

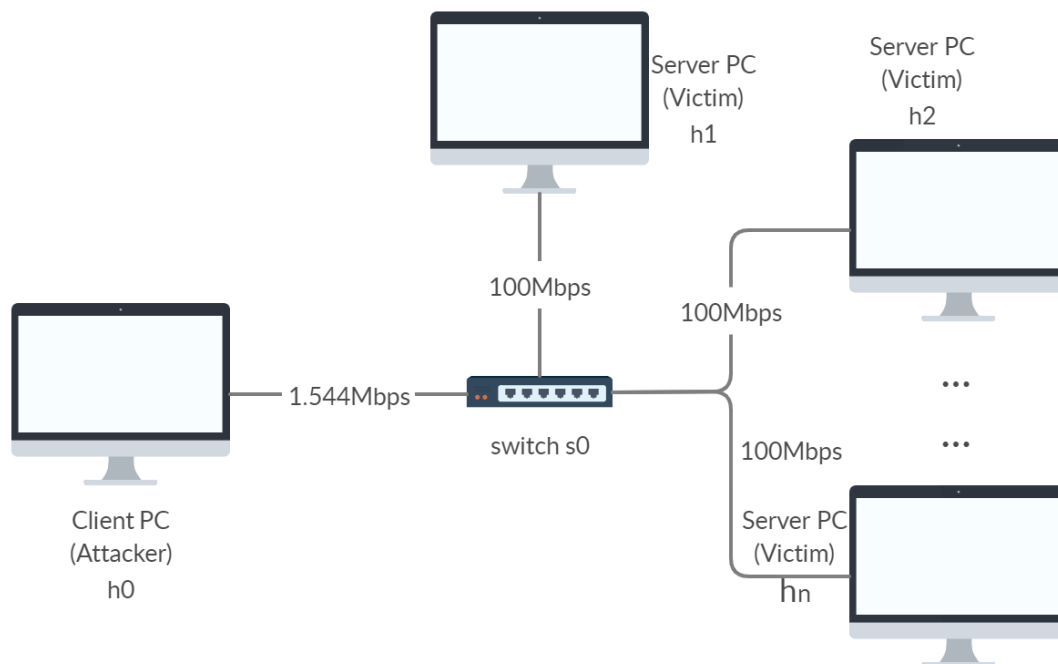
# **TCP Optimistic ACK Attack**

Sifat Ishmam Parisa

Student ID – 1505016

## Optimistic TCP ACK Attack

The underlying assumption of transmission control protocol - tcp, is the receiver end is trustworthy. As a result of the congestion control mechanism of tcp, the transmission rate is increased when there is no packet loss i.e positive ack. Henceforth, a misbehaving receiver who does not take data integrity into account could manipulate the sender into sending more packets even when data loss occurs by sending incorrect positive acknowledgment also known as optimistic acknowledgment or opt-ack. Opt ack attack can saturate the path from sender to receiver and potentially cause denial of service. The number of victims can be arbitrary.



Here the attacker PC is connected to n servers via a switch in a star topology.

## Steps of the attack:

I have implemented tcp opt-ack attack using mininet. Mininet creates a **realistic virtual network**, running **real kernel, switch and application code**, on a single machine (VM, cloud or native). I have used mininet VM and the code scripts are in python. Increase of network traffic is shown in a plot by gnuplot.

### Step 1:

#### Create Topology

The client h0 is connected to switch s0 with bandwidth 1.544Mbps . Then for n=1,2,4,8,16,32,64 servers h1, h2..., hn are connected to switch s0 with bandwidth 100Mbps.

### Step 2:

#### Start net

The servers send packets to the client until termination. The client receives the data from all servers and sends ack packets but not cause overrun (send ack for a packet that the server has not yet sent).

Mininet CLI:

```
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080
Experiment with 1 server finished
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080 10.0.0.3 8080
Experiment with 2 servers finished
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080 10.0.0.3 8080 10.0.0.4 8080 10.0.0.5 8080
Experiment with 4 servers finished
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080 10.0.0.3 8080 10.0.0.4 8080 10.0.0.5 8080 10.0.0.
6 8080 10.0.0.7 8080 10.0.0.8 8080 10.0.0.9 8080
Experiment with 8 servers finished
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080 10.0.0.3 8080 10.0.0.4 8080 10.0.0.5 8080 10.0.0.
6 8080 10.0.0.7 8080 10.0.0.8 8080 10.0.0.9 8080 10.0.0.10 8080 10.0.0.11 8080 1
0.0.0.12 8080 10.0.0.13 8080 10.0.0.14 8080 10.0.0.15 8080 10.0.0.16 8080 10.0.0
.17 8080
Experiment with 16 servers finished
Client Address 10.0.0.1
Server Addresses 10.0.0.2 8080 10.0.0.3 8080 10.0.0.4 8080 10.0.0.5 8080 10.0.0.
6 8080 10.0.0.7 8080 10.0.0.8 8080 10.0.0.9 8080 10.0.0.10 8080 10.0.0.11 8080 1
0.0.0.12 8080 10.0.0.13 8080 10.0.0.14 8080 10.0.0.15 8080 10.0.0.16 8080 10.0.0
.17 8080 10.0.0.18 8080 10.0.0.19 8080 10.0.0.20 8080 10.0.0.21 8080 10.0.0.22 8
080 10.0.0.23 8080 10.0.0.24 8080 10.0.0.25 8080 10.0.0.26 8080 10.0.0.27 8080 1
0.0.0.28 8080 10.0.0.29 8080 10.0.0.30 8080 10.0.0.31 8080 10.0.0.32 8080 10.0.0
.33 8080
```

Wireshark was started with loopback and OF filter. Wireshark output:

Capturing from Loopback: lo [Wireshark 1.10.6 (v1.10.6 from master-1.10)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: of Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
808	61.476407000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
809	61.498449000	10.0.0.1	10.0.0.2	OF 1.0	138	of_packet_in
810	61.498497000	10.0.0.1	10.0.0.2	OF 1.0	156	GET / HTTP/1.0 of_packet_in
811	61.499362000	10.0.0.1	10.0.0.3	OF 1.0	142	of_packet_in
812	61.500580000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
813	61.502152000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
815	61.503771000	127.0.0.1	127.0.0.1	OF 1.0	90	of_packet_out
819	61.524713000	10.0.0.3	10.0.0.1	OF 1.0	146	of_packet_in
820	61.525357000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
822	61.566130000	10.0.0.1	10.0.0.3	OF 1.0	138	of_packet_in
823	61.566172000	10.0.0.1	10.0.0.3	OF 1.0	156	GET / HTTP/1.0 of_packet_in
824	61.567452000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
826	61.569036000	127.0.0.1	127.0.0.1	OF 1.0	146	of_flow_add
975	65.595276000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_request
976	65.595723000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_reply
1755	70.596705000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_request
1756	70.598219000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_reply
2556	75.596206000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_request
2557	75.596800000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_reply
2789	80.596667000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_request
2790	80.597493000	127.0.0.1	127.0.0.1	OF 1.0	74	of_echo_reply

0000 00 00 00 00 00 00 00 00 00 00 00 08 00 45 c0 .....E.  
0010 00 8e b8 bf 40 00 40 06 82 e8 7f 00 00 01 7f 00 ....@.@.  
0020 00 01 94 bc 19 fd 37 00 83 0b d2 70 cf a2 80 18 .....7....p...  
0030 00 56 fe 82 00 00 01 01 08 0a 00 0c 39 c9 00 0c .V.....9...  
0040 39 c4 01 0a 00 5a 00 00 00 00 00 01 03 00 48 9.....Z.....H  
0050 00 01 00 00 8c 2d 99 b9 22 a2 7a 2c 72 82 63 4d -...#...rM

Loopback: lo: <live capture in progress> Fil... Packets: 2812 · Displayed: 52 (1.8%)

### Step 3:

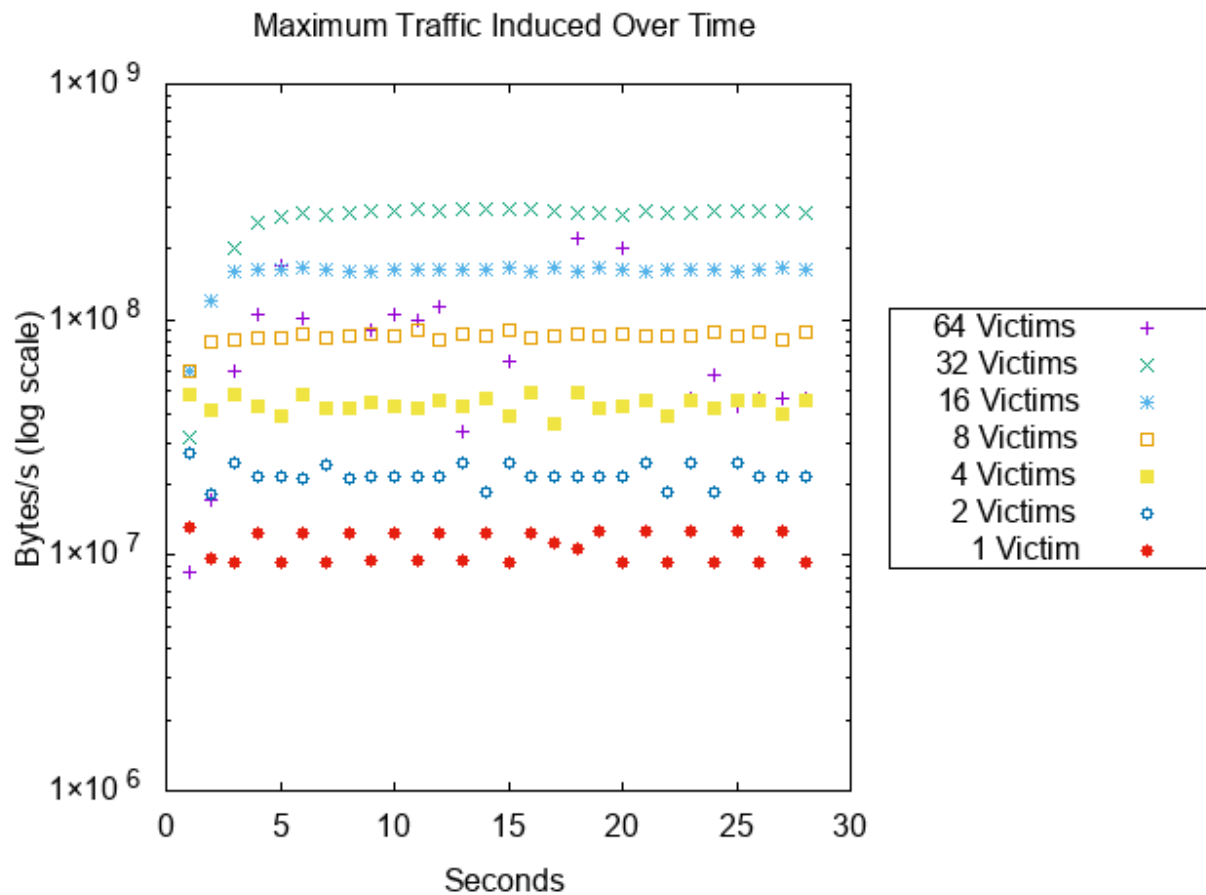
#### Log output:

Time and size(bytes) of sent data are logged in csv files

```
16.csv
1, 60280000
2, 119360000
3, 158940000
4, 163960000
5, 164560000
6, 165500000
7, 162160000
8, 160600000
9, 161120000
10, 163640000
11, 162100000
12, 164660000
13, 162080000
14, 164580000
15, 165920000
16, 160600000
17, 165080000
18, 160760000
19, 166920000
20, 164580000
21, 160120000
22, 164540000
23, 162660000
24, 162080000
25, 159100000
26, 162820000
27, 165000000
28, 163380000
```

#### Step 4:

Plot:



#### Success of the attack:

From the plot it is evident that opt-ack causes increase in traffic in the network. Packets received from the victim and retransmission detection are used to estimate congestion window and avoid overrun. Because of measurement issues there are fluctuations in the graph with less victims.

#### Countermeasures:

I did not implement any countermeasure. However, the following could be applied to defend against opt ack:

##### Selectively Send Packets:

The server can randomly choose not to send a packet and mark that packet. When a malicious client sends ack for that packet the server would therefore know the client is malicious.