Man in the Middle Attacks & DNS Spoofing

Cole Rees - A00741578

David Tran - A00801942

COMP 8505 Set 7D

British Columbia Institute of Technology

Aman Abdulla

Monday, November 3 2014

# 

# 

Table of Contents

# Introduction

We have discussed in our previous courses how it is important to understand the concepts of designing before programming. Often, a proof of concept allows the programmer to quickly design and put together a prototype that can be used as a tangible deliverable. Because of its how quickly it is able to be developed, it could easily show if the idea is worthwhile to pursue or not, increasing productivity and wasting minimal time and efforts. Once it has been shown that the prototype is promising, we can transition to the next stage in development for optimization using C programming, for example.

In this assignment, we are applying the proof of concept practice by designing our own DNS Spoofing program. We shall be using Ruby as our language of choice to provide this “prototype” as our deliverable. The components of our program involves an ARP Poisoning thread that acts as the Man in the Middle, carrying out MITM attacks. In turn, the victim machine will ultimately think that our MITM machine is a legitimate switch, and will be sending DNS requests to our attacking machine. All the while, our attacking machine will be responsible for the communication exchange between the switch and DNS requests and replies.

# Network Design & Pseudo Code

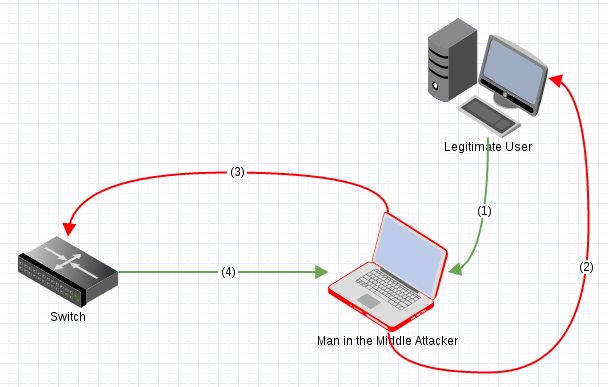
|  |  |
| --- | --- |
| **Names** | **IP Addresses** |
| Man in the Middle | 192.168.0.7 |
| Legitimate User | 192.168.0.8 |
| Switch | 192.168.0.100 |
| DNS Server | \*any arbitrary legitimate server\* |
| Malicious Server | 192.168.0.9 |

|  |
| --- |
| **“Driver” - main.rb** |
| #set some global variables  @interface = “em1”  @victim\_ip = victim’s ip address  @victim\_mac = victim’s mac address  @router\_ip = router’s ip  @router\_mac = router’s mac  @our\_info = grab our attacking machine’s info through PacketFu  ...  begin  Create our arpSpoof object with arguments  Create our dnsSpoof object with arguments  Create arpSpoof thread  Create dnsSpoof thread  Join threads  Catch “Ctrl+C” interrupt  kill threads  exit  end |

|  |
| --- |
| **ARP Spoof Class - “arpSpoof.rb”** |
| class ARPSpoof  def init (...)  create our ARP packet to victim machine  create our ARP packet to router  start arpSpoofing  end  def start  `echo 1 > /proc/sys/net/ipv4/ip\_forward`  drop all ICMP redirects from the attacking machine using IPtables  while caught == false do  sleep 1  send ARP packet to victim machine  send ARP packet to router  end  end  def stop  stop sending ARP packets  clear up IPtables  exit 0  end  end |

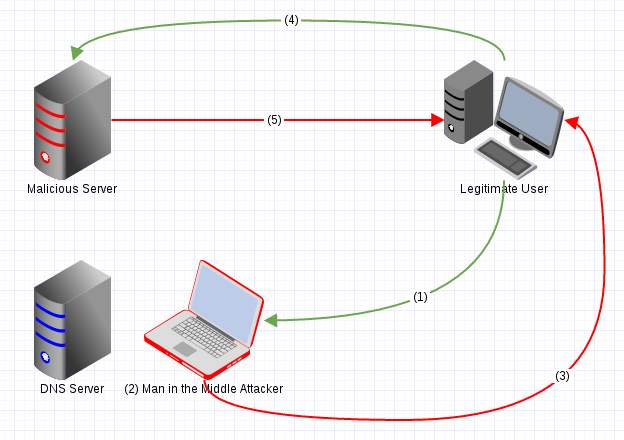
|  |
| --- |
| **DNS Spoof Class - “dnsSpoof.rb”** |
| class DNSSpoof  def init (...)  set some variables  start capturing  end  def capturing  set filter: “udp and port 53 and src “ + victim’s ip  set capturing module from PacketFu  check each packet coming in using filter  if the packet is UDP, parse it  if packet is DNS Query, grab it  parse domain name  send domain name and packet to send\_request  end  end  end  def send\_request(packet, domain name)  set some UDP variables  populate payload with DNS Response flags  populate payload with domain name  populate payload with more DNS Response flags  populate payload with Spoof’d IP (in hex)  recalculate UDP packet size  send udp packet to victim  end  end |

## ARP Poisoning Diagram



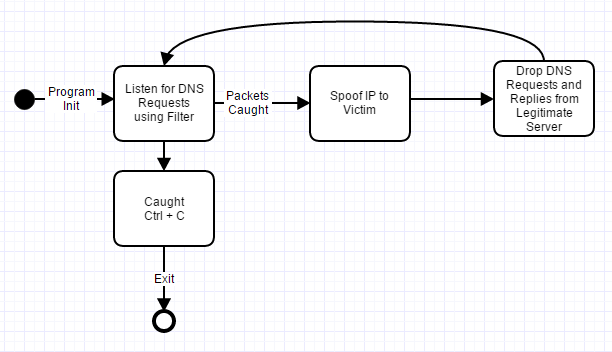
1. The Legitimate User sends an ARP request to the Switch.
2. The Man in the Middle intercepts this ARP request, and responds back to the Legitimate User, masquerading as the Switch.
3. Simultaneously, the Man in the Middle sends a fabricated ARP request to the Switch, masquerading as the Legitimate user.
4. The Switch sends back the ARP reply to the Man in the Middle, thinking that it is the Legitimate user. At this point, the Man in the Middle attacker relays the reply back to the Legitimate user but prepares the reply to look like the Switch.

## DNS Spoofing Diagram

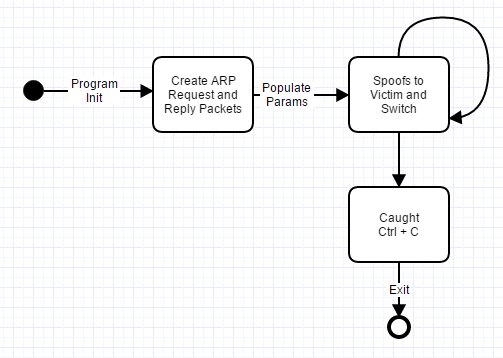


1. The Legitimate user wishes to visit a web page and sends a DNS query
2. The Man in the Middle intercepts this DNS query and fabricates its own DNS response. *The DNS Query does not even reach the legitimate DNS Server!*
3. The Man in the Middle then crafts its own values and sends back the DNS Response, with the destination pointing to the Malicious Server.
4. The Legitimate user, thinking that the DNS response is authentic, visits the Malicious Server.
5. The Malicious server responds by providing its web pages to the Legitimate user.

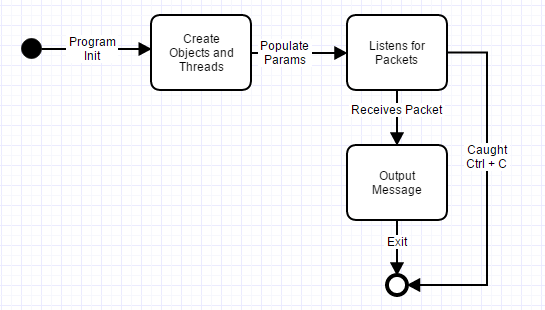
## dnsSpoof.rb State Chart Diagram



## arpSpoof.rb State Chart Diagram



## main.rb State Chart Diagram



# Tools & Equipment Used

## Hardware

|  |  |  |
| --- | --- | --- |
| * 8GB RAM | * Intel i5 Quad Core | * 500GB HDD |
| * Victim Machine | * Man in the Middle | * Switch |

## Software

|  |  |  |
| --- | --- | --- |
| * Fedora Linux 20 64-bit | * Ruby Programming | * Wireshark |
| * Terminal | * PacketFu Library | * IPtables |

# Testing & Documentation

## Test Cases

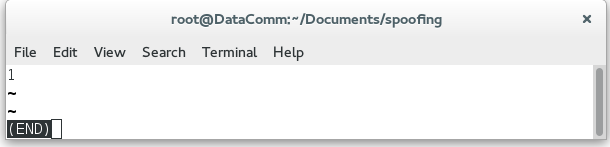
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1 | arpSpoof enables ip forwarding | less /proc/sys/net/ipv4/ | ip\_forwarding = 1 | PASSED. See results for details. |
| 2 | arpSpoof is able to create IPtable rules | iptables -L -n -v -x | Four distinct IPtable rules should exist | PASSED. See results for details. |
| 3 | arpSpoof is able to poison victim machine | terminal, arp | Victim machine will show two identical MAC addresses that belong to our attacking machine | PASSED. See results for details. |
| 4 | dnsSpoof is able to filter packets using the filter specified | terminal | Terminal output shows our filter | PASSED. See results for details. |
| 5 | dnsSpoof is able to spoof to intended website | terminal | Terminal will output intended spoof IP | PASSED. See results for details. |
| 6 | dnsSpoof is able to parse DNS requests when victim machine uses web browser | terminal | Terminal will output DNS requests | PASSED. See results for details. |
| 7 | Victim machine will be redirected to spoofed website | Web Browser, Wireshark | Wireshark will display filtered packets that have the DNS response with our spoofed IP; Web Browser will redirect to intended website | PASSED. See results for details. |
| 8 | Ctrl + C kills both threads | ps aux | grep main.rb, ps aux | grep arpSpoof.rb, ps aux | grep dnsSpoof.rb | All threads get killed and program exits | PASSED. See results for details. |
| 9 | Ctrl + C resets ip forwarding | less /proc/sys/net/ipv4/ | ip\_forwarding = 0 | PASSED. See results for details. |
| 10 | Ctrl + C resets iptables firewall rules | iptables -L -n -v -x | All tables should have accept policy | PASSED. See results for details. |

## Test Results

### arpSpoof enables ip forwarding

When initializing our program, we require IP forwarding to be enabled on Linux systems. Below, we confirm that our program does just that by running the following command to see that the file is inserted with a “1” to enable IP forwarding:

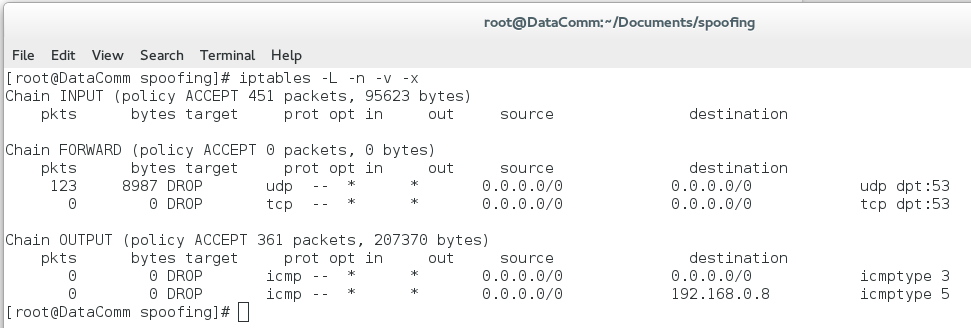
Command: less /proc/sys/net/ipv4/ip\_forward



### arpSpoof is able to create iptables rules

Our code executes four lines of iptables configurations. Here we run the following command to display our current firewall settings.

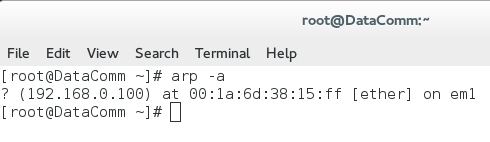
Command: iptables -L -n -v -x



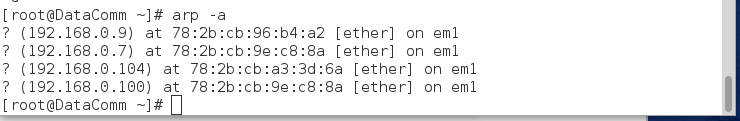
### arpSpoof is able to poison victim machine

In the following screenshot, we show what the ARP table looks like on our victim machine. Note that this is our legitimate router MAC address.

Before:



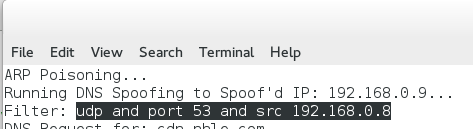
After Spoofing:



After we started the program to ARP poison our victim machine, we can clearly see that MAC addresses of 192.168.0.7 and 192.168.0.100 are the same. This is a sure signature of being ARP poisoned.

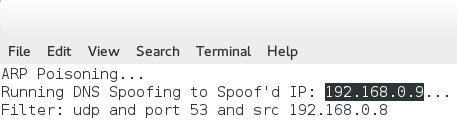
### dnsSpoof is able to filter packets using the filter specified

The below screenshot displays the filtering of packet captures and uses TCPdump-like notations for its filtering. Note that it is filtering packets originating from our victim IP address.



### dnsSpoof is able to spoof to intended website

The following screenshot displays that our DNS Spoofer will be redirecting all DNS replies with spoofed IP of 192.168.0.9...

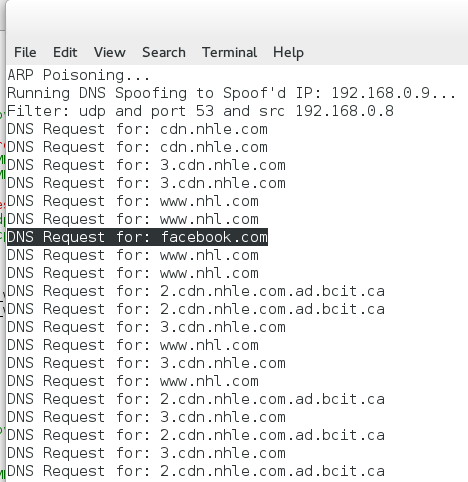


### 

### 

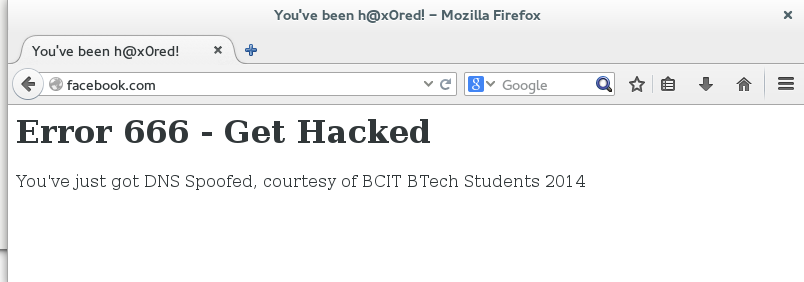
### dnsSpoof is able to parse DNS requests when victim machine uses web browser

When our victim opens a browser and visits a popular website such as Facebook, we can confirm that our DNS Spoofer is filtering and capturing packets based on our victim’s activity.

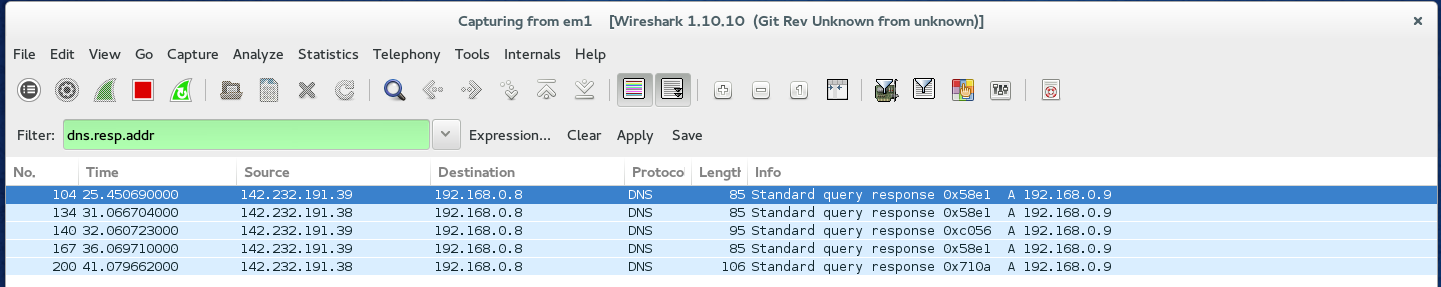


### Victim machine will be redirected to spoofed website

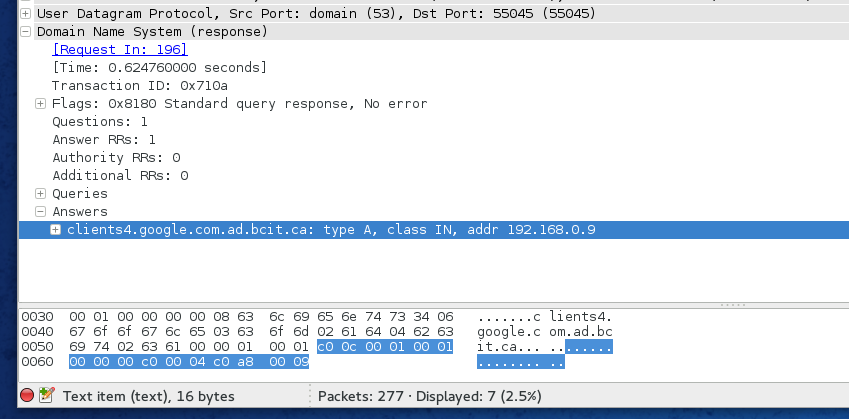
Below is what the victim sees when he or she visits our spoofed website.



The following are Wireshark captures of the DNS replies to our victim machine. Note how the domain name remains the same but the IP address is our spoofed IP.

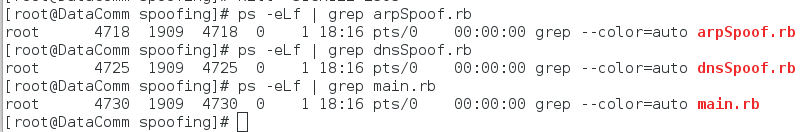


In further detail of a sample packet, we see that all captured packets that are DNS requests are being intercepted and crafted to send our victim to our spoofed address.



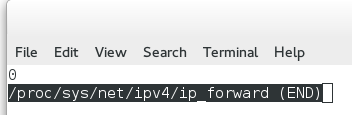
### Ctrl + C kills threads

Once we enter Control + C on our attacking machine, we should see no signs of any lingering processes or threads related to our program.



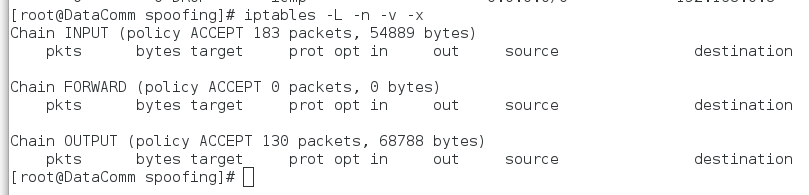
### Ctrl + C resets ip forwarding

IP forwarding is reverted back to its default settings of “0”.



### Ctrl + C resets iptables firewall rules

Firewall rules are reverted back to their default values. There are no special rules specified in the below screenshot.



# Limitations of Proof of Concept

Since we are using the Ruby language as a proof of concept for our DNS Spoofing, we ran into a problem where actual legitimate DNS replies came back faster than our attacking machine was able to even send its crafted packet. Also, once we applied firewall rules to delimit any forwarding of legitimate DNS replies, we saw that it took abnormally long to resolve a website name, even though we supplied the victim machine with a legitimate looking DNS reply packet. This is a serious drawback of using Ruby programming in this case. Suspecting victims will eventually grow tired, wearied and will definitely notice this strange lagging behavior of their once-fast machine if we used our program to DNS Spoof them. This violates practice of being covert and legitimate. However, while the drawbacks are visible, a proof of concept is still tangible in the sense that it can demonstrate simple weaknesses without having to provide rich and fully functional code.

# Conclusion

In more verbose programming languages like C, packet crafting such as we have done in this assignment may require code two or three times as much compared to the code in Ruby. However, because of the nature of C, it allows programmers to have much more control over their programming, rather than relying on automated features or libraries that may or may not execute code as intended. But sometimes spending all that effort in C may not surmount to a feasible project or program. In which case, time and effort is wasted. By quickly conjuring up a proof of concept as we have here, we can determine if the idea is worth pursuing without having to overextend.

A note on DNS spoofing, we can easily see how man in the middle attacks can exploit a network. Without preventive measures in place, a rogue employee with the know-how of the network infrastructure of the business organization, can easily take advantage of the other technologically-illiterate employees to providing or doing activities that can leverage the attacker.

Therefore, it is imperative that security is implemented on networks to check for DNS spoofing. Some solutions involve adding randomness to the queries, others include DNS request and reply checking. Whatever the case may be, DNS spoofing can be damaging to personal users and business organizations if left undetected.

# References

Queenan, L. (2012, October 29). Ruby DNS Spoofing using Packetfu. Retrieved November 1,

2014, from http://crushbeercrushcode.org/2012/10/ruby-dns-spoofing-using-packetfu/

# Appendices

## Appendix I - Files on Disk

Files located on-disk are the following:

* Man in the Middle Attacks & DNS Spoofing (.pdf)
* Ruby Code (directory)
  + main.rb
  + arpSpoof.rb
  + dnsSpoof.rb
* README.txt