Part 1

1. Describe the sequence used.

We will describe the sequence we used under KaliVM and UbuntuVM separately.
For sequence we used In Ubuntu:

- Using the provided dockerfile, we build the Docker container using: `$ sudo docker build -t twebsvr .`
- Then we start the Docker container using: `$ sudo docker run -d --name websvr --privileged -v /sys/fs/cgroup:/sys/fs/cgroup:ro --network host twebsvr:latest`
- After starting the container, we can login to the running container using: `$ sudo docker exec -it websvr bash`
- Note that we are in the running container now, and we can check the server status using: `# systemctl status nginx` We can see that there is a green light indicating the server is working now.

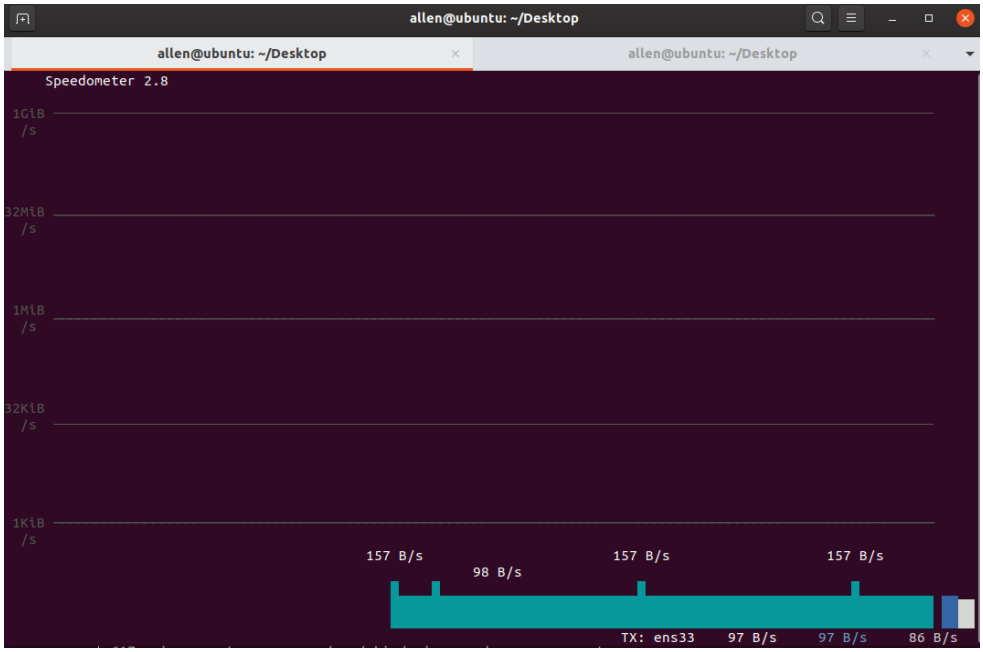
```
root@ubuntu:~# systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
   Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2020-04-12 20:12:07 UTC; 54min ago
     Docs: man:nginx(8)
  Process: 614 ExecStop=/sbin/start-stop-daemon --quiet --stop --retry QUIT/5 --pidfile /run/nginx.pid (code=exited, status=0/SUCCESS)
  Process: 616 ExecStart=/usr/sbin/nginx -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
  Process: 615 ExecStartPre=/usr/sbin/nginx -t -q -g daemon on; master_process on; (code=exited, status=0/SUCCESS)
 Main PID: 617 (nginx)
    Tasks: 3 (limit: 4020)
   CGroup: /system.slice/containerd.service/system.slice/nginx.service
           ├─617 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
           ├─618 nginx: worker process
           └─619 nginx: worker process

Apr 12 20:12:07 ubuntu systemd[1]: Starting A high performance web server and a reverse proxy server...
Apr 12 20:12:07 ubuntu systemd[1]: Started A high performance web server and a reverse proxy server.
root@ubuntu:~#
```

- If we modify the configure file, we can restart the Nginx server using: `# systemctl restart nginx`
- Finally, we use a network speed monitor command-line based tool called Speedometer to observer traffic across the network connection. We can download the tool simply using: `$ sudo apt-get install speedometer` Then we can use `speedometer -t network-interface` or `speedometer -r network-interface` to display bytes transmitted or received on a certain network interface. Here we use `speedometer -t ens33` to observe the rate of bytes transmitted on interface ens33.

```
ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:3a:6a:b7 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.176/24 brd 10.0.0.255 scope global dynamic noprefixroute ens33
       valid_lft 571624sec preferred_lft 571624sec
    inet6 2601:14d:4280:44::41eb/128 scope global dynamic noprefixroute
       valid_lft 571625sec preferred_lft 571625sec
    inet6 fe80::c2:87d4:6a92:5c7e/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

We can see that on the network interface ens33 we have IP address 10.0.0.176. And we will later send Ping requests to or initiate DoS attacks on this address.

The screenshot below shows the interface of Speedometer.



For sequence we used In KaliVM:

- After the Nginx server is started on UbuntuVM (within the docker container we built more specifically) , we can use telnet to fetch content from nginx server using `telnet 10.0.0.176 81` and input the HTTP request using
  `GET / HTTP/1.1\r\n`
  `Host: 10.0.0.176\r\n\r\n`

```
allenchan@kali:~$ telnet 10.0.0.176 81
Trying 10.0.0.176 ...
Connected to 10.0.0.176.
Escape character is '^]'.
GET / HTTP/1.1
Host: 10.0.0.176

HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Sun, 12 Apr 2020 21:30:31 GMT
Content-Type: text/html
Content-Length: 290
Last-Modified: Fri, 03 Apr 2020 13:46:41 GMT
Connection: keep-alive
ETag: "5e873e41-122"
Accept-Ranges: bytes

<!-- Place in /var/www/nwsec -->

<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Hello, Students!</title>
</head>
<body>
    <h1>Hello, EN.650.624 Network Security Students!</h1>
    <p>We have just configured our Nginx web server on Ubuntu Server!</p>
</body>
</html>
```

- We use hping3 to initiate DoS SYN flood attack on Nginx server. We just need to first install hping3 using `sudo apt-get install hping3`. After successfully installing hping3, we can initiate DoS SYN flood attack by using the following command:

```
allenchan@ubuntu:~/Documents/websvr/UbuntuServerRPI4B$ sudo hping3 -c 10000 -d 120 -S
 -w 64 -p 81 --flood --rand-source 10.0.0.176
[sudo] password for allenchan:
HPING 10.0.0.176 (ens33 10.0.0.176): S set, 40 headers + 120 data bytes
hping in flood mode, no replies will be shown
^C
--- 10.0.0.176 hping statistic ---
7363441 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Now let's explain each parameter in this command in detail!

- sudo: run hping3 with root privilege.
- hping3 = call hping3 program
- -c 100000 = number of packets to send.
- -d 120 = size of each packet
- -S = specifies SYN packets only.
- -w 64 = TCP window size.
- -p 81 = the destination port number is 81, which is the port used by nginx.
- --flood = flood mode, replies will be ignored and packets are sent as fast as possible.
- --rand-source = hide IP address and use random source IP addresses.
- 10.0.0.176 = target IP address.

- Finally, we used the Ping program to observe the status of the Ubuntu server, if the server is down then we could probably receive no reply or few replies from the server. The command is Ping hostname, such as Ping 10.0.0.176.

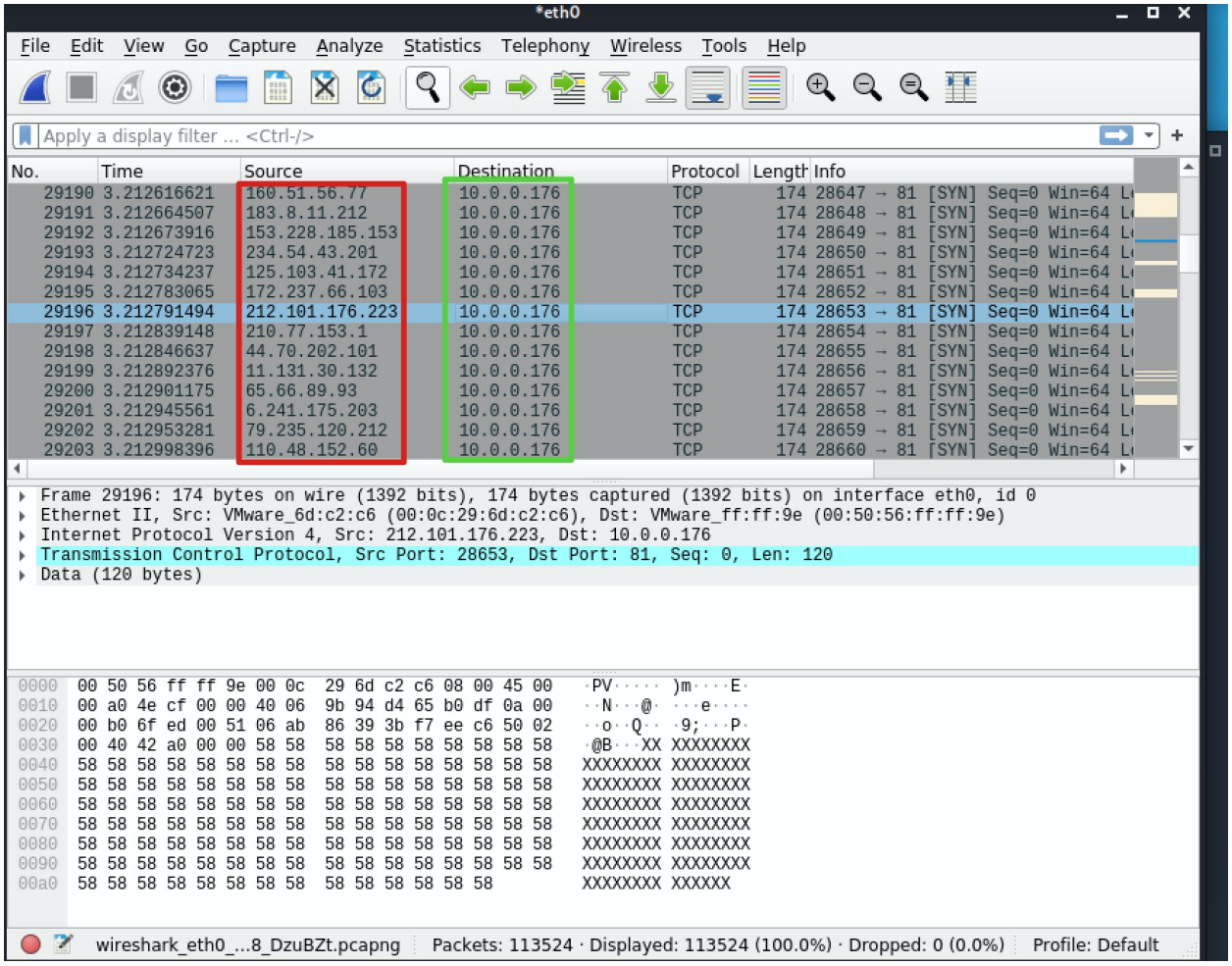
```
allenchan@kali:~$ ping 10.0.0.176
PING 10.0.0.176 (10.0.0.176) 56(84) bytes of data.
64 bytes from 10.0.0.176: icmp_seq=1 ttl=128 time=19.8 ms
64 bytes from 10.0.0.176: icmp_seq=2 ttl=128 time=11.3 ms
64 bytes from 10.0.0.176: icmp_seq=3 ttl=128 time=11.4 ms
64 bytes from 10.0.0.176: icmp_seq=4 ttl=128 time=10.8 ms
64 bytes from 10.0.0.176: icmp_seq=5 ttl=128 time=11.2 ms
^C
--- 10.0.0.176 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4008ms
rtt min/avg/max/mdev = 10.757/12.894/19.797/3.458 ms
```

2. Provide a python script that executes your attack

As described above, we use hping3 -c 10000 -d 120 -S -w 64 -p 81 --flood --rand-source 10.0.0.176 to execute our attack. Also, we tried ICMP attack using hping 3, which is not much powerful, though. The command of ICMP flood attack we used is: sudo hping3 -S -a 10.0.0.196 --flood -p 81 192.168.20.140.

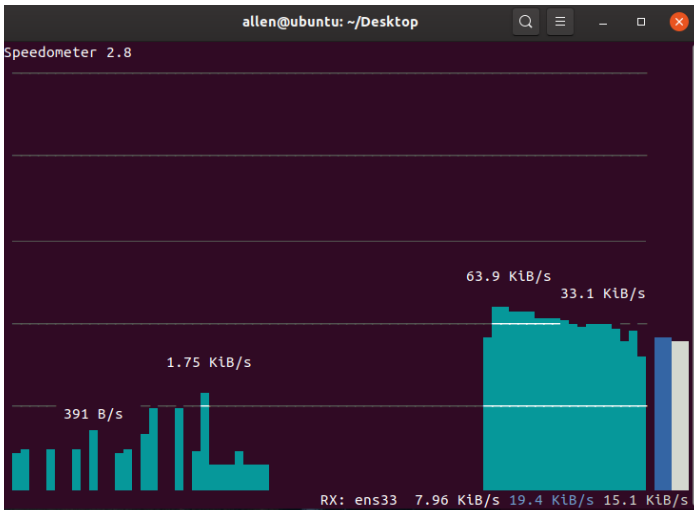
3. Provide *.pcap network capture file from traffic between attack host and target running the application server

We provide the supplementary .pcap file in a single .zip archive. Here is a screenshot of the traffic captured by Wireshark between attack host and target. We can see that the Destination host is always 10.0.0.176 but the Source IP addresses keep changing, which is due to the mechanism that hping3 spoofs random Source IP addresses to conduct the DoS attack.

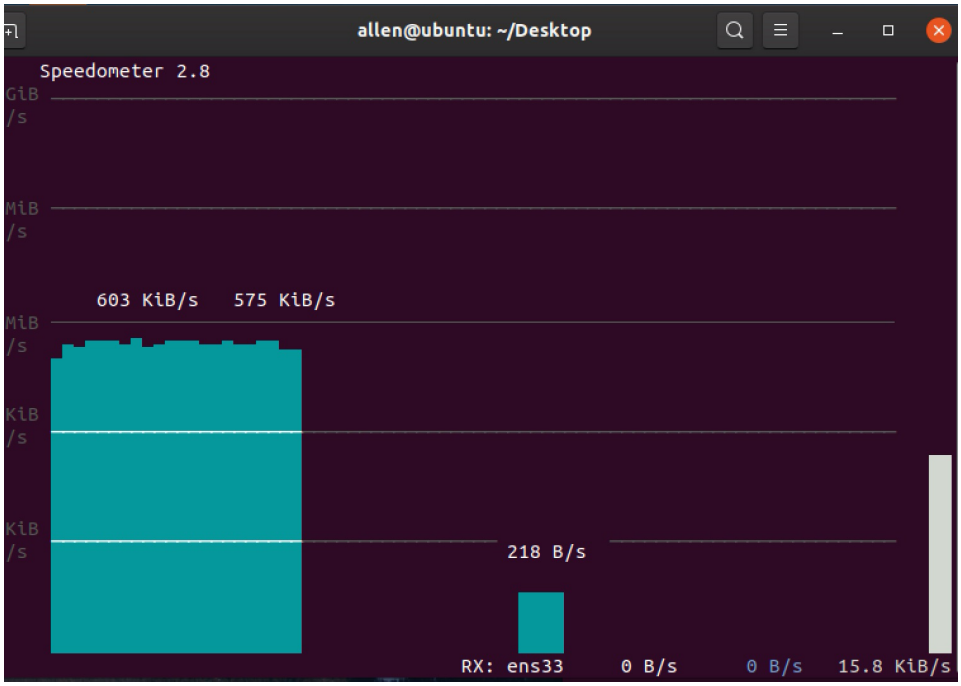


4. Describe the results of the attack

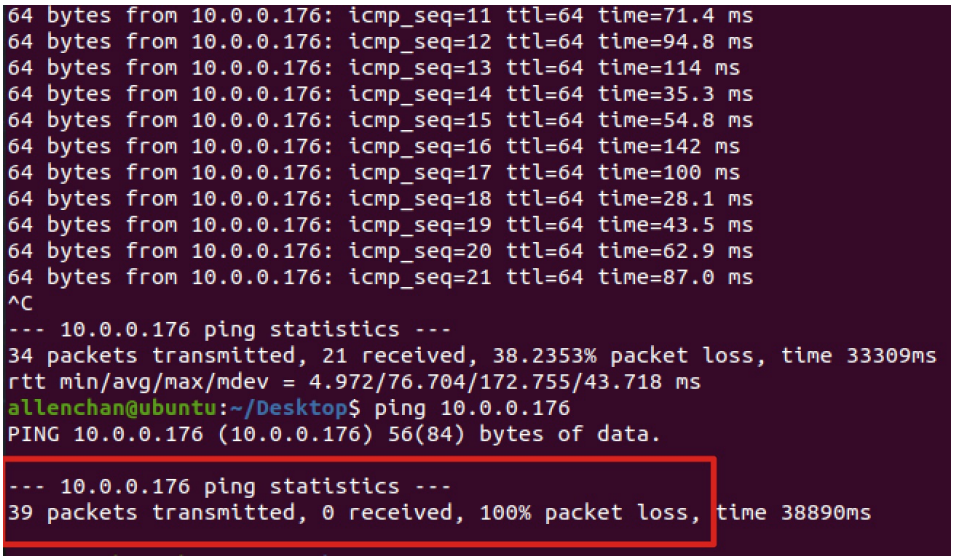
From the interface of Speedometer below we can see that after the flood attack begins, there is a surge of data received in Ubuntu server.



After conducting the DoS attack for a while (about one minute), the traffic on the server suddenly drops to near 0. Actually, the server is out of service now!



Another obvious evidence is that when we send Ping requests to the server now, we get no Ping reply from the server. All packets are lost.



To conclude, we have successfully performed DoS attacks on Nginx running on the Ubuntu server.

Part 2

1. Provide the details for updated Nginx settings

We tried 2 ways to improve our defense against DoS attacks, or more specifically, SYN flood attacks we performed here using hping3. First, we tried to modify the configuration file of Nginx in /etc/nginx/nginx.conf. We mainly tried to limit the rate of requests and the rate of connections of Nginx. Rate limiting is one of the common ways to defend against DDoS in Nginx, it can limit the number of requests that can be made from a particular client IP address within a certain period. And, if this limit is exceeded, Nginx denies these requests.

Specifically, we modify the limit_req_zone directive in the Nginx configuration file to limit the rate of requests:

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
server {
    location /search/ {
        limit_req zone=one burst=5;
    }
}
```

This command creates a memory area of 10m and its name is one, and the 1r/s means that 1 request can be permitted by a particular client for every second. If the requests rate exceeds the rate configured for a zone, their processing is delayed such that requests are processed at a defined rate. Excessive requests are delayed until their number exceeds the maximum burst size in which case the request is terminated with an error.

Also, we modified the limit_conn_zone and limit_conn directives to limit the number of connections per particular IP address.

```
limit_conn_zone $binary_remote_addr zone=addr: 10m;
server {
        location /download/ {
                limit_conn addr 1;
        }
}
```

In this way, the directives allow only one connection per an IP address at a time.

However, the way of modifying the configuration file of Nginx as shown above does not work so well, and we analyze that it is probably due to the type of attack we performed. Since we used hping3 to perform SYN flood attack, a large amount of different spoofed IP addresses are used to attack the server, which resulted in limiting connections or requests of each IP address is not effective enough. Therefore, we had to find the second way to improve our defense.

Here, we used iptables to set up specific network rules on firewalls and tried to improve defense against DoS attack on OS level.

First, we installed iptables in Docker container using command: `$ sudo apt-get install iptables`
Then we can set up multiple network rules on the firewall. For example, we use:

`# iptables -A INPUT -p tcp --syn -m limit --limit 1/s --limit-burst 3 -j RETURN`
We limit the number of incoming tcp connection or syn-flood attacks, and all incoming connection are allowed till limit is reached:

- limit 1/s: Maximum average matching rate in seconds.
- limit-burst 3: Maximum initial number of packets to match.

We also use the following rule to prevent DoS attack:

`# iptables -A INPUT -p tcp --dport 81 -m limit --limit 25/minute --limit-burst 20 -j ACCEPT`
Where:
-m limit: This uses the limit iptables extension
limit 25/minute: This limits only a maximum of 25 connections per minute.
--limit-burst 20: This value indicates that the limit/minute will be enforced only after the total number of connections have reached the limit-burst level.

```
root@ubuntu:~# iptables -nL INPUT
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:81 limit: avg 25/min burst 20
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:81
RETURN     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp flags:0x17/0x02 limit: avg 1/sec burst 3
root@ubuntu:~#
```

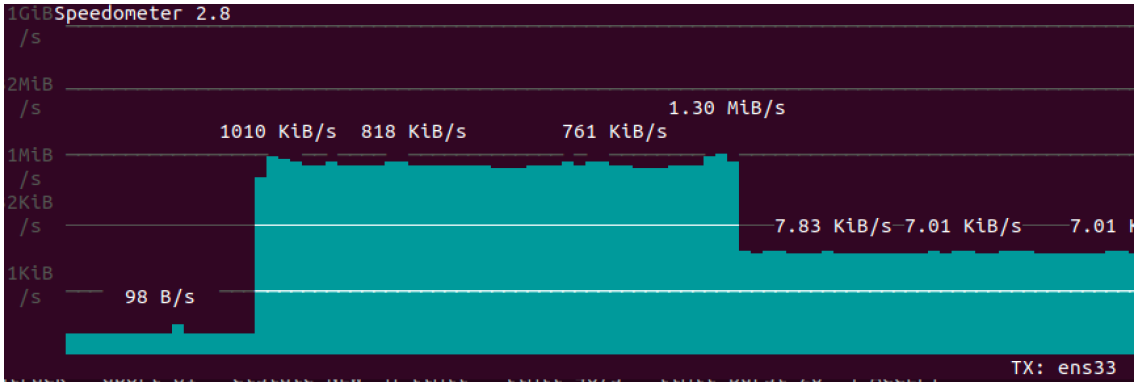
Finally, we can use `# iptables -nL INPUT` to list the current network rules we set in iptables, as shown above.

2. Provide *.pcap network capture file from traffic between attack host and target running the application server

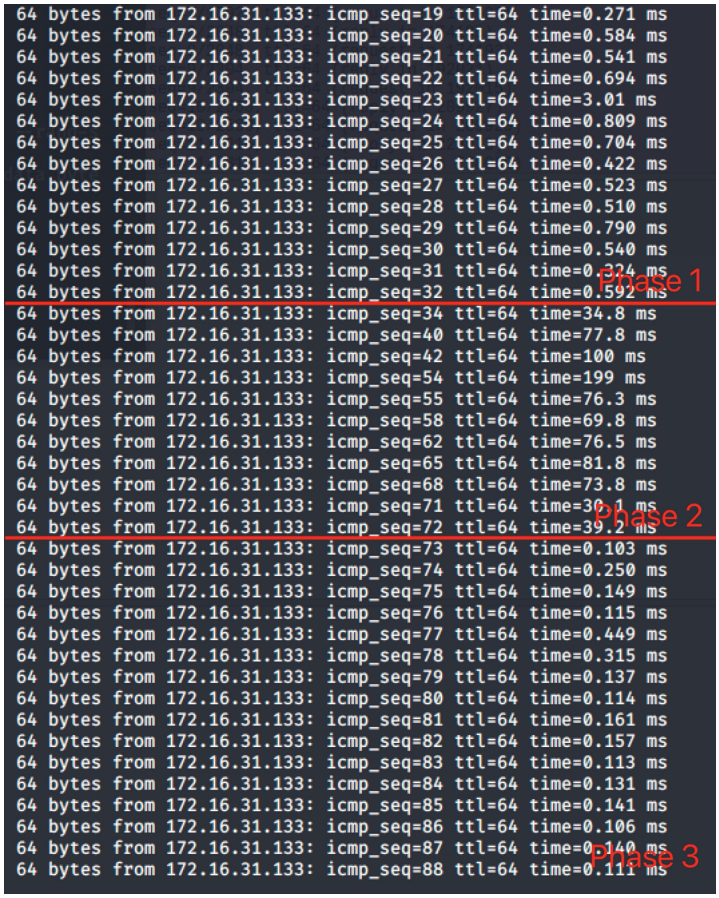
Please see attached .pcap file.

3. Describe the results of the attack

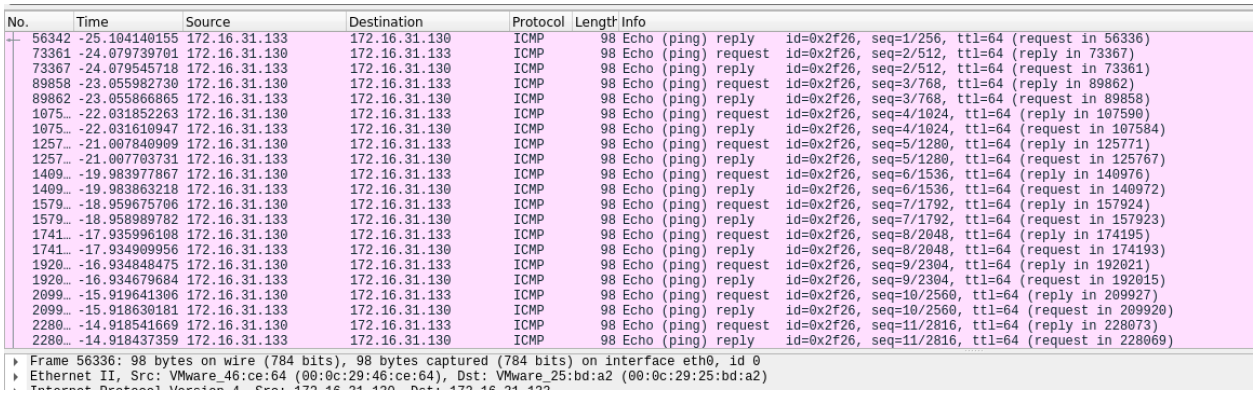
First, let's see the traffic change displayed by Speedometer. After performing DoS attack on Nginx server, the rate of data transmitted by the server surges from approximately 98 B/s to 1MB/s. However, after we improved defense against DoS attacks by improving os network security using iptables, the rate of data transmitted by the server drops down to only approximately 7 KB/s, which is a significant change.



Also, from the Ping program we can see that there are 3 distinct phases. In Phase 1, we just Ping our server without flood attack or defense improvement on the server, and the Round Trip Time is small. In Phase 2, we performed DoS attacks on Nginx from KaliVM, and the Round Trip Time increased significantly, which means the Nginx server became really slow to respond. In Phase 3, we improved defense against DoS attacks by using iptables to set up specific network rules on the Ubuntu server running Nginx. We can see that the server is again able to respond quickly to Ping requests and the Round Trip Time is small.



From Wireshark captures, we can also observe that almost every Ping request can be replied by the server now.



Part 3

1. Provide the details used to install and configure Suricata so that it could be replicated by someone else

Installation

First, we install Suricata in a Docker container on the Ubuntu server following the steps:

1) Build the Docker image using: `$ sudo docker build -t tsuricata .`
2) Build the Docker container using: `$ sudo docker run -d --name suricata --privileged -v /sys/fs/cgroup:/sys/fs/cgroup:ro --network host tsuricata:latest`
3) Run the Docker container using: `$ sudo docker exec -it suricata bash`

Configuration

After the installation, we will have Suricata rules under `/etc/suricata/rules`/ and the main configuration file under `/etc/suricata/suricata.yaml`. Next step is to configure the Suricata in the Docker container for our usage.

4) Configure Suricata to differentiate between your internal network to be protected and the external network in the configuration file. This can be done by defining the correct values for the `HOME_NET` and `EXTERNAL_NET` variables respectively under the address groups. We put our IP address in `HOME_NET` and `"!$HOME_NET"` for `EXTERNAL_NET` as default:

```
$ vim /etc/suricata/suricata.yaml
      HOME_NET: "[192.168.43.220]" ($Host IP Address$)
  ...
      EXTERNAL_NET: "!$HOME_NET"
```

5) Next, we want to create a .rules file to set up the regulations for Suricata to detect the possible SYN flood:

```
$ vim /etc/suricata/rules/test-ddos.rules
```

And put these in the file:

```
alert tcp any any -> $HOME_NET 81 (msg: "Possible DDoS attack"; flags: S; flow: stateless; threshold: type both, track by_dst, count 100, seconds 1; sid:1000001; rev:1;)
```

The rule basically fires when there are 100 attempted connections to the local network with port 81 in 1 seconds. The parameter can be changed accordingly.

6) Next, we need to edit the Suricata configuration file to configure Suricata to include this rule:
    a) Change the default rule path to: `/etc/suricata/rules`
    b) Add the rules file under the `rule-files:` section:

```
$ vim /etc/suricata/suricata.yaml
```

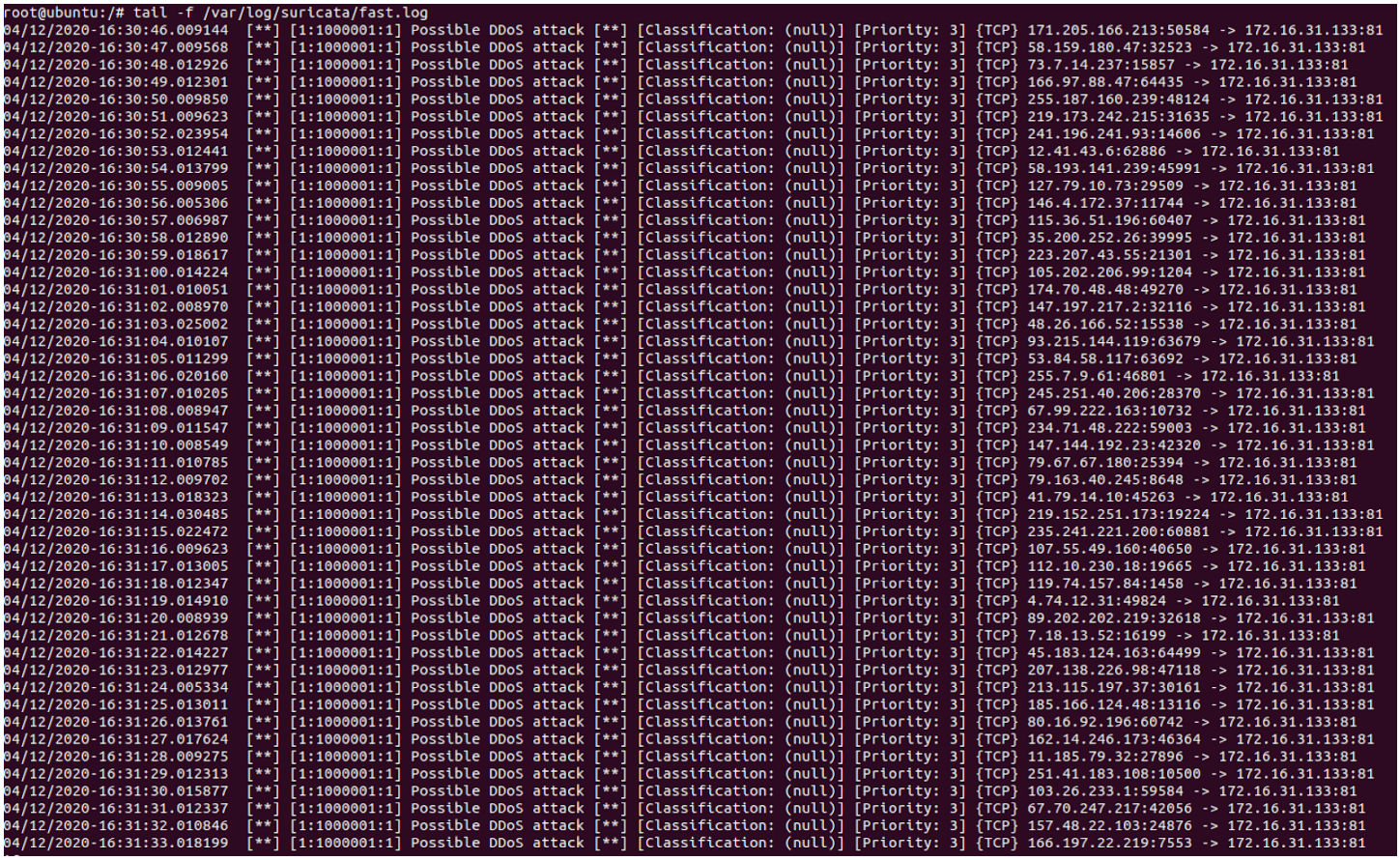
```
Default rule path: /etc/suricata/rules
rule-files:
# - Custom Test rules
 - test-ddos.rules
```

Running Suricata to Detect Possible DDoS Attack

7) After configuration, we firstly need to check our network interface using: `$ ifconfig`
8) Then running the Suricata to detect possible DDos Attack:
   `$ suricata -c /etc/suricata/suricata.yaml -i ens33`(The interface get from last step)

2. Provide reporting details from Suricata that show it detected the attack

1) While the Suricata is running on the HTTP server, we perform DDoS attack from client in Kali VMWare using: `$ hping3 -c 10000 -d 120 -S -w 64 -p 81 --flood --rand-source 172.16.31.133` (where 172.16.31.133 is the IP address of my Nginx HTTP Server.)
2) Meanwhile, we open another terminal for the same Suricata container, and tail the Suricata logs to monitor its behavior using:
   `$ sudo docker exec -it suricata bash`
   `$ tail -f /var/log/suricata/fast.log`



From the screenshot above we can notice that Suricata has successfully captured several Possible DDoS attacks on Port 81 of 172.16.31.133, where the Nginx server locates on the host.

Part 4

1. Discuss what else could be performed to defend against these attacks

In general, DDoS attacks can be segregated by which layer of the Open Systems Interconnection (OSI) model they attack. They are most common at the Network (layer 3), Transport (Layer 4), Presentation (Layer 6) and Application (Layer 7) Layers.

Infrastructure Layer Attacks

Attacks at Layer 3 and 4, are typically categorized as Infrastructure layer attacks. These are also the most common type of DDoS attack and include vectors like synchronized (SYN) floods and other reflection attacks like User Datagram Packet (UDP) floods. These attacks are usually large in volume and aim to overload the capacity of the network or the application servers. But

fortunately, these are also the type of attacks that have clear signatures and are easier to detect.

Application Layer Attacks

Attacks at Layer 6 and 7, are often categorized as Application layer attacks. While these attacks are less common, they also tend to be more sophisticated. These attacks are typically small in volume compared to the Infrastructure layer attacks but tend to focus on particularly expensive parts of the application thereby making it unavailable for real users. For instance, a flood of HTTP requests to a login page, or an expensive search API, or even Wordpress XML-RPC floods (also known as Wordpress pingback attacks).

Thus, for thinking about mitigation techniques against these attacks, it is useful to group them as Infrastructure layer (Layers 3 and 4) and Application Layer (Layer 6 and 7) attacks.

- Reduce Attack Surface Area
  One of the first techniques to mitigate DDoS attacks is to minimize the surface area that can be attacked thereby limiting the options for attackers and allowing you to build protections in a single place. We want to ensure that we do not expose our application or resources to ports, protocols or applications from where they do not expect any communication. Thus, minimizing the possible points of attack and letting us concentrate our mitigation efforts. In some cases, you can do this by placing your computation resources behind Content Distribution Networks (CDNs) or Load Balancers and restricting direct Internet traffic to certain parts of your infrastructure like your database servers. In other cases, you can use firewalls or Access Control Lists (ACLs) to control what traffic reaches your applications

- Plan carefully for scale
  Transit capacity. When architecting your applications, make sure your hosting provider provides ample redundant Internet connectivity that allows you to handle large volumes of traffic. Since the ultimate objective of DDoS attacks is to affect the availability of your resources/applications, you should locate them, not only close to your end users but also to large Internet exchanges which will give your users easy access to your application even during high volumes of traffic. Additionally, web applications can go a step further by employing Content Distribution Networks (CDNs) and smart DNS resolution services which provide an additional layer of network infrastructure for serving content and resolving DNS queries from locations that are often closer to your end users.

  Server capacity. Most DDoS attacks are volumetric attacks that use up a lot of resources; it is, therefore, important that you can quickly scale up or down on your computation resources. You can either do this by running on larger computation resources or those with features like more extensive network interfaces or enhanced networking that support larger volumes. Additionally, it is also common to use load

balancers to continually monitor and shift loads between resources to prevent overloading any one resource.

- Be alert with normal and abnormal traffic

  Whenever we detect elevated levels of traffic hitting a host, the very baseline is to be able only to accept as much traffic as our host can handle without affecting availability. This concept is called rate limiting. More advanced protection techniques can go one step further and intelligently only accept traffic that is legitimate by analyzing the individual packets themselves. To do this, you need to understand the characteristics of good traffic that the target usually receives and be able to compare each packet against this baseline.

- Deploy firewalls with sophisticated application attacks

  A good practice is to use a Web Application Firewall (WAF) against attacks, such as SQL injection or cross-site request forgery, that attempt to exploit a vulnerability in your application itself. Additionally, due to the unique nature of these attacks, you should be able to easily create customized mitigations against illegitimate requests which could have characteristics like disguising as good traffic or coming from bad IPs, unexpected geographies, etc. At times it might also be helpful in mitigating attacks as they happen to get experienced support to study traffic patterns and create customized protections.ploy firewalls for sophisticated attacks