COMP 6841 Something Awesome Project Deliverables

https://github.com/YaBashar/Something-Awesome-Project

Table of Contents

Problem Statement

With the Internet being an integral aspect of our lives, it is crucial to understand how vulnerabilities can be exploited within the network. This is particularly the case when connecting to public LANs and WIFI networks, in which attackers can easily join the network and perform man-in-the-middle attacks, including ARP and DNS poisoning. I aim to raise awareness among university students and working-class individuals who are highly vulnerable to insecure or improperly configured LAN networks, particularly at universities and offices.

My project aims to simulate a man-in-the-middle attack on my home network to demonstrate the scope and impact of such an attack, thereby increasing awareness, while also investigating existing security tools that can automate these processes. I will also be reporting on standard security measures to mitigate such attacks.

Project Context

My project involves investigating Network Security, particularly regarding Layer 2 Ethernet Frames within the Data Link Layer. My project aims to simulate a man-in-the-middle attack through ARP Poisoning and DNS Poisoning, which allows an attacker to observe all traffic between two hosts and potentially modify and alter network packets.

Background Information

Computer Networks are structured using the OSI model, which divides a network into several layers. The major layers, in order, are Application, Transport, Network, Data Link, and Physical Layer.

When an application makes a request, a packet is sent through the network using this layered approach, in which additional header information is encapsulated at each layer so that the next layer can process the packet.

ARP Poisoning is a security exploit conducted on the Data Link layer, which exploits the mechanisms of the ARP protocol. ARP (Address Resolution Protocol) is a protocol used to map IP addresses to their corresponding MAC addresses, enabling communication at the Link Layer. The mappings are stored in an ARP table of each host.

My attacking machine will be positioned between my victim machine and the default gateway router of my home network, involving the sending of forged ARP packets to corrupt the ARP tables. This will cause the victim machine to assume that the MAC address of my machine corresponds to the IP address of the router. This will allow me to observe all traffic passing between the victim and the router, as well as modify DNS packets, ultimately redirecting the victim to unintended websites.

I will be implementing my solution from scratch using Python modules, such as Scapy and NetfilterQueue, to create an exploit.

Challenges and Iterations of Implementation

Initial Setup with WSL2

I began by creating an initial test script, which involved sending a simple packet using Python's Scapy module through WSL on my Windows Laptop. However, I encountered errors (See Appendix A Figure 1) that highlighted the numerous limitations of WSL for network-related scripts.

I reviewed the documentation to understand the response type for SRP (used to send Layer 2 packets). I found out that SRP returns two lists, one for answered packets and one for unanswered packets. The line

$$\text{answer} = \text{srp}(\text{entire\_packet}, \text{timeout} = 2, \text{verbose} = \text{True})[0]$$

Accesses the answers list, which itself contains a list of tuples as (sent pkt, received packet)

$$\text{answer} = [\ (\text{sent packet},\ \ \text{recieved packet})\ \ldots\ ]$$

So $answer[0][0]$ Refers to the sent packet, and $answer[0][1]$ Refers to the packet that we should have received when performing an ARP request to determine the MAC address of our victim machine. Since $answer[0][1]$ Gives an index out of range error, which means we did not receive an ARP response when we should have.

I suspected that this was due to the virtualisation implementation of WSL, which may be blocking responses.

Looking at the documentation provided by Microsoft, which maintains WSL2, I found that WSL2 runs as a Hyper-V virtual machine, utilising a virtual Network Interface Card in conjunction with NAT instead of bridging to the host's physical Network Interface Card. This may have been why my Ethernet frames were being blocked. I wasn't able to find a concrete answer, but I thought it was best to use a full virtual machine instead of WSL2, which uses a lightweight Linux kernel.

VirtualBox Set Up with Linux

I started by first setting up a virtual Linux Ubuntu Machine with Oracle VirtualBox (See Appendix A Figure 2). I downloaded the ISO image of the Linux Machine and set it up within VirtualBox, then booted it up for the first time. Initially, the machine felt extremely slow and laggy with the default settings, so I had to increase both the Video Memory and the number of processors to improve its performance. I also made sure I was using a bridged network interface instead of a virtual network interface. This also exposed me to the concept of virtualisation, in which multiple Operating Systems can be run on one machine's hardware as Virtual Machines.

My initial setup consisted of the Linux Machine as well as my Windows Operating System running on my Lenovo ThinkPad laptop.

Using this setup, I created the following script. (See Appendix A Figure 3)

However, I realised that websites would not load, meaning that packets were being dropped. After conducting some research, I discovered that I needed to enable IP forwarding on my Linux machine to forward packets between the victim and the gateway router. (See Appendix A Figure 4)

Another thing I realised while reading the sniff logs was that my victim machine was sending packets to me, but I wasn't receiving packets from the gateway router, even though websites seemed to be loading. However, what I didn't account for was that I would also have to ARP spoof the gateway router into thinking that it was responding to the victim when it sends packets to my machine running the script.

With these changes in mind, I successfully corrupted the ARP tables of both the victim machine and the gateway router.

The next challenge was that I was unable to receive DNS responses when corrupting the ARP table of my Windows machine and the gateway router. I applied packet filters to help with logging; however, none of them worked. I could see DNS requests being sent by the victim machine, but the gateway router never sends a DNS response back. I also thought that some browser security settings might be the cause, so I disabled them. See Appendix B Figure 1.1 and 1.2

After failing to find any luck, I decided to research whether there were any secure protocols regarding DNS, as there are with the transport layer, which uses TLS.

I discovered that Windows 11 machines utilise both DOT (DNS over TLS) and DOH (DNS over HTTPS). This meant that all responses would be encrypted and that it would be impossible for me to use Scapy to inspect DNS response packets.

I was able to confirm that DOT and DOH were enabled at the Operating System Level.

This also highlights how Windows uses external DNS servers, which prevent downgrading to unencrypted UDP. See Appendix B Figures 2.1 and 2.2

Using Two Virtual Machines

I then set up another virtual Linux machine, this time installing the Kali Linux ISO file and running it on VirtualBox. See Appendix B Figure 3. To my surprise, I was now able to receive DNS responses, as Linux supports both DOT and DOH, but these protocols are not enabled by default, allowing me to inspect DNS response packets from the gateway router.

Modifying Packets

Now I needed a way to modify packets before sending them back to the victim machine. This article https://medium.com/@SpoofIMEI/modifying-packets-on-the-fly-python-76076c5d6e1e suggested using netfilterqueue, a Python module that would assist with modifying in-flight packets as a man-in-the-middle.

NetfilterQueue enables us to access and manipulate packets present in the Linux machine's nfqueue, where packets are queued based on an iptables rule. An iptable rule tells the machine how to handle specific traffic and can be configured to reject, accept, or forward particular packets.

This packet (See Appendix C, Figure 1), logged by the sniff function in Python Scapy, displays the fields associated with a DNS response packet. Our goal was to modify the IP address to redirect to a webpage hosted on our IP address.

My final script successfully corrupted ARP tables, logged all packet information between the victim and gateway router, and attempted to modify DNS response packets sent to the victim.

Notice how it assumes that 192.168.0.1 (the router's IP address) is associated with the MAC address of my attacking machine, which has the IP address 192.168.0.198. See Appendix C Figure 2

The issue was that HTTPS sites recognised that the DNS cache was corrupted and would abort the connection due to invalid certificates. I attempted to host my own HTTP site using Apache2 as the web server; however, I was unable to redirect users to my website when they searched for something else.

I realised the issue was because of an invalid certificate for our website. Usually, Digital Certificates are obtained by a Certificate Authority. However, we could self-sign a valid certificate. I followed this tutorial from Medium https://medium.com/@pasanglamatamang/configuring-a-self-signed-ssl-certificate-on-a-apache-server-cbcd6eefdf1a to create a self-signed certificate for my website hosted on Apache2 (See Appendix C Figure 3). This allowed the browser to generate a warning that my website is untrusted, allowing (See Appendix C Figure 4) the user the choice of accepting the risk and being redirected to my malicious webpage.

It was interesting to note that browsers on different Linux distributions exhibited varying behaviour. On Kali Linux, the browser did not give the option to accept the risk and continue. This could be because Kali is usually used for penetration testing and has hardened browser security. However, Fedora Linux provided the option only after disabling the firewall and SELinux Security, which were enabled by default.

Finalised Python Script

My final Script involves five functions, which include:

- Mac_addr_discovery(): This function sends one ARP broadcast frame to both the router and victim to receive the MAC addresses of the two machines.

- Poison(victim_pkt, router_pkt): This function takes in two packets, which will be sent to both machines, the victim and router, respectively, to corrupt their ARP tables. It must be repeatedly sent as the ARP cache resets after a specific period

- Packet_listener(packet): This function takes in a packet and determines whether it is a DNS query packet. If it is, then we create a fake DNS response packet and send it to perform DNS spoofing.

- Packet_sniff(victim_mac_addr, router_mac_addr, attacker_mac_addr): This function deals with filtering packets only concerned with the connection between our router and the victim machine and logs all packet information into a file.

- The main first uses os to enable IP forwarding and IP table rules. Then it creates two packets, which will be passed into poison(). Both poison() and packet_sniff() run on two separate threads, as packet_sniff() is a blocking process, and poison () should be performed continuously in the background. We then initiate and run a queue using the nfq module, which queues packets within the operating system, allowing us to modify them before they are sent.

ARP Poisoning with Ettercap.

I was close to what I set out to achieve. I also decided to research open-source solutions that have already been implemented to gain a better understanding of different approaches. I used Ettercap, which had an easy-to-use interface, providing me with the same capabilities of ARP spoofing as my Python script.

Findings and Conclusion

The project aimed to simulate a man-in-the-middle attack through ARP poisoning and DNS spoofing, highlighting and raising awareness of the impact on students and the working class who connect to LAN networks that may be insecure or not correctly configured.

Overall, the project outcomes turned out to be slightly different from what I envisioned, as I encountered new concepts such as virtualisation and security measures applied to DNS and web browsers.

My findings from this project indicate that this attack is highly likely. My machine running the malicious script was able to easily corrupt ARP tables for both Windows and Linux victim machines, highlighting a low technical barrier to such an attack.

DNS spoofing was a bit more challenging to simulate, as Windows has incorporated security measures such as encrypting DNS packets at the operating system level, making it much more complicated. I found that Linux machines do not encrypt DNS packets at the operating system level, making them susceptible to DNS spoofing. Both Windows and Linux machines have browsers that also implement DNS security. The use of HTTPS prevents redirection to potentially malicious sites that have invalid or untrusted digital certificates.

In Conclusion, students and individuals of the working class who frequently connect to LAN networks outside of home, such as university and office networks, are highly susceptible to ARP poisoning as well as DNS spoofing. Therefore, it is recommended that users be able to notice and identify suspicious activity. Moreover, institutions and facilities should implement port security, which can inspect ARP packets and drop potentially malicious ARP packets.

References

scapy.readthedocs.io. (n.d.). *Usage — Scapy 2.4.4. documentation*. [online] Available at: https://scapy.readthedocs.io/en/latest/usage.html.

craigloewen-msft (n.d.). *Windows Subsystem for Linux Documentation*. [online] learn.microsoft.com. Available at: https://learn.microsoft.com/en-us/windows/wsl/.

IT Developer (2023). *Manually Poisoning Targets ARP Cache With Scapy 🕵️Man In The Middle - MITM 15.4*. [online] YouTube. Available at: https://www.youtube.com/watch?v=C_FKDs7a-mk [Accessed 22 Jul. 2025].

Pasang Lama Tamang (2023). *Configuring a Self-Signed SSL Certificate on a Apache Server*. [online] Medium. Available at: https://medium.com/@pasanglamatamang/configuring-a-self-signed-ssl-certificate-on-a-apache-server-cbcd6eefdf1a.

Spoofimei (2022). *Modifying packets on the fly (python) - Spoofimei - Medium*. [online] Medium. Available at: https://medium.com/@SpoofIMEI/modifying-packets-on-the-fly-python-76076c5d6e1e [Accessed 22 Jul. 2025].

Appendix A

```
mubashir@Mubashir-Laptop:~/work/Something-Awesome-Project$ sudo python3 arpPoisoner.py
Begin emission:
Finished sending 1 packets.

Received 1 packets, got 0 answers, remaining 1 packets
Traceback (most recent call last):
  File "/home/mubashir/work/Something-Awesome-Project/arpPoisoner.py", line 15, in <module>
    target_mac_addr = answer[0][1].hwsrc
  File "/usr/lib/python3/dist-packages/scapy/plist.py", line 133, in __getitem__
    return self.res.__getitem__(item)
IndexError: list index out of range
```

Figure 1

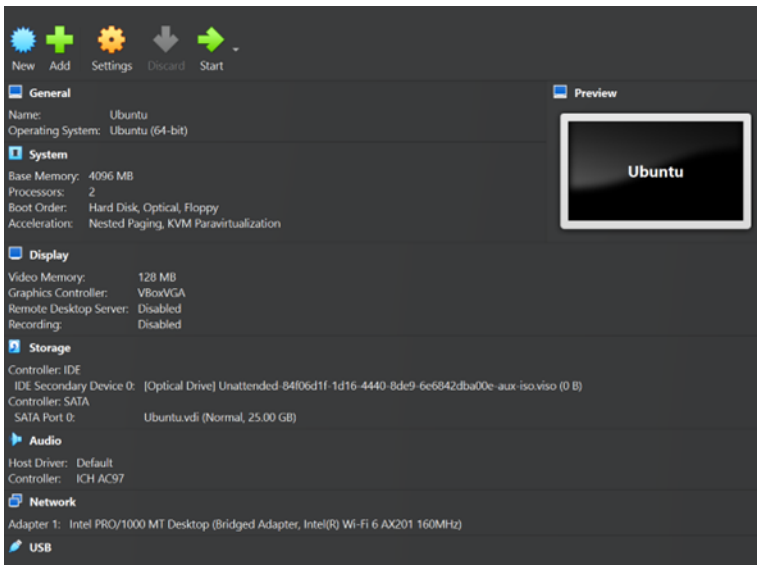

Figure 2

```python
from scapy.all import *
ether = Ether()

## Discovering MAC Address of Target Machine
broadcast = Ether(dst = 'ff:ff:ff:ff:ff:ff')
arp_layer = ARP(pdst = '192.168.0.49')
entire_packet = broadcast / arp_layer
answer = srp(entire_packet, timeout = 2, verbose = True)[0]

## Creating Malicous ARP Response to victims machine
target_mac_addr = answer[0][1].hwsrc
packet = ARP(op=2, hwdst = target_mac_addr, pdst='192.168.0.49', psrc='192.168.0.1')
send(packet, verbose = False)
pkt = sniff(count = 10, prn = lambda x: x[0].show())
```

Figure 3

```
os.system('echo 1 > /proc/sys/net/ipv4/ip_forward')
```

Figure 4

Appendix B

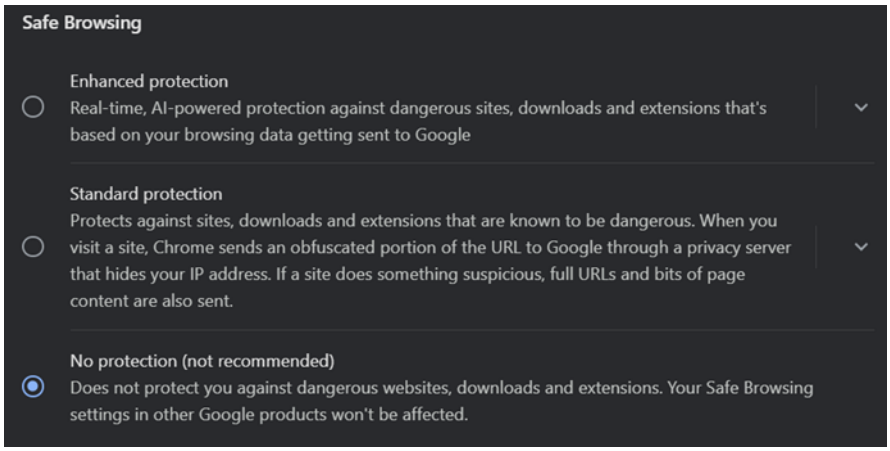

Figure 1.1 Disabling Chrome Security

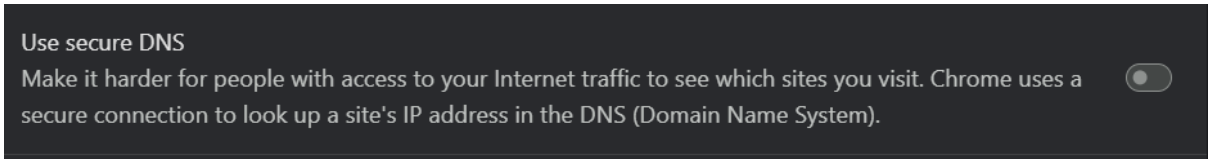

Figure 1.2 Disabling Chrome DNS Security

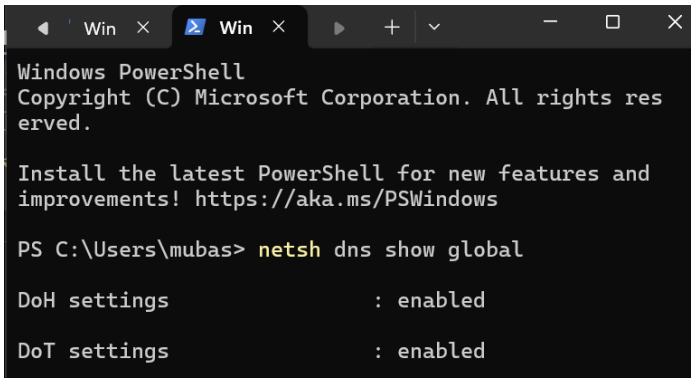

Figure 2.1 Windows DNS Settings

```
PS C:\Users\mubas> Get-DnsClientDohServerAddress

ServerAddress        AllowFallbackToUdp AutoUpgrade
-------------        ------------------ ----------
149.112.112.112      False              False
9.9.9.9              False              False
8.8.8.8              False              False
8.8.4.4              False              False
1.1.1.1              False              False
1.0.0.1              False              False
2001:4860:4860::8844 False              False
2001:4860:4860::8888 False              False
2606:4700:4700::1001 False              False
2606:4700:4700::1111 False              False
2620:fe::9           False              False
2620:fe::fe          False              False
2620:fe::fe:9        False              False
```

Figure 2.2 More Windows DNS Settings

Figure 3 Setting up Kali Linux VM

Appendix C

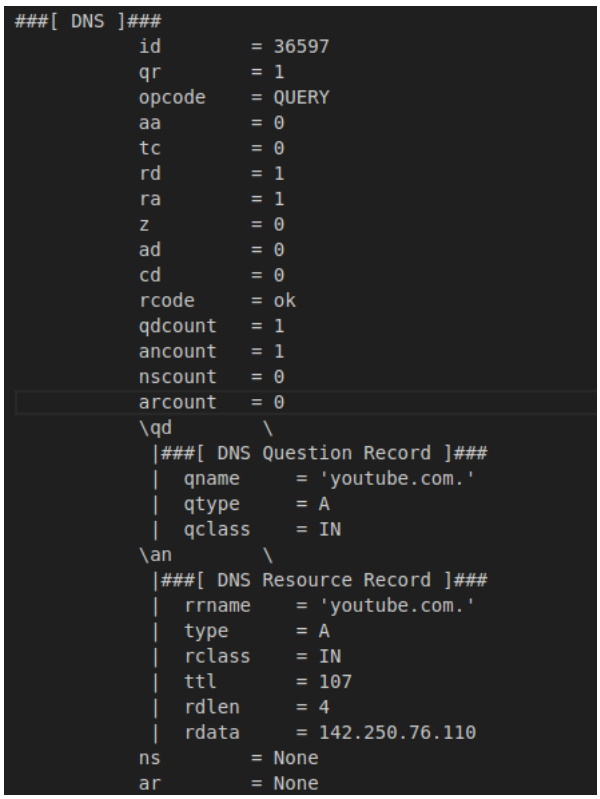

Figure 1

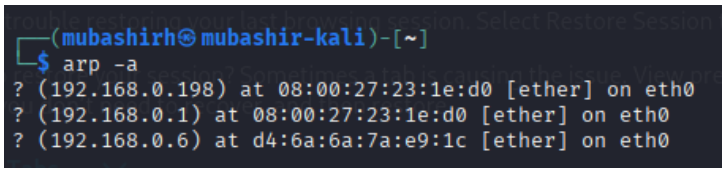

Figure 2

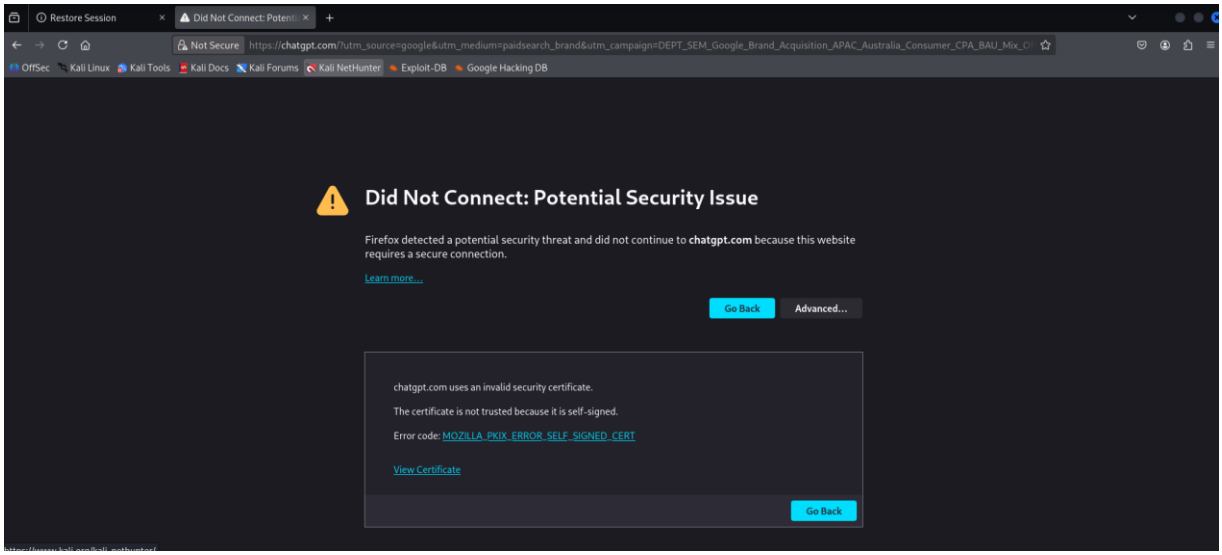

Figure 3

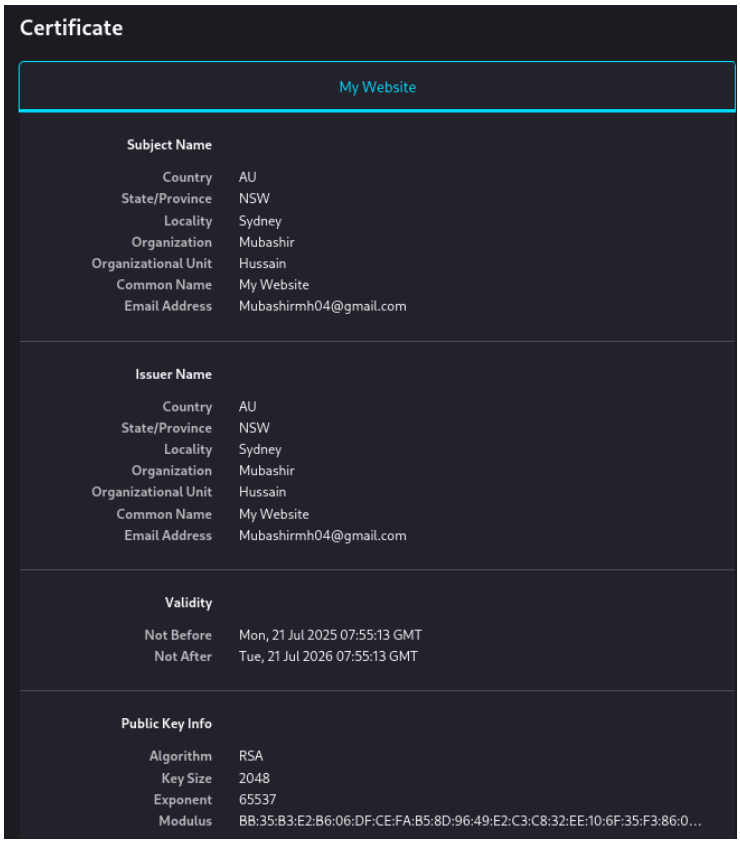

Figure 4

Appendix D

https://github.com/YaBashar/Something-Awesome-Project