SEEDs Lab Report

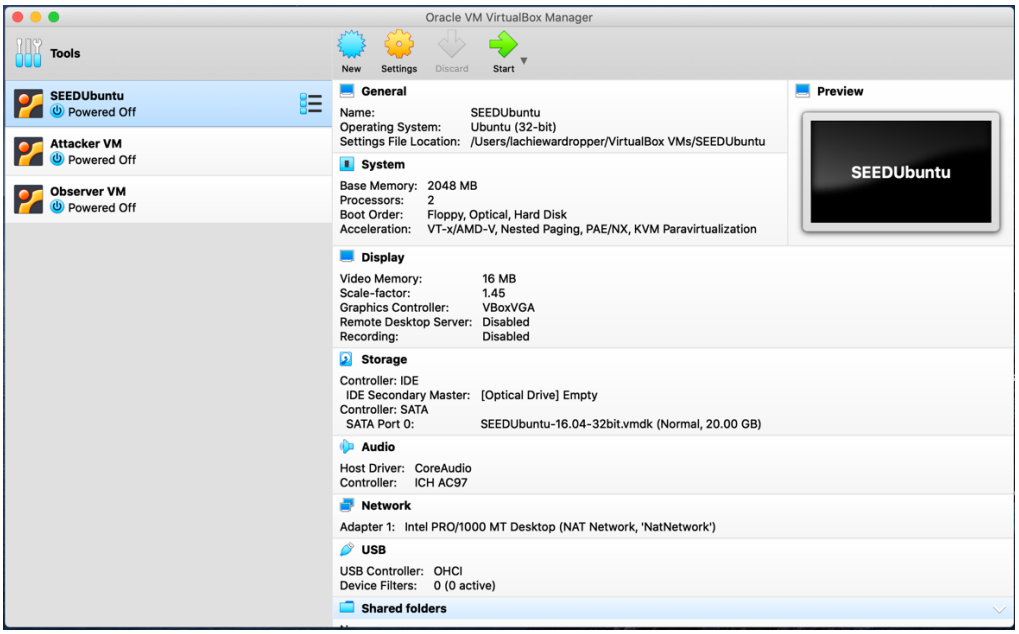
LACHLAN WARDROPPER
44397580

Executive Summary

This is the SEEDs Lab Report that help students gaining their first-hand experience with: **Buffer Overflow Attack**, **Attacks On TCP/IP Protocol** and **RSA Public-Key Encryption and Signature.**

Introduction

I set up my SEED Ubuntu16.04 VM and VirtualBox on my 2015 MacBook Pro, as shown below.



Objective

After finished the Labs, I now understand how the RSA encryption algorithm can be used to derive a private key. I also understand how to encrypt and decrypt a message using RSA encryption, and how a message signature can be created and verified. Solving these tasks in a practical environment also provided me with great insight into how the BIGNUM API can be used to handle large numbers, which is an accurate reflection of actual encryption used in real-world scenarios.

After completing the buffer overflow lab, I learned how to compile and execute shellcode within a c program. I also learned how different root permissions can be modified to allow for buffer overflow to occur, including the non-executable stack and address space randomization. I also gained insight into how attackers would exploit a vulnerability in code to access the shell, and also how the *setuid* command can change the permissions a user has when accessing the shell.

Finally, from completing the TCP IP attack lab, I learned how the SYN Cookies protection countermeasure prevents a SYN flood attack. Additionally, I understand how an RST TCP

attack can be performed over both telnet and ssh connections, and I saw the effects this has on said connection. I also understand how an RST TCP attack can affect a machines connection with a video streaming server, and how it will prevent a video from playing due to a constant stream of TCP packets.

Timeline

I was making this report from 25/08/2020 to 15/09/2020.

Processes

There are 11 tasks that I have completed in this report:

1. Deriving the Private Key
2. Encrypting a Message
3. Decrypting a Message
4. Signing a Message
5. Verifying a Signature
6. Running Shellcode
7. Exploiting the Vulnerability
8. Defense **dash's** Countermeasure
9. SYN Flooding Attack
10. TCP RSA Attacks on telnet and ssh Connections
11. TCP RST Attacks on Video Streaming Applications

Report

RSA Public-Key Encryption And Signature Lab

Task 1: Deriving the Private Key

In order to complete this task, it was important to have a solid understanding of how the RSA algorithm works. Firstly, it was necessary to convert all of the parameters used into BIGNUM types, in order to allow for arithmetic operations of such large integers. After declaring BIGNUM variables for *p, q* and *e,* the hexadecimal values provided were assigned to these variables using the *BN_hex2bn* function. This function converts a hexadecimal string into a BIGNUM datatype, before storing it at a specified memory position.

Following this, I knew that in order to calculate the value of ϕ, where $\Phi = (p-1)(q-1)$, it was important to convert the integer 1 into a BIGNUM type to allow for arithmetic operations. After doing this, I was able to use the *BN_sub* method to calculate (p – 1) as well as (q – 1). Then, by simply using the *BN_mul* function, I was able to find the value of ϕ (defined as *x* in the code in *Figure 1)*.

The final step required to derive the private key was to find the inverse (d) of $e \; mod \phi$ such that $(e * d) \; \% \; \phi \; = 1$. The result of these steps can be seen in *Figure 1* below.

RSA.c

```
#include <stdio.h>
#include <openssl/bn.h>

int main() {
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *p1 = BN_new();
    BIGNUM *q1 = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *x = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *one = BN_new();

    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");

    BN_dec2bn(&one, "1");

    BN_sub(p1, p, one);
    BN_sub(q1, q, one);
    BN_mul(x, p1, q1, ctx);

    BN_mod_inverse(d, e, x, ctx);

    char *number = BN_bn2hex(d);
    printf("d = %s\n", number);

    return 0;
}
```

Figure 1: Code to derive the private key using RSA

As is clear in *Figure 1*, the variable *x* was used to replace *ϕ.* Furthermore, a string variable called *number* was declared and assigned the hexadecimal value of the private key *d*. This was achieved by using the *BN_bn2hex* function, which converts a BIGNUM type back into a

hexadecimal string. Finally, the value of *number* was printed to console. The resulting output from the code in *Figure 1* can be seen in *Figure 2.*

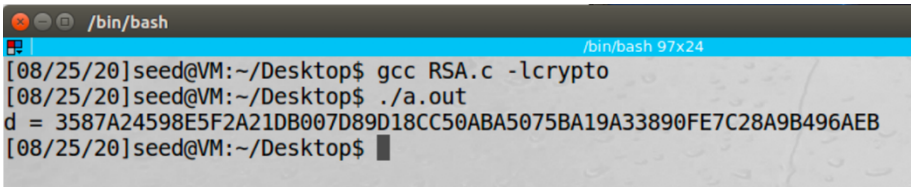


Figure 2: Output of code used to derive the private key

Task 2: Encrypting a Message

The required formula to encrypt a message using RSA is $C = M^e mod(n)$, where C is the ciphertext, M is the plaintext message as a hexadecimal value such that $1 < M < n$, and (n, e) are the recipient's public key. For this task, the values e and n were provided. Therefore, in order to find the value of M, the provided python command was used in the console. This outputted a hexadecimal string that was then converted to a BIGNUM type using the *BN_hex2bn* function. The values of e and n were also converted to BIGNUM datatypes using the same method. *Figure 3* shows how this function was used to convert hex these strings.

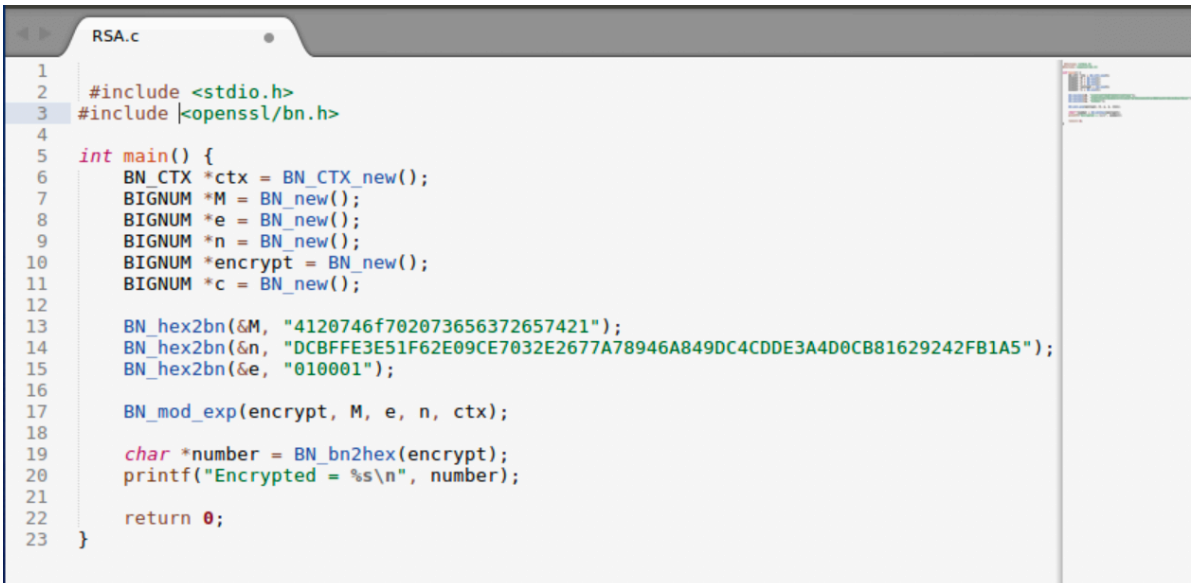


Figure 3: Code to encrypt a message

As is also shown in Figure 3, the *BN_mod_exp* function was used to perform the arithmetic operation $C = M^e mod(n)$. This value was then stored in a variable called *encrypt*, which was then outputted to the terminal. The results of this can be seen in *Figure 4.*

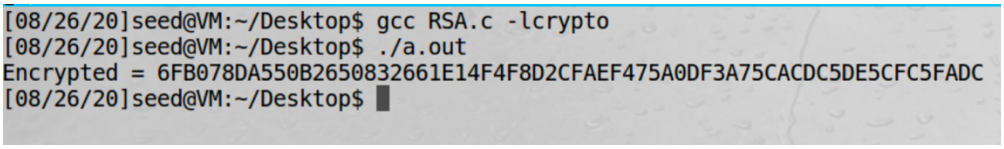


Figure 4: Output of message encryption code

In order to verify the output of *Figure 4,* the private key d was used to decrypt the newly encrypted ciphertext. The formula $M = C^d mod(n)$ was used, where M is the hexadecimal value of the original message, C is the encrypted ciphertext, and d is the recipient's private key. If the encrypted cipher text is correct, then the decoded message M should be the same as the original hexadecimal message string. *Figure 5* shows the additional code used to verify the encrypted message, and *Figure 6* shows the output of this code.

```
BIGNUM *d = BN_new();
BIGNUM *decrypt = BN_new();
BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

BN_mod_exp(decrypt, encrypt, d, n, ctx);

char* decryptedNum = BN_bn2hex(decrypt);
printf("Decrypted = %s\n", decryptedNum);
```

Figure 5: Code to verify encrypted message using private key

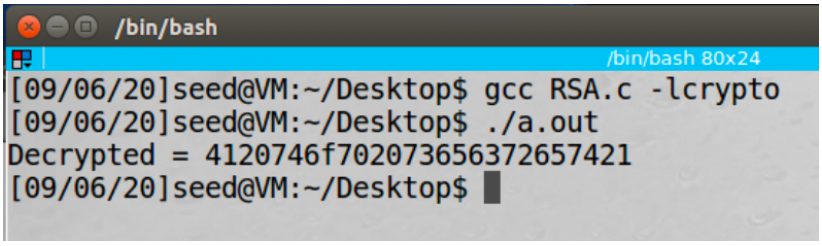


Figure 6: Output of code to verify encrypted message

As is evident from *Figure 6,* the encryption result for this task has been verified, as the resulting message M is the same as the original hexadecimal message value. Therefore, the message *"A top secret!"* was successfully encrypted. I learnt from this task how to use RSA to both encrypt a message as well as verify an encrypted message by using a private key.

Task 3: Decrypting a Message

This task required the same formula as was used to decrypt the message produced in the previous task. This formula is represented as $M = C^d mod(n)$, where M is the hexadecimal value of the original message, C is the encrypted ciphertext, and d is the recipient's private key. The same private and public keys were used for this task as were used for Task 2, with the only differing element being the ciphertext C. The code snippet provided in *Figure 7* shows how the formula $M = C^d mod(n)$ was implemented using BIGNUM types.

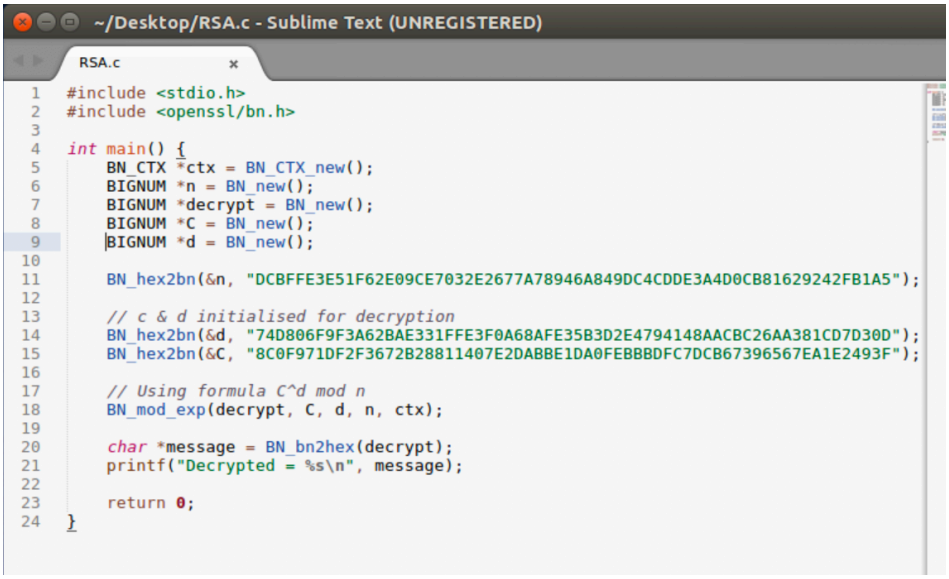


Figure 7: Code to decrypt a message

Once the string representing an integer format of some message M was printed to the console, the python command provided was used. Within this command, the hex value outputted by the program was substituted, and the result was the message *"Password is dees"*. This can be seen in *Figure 8* below.

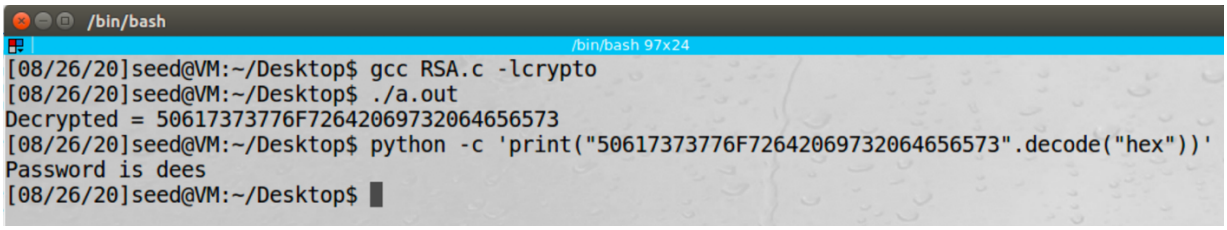


Figure 8: Output of decryption code and use of python command

As is evident by *Figure 8,* the decrypted hexadecimal value was decoded using the *decode* python function. I found it very interesting how an RSA key pair works when conducting this task and was surprised by the simplicity of the decryption process when an effective key pairing is used.

Task 4: Signing a Message

A digital signature is used as a means of verifying the authenticity of a message. Specifically, it uses a sender's private key on a message to create a signature. This signature can then be verified by applying the pairing public key, thus verifying the integrity of the original message. In order to generate a digital signature, a cryptographic hash function is used to alter the length of a message to a short number [1]. Furthermore, it is essential for a cryptographic hash function to map different letter combinations to different hashes, in order to avoid overlap.

For this task, the digital signature was found by using the formula $s = M^d mod(n)$, where M is a message represented as an integer, d is a sender's private key, and n is the RSA modulus.

All of these values apart from M were provided for this task, and thus made generating the digital signature relatively simple. Firstly, all of the numerical values were converted to BIGNUM types, before using the *BN_mod_exp* function to perform the required arithmetic operation to find the value of *s*. The specific code used for this task can be found in *Figure 9*.

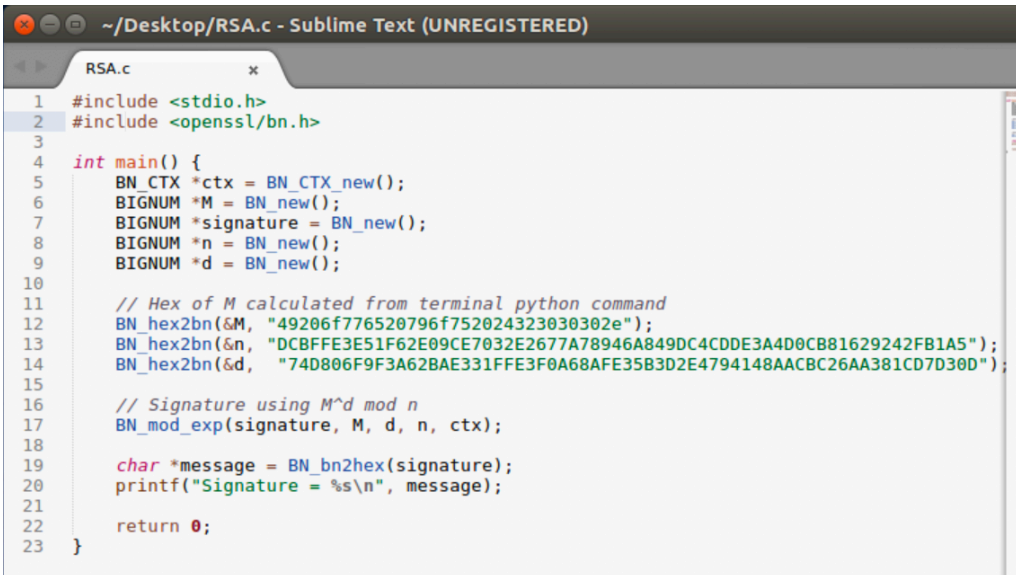


Figure 9: Code used to generate a digital signature

The *M* value used in the program *RSA.c* seen in *Figure 9* was generated through the python *encode* function on the message *"I owe you $2500."*. The signature produced by the key pair *(n, d)* and message code *M* can be seen below in *Figure 10.*

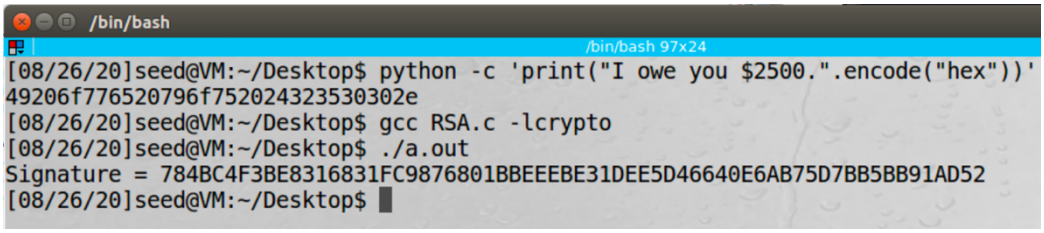


Figure 10: Output of code to produce digital signature of message (M)

Once this signature was produced, I thought it would be interesting to see how a small change to the original message would affect the signature produced. In order to do this, I altered the amount owed in the message from $2500 to $2000. It was very interesting to observe that whilst there was only one-byte difference between the original messages, the resulting signatures produced were completely different. *Figure 11* shows the newly produced digital signature after altering the original message *M*.

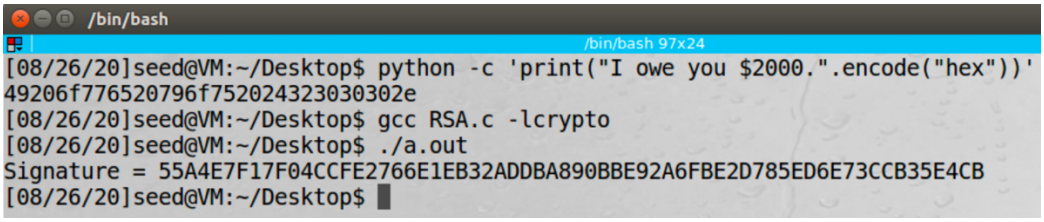


Figure 11: Output of code to produce digital signature with altered message (M)

I believe the large difference between these messages highlights the effectiveness of an RSA signature when using a suitable key pairing. It very clearly shows how a cryptographic hash function is used to map messages and shows why it is so difficult to decrypt such signatures without access to a private key.

Task 5: Verifying a Signature

In order to verify a digital signature, a recipient must have access to the sender's public keys n and e. With this information, a message v can be computed using the formula $v = S^e mod(n)$, where S is the signature received. Once the integer value v was obtained, I compared this value with the encoded value of the original message M. If both of these values are equal, then it is clear that the signature is valid, and the authenticity of the message was maintained. I computed the hexadecimal encoding of the original message *"Launch a missile."* by using the python *encode* function. I compared both messages by converting the BIGNUM types into strings and checking for equality. The completed code can be seen below in *Figure 12.*

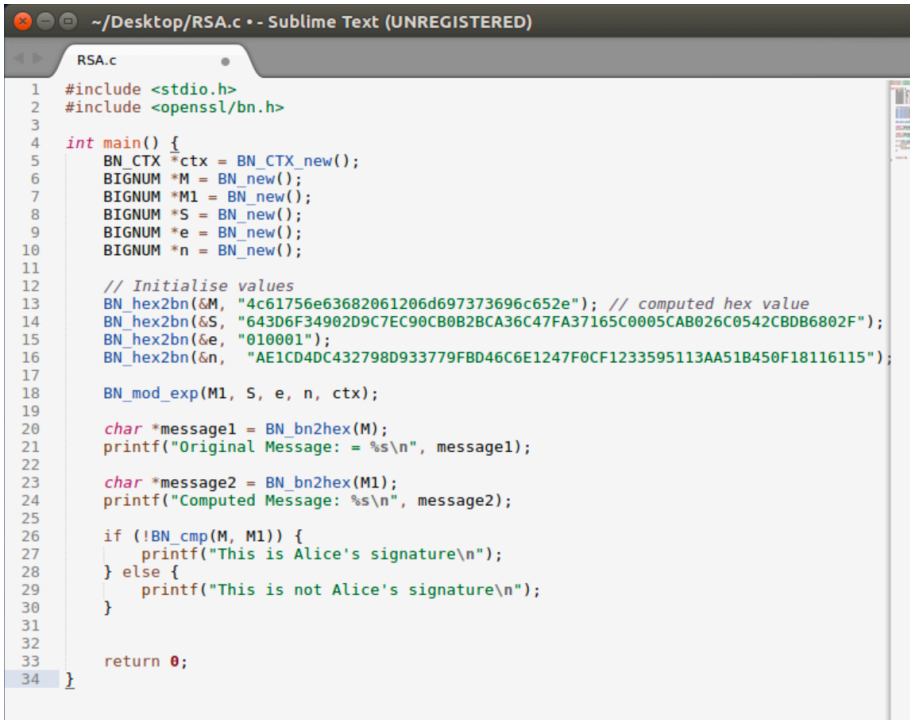


Figure 12: Code to verify a digital signature

The output of this code can be seen in *Figure 13* below. This figure clearly shows that the signature Bob received from Alice was in fact valid, and thus he can be confident that the authenticity of this message is maintained.

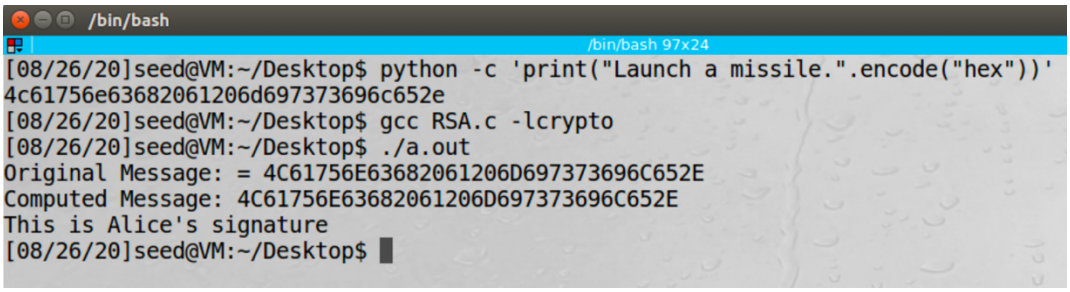


Figure 13: Output showing Bob receiving a valid signature from Alice

In order to further investigate how digital signatures work, I then altered the final two digits of the signature *S* from 2F to 3F. It was very interesting to see that despite there only being one-bit difference, the computed message was completely different to the original message. Consequently, confirming the authenticity of this message is corrupted. The output of such changes is highlighted below in *Figure 14.*

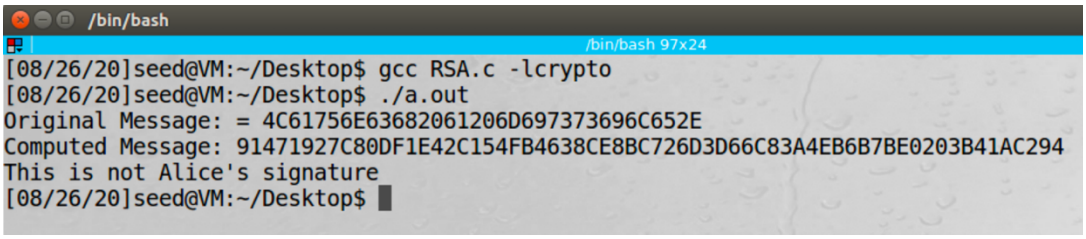


Figure 14: Output showing Bob receive an invalid signature from Alice

The results of running such a test show how effective RSA signature verification is. Despite such a minute change in the signature, the results of the operation $v = S^e mod(n)$ produced a completely new output. Consequently, this is an effective way to verify that a message was not altered in transit, since Alice's signature would not have changed unless the integrity of the message was compromised.

Buffer Overflow

Task 1: Running the Shell Code

The goal of this task is to launch a root shell through buffer overflow. In order to do this, the program *call_shellcode.c* was compiled and run. Prior to compiling and running the program, I disabled address space randomization. This is one of the counter measures to a buffer overflow attack, as it randomly arranges the address space positions of data areas of a process [2]. This prevents an attacker from jumping to a particular exploited function in a program. In Linux, this feature is automatically enabled, and therefore had to be disabled using root privileges (see *Figure 15*).

Furthermore, */bin/sh* was configured by linking it to zsh. What this meant was that due to the victim program being a Set-UID program, an attack on */bin/sh* was more difficult. Consequently, by linking */bin/sh* to another shell without such countermeasures, a buffer overflow attack is simpler. The results of such a command made with root privilege can be seein in *Figure 15*.

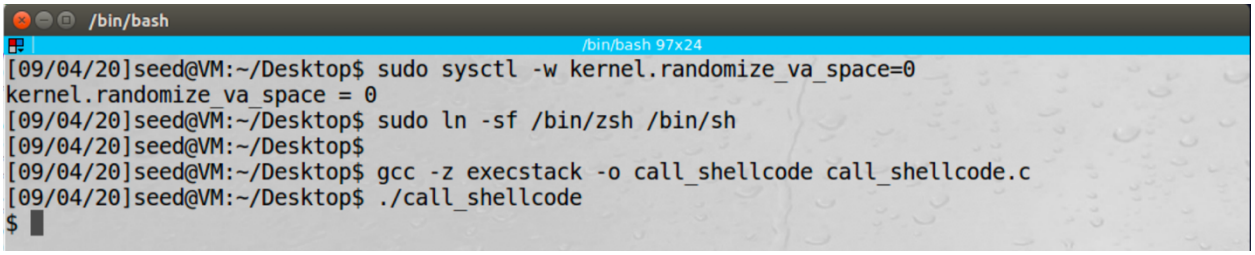


Figure 15: Running the shell code to cause buffer overflow

The *execstack* command was used in order to turn off the non-executable stack. Once the shell code was run, it is clear by the *$* symbol that the program launched a shell – although not in root form. Therefore, it is clear from completing this task that the shell code provided successfully launched a shell through buffer overflow. As a result, any vulnerable program can be executed.

Task 2: Exploiting the Vulnerability

The goal of this task was to exploit a buffer-overflow vulnerability in order to gain root privilege. In order to complete this task, the files *stack.c* and *exploit.c* were used. The program *stack.c* contains a vulnerable buffer that aims to be attacked, and *exploit.c* is program that outputs a *badfile* which is then imported into *stack.c* and is written to one of its buffers.

The first step in the process was to compile the vulnerable program, being sure to turn off non-executable stack and turning off the StackGuard protection schema. If either of these features were left enabled, then the buffer overflow attack would not be successful. The StackGuard protection schema works by inserting a small value (canary) between buffers and the function return address. If the value of this canary changes during the function return, then the program is terminated. Once these features had been disabled, I then changed the ownership of the program to the root and changed the permission to 4755 go enable the Set-UID bit. Without this alteration, the program cannot be run with the required

privileges to carry out a buffer overflow attack. The results of these commands can be seen in *Figure 16* below.

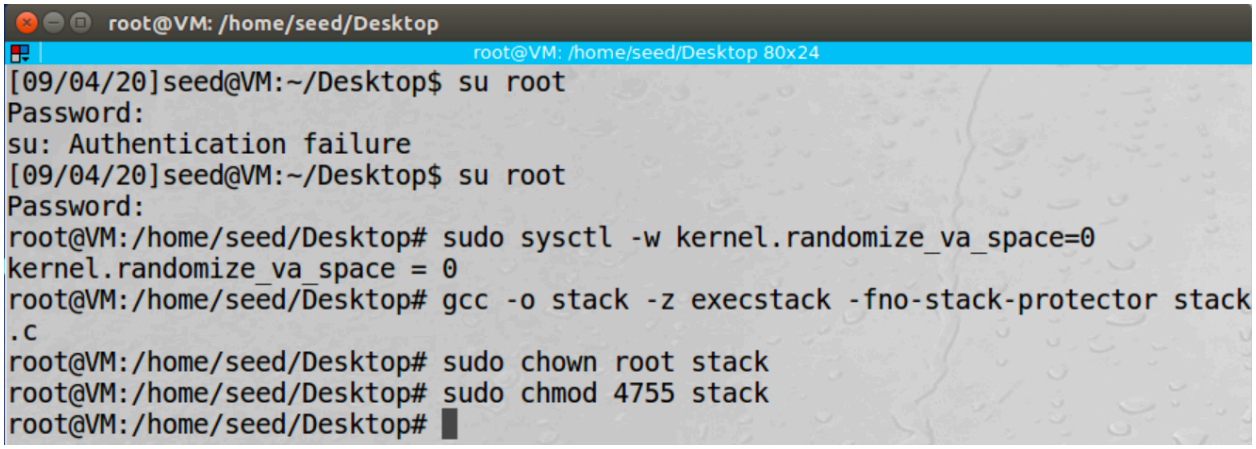


Figure 16: Compiling the vulnerable program and changing permissions

Once the vulnerable program had been compiled, the next step was to find the return address relative to the buffer such that it could be overridden when the function returns. In order to achieve this, *gdb* was used, as it allowed me to determine the base pointer address in the *bof* function in *stack.c*. By running the *disassemble* command, I was able to determine where this base pointer address was by adding a breakpoint at a position right after the initialization of the stack and base pointers. The output of the *disassemble* command can be seen below in *Figure 17.*

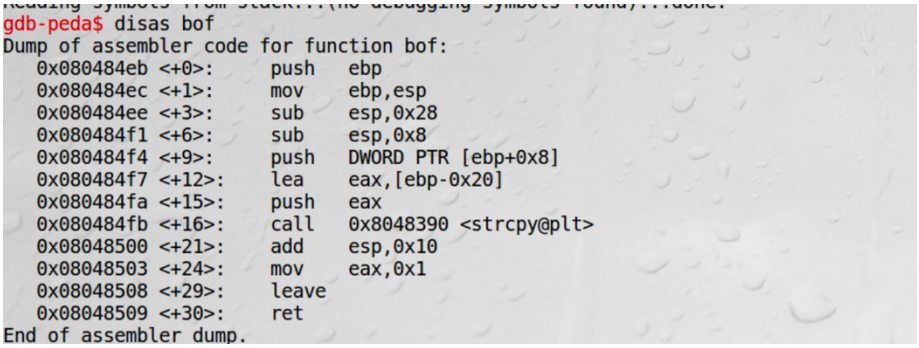


Figure 17: Result of disassembling the bof function inside stack.c

Next I was able to see the address of the base pointer by printing it to the console. I knew that the return address was just above the base pointer in the program stack. Furthermore, looking at the line at memory location *0x080484f7*, which states eax, $[ebp - 0x20]$, it is clear that the buffer is exactly 32 bytes below the base address, since it is the first local variable defined in this function. The address found at memory location can be seen in *Figure 18.*

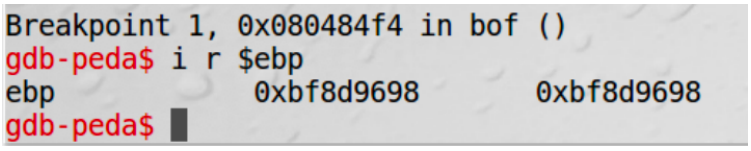


Figure 18: Address of base pointer found using gdb

I also knew that the base pointer address is 4 bytes below the return address due to its size, meaning that the location of the return address is exactly 36 bytes above the initial address of the buffer. Consequently, in my *exploit.c* file, I overwrote the return address by writing 36 bytes above the buffer for four sequential bytes. I determined the address to overwrite the return address with based on my knowledge of the contents of *badfile*. In *exploit.c*, I could see that *badfile* was 517 bytes in length, with the first 36 bytes being the base pointer contents. Following these bytes are 4 for the return address, and then a series of bytes containing NOP instructions followed by shell code.

Initially, I tried to redirect my return address to the beginning of the shellcode section of *badfile*, however found that this was not successful. Through additional research, I found that this was due to a slight alteration in memory addresses when using gdb. As a result, I instead redirected my return address to land somewhere within the chunk of memory containing NOP instructions, as these would simply execute before reaching the shellcode. Through some trial and error, and the knowledge that the NOP instructions contained roughly 450 bytes, I found that overwriting the return address with *0xbf8d9792* was successful. My code for exploit.c can be seen in *Figure 19* and shows how I then overwrote the buffer with my shellcode.

```
27    /* You need to fill the buffer with appropriate contents here */
28    *(buffer + 36) = 0x92;
29    *(buffer + 37) = 0x97;
30    *(buffer + 38) = 0x8d;
31    *(buffer + 39) = 0xbf;
32
33    int final = sizeof(buffer) - sizeof(shellcode);
34    int i;
35    for (i = 0; i < sizeof(shellcode); i++)
36        buffer[final+i] = shellcode[i];
```

Figure 19: Code that shows the return address being overwritten to access the shellcode

Once I compiled and ran the *exploit.c* file, I was able to successfully launch a root shell. The output of running these commands can be seen by the # in *Figure 20.*

Figure 20: Launched root shell

Task 3: Defeating dash's Countermeasure

This task required me to run the *dash_shell_test.c* program to see how the dash shell in Ubuntu drops privileges when the effective UID and the real UID are not equal. Firstly, I ran the program such that UIDs were not equal, and the output can be seen in *Figure 21.*

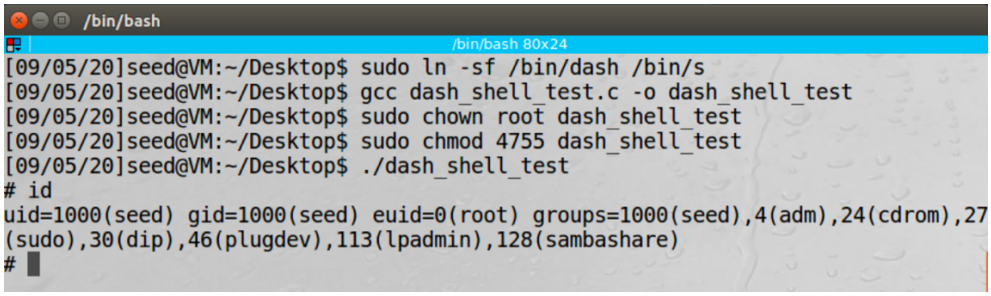


Figure 21: Output showing root privilege dropped due to UID inequality

As is shown in the figure above, the UID is equal to seed, rather than the root. This is expected given how dash checks for UID equality. *Figure 22* however shows the difference when the UID is set to zero.

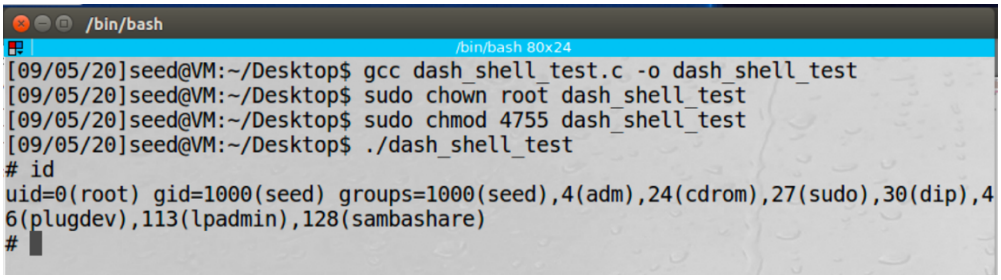


Figure 22: Output showing root privilege due to UID equality

It is clear from observing the differences in outputs that the setUID command successfully invokes the root shell. By running the program in root mode, the effective user EUID is set to root, which means that UNIX requires the actual user UID to be root as well. The line of code in the brief that explicates this is:

$$if\ (!pflag\ \&\&\ (uid\ !=\ geteuid()\ ||\ gid\ !=\ getegid()))$$

Following this, the task required me to modify the *exploit.c* program to include the *setuid(0)* system call in the shellcode. When executing the *stack.c* program, I observed that again the root shell was invoked, as seen in *Figure 23.*

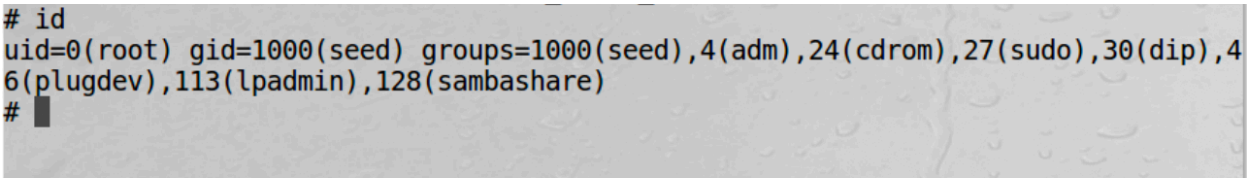


Figure 23: Root shell invoked from stack.c

This is an expected observation, as again the stack.c program is given root privileges, meaning that the EUID is set to zero. Thus, without changing the UID when executing the program, UNIX would detect an inequality and would instead launch the shell without root privileges.

Attacks on TCP/IP Protocol

Task 1: SYN Flooding Attack

The first task involved sending many SYN requests to a victim's TCP ports without completing the three-way handshake procedure. In order to complete this task, three separate VMs were created – one for the victim (10.0.2.15), one for the attacker (10.0.2.5) and one for the observer (10.0.2.4). Before beginning, it was essential to understand how a SYN flooding attack works. From reading the brief and external sources, I knew that this attack works by sending multiple SYN requests that spoof the IP address of another machine. That is why, the first step was to open a telnet connection between the two machines. This step worked by the observer sending a SYN request, with the victim then acknowledging it was sent and finally the observer again confirming the three-way connection. This created an open connection between both machines, and the command to achieve this can be seen in *Figure 24* below.

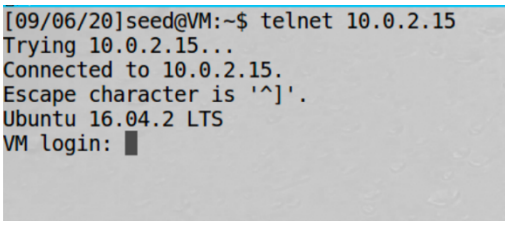


Figure 24: Connection between Observer and Victim machines

Once the telnet connection was established, the next step was to disable the SYN Cookie countermeasure to ensure a successful attack. This was simply disabled by using the command shown in *Figure 25*.

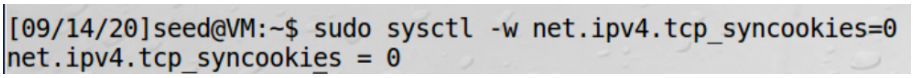


Figure 25: Disabling of SYN Cookie countermeasure

Next, since the initial connection between the observer and victim had been made, it was possible to carry out the SYN flood attack. This was done by using the *netwox 76* command, as well as the IP address of the victim machine and the port destination. This command is shown below in *Figure 26.*

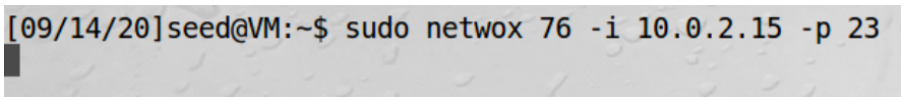


Figure 26: Command for SYN Flood from Attacker machine

As shown in *Figure 27*, the result of this command was multiple SYN requests being sent to port 23 of the victim machine. However, since the attacker was not acknowledging the victim's acknowledgement, the requests were not being completed. As a result, when running the command *netstat -tna*, there were many SYN_RECV connections. Consequently, the queue for requests was filled with IP address connections from the attacker.

```
[3;J
[09/06/20]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:*               LISTEN
tcp        0      0 10.0.2.15:53            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:953           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp        0      0 10.0.2.15:23            253.76.243.58:56981     SYN_RECV
tcp        0      0 10.0.2.15:23            249.169.253.204:15873   SYN_RECV
tcp        0      0 10.0.2.15:23            247.3.192.15:58171      SYN_RECV
tcp        0      0 10.0.2.15:23            255.218.231.109:49157   SYN_RECV
tcp        0      0 10.0.2.15:23            245.56.114.196:27386    SYN_RECV
tcp        0      0 10.0.2.15:23            241.55.56.246:50939     SYN_RECV
tcp        0      0 10.0.2.15:23            242.239.208.22:54467    SYN_RECV
tcp        0      0 10.0.2.15:23            241.183.108.121:28716   SYN_RECV
tcp        0      0 10.0.2.15:23            244.229.123.142:36317   SYN_RECV
tcp        0      0 10.0.2.15:23            255.83.106.88:64296     SYN_RECV
tcp        0      0 10.0.2.15:23            241.217.168.65:2701     SYN_RECV
tcp        0      0 10.0.2.15:23            250.10.237.104:14083    SYN_RECV
```

Figure 27: Resulting connections to victim machine after SYN Flooding

In order to confirm the success of the SYN flood attack, the observer VM was used to try and connect to the victim machine. If the SYN flood attack was successful, then this should not be able to occur as the queue of requests is too large to process any additional connections. As is shown by *Figure 28,* this was what occurred. This is an expected outcome given the disabling of the SYN cookie countermeasure, meaning that the victim machine had no real counter measure to the attack. It was interesting observation that the observer tried to connect to the attacker for some time, before timing out.

```
[09/14/20]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
telnet: Unable to connect to remote host: Connection timed out
```

Figure 28: Successful SYN Flood shows Observer unable to connect to Victim

Once I had a good idea of the impacts of the SYN cookie countermeasure, I thought that enabling it and retrying the attack may output interesting results. In order to do this, the countermeasure was re-enabled using the same command as previously, however setting the bit to one instead of zero. This can be seen in *Figure 29.*

```
[09/06/20]seed@VM:~$ sudo sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
[09/06/20]seed@VM:~$
```

Figure 29: Enabling of SYN Cookie countermeasure

After going through the same steps as detailed earlier, and again running the *netstat -tna* command on the victim's virtual machine, it was interesting to see that this time the observer was able to establish a telnet connection. This clearly explicates the success of the SYN cookie countermeasure, as it prevented the victim machine's queue from filling with unfinished SYN requests. This was due to the victim machine not sending SYN-ACK requests back to the attacker machine, meaning it wasn't constantly waiting for confirmation. Consequently, the victim was able to acknowledge the observer's connections requests, hence completing the three-way handshake protocol.

```
tcp        0      0 10.0.2.15:23            244.190.76.70:19043     SYN_RECV
tcp        0      0 10.0.2.15:23            254.210.56.33:22849     SYN_RECV
tcp        0      0 10.0.2.15:23            251.127.175.12:51805    SYN_RECV
tcp        0      0 10.0.2.15:23            246.119.11.16:25754     SYN_RECV
tcp        0      0 10.0.2.15:23            244.130.170.217:32123   SYN_RECV
tcp        0      0 10.0.2.15:23            243.20.213.168:18172    SYN_RECV
tcp        0      0 10.0.2.15:23            246.31.12.97:61402      SYN_RECV
tcp       27      0 10.0.2.15:23            10.0.2.5:51170          ESTABLISHED
tcp        0      0 10.0.2.15:23            247.185.67.237:49006    SYN_RECV
tcp        0      0 10.0.2.15:23            250.117.215.11:18627    SYN_RECV
tcp        0      0 10.0.2.15:23            249.200.76.233:15405    SYN_RECV
tcp        0      0 10.0.2.15:23            250.33.206.192:40195    SYN_RECV
tcp        0      0 10.0.2.15:23            249.175.36.61:13795     SYN_RECV
tcp        0      0 10.0.2.15:23            252.215.140.211:5208    SYN_RECV
tcp        0      0 10.0.2.15:23            250.65.64.89:5215       SYN_RECV
tcp        0      0 10.0.2.15:23            247.102.154.170:29005   SYN_RECV
tcp        0      0 10.0.2.15:23            255.67.173.131:35385    SYN_RECV
tcp        0      0 10.0.2.15:23            248.37.136.37:61003     SYN_RECV
```

Figure 30: Successful connection to victim server after SYN Flood attack

Task 2: TCP RST Attacks on telnet and ssh Connections

This task involved carrying-out a TCP RST attack on both telnet and ssh connections. Ideally, the attack machine should spoof an RST packet when two machines are already connected. Again, this task required the use of three virtual machines, however for simplicity, they were renamed to Alice (10.0.2.15), Bob (10.0.2.4) and Attacker (10.0.2.5). This is shown explicitly in the figure below.

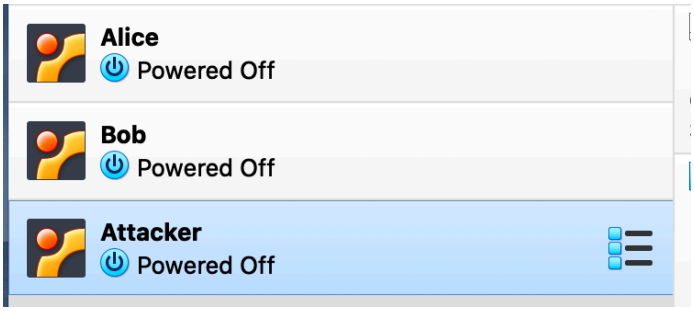


Figure 31: Setup of VMs for task

The first step for this task was to establish a telnet connection between Alice and Bob. This was achieved from Bob's machine, and used the command shown below in *Figure 32.*

```
[09/14/20]seed@VM:~$ sudo telnet 10.0.2.15
Trying 10.0.2.15...
Connected to 10.0.2.15.
```

Figure 32: Telnet connection between Bob and Alice

Next, the Attacker used the *netwox 78* command to carry out the TCP RST attack. The tags for this command specified the device as *"Eth0"*, the filter to be the host Bob, and the target IP to be Alice. At first, I found it interesting that Bob was recipient of this command instead of

Alice. However, after further research I understood that the Attacker used Bob's connection with Alice to obtain the IP, and then used Bob to construct the RST packets that break the connection. The command that was used including the tags are outlined in *Figure 33.*

```
[09/14/20]seed@VM:~$ sudo netwox 78 --device "Eth0" --filter "host 10.0.2.4" --s
poofip "raw" --ips 10.0.2.15
```

Figure 33: TCP RST attack on Alice

Once the command was executed from the Attacker machine, *Wireshark* was opened in Alice's VM. The results of running the program were interesting, and better visualized the effects of the attack. What was evident immediately after opening *Wireshark* was the number of spoofed TCP RST packets. This made it very clear that the attack had been successful (see *Figure 34*).

70	2020-09-14 21:26:20.6986939…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
71	2020-09-14 21:26:20.6987054…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
72	2020-09-14 21:26:20.6987062…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
73	2020-09-14 21:26:20.6987683…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
74	2020-09-14 21:26:20.6989520…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
75	2020-09-14 21:26:20.6990320…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23
76	2020-09-14 21:26:20.6990339…	10.0.2.4	10.0.2.15	TCP	62	35036 → 23

Figure 34: Wireshark capture showing Alice incoming packets

In addition to using *Wireshark* to determine the attack's success, it was further explicated by the terminal message outputted in both Alice and Bob's machines. This message is shown in *Figure 35* and makes it clear that due to the spoofed packets sent to Alice, the connection was broken. I believe that this was due to the reset packet header being set to reset in the RST TCP packets.

```
[09/14/20]seed@VM:~$ netstat -tnaConnection closed by foreign host.
[09/14/20]seed@VM:~$
```

Figure 35: Closed connection after attack on Alice

Once the TCP RST attack on a telnet connection had been shown to work as intended, the next step was to essentially repeat the process on an ssh connection. This process was relatively similar to the last, but instead of connecting Alice and Bob using telnet, an ssh command was instead used. For this example, Alice sent an ssh request to Bob using the following command (*Figure 36*).

```
[09/14/20]seed@VM:~$ ssh 10.0.2.4
```

Figure 36: ssh connection from Alice to Bob

The Attacker VM was then used again to carry-out the TCP RST attack. A slight difference was that the filter was set to port 22, which is the port used to establish ssh connections. These changes are shown in *Figure 37.*

```
[09/14/20]seed@VM:~$ sudo netwox 78 --device "Eth0" --filter "port 22" --spoofip
 "raw" --ips 10.0.2.4
```

Figure 37: SSH TCP RST attack

After running the command from the Attacker VM, the next step was to again use *Wireshark* to visualize the results of the attack. By setting the IP of the attack to spoof that of Bob, it meant that Alice again received many packets via their connection. Furthermore, by specifying the use of port 22 in the attack, it meant that these attacks could be carried out, since there was a valid ssh connection. Consequently, *Wireshark* confirmed that Alice had received many spoofed TCP RST packets. Additionally, when looking at more information about the TCP, it showed that these packets were being sent via port 22. This is visualized in *Figure 38.* Much like the original telnet task, the ssh connection between Alice an Bob was broken after the command was executed.

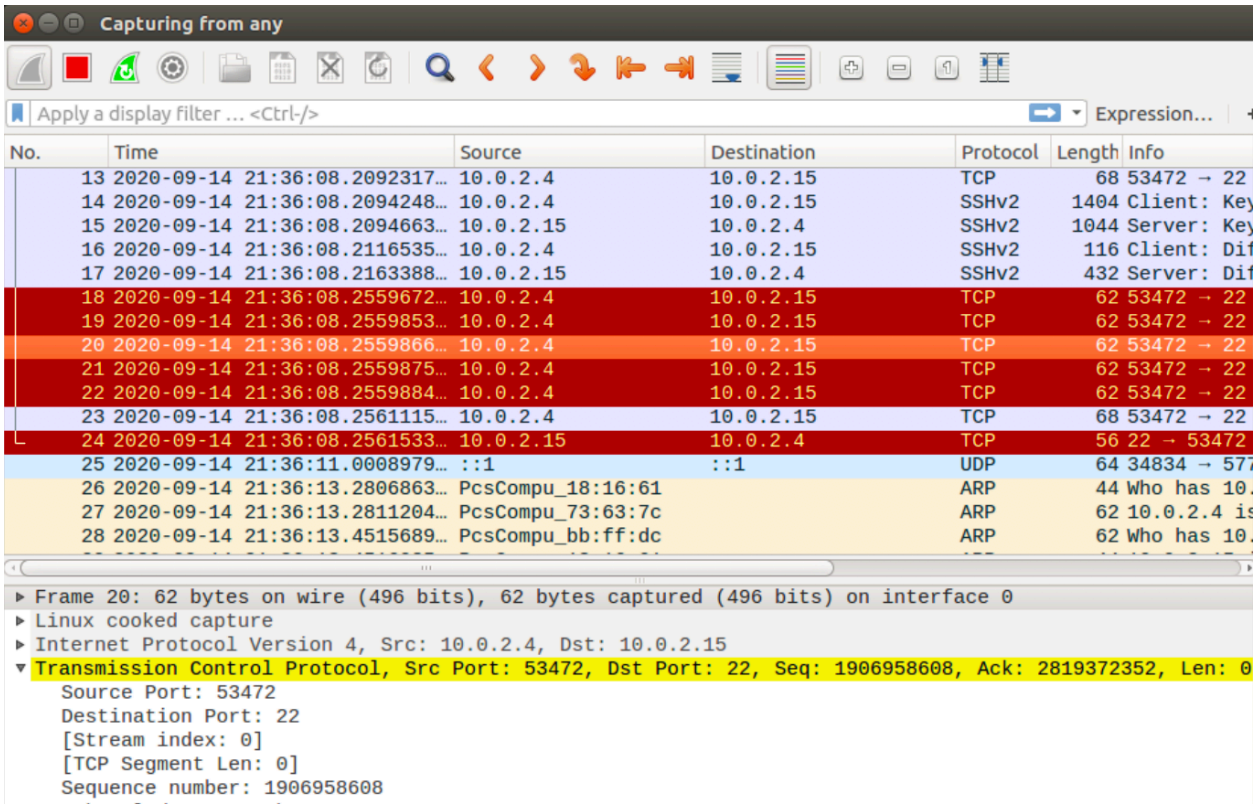


Figure 38: Wireshark capture from Alice machine showing destination port

Task 3: TCP RST Attacks on Video Streaming Applications

The final task involved conducting a TCP RST attack on a video streaming application. This task definitely helped me better understand how disrupting TCP sessions impacts a victim's ability to perform tasks. For this task, Alice (10.0.2.15) and the Attacker (10.0.2.5) VMs were

used. I decided to use YouTube as the video streaming platform, as I found the loading buffer to be clear to see and the videos performed better on my virtual machine. A random video was found from YouTube and played at 720p prior to any attack, to confirm that the results of this task were accurate. The video being played before the TCP RST attack is shown below in *Figure 39.*

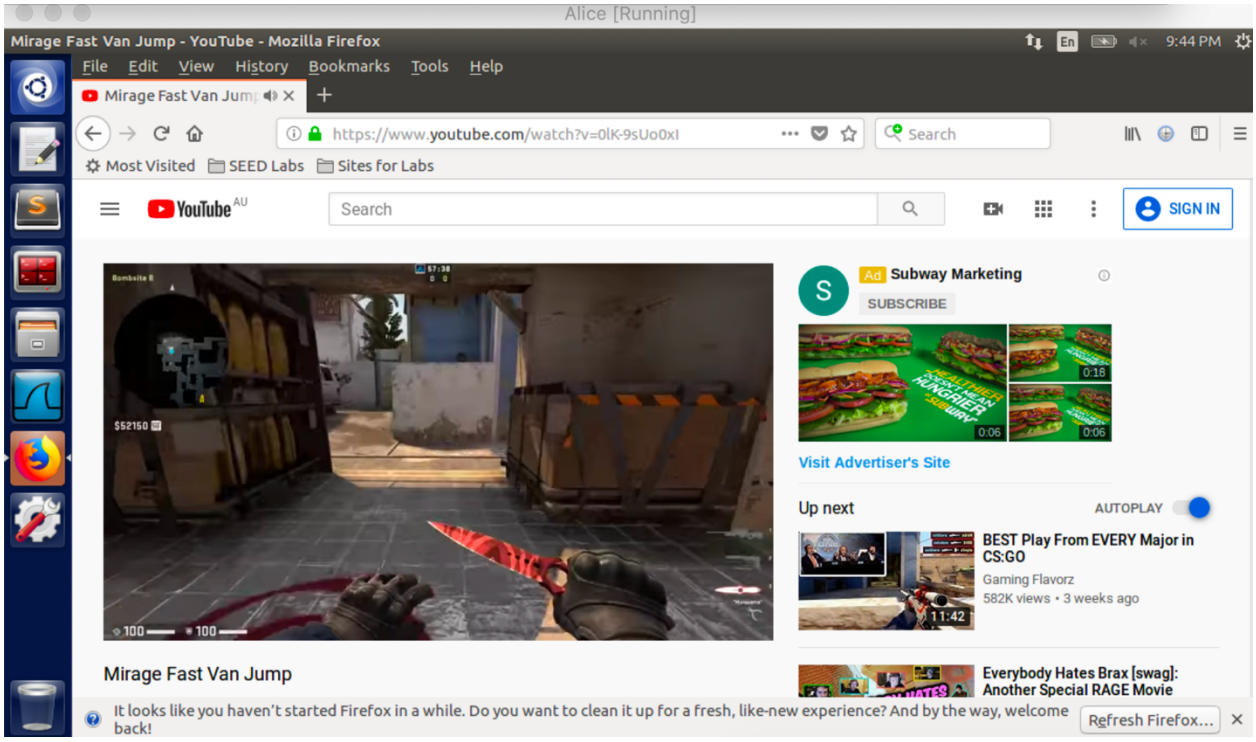


Figure 39: Normal streaming of Youtube video before attack

After forming a TCP connection with the YouTube server, the next step was to target a TCP RST attack towards Alice through the Attacker VM. Again, the *netwox 78* command was used, and the victim IP address was set to that of Alice (*Figure 40*).

```
[09/14/20]seed@VM:~$
[09/14/20]seed@VM:~$ sudo netwox 78 --filter "src host 10.0.2.15"
```

Figure 40: Attacker command for TCP RST attack on victim Alice

The impacts of this command became clear very quickly, with the video beginning to buffer. An explanation for this is that the queue of Alice was being filled with packets from the attacker, rather than ones from the YouTube server. As a result, the video was unable to be played, resulting in it being stuck in a constant buffer (*Figure 41*).

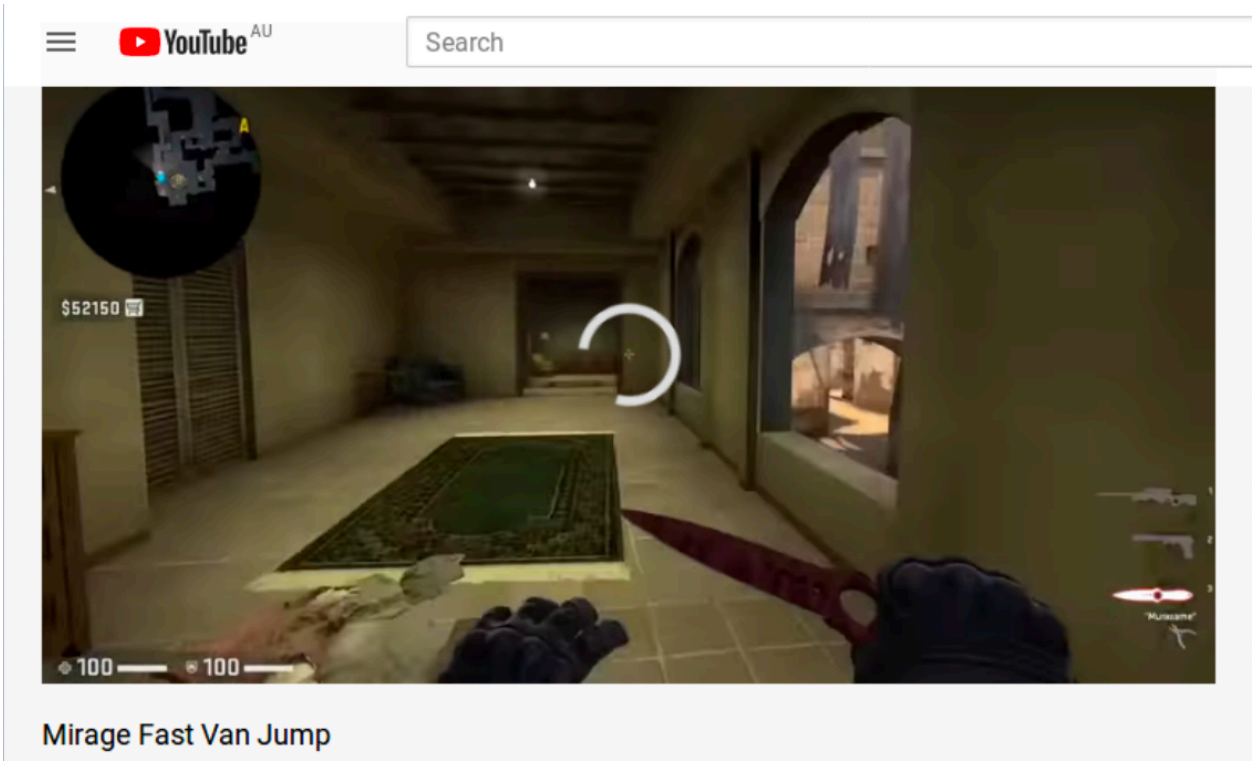


Figure 41: Buffering video on victim VM due to TCP RST attack

In addition to the obvious video buffering, a look at *Wireshark* confirmed the attacker's success. There was a constant stream of packets being sent to Alice after the command was executed by the attacker, as is shown in *Figure 42*, with the high number of RST packets disrupting the TCP connection with YouTube.

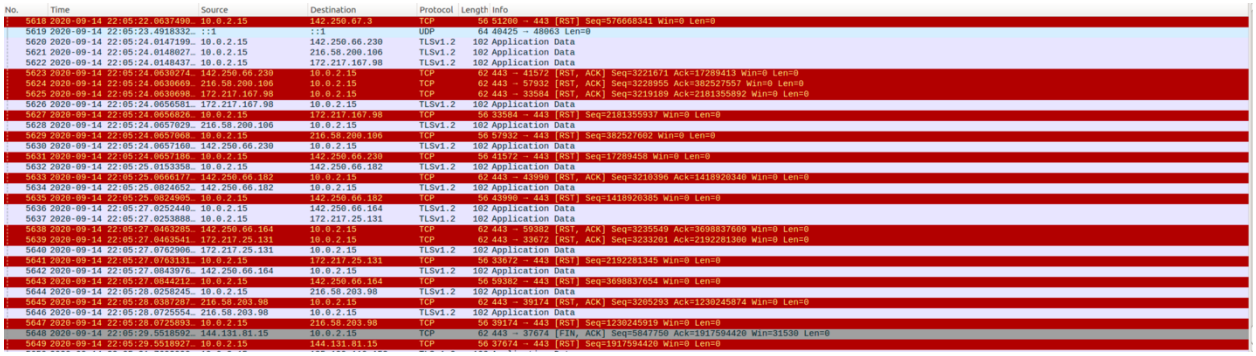
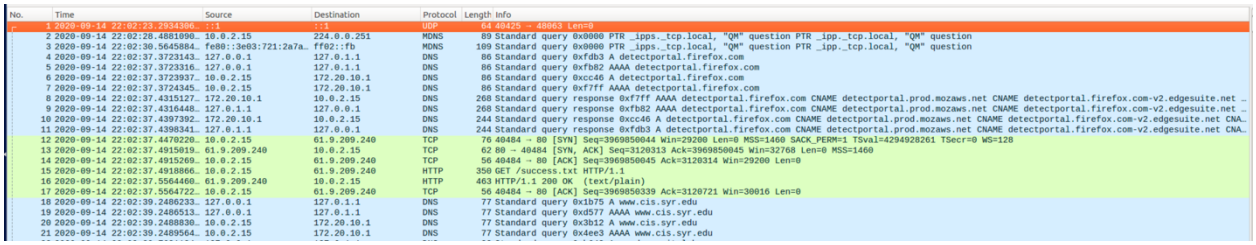


Figure 42: Wireshark capture showing packets sent to victim

To further investigate the impacts of the attack, I thought it would be interesting to kill the command being run by the Attacker VM. After a small amount of time, whereby Alice was processing and removing the large number of packets, the video began to play normally. Additionally, *Wireshark* appeared to display a normal, undisrupted connection between Alice and the YouTube server (*Figure 43)*.



No.	Time	Source	Destination	Protocol	Length	Info
1	2020-09-14 22:02:23.2934306…	::1	::1	UDP	64	40425 → 48063 Len=0
2	2020-09-14 22:02:28.4881090…	10.0.2.15	224.0.0.251	MDNS	89	Standard query 0x0000 PTR _ipps._tcp.local, "QM" question PTR _ipp._tcp.local, "QM" question
3	2020-09-14 22:02:30.5645884…	fe80::3e03:721:2a7a…	ff02::fb	MDNS	109	Standard query 0x0000 PTR _ipps._tcp.local, "QM" question PTR _ipp._tcp.local, "QM" question
4	2020-09-14 22:02:37.3723143…	127.0.0.1	127.0.1.1	DNS	86	Standard query 0xfdb3 A detectportal.firefox.com
5	2020-09-14 22:02:37.3723316…	127.0.0.1	127.0.1.1	DNS	86	Standard query 0xfb82 AAAA detectportal.firefox.com
6	2020-09-14 22:02:37.3723937…	10.0.2.15	172.20.10.1	DNS	86	Standard query 0xcc46 A detectportal.firefox.com
7	2020-09-14 22:02:37.3724345…	10.0.2.15	172.20.10.1	DNS	86	Standard query 0xf7ff AAAA detectportal.firefox.com
8	2020-09-14 22:02:37.4315127…	172.20.10.1	10.0.2.15	DNS	268	Standard query response 0xf7ff AAAA detectportal.firefox.com CNAME detectportal.prod.mozaws.net CNAME detectportal.firefox.com-v2.edgesuite.net …
9	2020-09-14 22:02:37.4316448…	127.0.1.1	127.0.0.1	DNS	268	Standard query response 0xfb82 AAAA detectportal.firefox.com CNAME detectportal.prod.mozaws.net CNAME detectportal.firefox.com-v2.edgesuite.net …
10	2020-09-14 22:02:37.4397392…	172.20.10.1	10.0.2.15	DNS	244	Standard query response 0xcc46 A detectportal.firefox.com CNAME detectportal.prod.mozaws.net CNAME detectportal.firefox.com-v2.edgesuite.net CNA…
11	2020-09-14 22:02:37.4398341…	127.0.1.1	127.0.0.1	DNS	244	Standard query response 0xfdb3 A detectportal.firefox.com CNAME detectportal.prod.mozaws.net CNAME detectportal.firefox.com-v2.edgesuite.net CNA…
12	2020-09-14 22:02:37.4470220…	10.0.2.15	61.9.209.240	TCP	76	40484 → 80 [SYN] Seq=3969850044 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=4294928261 TSecr=0 WS=128
13	2020-09-14 22:02:37.4915019…	61.9.209.240	10.0.2.15	TCP	62	80 → 40484 [SYN, ACK] Seq=3120313 Ack=3969850045 Win=32768 Len=0 MSS=1460
14	2020-09-14 22:02:37.4915269…	10.0.2.15	61.9.209.240	TCP	56	40484 → 80 [ACK] Seq=3969850045 Ack=3120314 Win=29200 Len=0
15	2020-09-14 22:02:37.4918866…	10.0.2.15	61.9.209.240	HTTP	350	GET /success.txt HTTP/1.1
16	2020-09-14 22:02:37.5564460…	61.9.209.240	10.0.2.15	HTTP	463	HTTP/1.1 200 OK (text/plain)
17	2020-09-14 22:02:37.5564722…	10.0.2.15	61.9.209.240	TCP	56	40484 → 80 [ACK] Seq=3969850339 Ack=3120721 Win=30016 Len=0
18	2020-09-14 22:02:39.2486233…	127.0.0.1	127.0.1.1	DNS	77	Standard query 0x1b75 A www.cis.syr.edu
19	2020-09-14 22:02:39.2486513…	127.0.0.1	127.0.1.1	DNS	77	Standard query 0xd577 AAAA www.cis.syr.edu
20	2020-09-14 22:02:39.2488830…	10.0.2.15	172.20.10.1	DNS	77	Standard query 0x3b12 A www.cis.syr.edu
21	2020-09-14 22:02:39.2489564…	10.0.2.15	172.20.10.1	DNS	77	Standard query 0x4ee3 AAAA www.cis.syr.edu

Figure 43: Wireshark showing normal TCP connection after attack ended

Bibliography

[1] E. Altili, "Cryptography, Encryption, Hash Functions and Digital Signature," 18 February 2018. [Online]. Available: https://medium.com/@ealtili/cryptography-encryption-hash-functions-and-digital-signature-101-298a03eb9462. [Accessed 26 August 2020].

[2] D. Stewart, "What Is ASLR, and How Does It Keep Your Computer Secure?," 26 October 2016. [Online]. Available: https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/. [Accessed 4 September 2020].