the client sending packets to the VPN server and vice versa. However, information regarding packet source and destination were still visible. Finally, using TOR, the captured network traffic showed mostly packets being sent to and from `127.0.0.1` and not the IP of, say, `wikipedia.org`. As a result, it was much harder to tell exactly what websites were visited just by looking at network traffic in Wireshark.

Although difficult, it is possible to determine which of the 10 websites was visited simply based on connection statistics. For example, in Figure 4, UC Berkeley's website has, on average, larger packet sizes than most of the other websites, so given only these basic connection statistics, guessing UC Berkeley's website as the site visited due to large packet sizes is not unreasonable. Additionally, connection statistics that show a consistent rate packets sent and a consistent packet size would most likely imply that a video or stream was being watched.

Some other observations of note were in regards to the number of packets sent per connection. The average number of packets sent each iteration of visiting the same website generally decreased as the website was visited more. Also, since the websites were visited in a consistent order (from left to right on Figure 3 and Figure 4), there seemed to be a consistent trend of decreasing average number of packets per connection.

Part 4 - Splitting Packets

Using `tcpdump` to record network traffic and Wireshark to view the packets, it is possible to see certain details of a connection. In this part, these tools were used to analyze a connection to http://baiwanzhan.com. Performing

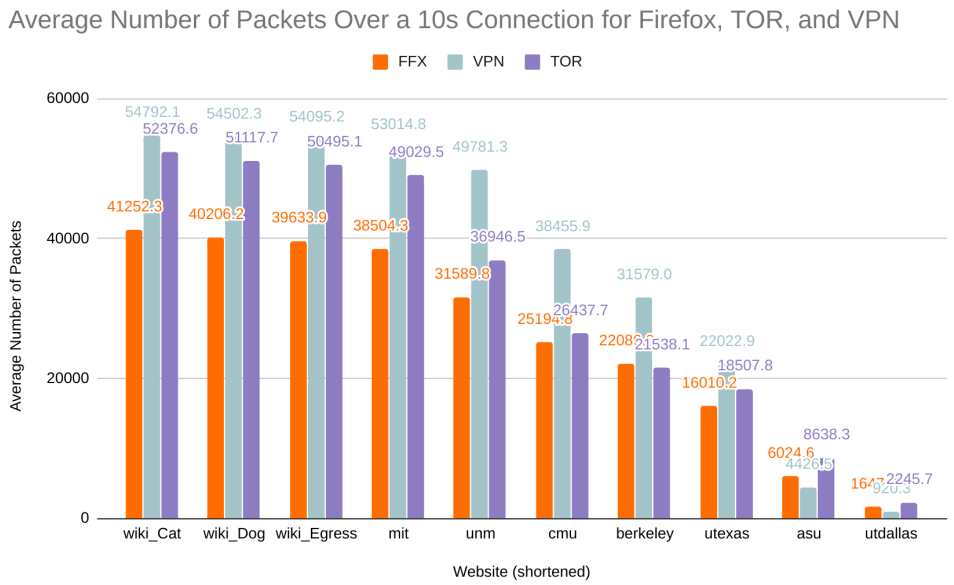

Figure 3: The average number of packets sent over a 10 second connection for 10 different websites on 3 different connections

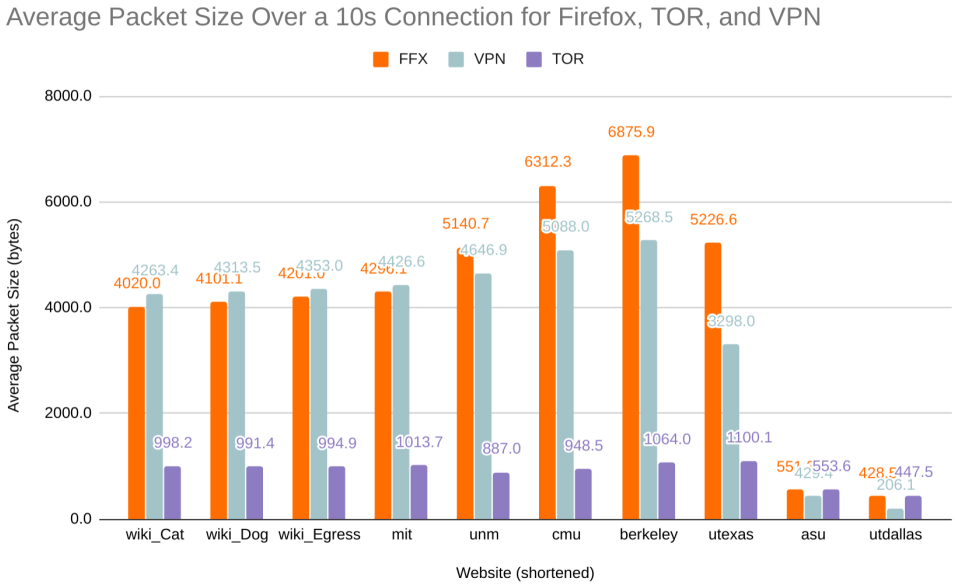

Figure 4: The average size of packets sent over a 10 second connection for 10 different websites on 3 different connections

a generic search on the website resulted in an HTTP GET request visible among the packets. Searching the phrase '法轮' in the search bar resulted in a connection reset. This is also visible in the `.pcap` file as the client receiving a reset TCP packet, seen as `[RST]` in Wireshark. One way to get around this censorship would be to split the packet into 2 and send the request split up in order to bypass the relay node that is intercepting traffic and sending resets to censor certain words or phrases. This way, when the split packets reach the destination, the server will reassemble the packets and view the result as one complete packet. This is visible in Figure 5 and Figure 6.

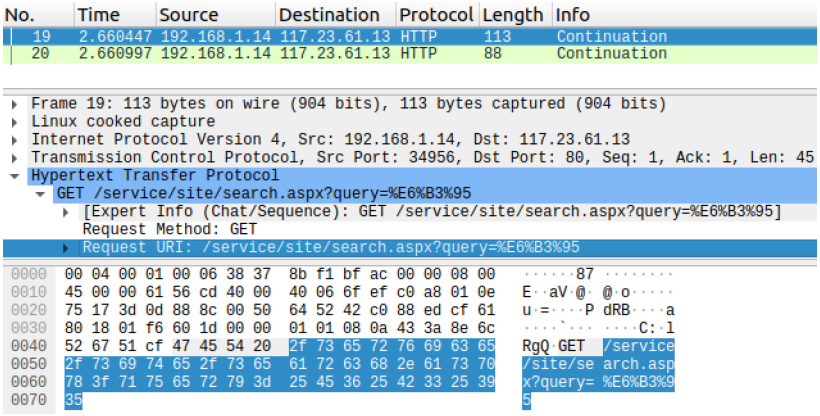

Figure 5: The first half of the GET request. This includes the encoded first character.

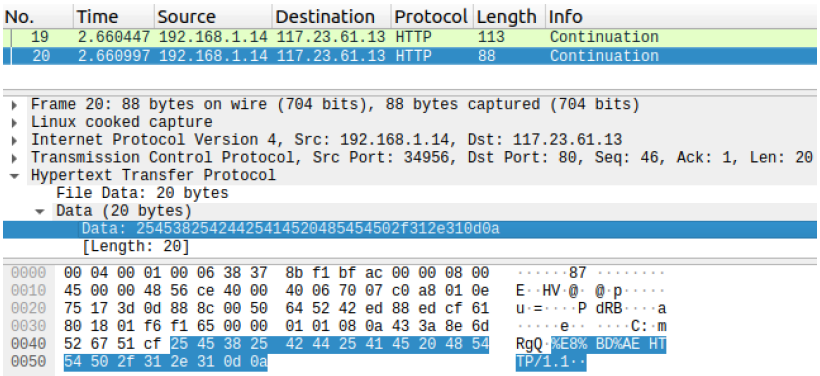

Figure 6: The second half of the GET request. This includes the encoded second character.

However, the result of this was that the request still did not get through. This is certainly a possibility since it would make sense for the censorship technology to be updated and close this loophole.

Using Wireshark to examine the packets from steps 2, where a generic search was performed, and from step 3, where a search for the censored phrase was performed, it is revealed that the time to live (TTL) of packets sent from the server is generally 43 seconds or 45 seconds. This is a relatively short TTL and is probably in place to effectively balance traffic load and redirect traffic faster since the host might be implementing a rerouting method where the IP on the authoritative DNS needs to be modified.