CSE 406

Computer Security Sessional

Final Report

GROUP → 01

Optimistic TCP ACK Attack

1605001, Md. Ashraful Islam

ICMP Connection-Reset & Throughput Reduction

1605011, Mir Mahathir Mohammad

DHCP Spoofing

1605026, Mohammad Abser Uddin

ARP Cache Poisoning with MITM

1505004, All Rubayet Ahmed Tusher

Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology
Dhaka, Bangladesh

July 25, 2021

Contents

List of Figures

List of Source Codes

Chapter 1

Optimistic TCP ACK attack (streaming server) → 1605001

1.1 Definition & Topology Diagram

We start our discussion by reviewing the background of the Optimistic TCP ACK attack, then we'll see what is the Optimistic TCP ACK attack.

TCP Congestion Control:

Transmission Control Protocol (TCP) → is a connection-oriented communication protocol that facilitates the exchange of messages between computing devices in a network. It is the most common protocol in networks that use the Internet Protocol (IP); together they are sometimes referred to as **TCP/IP** [1].

The number of TCP packets allowed to be outstanding during a TCP communication session before requiring an ACK is known as a **congestion window**.

When a server receives ACKs from a client, it dynamically adjusts the congestion window size to reflect the estimated bandwidth available. This window size grows when ACKs are received, and shrinks when segments arrive out of order or are not received at all, indicating missing data. In this way, TCP handles the transmission speed. But attackers use this technique to perform an attack called ***Optimistic TCP ACK attack***.

Optimistic TCP ACK Attack:

An optimistic **TCP ACK** attack is a **Denial-of-Service** attack that makes the congestion-control mechanism of TCP work against itself. In this attack, a rogue client tries to make a server increase its sending rate until it runs out of bandwidth and cannot effectively serve anyone else. After connecting with a server using a three-way handshake, a rogue client sends too many ACK packets to the server without even seeing these packets in the first place. When the server receives ACK packets, it thinks that the client receives all the packets; so it increases the congestion window size. This means server sends more data to that client and at a certain point, the server runs out of bandwidth and can not serve other clients. The topology diagram of this attack is shown in Figure 1.1.

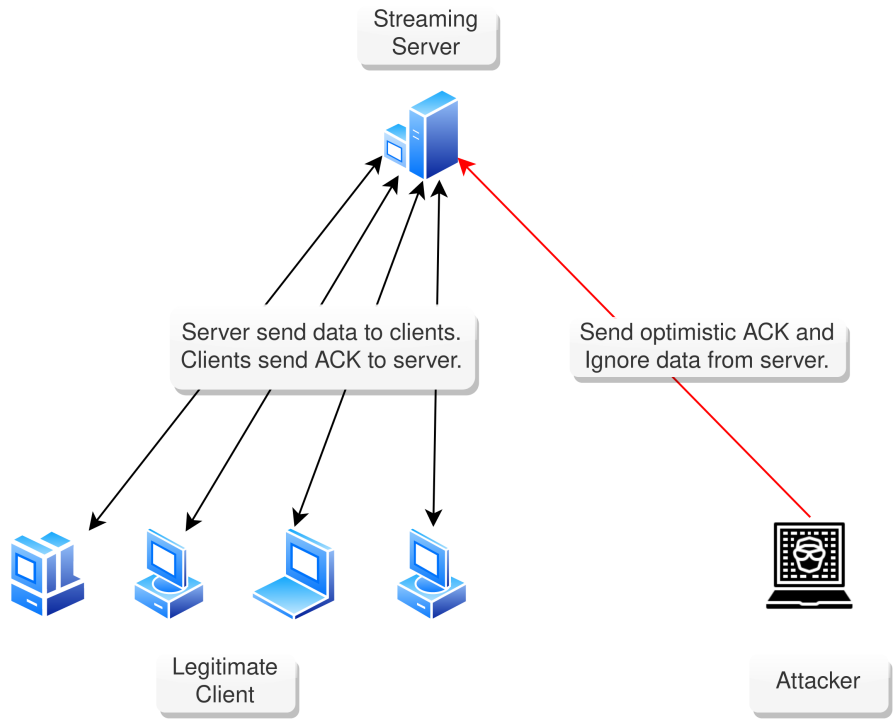
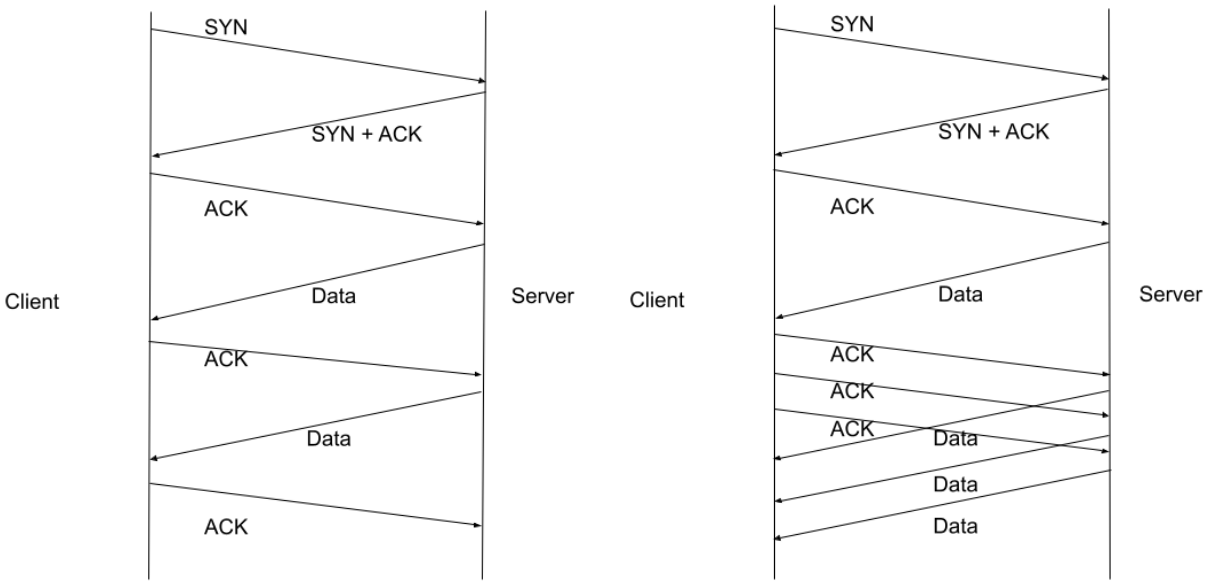


Figure 1.1: Topology Diagram of Optimistic TCP ACK Attack

1.2 Attack Strategies

Let us see the timing diagram of the original protocol in Figure 1.2a and the timing diagram for an attacker in Figure 1.2b.



(a) Timing Diagram of the original protocol (b) Timing diagram of the attack

Figure 1.2: Timing Diagram of Optimistic TCP ACK Attack

Now, let us see the Algorithm 1 which is used to successfully perform this attack. This algorithm is a slightly modified version of the algorithm proposed by Sherwood et al. [2] for optimistic TCP ack attack for multiple servers. The algorithm takes the server's IP address, PORT, Maximum Segment Size (MSS), and Maximum Window Size (MWS) as parameters. Server IP address and PORT can easily be collected. Now, for maximum segment size (MSS) and maximum window number (MWS), we need to inspect the TCP segments of a legitimate client and server in Wireshark.

Algorithm of Optimistic TCP ACK attack:

Algorithm 1 Optimistic_TCP_ACK

Require: IP, $PORT$, MSS, MWS
  Establish connection with the server using IP and PORT
  Wait until receiving a segment from the server
  ack_number $\Leftarrow$ *seq number from the segment*
  while *true* **do**
    send an ack to the server with ack_number in the acknowledgement number field
    if *ack_number* $< MWS$ **then**
      ack_number $\Leftarrow$ *ack_number* $+ MSS$
    end if
    Sleep for a while
  end while

1.3 Attack Details

For successfully performing this attack I have used a network which is shown in Figure 1.3. I have used mininet to implement this network in my Linux (Ubuntu 20.04) machine.

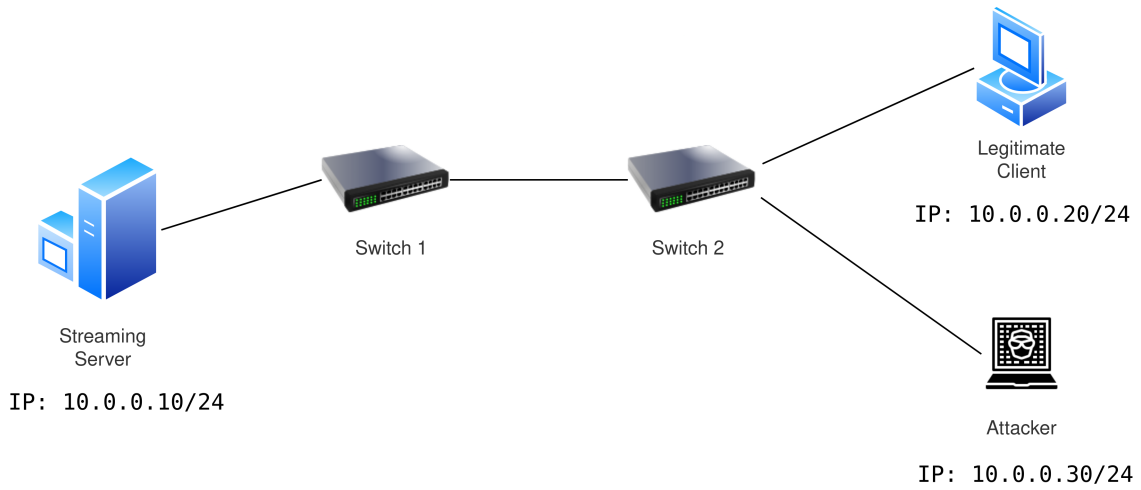


Figure 1.3: Topology Diagram of Optimistic TCP ACK Attack in mininet

My server and legitimate client are written in python. I have tried to implement this two in C or C++. But the problem is that my server and client need OpenCV to transfer and receive image data respectively. In C or C++, I can not properly configure

the OpenCV in my machine so I use python. But my attacker is written in C. I have used **RAW_SOCKET** to change the segment and packet header of TCP/IP protocol. Let us see the result of the attack.

First, we will see the normal case. So, we run the server and a legitimate client. We can see in Figure 1.4 that in this case, the client can get the image frame successfully from the server. I have also shown the bandwidth of received data and total frame received to easily understand the whole scenario.

Figure 1.4: Attack Statistics

Now, I have run the server and the attacker. In the wireshark, we can see in Figure 1.5 that attacker successfully connects with the server using three-way handshake protocol and in Figure 1.6 that attacker can successfully send optimistic ACK to the server and the server sends packets in response of the optimistic ACKs. This process will continue. Figure 1.5 and Figure 1.6 illustrate that my attack is successfully performed.

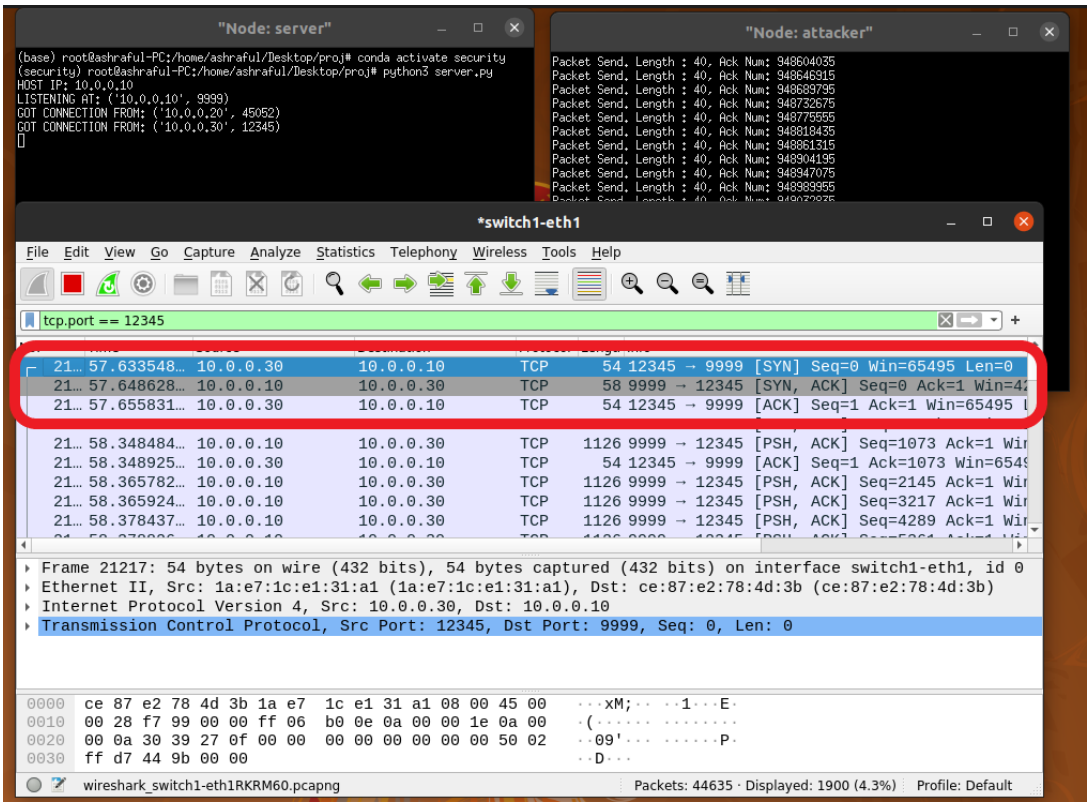


Figure 1.5: Attack Statistics

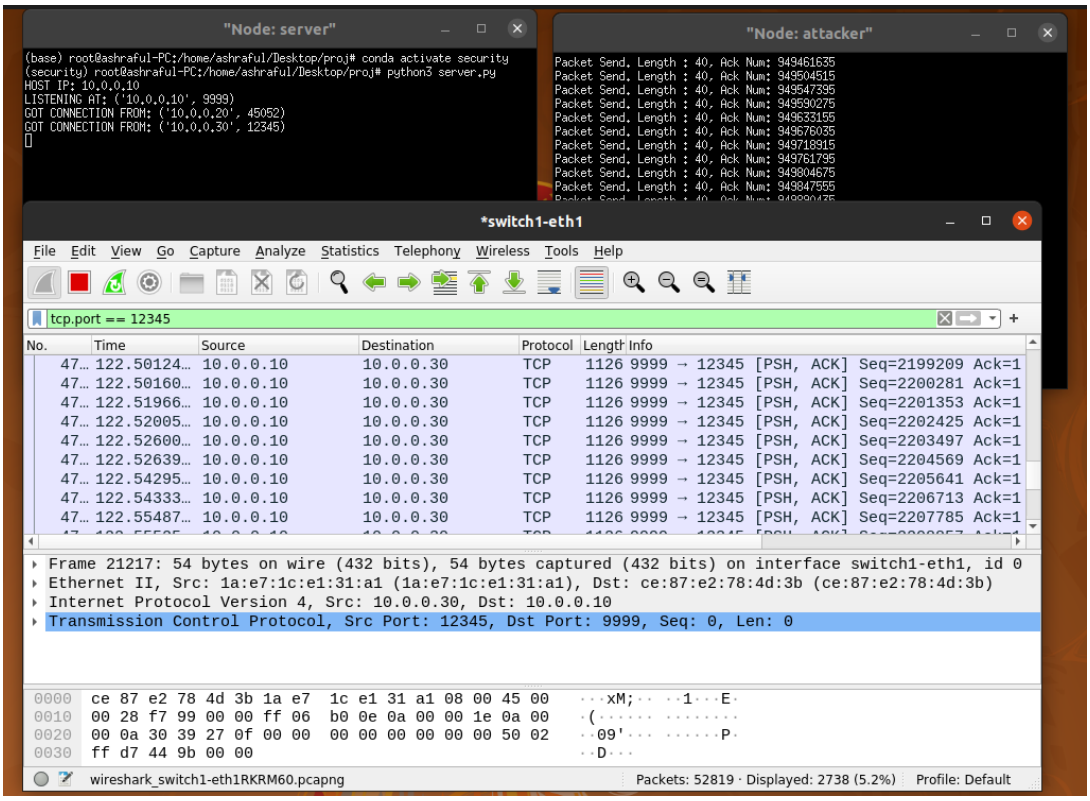


Figure 1.6: Attack Statistics

Now, let us generate some statistics so that we can easily understand that my attack successfully performs in the case of a large number of clients. I have modified my server and client code to easily generate the statistics. I have taken the average bandwidth of the received data by the client in the presence and the absence of the attacker. I have experiment for $number\ of\ clients = \{5, 10, 15, 20, 25, 30\}$. I have plotted the graph using another script and the plot is shown in Figure 1.7. The Figure 1.7 demonstrate that my attack is successful. All my code can be found in this link.

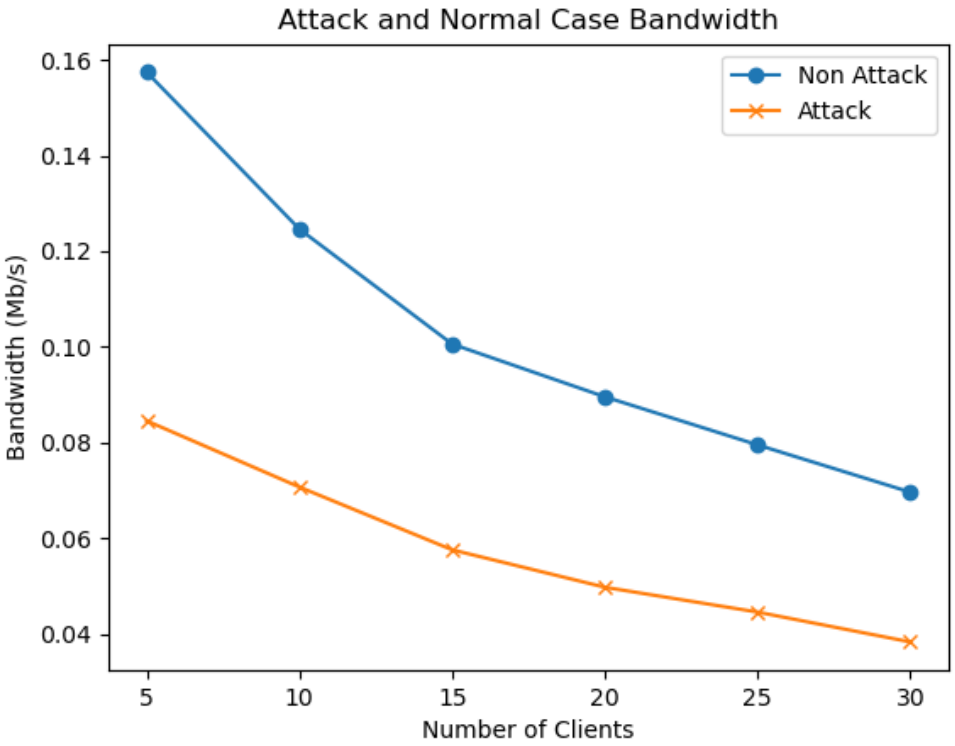


Figure 1.7: Attack Statistics

1.4 Conclusion

1.4.1 Is my attack successful?

Yes, my attack is successful because the attacker successfully reduces the average bandwidth of the legitimate clients. And if we see the Figure 1.7 then we can clearly understand that in the attack case the bandwidth is less than the non-attack case. So, the Figure 1.7 is the evidence of the successful attack.

1.4.2 Challenges

In this project, I have faced so many problems that I need to address to perform this successful attack. They are listed below-

1. I need to work with RAW_SOCKET. It has very little documentation so I need to experiment a lot to get success.

2. When I use RAW_SOCKET Linux kernel automatically sends an RST after SYN+ACK packet of the server. I need to disable this feature to successfully run my code.

3. I need to select the sleep time between two ACK packets so carefully because if it is too big then no advantage of this attack, and if it is too small then an unseen TCP segment will be received by the server.

1.4.3 Countermeasures

A real-life solution to this attack can be mitigated by implementing maximum traffic limits per client at the server level, and by promptly blocking traffic from clients whose traffic patterns indicate denial-of-service attempts.
Another approach proposed by Sherwood et al. [2] to mitigate this vulnerability. In particular, the authors suggest a strategy of randomly dropping segments on the sender. The authors have provided a patch that implements this workaround for Linux 2.4.24 kernel.

Chapter 2

ICMP Blind Connection-Reset + Blind Throughput Reduction Attack against TCP → 1605011

2.1 Definition & Topology Diagram

ICMP Blind Connection Reset:

There are two types of error messages that a TCP connection can receive: hard error and soft error. Whenever TCP receives a hard error, it will terminate the connection. In case of soft error, TCP will log and continue sending data until an acknowledgement is received.
Thus, if we wish to terminate a TCP connection as a third party, we can send any ICMP error message that indicates a hard error to the server or client.

Blind throughput reduction attack against TCP:

Whenever a host receives ICMP source quench messages, the host/ server must slow down the transmission of the connection. This can be utilized by an attacker who can send his/her own forged ICMP source quench message to a host or TCP endpoint to make the host reduce the rate of transmission.

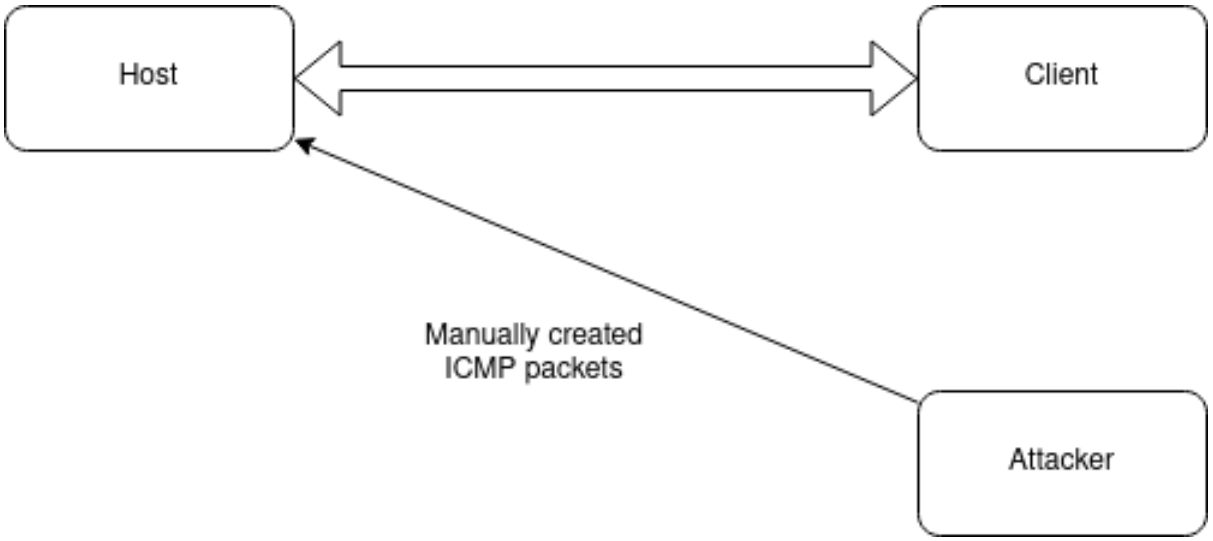


Figure 2.1: Attacking an endpoint using ICMP packets

2.2 Packet Details & Header Modifications

The following figures show the structure of an ICMP packet for both of the mentioned attacks:

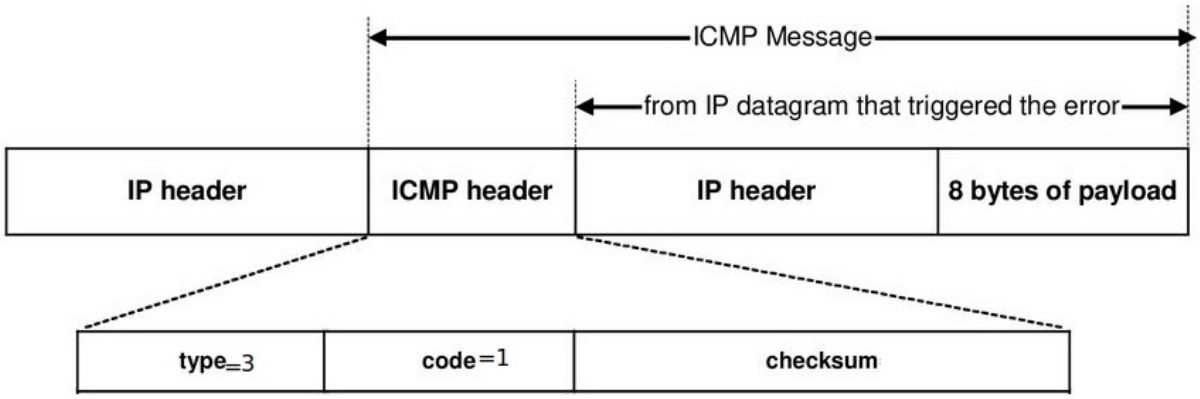


Figure 2.2: ICMP packet for ICMP blind reset

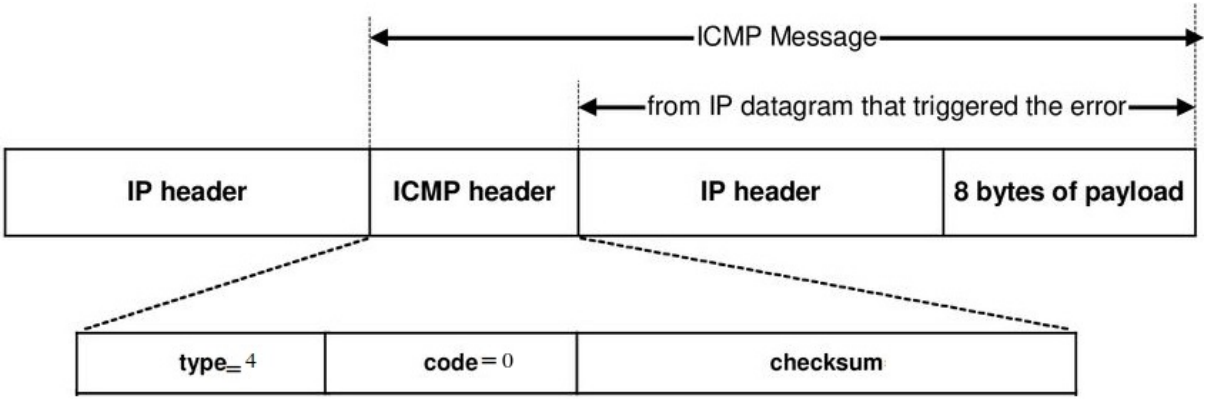


Figure 2.3: ICMP packet for throughput reduction

2.3 Steps of Attack:

ICMP Blind Connection Reset attack against TCP:

I have attempted this attack on two scenarios:

Attack on Java Socket Connection:

I made a simple server-client socket connection using java sockets in two VMs

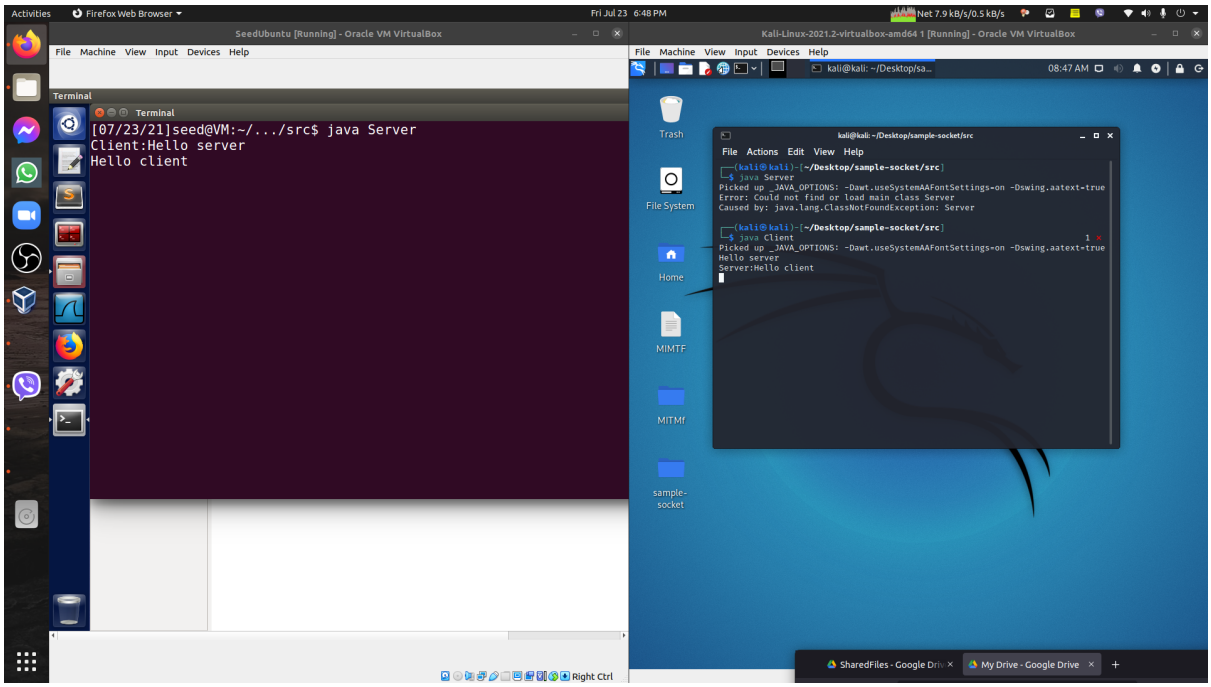


Figure 2.4: Setup of a Java server client communication

And I did the blind connection reset attack from a third VM. The packet traces are shown side by side

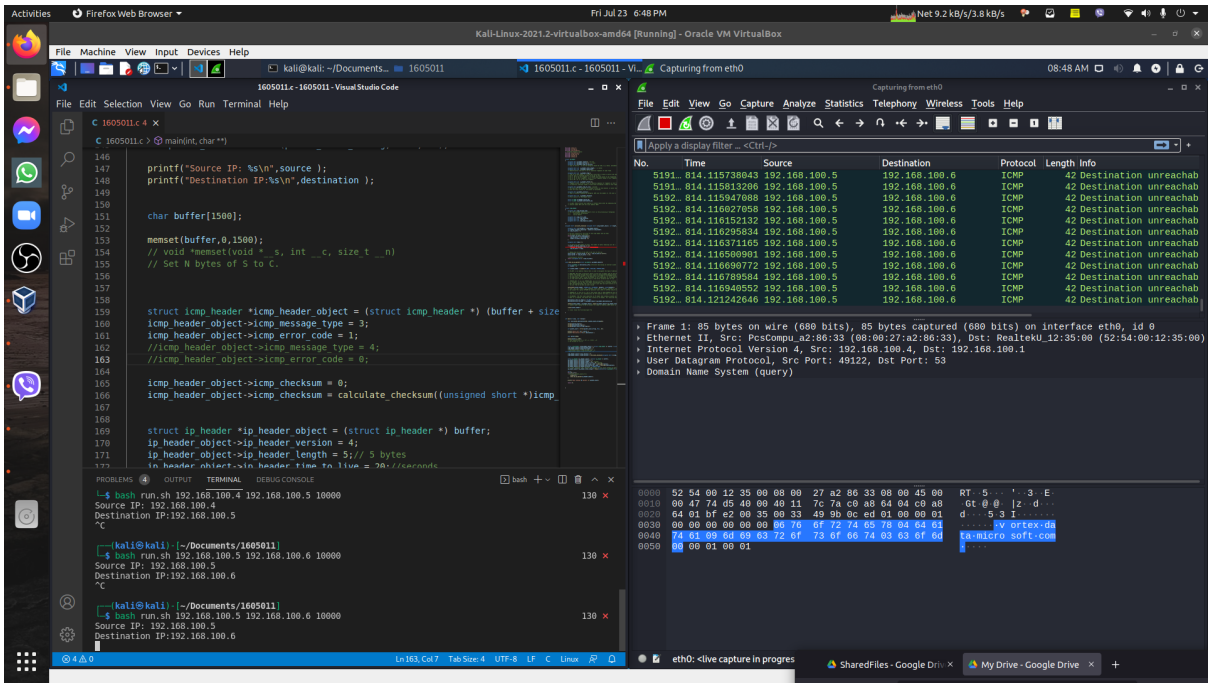


Figure 2.5: Attempt of ICMP blind connection reset

Apparently, the connection was not broken as per the following screenshot

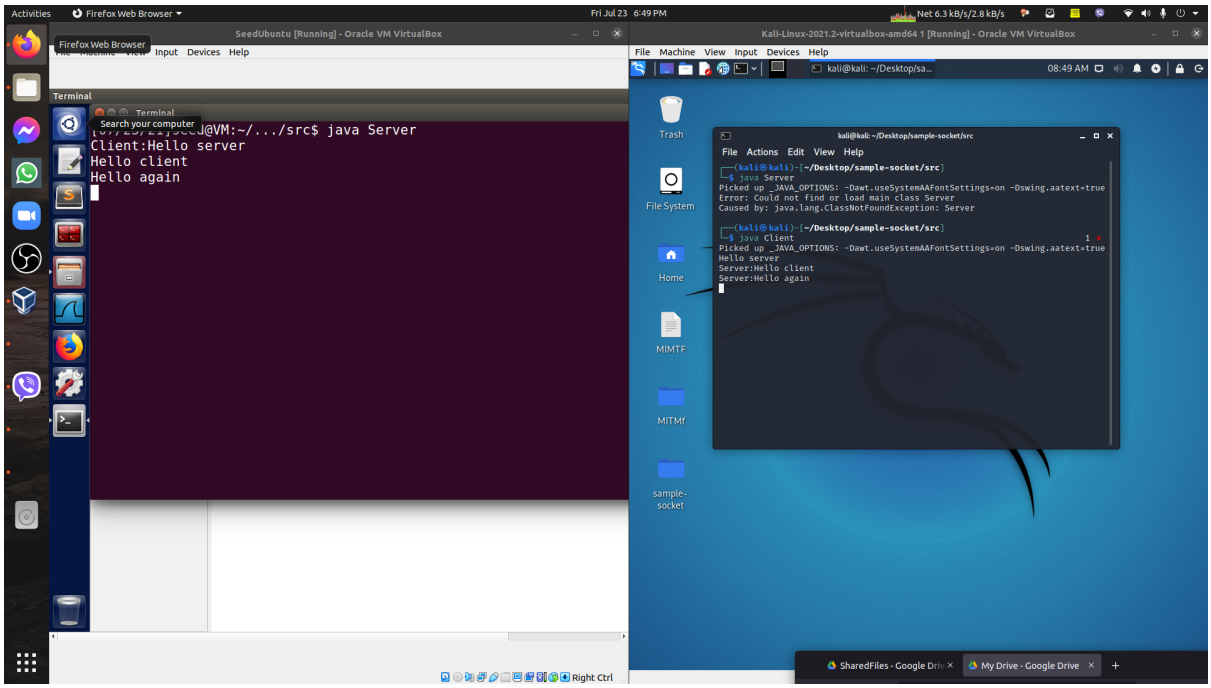


Figure 2.6: Uninterrupted socket communication after attack attempt

Attack During Browsing the Web:

When I visited google.com on the target VM under normal circumstances, the page took only 3 seconds to load. But under this attack, the page inside the browser of the victim

machine took an average of 8 seconds to load.

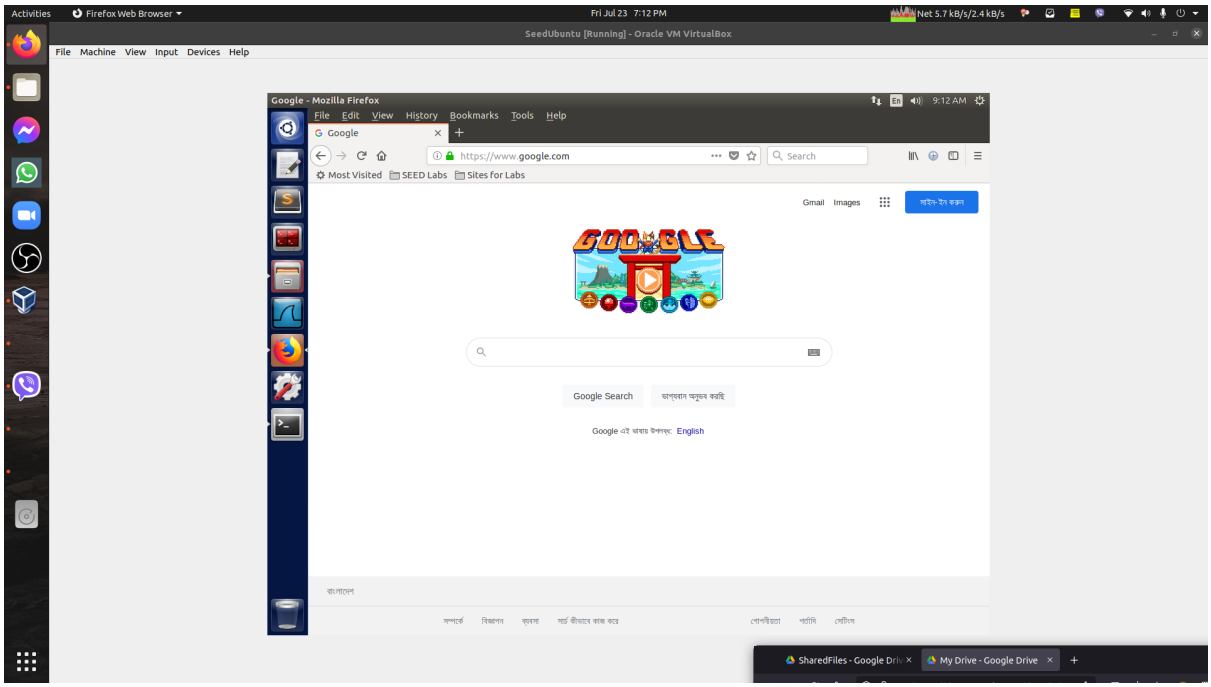


Figure 2.7: Search page of Google loads for a longer period of time under the attack

Blind throughput reduction attack against TCP:

I have tested this attack on two different scenarios:

Attacking on large file download:

I attempted to download a large zip file using wget as the following screenshot:

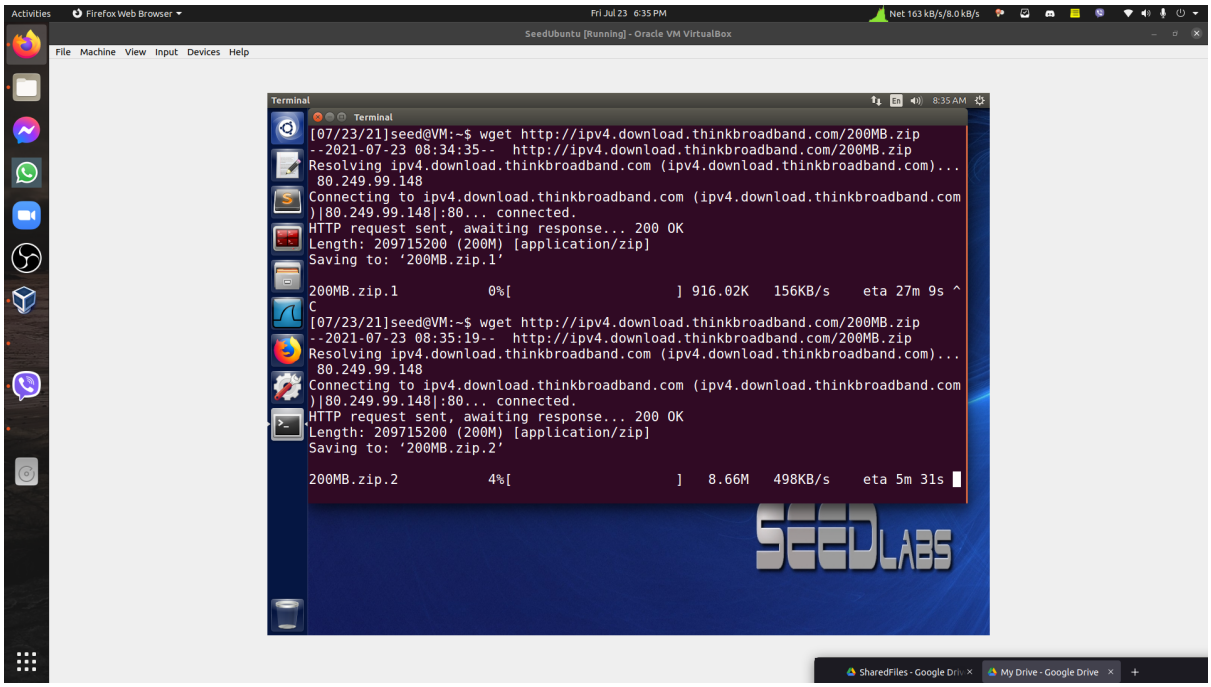


Figure 2.8: Sample download of large file using wget

And then I did the ICMP source quench attack from an attacker VM. The packet traces have been shown side by side.

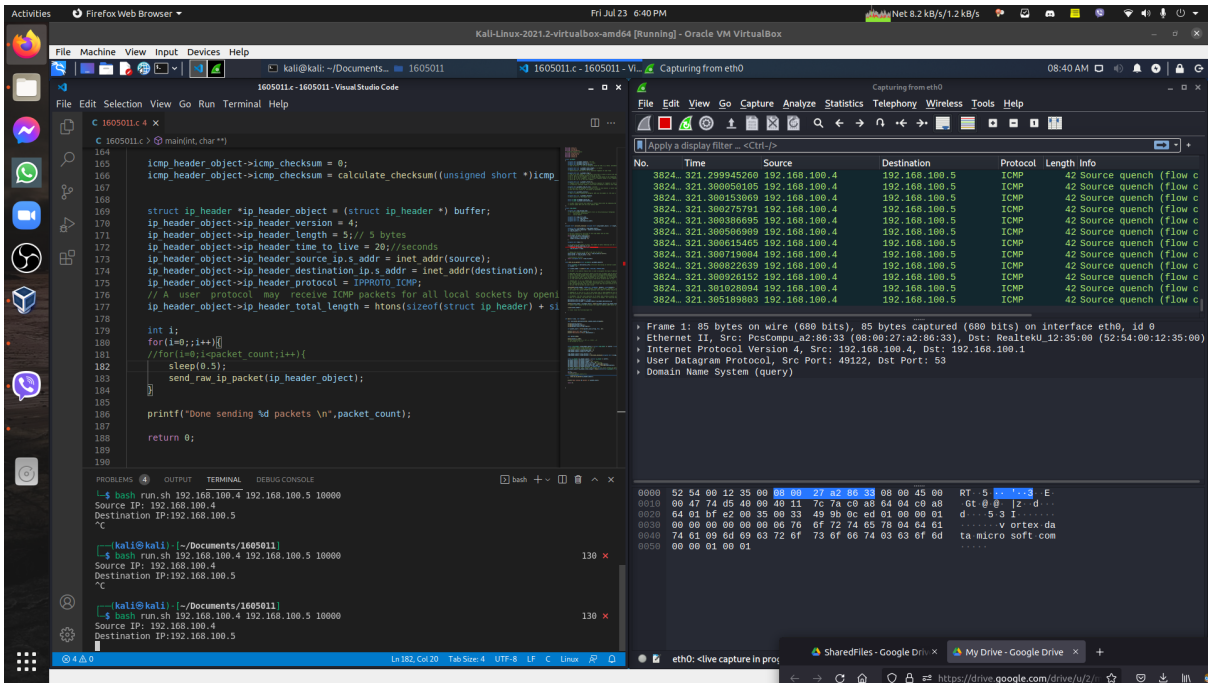


Figure 2.9: Attack attempt to slow down download speed using source-quench attack

Apparently, the internet speed of the target machine did not face any observable disruption.

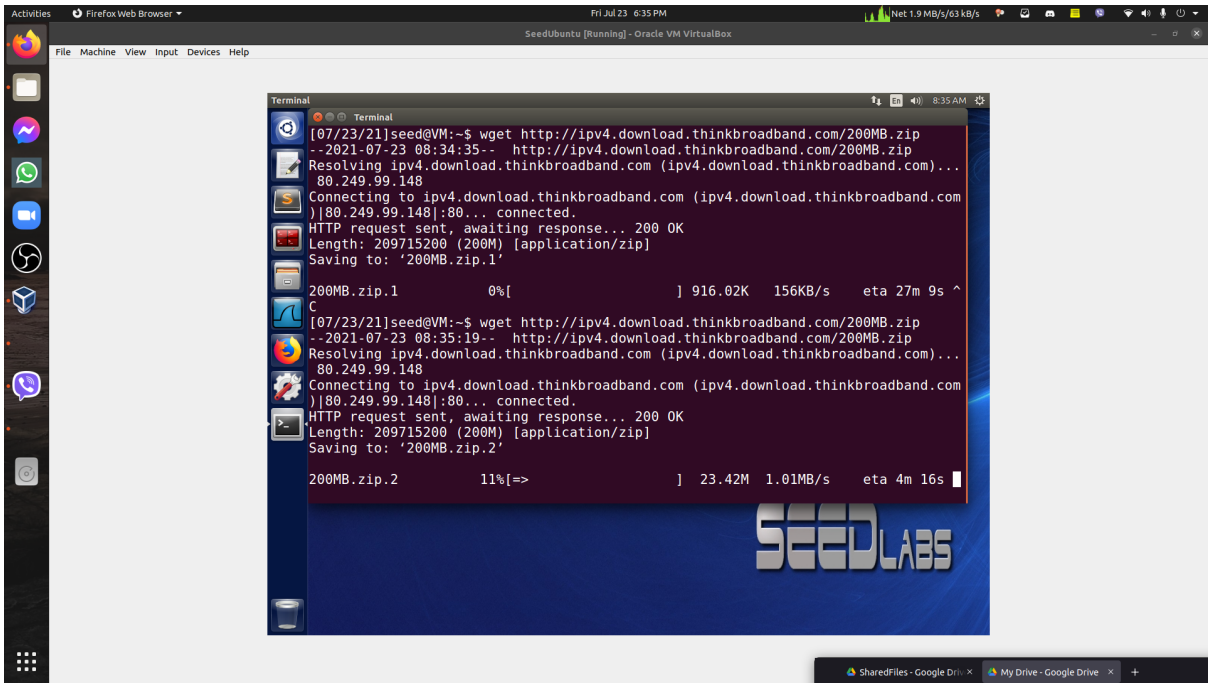


Figure 2.10: Uninterrupted download speed even being under attacked

Attach while browsing the web:

Similar to the ICMP blind reset attack, I did the ICMP blind throughput reduction attack against a browser that was running on the target machine. Under the attack, google.com or any other website takes twice the time to load.

2.4 Justification:

Both of the attacks did not succeed as intended. The issues behind them might be the followings:

- Recent updates to operating systems have prevented the security issues and thus the vulnerabilities are no longer there. The Linux kernel has implemented the protection policies for more than ten years.
- I could not disable the security features of Ubuntu OS that prevent these attacks due to lack of resources.
- The attacks may be incorrectly implemented.

2.5 Counter Measures

I did not implement any counter measures for the ICMP blind connection reset attack and ICMP blind throughput reduction attacks. However, the following countermeasures have already been implemented by the open source community:

2.5.1 Prevention of Blind Reset Attack

- TCP should treat all of the above messages as indicating "soft errors", rather than "hard errors", and thus should not abort the corresponding connection upon receipt of them. This policy is based on the premise that TCP should be as robust as possible.
- A counter-measure could be to delay the connection reset. Rather than immediately aborting a connection, a TCP would abort a connection only after an ICMP error message indicating a hard error has been received, and the corresponding data have already been retransmitted more than some specified number of times.

2.5.2 Prevention of Blind Throughput Reduction Attack

- Research suggests that ICMP Source Quench is an ineffective (and unfair) antidote for congestion. RFC 1812 further states that routers should not send ICMP Source Quench messages in response to congestion.Thus, hosts should completely ignore ICMP Source Quench messages meant for TCP connections.

Chapter 3

DHCP Spoofing → 1605026

3.1 Definition & Topology Diagram

Dynamic Host Configuration Protocol (DHCP) is a network management protocol used to automate the process of configuring devices on IP networks. A DHCP server dynamically assigns an IP address and other network configuration parameters to each device on a network so they can communicate with other IP networks.

DHCP Spoofing Attack:

DHCP Spoofing attack is an attack in which attackers set up a rogue DHCP server and use that to send forged DHCP responses to devices in a network. Attackers often use this attack to replace the IP addresses of Default Gateway and DNS servers and thereby divert traffic to malicious servers.

Topology Diagram:

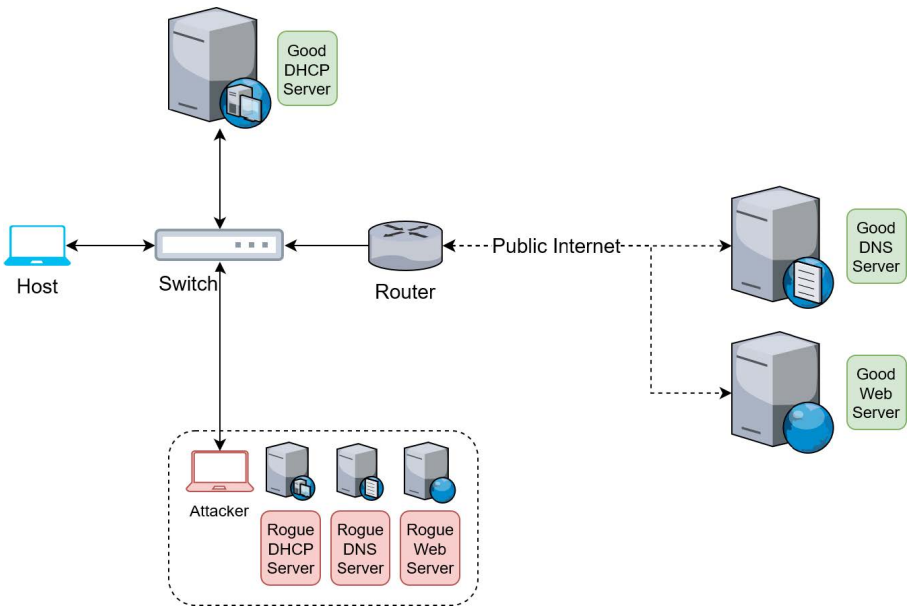
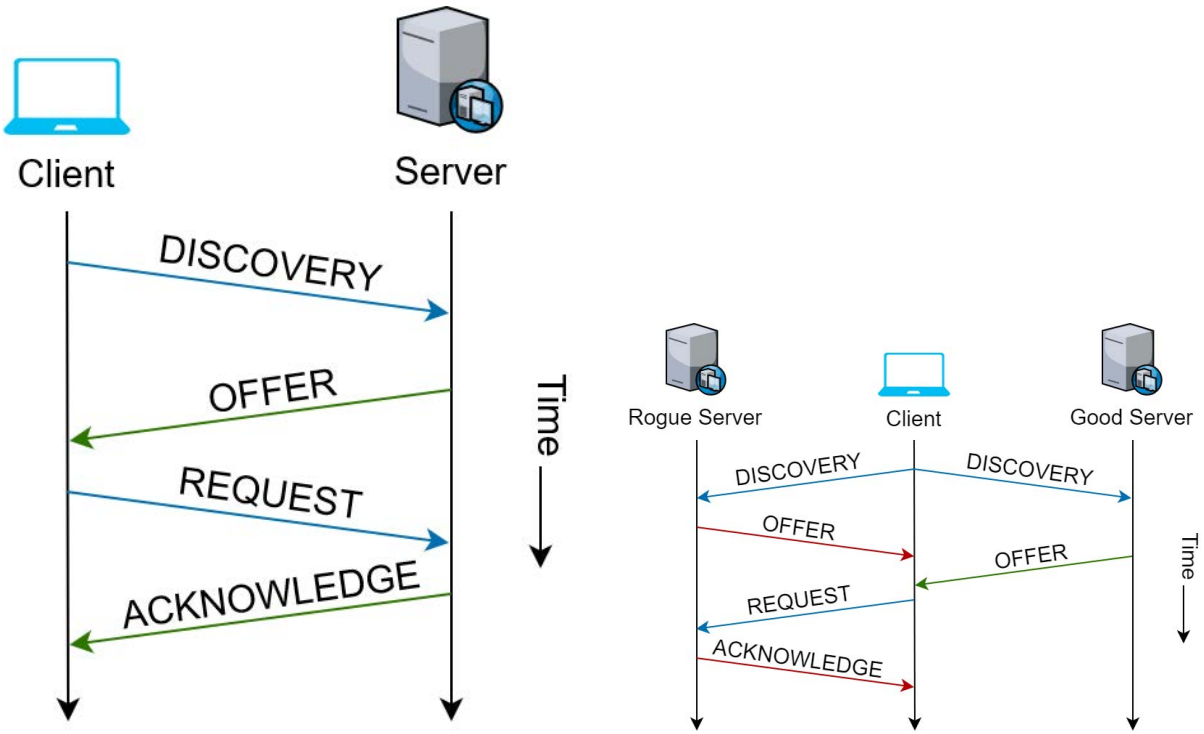


Figure 3.1: Topology Diagram of DHCP Spoofing Attack

3.2 Timing Diagram & Attacking Strategies

First, we'll see the timing diagram of the original protocol and then we'll see the timing diagram for an attack.



(a) Timing Diagram of the original protocol (b) Timing diagram of the attack

Figure 3.2: Timing Diagram of DHCP Spoofing Attack

Strategies:

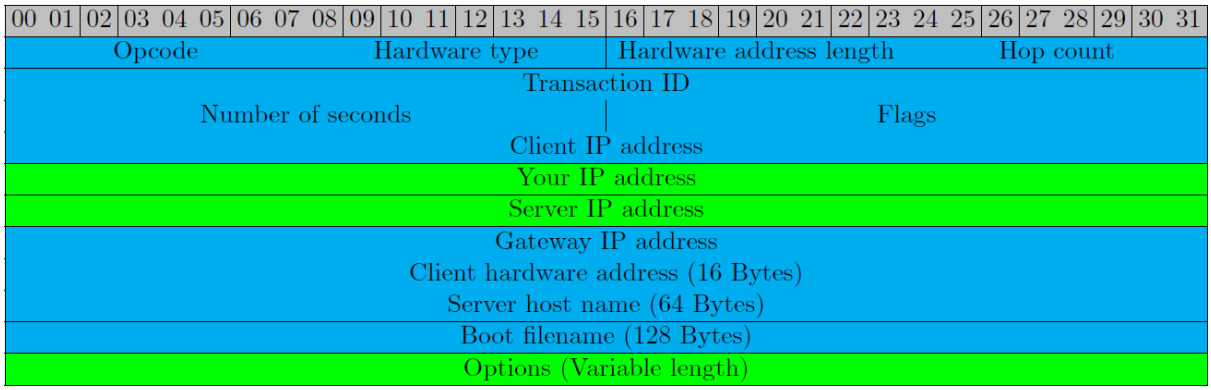
- Set up a rogue DNS server
- Wait for sniffing DHCP discovery packet
- Reply the DHCP discovery packet as early as possible with own IP address as server address and rogue DNS server address as DNS server
- Wait for DHCP request message
- Send DHCP acknowledge

3.3 Packet Details & Header Modifications

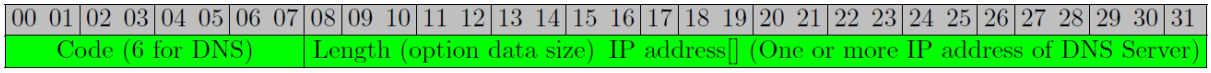
We are only concrened with victim's IP address, Server IP address and DNS IP address in DNS configuration option. We will fill →

- Victim's IP address with an IP from a given range
- Server IP address with IP address of our rogue DHCP server
- DNS IP address with IP address of our rogue DNS server

(a) Structure of a DHCP Frame



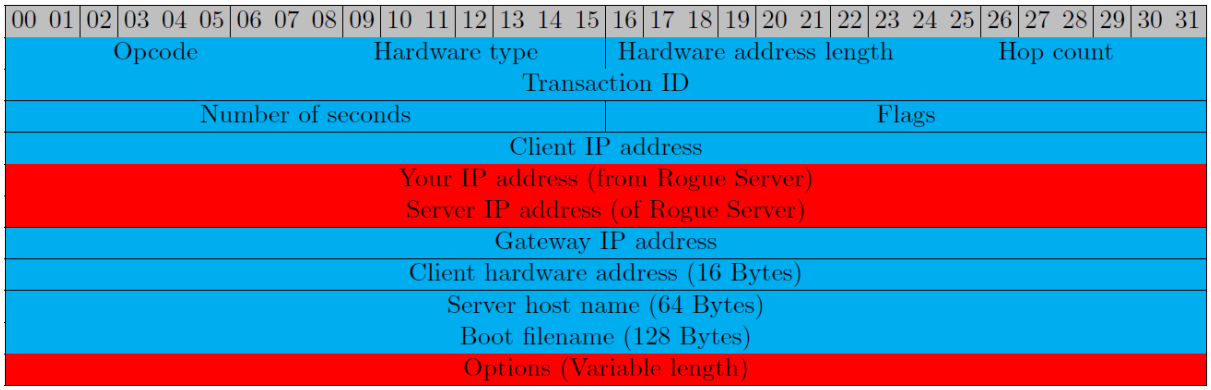
(b) Structure of a DHCP Header



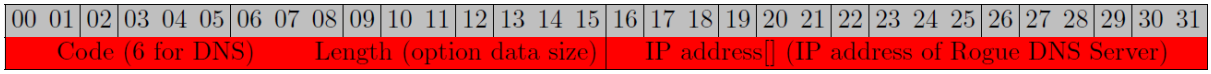
(c) Structure of DNS configuration option of DHCP Header

Figure 3.3: DHCP Protocol

(a) Structure of a DHCP Frame for spoofing



(b) Structure of a DHCP Header for spoofing



(c) Structure of DNS configuration option of DHCP Header for spoofing

Figure 3.4: DHCP Protocol with spoofing

3.4 Network Creation & Steps of Attack

Network Creation

We use mininet to create to create a virtual network. We create a good DHCP server there. Run the file with following command:

```
sudo python goodDhcp.py
```

Then two xterm terminal will be opened. One for rogue DHCP server and another for victim.

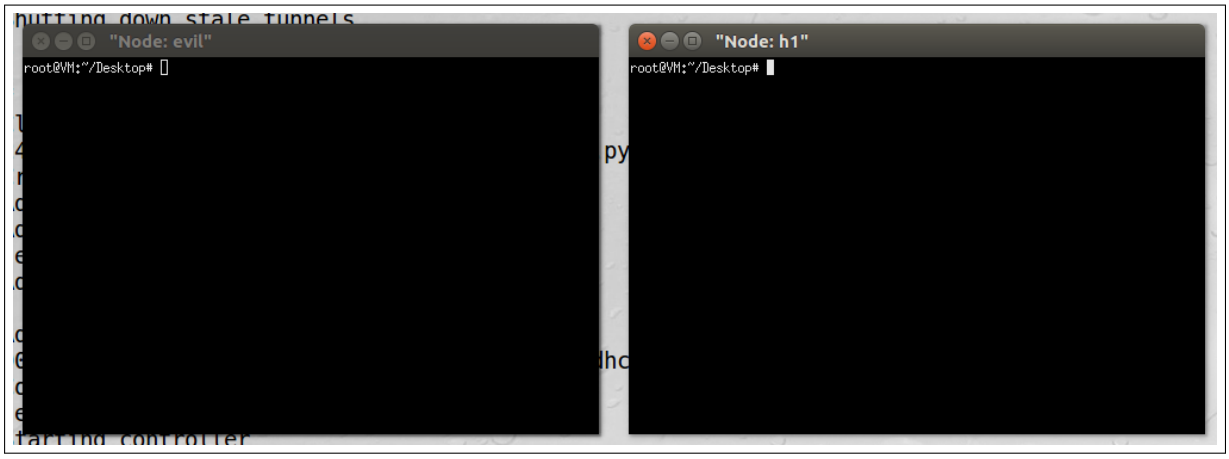


Figure 3.5: Network Setup

Steps of Attack

To implement a rogue DHCP server, we run a c file using following command:

```
gcc attack.c
sudo ./a.out
```

It sniffs the network for DHCP DISCOVER packet. As soon as getting the packet it responses with a DHCP OFFER packet before the good DHCP server.

So, the victim responses with DHCP REQUEST (hence, getting spoofed) and the server sends back a DHCP ACKNOWLEDGE packet. The victim keeps sending DHCP REQUESTs to the server as the lease time gets near to end. The server sends DHCP ACKNOWLEDGEs back for each of them.

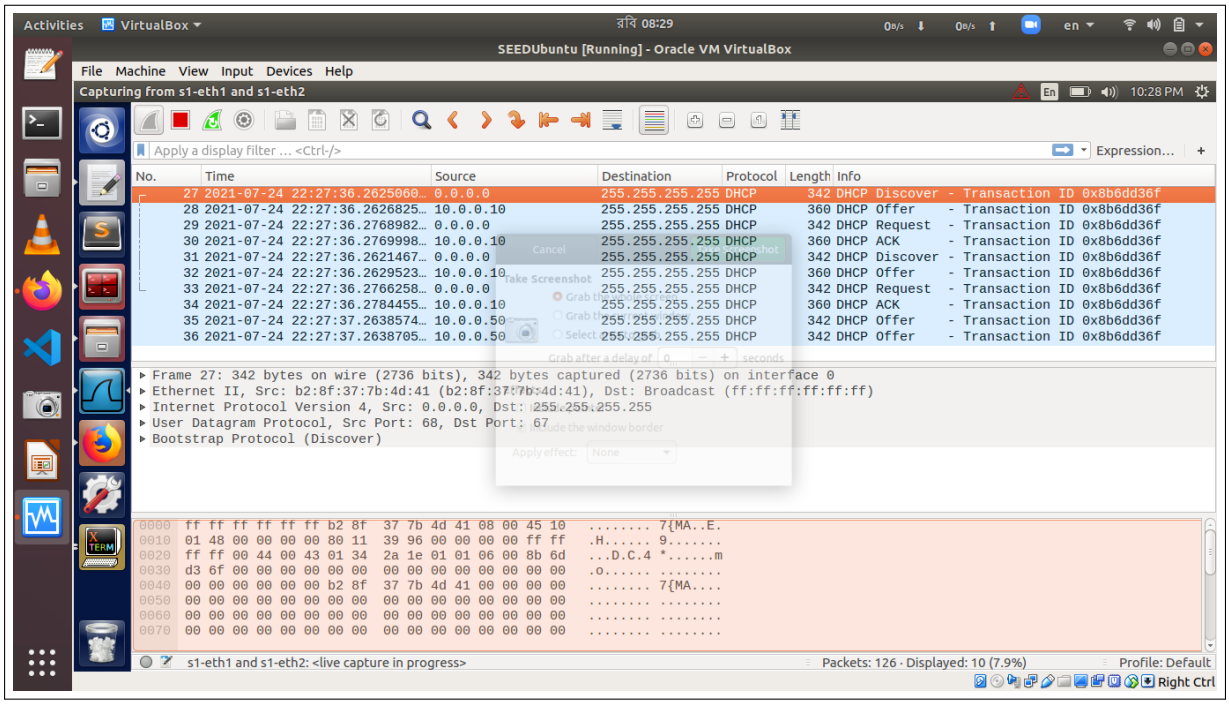
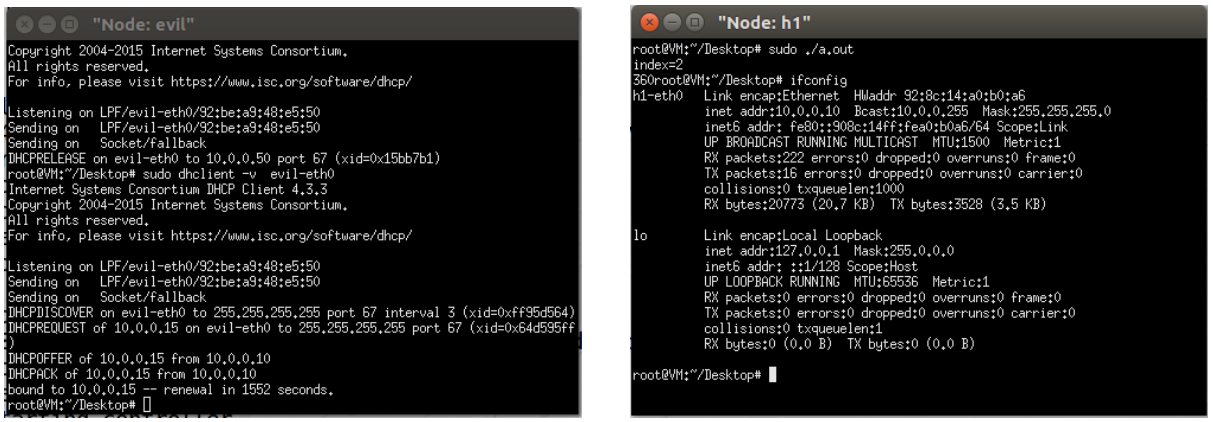


Figure 3.6: Steps of Attack

3.5 Observed Output



(a) Victim's Terminal (b) Attacker's Terminal

Figure 3.7: Observed Output from both Victim & Attacker

Now we can do two types of attacks:

Masquerade: Rogue server can send ip of rogue nameserver to the victim. This rogue nameserver can send the ip of fake website.

Evesdroping: Rogue server can set the default gateway address of victim. So, if attacker set his ip as a default gateway address, each packet sent by victim will go through attacker pc.

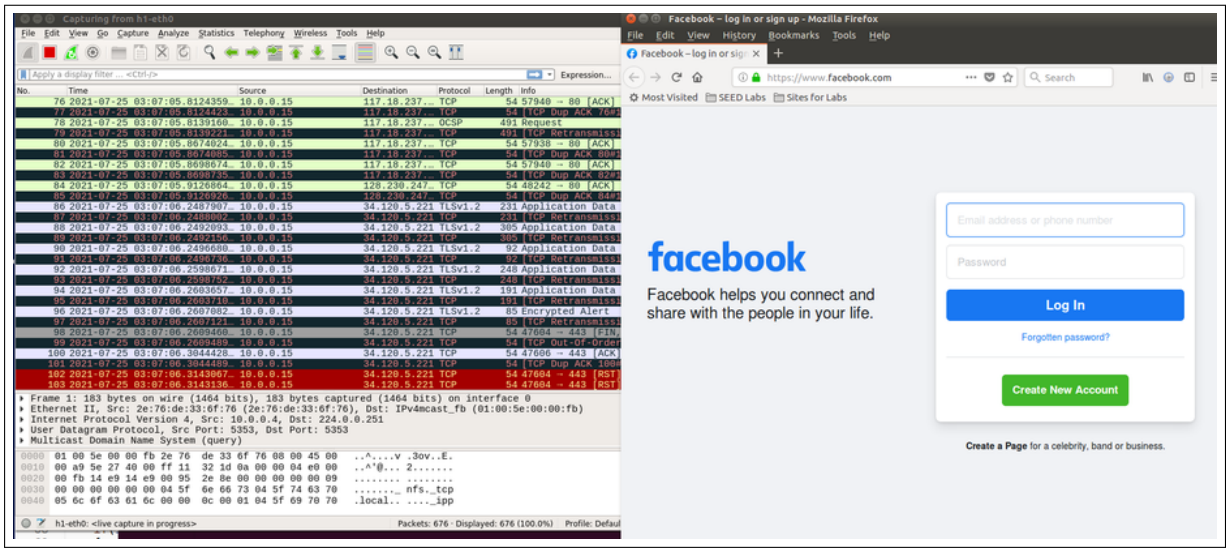


Figure 3.8: Evesdroping

3.6 Was my attack successful? Why?

When a device needs to obtain an IP address, it sends the **DHCP Discover** packet to the broadcast IP address **255.255.255.255**. Packets sent to this IP address are received by all hosts on the network. So, our attack will be able to spoof the DHCP Discover packet. Then we send the IP of a Rogue DNS Server as *only* DNS server. So, when the Host tries to connect to the internet, it uses that Rogue DNS Server. Therefore, our attack will be successful.

3.7 Countermeasures

DHCP Snooping is the best mechanism to avoid this kind of attacks. The idea of this feature is to differentiate between two types of ports in a switched environment: a reliable port side (trusted port) and, secondly, untrusted ports (untrusted host). The first have no restrictions on the type of DHCP messages that can receive, as they will be those connected to a controlled environment (in this case the server / DHCP servers). However, the latter can only send those packets that under normal conditions a client needs to send to get its DHCP configuration (DHCP Discover, DHCP Request, DHCP Release). Therefore, the untrusted ports will correspond to those ports connected to end users, and in the case that one of those ports receives a spoofed DHCP offer packet or a DHCP ack (as in our case), it will be blocked.

Chapter 4

ARP Cache Poisoning with Man in the Middle Attack → 1505004

4.1 Definition & Topology Diagram

ARP stands for *Address Resolution Protocol*. It's designed to discover MAC addresses and then map them to corresponding IP addresses.

When computers communicate, they primarily use IP addresses. But before they can send any data to the *layer 2* switched network, they first need to find the MAC address of the destination.

To do this, they use ARP requests to shout out to the entire network asking **"who is `192.168.0.105`? Tell me your MAC address"**. All other computers will ignore this request except for `192.168.0.105` who will respond with its own MAC address.

Sender computer will then take note of this MAC address and associated IP address & keep them in its ARP cache for future use to increase efficiency.

```
C:\Users\User>arp -a

Interface: 192.168.0.102 --- 0x9
  Internet Address      Physical Address      Type
  192.168.0.1           cc-32-e5-03-c7-89     dynamic
  192.168.0.105         08-00-27-d5-05-a3     dynamic
  192.168.0.107         08-00-27-e0-aa-9f     dynamic
  192.168.0.108         08-00-27-37-e9-0c     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Figure 4.1: A Normal ARP cache

How ARP Protocol works:

Before Jumping on to the description of *ARP cache poisoning*, let's first refresh how ARP protocol works. ARP protocol mainly consists of the following 4 basic messages:

ARP request: Computer 'A' asks on the network, "who has this IP?"

ARP reply: All the other computers ignore the request except the computer which has the requested IP. This computer, let's say 'B' says, I have the requested IP address and here is my MAC address.

RARP request: This is more or less same as ARP request, the difference being that in this message a MAC address is broad-casted on network.

RARP reply: Same concept. Computer 'B' tells that the requested MAC is mine and here is my IP address.

All the devices that are connected to that network have an ARP cache. This cache contains the mapping of all the MAC and IP address for the network devices this host has already communicated with.

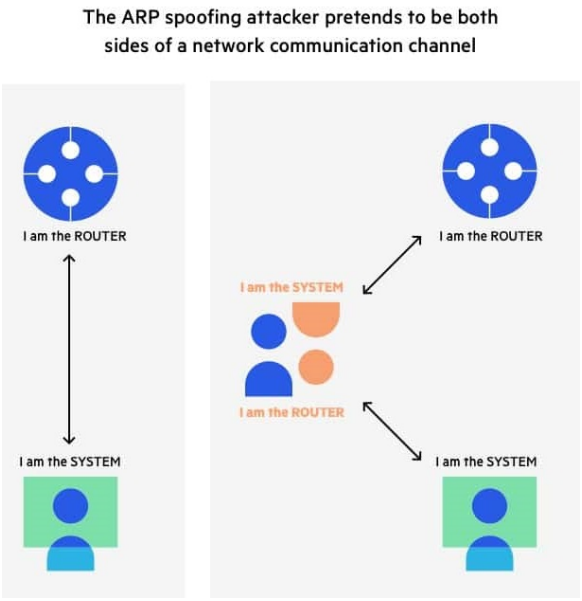
ARP protocol was designed to be simple and efficient but ***a major flaw* in the protocol is lack of authentication**. No authentication was added to its implementation and as a result, there is no way to authenticate the IP to MAC address mapping in the ARP reply. ***Further***, the host does not even check whether it sent an ARP request for which it is receiving ARP reply message. And, that leads to...

ARP Cache Poisoning:

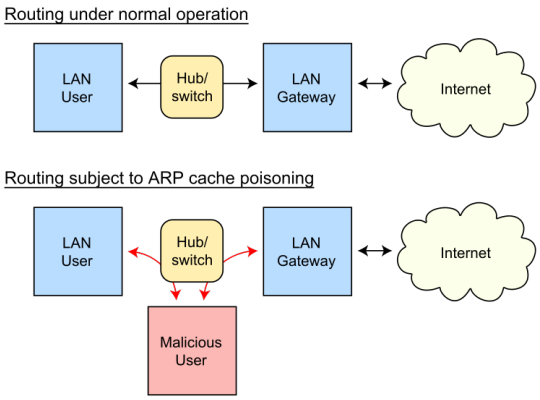
If computer 'A' has sent an ARP request and it gets an ARP reply, then ARP protocol by no means can check whether the information or the IP to MAC mapping in the ARP reply is correct or not. ***Also, even if a host did not send an ARP request*** and gets an ARP reply, then also it trusts the information in reply and updates its ARP cache.

Suppose 'A' and 'B' are very good friends and 'A' shares all his secrets with 'B'. Now if a guy 'C' comes in and fakes as if he is 'B', can you imagine what could happen? Yes, 'A' could tell all his secrets to 'C' and 'C' could *misuse* it.

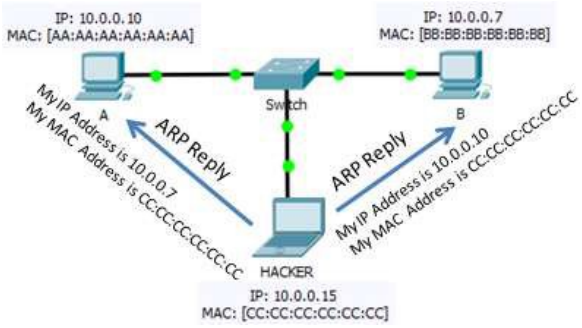
In a layman's language, this is what we mean by ***ARP cache poisoning***.



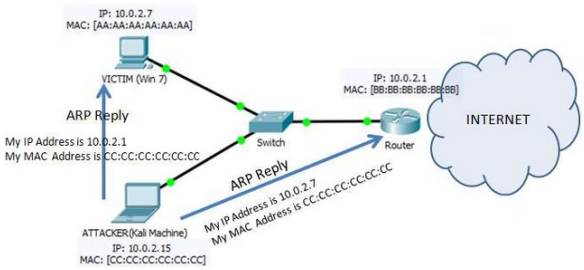
(a) Attacker as both system & router



(b) Routing before & after cache poisoning



(c) Attacker poisoning two victim's ARP cache



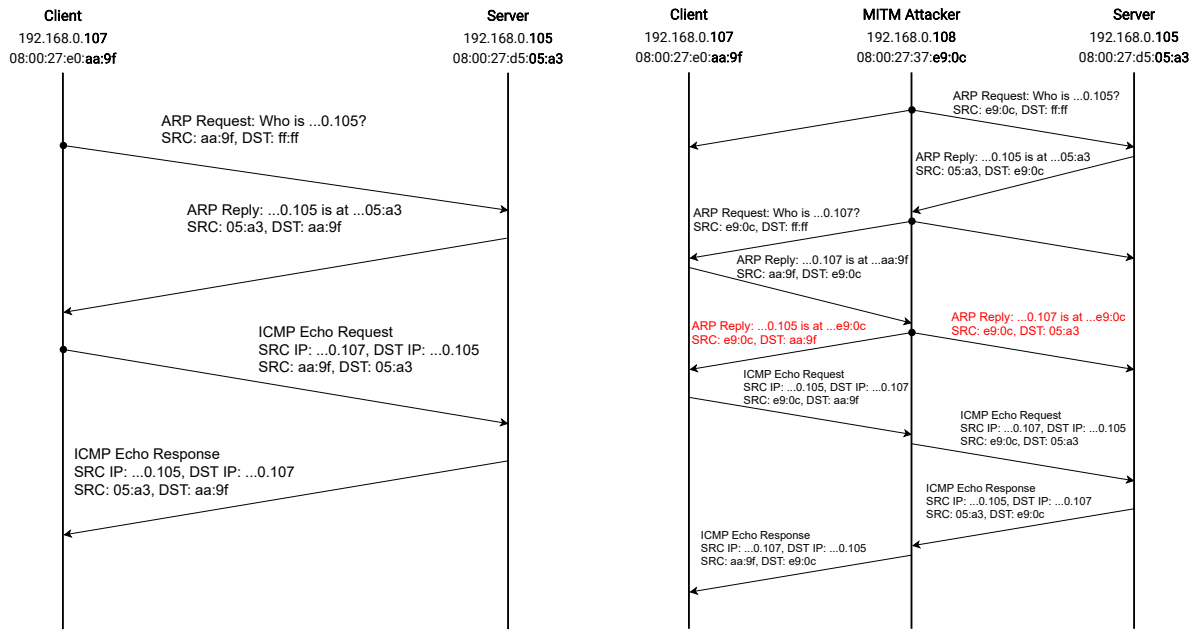
(d) Attacker poisoning Router's ARP cache

Figure 4.2: Topology Diagram of ARP Cache Poisoning Attack

So, its easy to exploit this weakness of ARP protocol. An evil hacker can craft a valid ARP reply in which any IP is mapped to any MAC address of the hackers choice and can send this message to the complete network. All the devices on network will accept this message and will update their ARP table with new Information and this way the hacker can gain control of the to and fro communication from any host in network.

4.2 Timing Diagram

First, we'll see the timing diagram of the original protocol and then with a MITM attacker poisoning the ARP cache of the server & client.



(a) Timing Diagram of the original protocol (b) Timing diagram of the attack

Figure 4.3: Timing Diagram of ARP cache poisoning with MITM Attack

4.3 Packet Details & Header Modifications

Internet Protocol (IPv4) over Ethernet ARP packet

Octet offset	0	1
0	Hardware type (HTYPE)	
2	Protocol type (PTYPE)	
4	Hardware address length (HLEN)	Protocol address length (PLEN)
6	Operation (OPER)	
8	Sender hardware address (SHA) (first 2 bytes)	
10	(next 2 bytes)	
12	(last 2 bytes)	
14	Sender protocol address (SPA) (first 2 bytes)	
16	(last 2 bytes)	
18	Target hardware address (THA) (first 2 bytes)	
20	(next 2 bytes)	
22	(last 2 bytes)	
24	Target protocol address (TPA) (first 2 bytes)	
26	(last 2 bytes)	

Figure 4.4: Ethernet ARP Packet Details

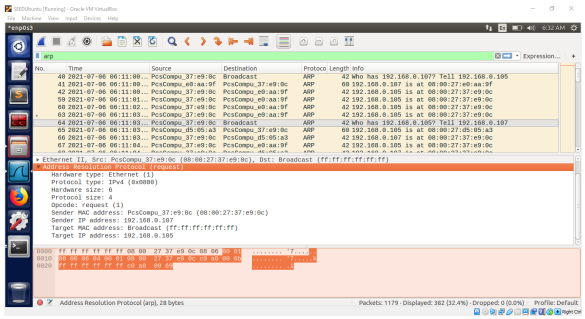
To poison Client's ARP cache, in the Ethernet ARP packet given above, I need to set `OPER = 2` for `ARP_Reply` and `SHA` to attacker's MAC Address, `SPA` to Server's Protocol Address, `THA` to Client's MAC Address, `TPA` to Client's Protocol Address. Then, similar but opposite thing to poison Server's ARP cache.

Well, how am I going to know the MAC Addresses of Server & Client? For that, firstly I need to send two well-crafted ARP broadcast request (`OPER = 1` for `ARP_Request`) to Server & Client to obtain their MAC Addresses from their corresponding replies. Please refer to figure 4.5a, 4.5b, 4.5c, 4.5d, for detailed clarification of the packets.

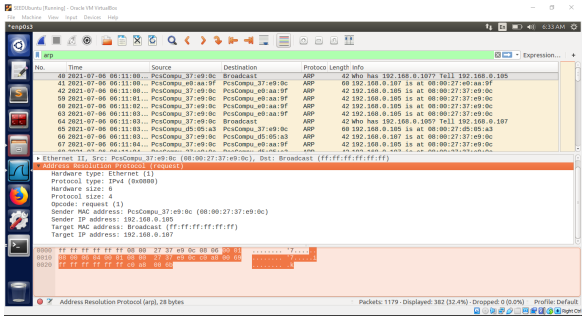
4.4 Attacking Strategies & Steps of Attack

In order to capture both parties' MAC Address, first I'll send a (broadcast) ARP Request to the network asking *"who has that IP address?"* and then I'll listen for any ARP Reply coming from the target in order to extract and obtain his MAC address.

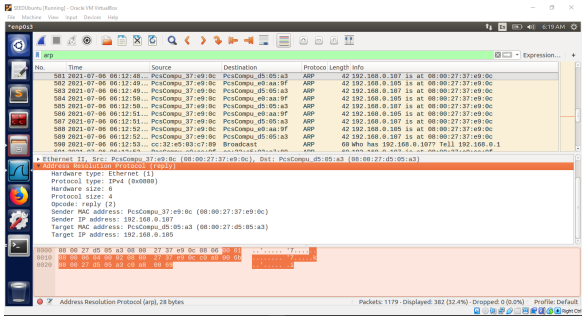
Then, to be able to perform a MITM attack by *(ab)using* the ARP protocol, spoofed ARP Reply packets need to be sent to both parties. I'll set the source IP address to the IP address of the *other party* ***but* source MAC address to attacker's (my) MAC address**. Then by sending ***unicast* ARP replies** to both victims, I will be able to associate on those machines the *other victim's IP* with *my MAC address* and ***thus rendering myself into a Middleman of all their communications***.



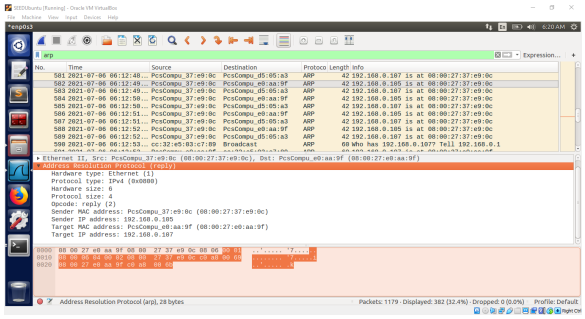
(a) ARP request to get Server's MAC Address



(b) ARP request to get Client's MAC Address



(c) Forged ARP Reply to poison Server's cache



(d) Forged ARP Reply to poison Client's cache

Figure 4.5: ARP request & reply packets for ARP cache poisoning

I have targeted the Dynamic entries of the ARP cache as my victims. Also, as dynamic entries are not permanent rather they are flushed out periodically, I have planned to send forged ARP replies frequently to both Server & Client so that their ARP caches *remain poisoned*.

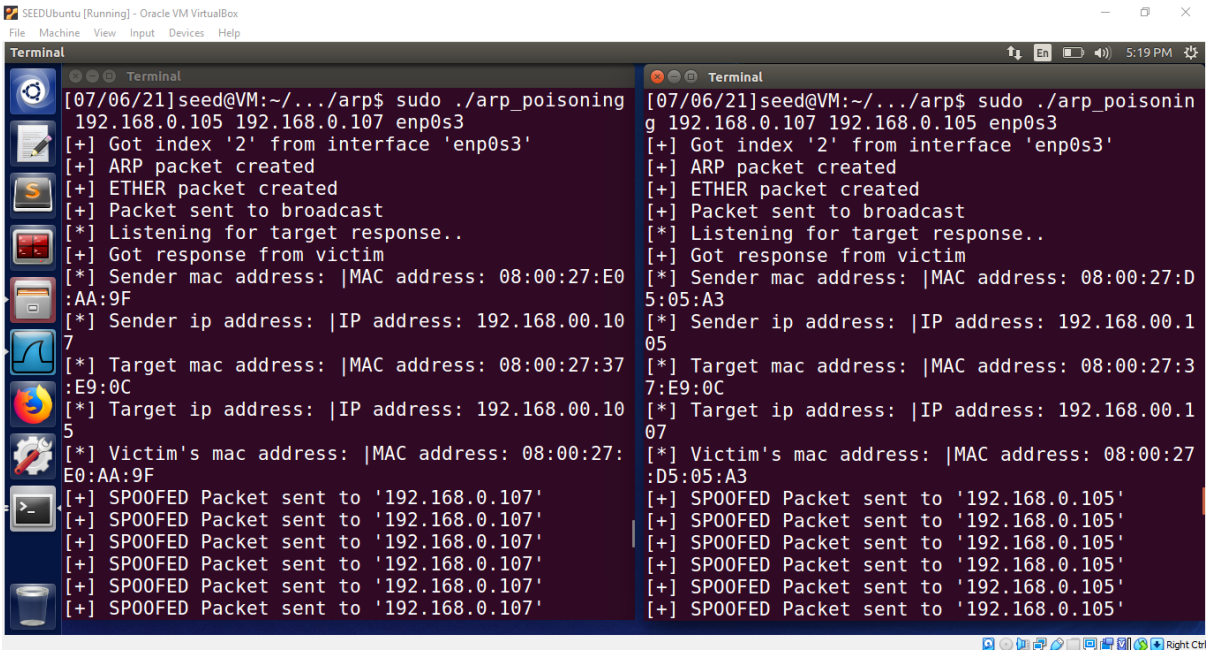


Figure 4.6: Frequently sending forged ARP replies to avoid getting flushed out

Steps

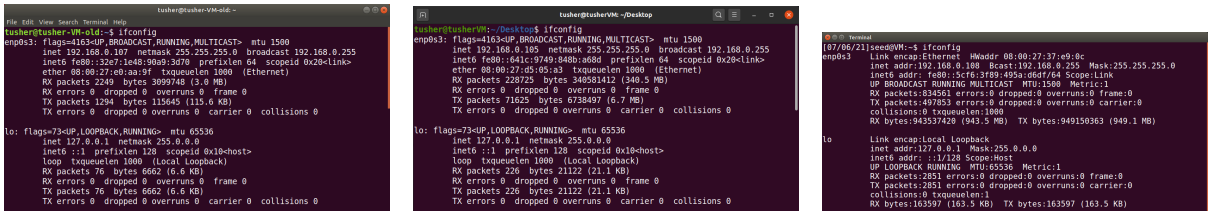
- Get Hardware Address
- Get My MAC Address
- Get Index From Interface
- Create ARP Packet
- Create Ethernet Packet
- Send Packet to Broadcast
- Get Victim's Response & Extract it's MAC Address
- Send Payload to Victim to Poison ARP Cache & keep poisoning
- Sniff Packets, Steal Data & Alter as required

```
gcc arp_poisoning.c -o arp_poisoning
sudo rm ./mac.txt
sudo sysctl net.ipv4.ip_forward = 0
sudo ./arp_poisoning 192.168.0.105 192.168.0.107 enp0s3
sudo ./arp_poisoning 192.168.0.107 192.168.0.105 enp0s3
gcc sniffer.c -o sniffer && sudo ./sniffer
```

4.5 Observed Output in Victim's & Attacker's PC

Configuration

- **Server IP:** 192.168.0.105, **Server MAC:** 08:00:27:d5:05:a3
- **Client IP:** 192.168.0.107, **Client MAC:** 08:00:27:e0:aa:9f
- **Attacker IP:** 192.168.0.108, **Attacker MAC:** 08:00:27:37:e9:0C



(a) Configuration of Client (b) Configuration of Server (c) Configuration of Attacker

Figure 4.7: Configuration of Client, Server & Attacker

VM Setup

- Linux (Ubuntu 20.04 LTS) as Server
- Linux (Ubuntu 18.04 LTS) as Client
- SEEDUbuntu (Ubuntu 16.04 from Seed Labs) as Attacker

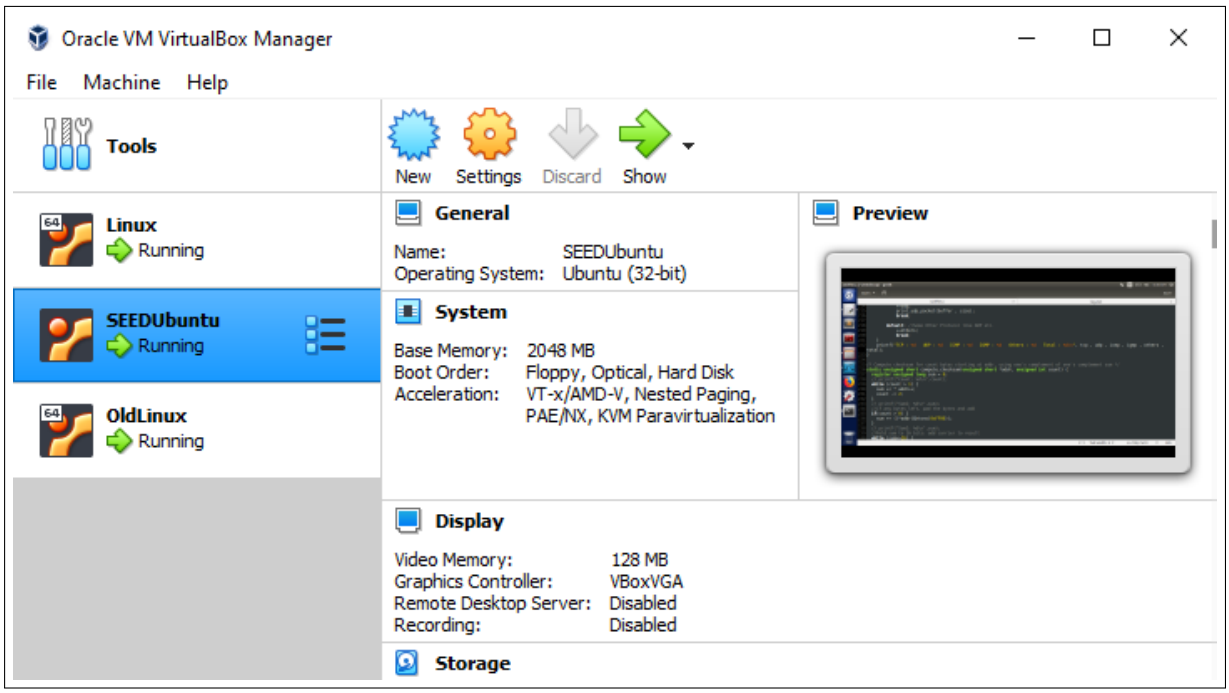
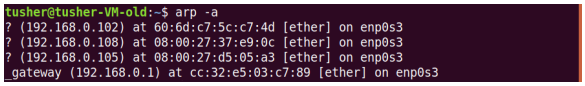
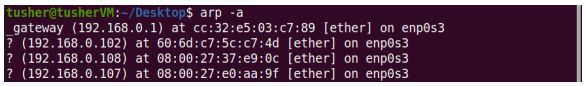


Figure 4.8: Virtual Machines Setup

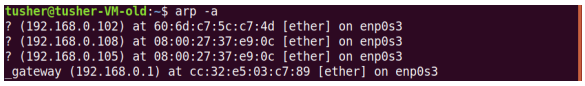
ARP Cache Poisoning



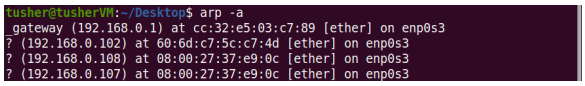
(a) ARP cache of Client *before* attack



(b) ARP cache of Server *before* attack



(c) ARP cache of Client *after* attack

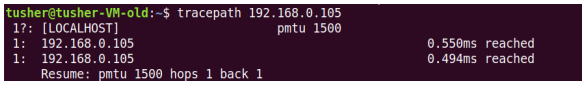


(d) ARP cache of Server *after* attack

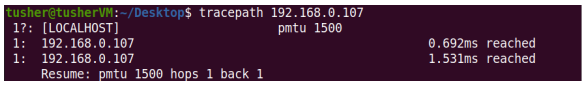
Figure 4.9: ARP caches of Client & Server *before* & *after* attack

Man In The Middle

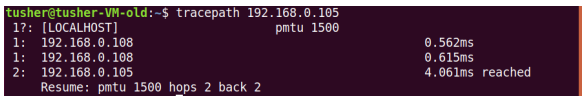
Tracepath



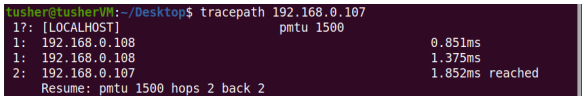
(a) Normal Communication Client → Server



(b) Normal Communication Server → Client



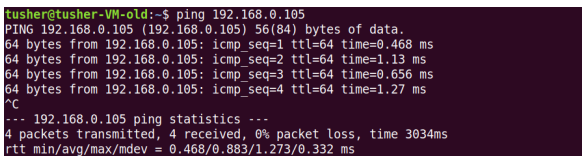
(c) Attacker in Middle from Client → Server



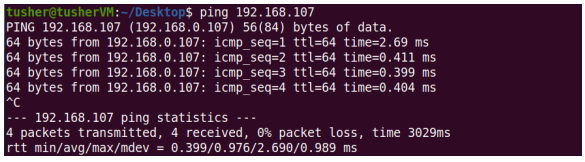
(d) Attacker in Middle from Server → Client

Figure 4.10: `tracepath` from both Client & Server *before* & *after* attack

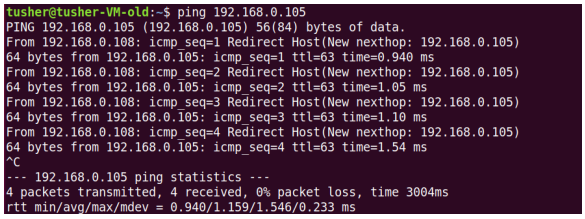
Ping



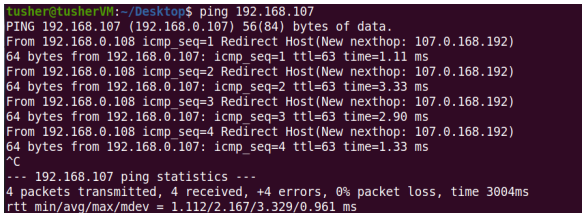
(a) Normal Communication Client → Server



(b) Normal Communication Server → Client



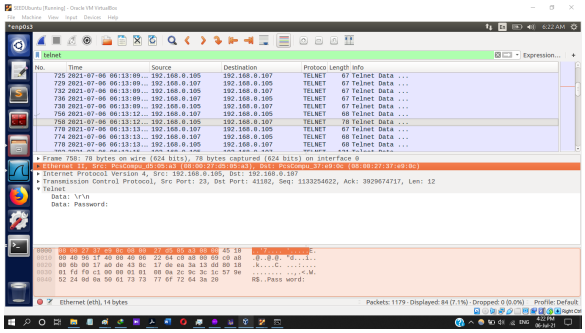
(c) Attacker in Middle from Client → Server



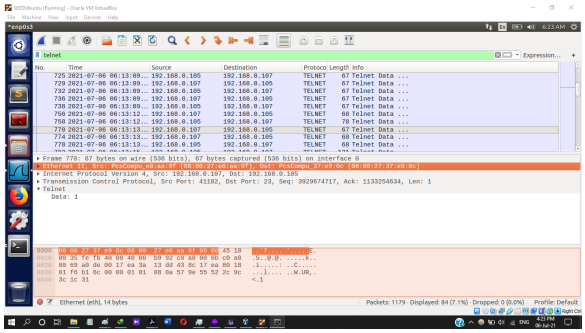
(d) Attacker in Middle from Server → Client

Figure 4.11: `ping` from both Client & Server *before* & *after* attack

Stealing Sensitive Information



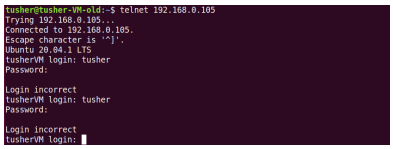
(a) Server sending data to attacker



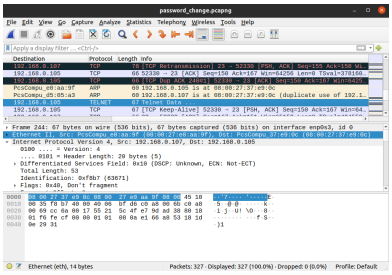
(b) Client sending Password to attacker

Figure 4.12: Attacker has become the Destination MAC Address for both server & client

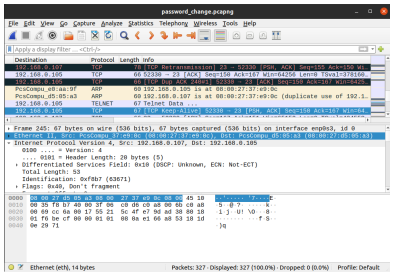
Altering Data



(a) Login Incorrect! ***or, is it?***

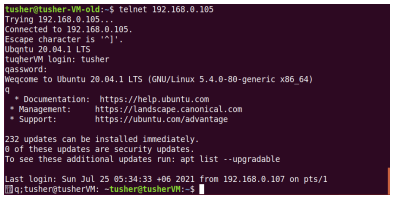


(b) Client sending Attacker **1**



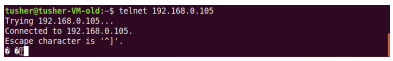
(c) Attacker sending Server **q**

Figure 4.13: Client is unable to login even after typing correct password



(a) **Ubqntu, qassword**; *what even are they?*

(b) Total echo!



(c) Abrupt halt from Server

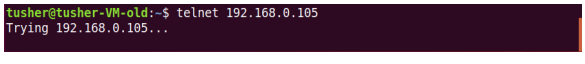
Figure 4.14: *Seemingly* weird behaviors from Server

This Code Snippet is responsible for all these chaos.

Stopping Communication (DoS)

```
sudo sysctl net.ipv4.ip_forward = 0
```

(a) Setting `ip_forward = 0` as `root`



(b) Server is *seeming* to be down

Figure 4.15: Stopping the communication altogether between server & client

4.6 Implementation Details

ARP Cache Poisoning part → `arp_poisoning.c`

Setting up Constants

Source Code 4.1: Setting up Constants

Printing Utility

Source Code 4.2: Printing Utility

Error Messages

Source Code 4.3: Error Messages

Packet Structure

Source Code 4.4: Packet Structure

Get Hardware Address

Source Code 4.5: Get Hardware Address

Get My MAC Address

Source Code 4.6: Get My MAC Address

Get Index From Interface

Source Code 4.7: Get Index From Interface

Create ARP Packet

Source Code 4.8: Create ARP Packet

Create Ethernet Packet

Source Code 4.9: Create Ethernet Packet

Send Packet to Broadcast

Source Code 4.10: Send Packet to Broadcast

Get Victim's Response

Source Code 4.11: Get Victim's Response

Send Payload to Victim to Poison ARP Cache & keep poisoning

Source Code 4.12: Send Payload to Victim to Poison ARP Cache & keep poisoning

Main Function of `arp_poisoning.c`

Source Code 4.13: Main Function of `arp_poisoning.c`

Man In The Middle Attack part → `sniffer.c`

Calculating IP Checksum

Source Code 4.14: Calculating IP checksum

Calculating TCP Checksum

Source Code 4.15: Calculating TCP checksum

Should I Forward or Change?

Source Code 4.16: Calculating TCP checksum

Set Source & Destination appropriately before Forwarding

Source Code 4.17: Calculating TCP checksum

Decreasing `TTL` by 1

Source Code 4.18: Calculating TCP checksum

Altering Data before forwarding

Source Code 4.19: Altering Data before forwarding

Process Packets according to Protocol

Source Code 4.20: Process Packets according to Protocol

Main Function of `sniffer.c`

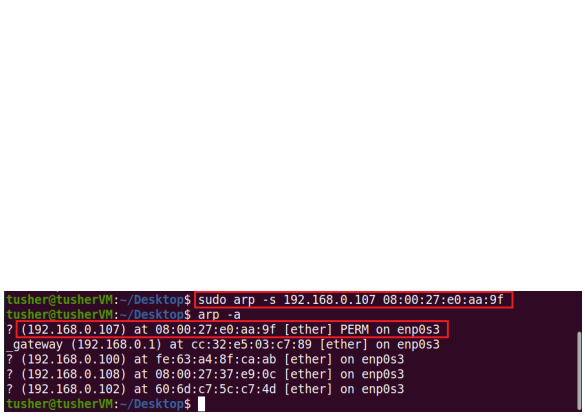
Source Code 4.21: Main Function of `sniffer.c`

4.7 Countermeasure

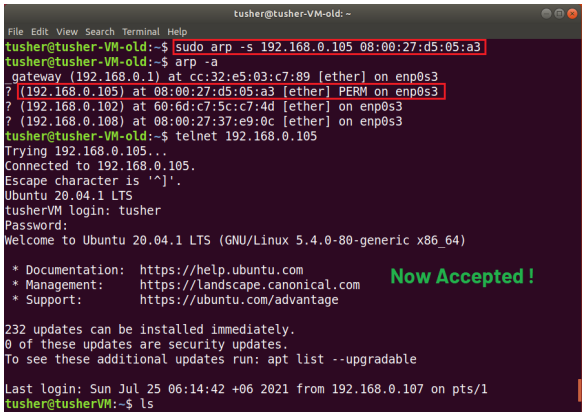
In case of *static* ARP entries or if **DAI** (Dynamic ARP Inspection) is enabled, then my attack will not work.

Static ARP Entries

Static Client Entry from Server Side:
`sudo arp -s 192.168.0.107 08:00:27:e0:aa:9f`
Static Server Entry from Client Side:
`sudo arp -s 192.168.0.105 08:00:27:d5:05:a3`



(a) Static Client Entry from Server Terminal



(b) Static Server Entry from Client Terminal

Figure 4.16: My Attack can NOT affect Static ARP entries

Dynamic ARP Inspection (DAI)

In Ubuntu, there is a module **ArpON (ARP handler inspection)**, a Host-based solution that make the ARP standardized protocol secure in order to avoid the Man In The Middle (MITM) attack through the ARP spoofing, ARP cache poisoning or ARP poison routing attack.

This is possible using three kinds of anti ARP spoofing techniques:

1. SARPI (Static ARP Inspection) for statically configured networks without DHCP;
2. DARPI (Dynamic ARP Inspection) for dynamically configured networks with DHCP;
3. HARPI (Hybrid ARP Inspection) for statically & dynamically configured networks with DHCP.

Also in the TP-Link Router in my home, **DAI** (Dynamic ARP Inspection) was enabled. That's why my attack could Not poison it's ARP cache to demonstrate `http` vulnerabilities. To disable DAI in router, I had to know the password which ISP doesn't want to inform me.

References

[1] Transmission control protocol (tcp). https://www.sdxcentral.com/resources/glossary/transmission-control-protocol-tcp/, 2021.

[2] Rob Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving tcp receivers can cause internet-wide congestion collapse. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 383–392, 2005.