Assignment 2 Sliding Part

Question 1

- To get the public/private key pair, we used the command "openssl genrsa -des3 -out ca.key 4096". This command creates a key of 4096 bits using des3. We then use this key to create the certificate using the command "openssl req -config ~/../etc/ssl/openssl.cnf -new -x509 -extensions v3_ca -keyout ca.key -out ca.crt -days 365"
- Password to key files is "KitHpwuaaUni". "ca.crt" is the Certificate Authorities crt, "ca.key" is the Certificate Authorities key. The ca.crt file is important because it is required to be used with the ca.key to sign a server certificate.

Question 2

- Using the command "openssl genrsa -des3 -out server.key 4096" we create a key with 4096 bits long using des3. This server.key file is then used to create the Servers Certificate request file using the command "openssl req -new -key server.key -out server.csr". This will take the key and use it to create the .csr file.
- "server.csr" is the Servers Certificate request. This file is used as an in parameter with ca.key and ca.crt to sign the server certificate so that it will be an "official and trusted" certificate.

Question 3

- Using the command "openssl ca -policy policy_anything -config /etc/ssl/openssl.cnf -days 365 -in server.csr -CA ca.crt -CAkey ca.key -set_serial 01 -out server.crt" we create the Servers Certificate. This uses the certificate request file along with the the key and certificate that is used to be the certificate authority named ca.cy and ca.key. We allow the certificate to exist for one year until you it expires and a new one has to be created. We set serial to be 01 so that when it comes time to renew the certificate we can set it to 0n where n is the newest number of certificates (02 for 2nd cert, 03 for 3rd cert).
- We need to use policy_anything as the policy because the default is policy_match which needs country Name, state or Province Name, and organization Name to match exactly with the addition of common Name being provided. Policy_anything allows that only common Name is supplied and all other parameters are optional. So if we did not set the policy to be policy_anything then it would have failed because of the exact matching

of the country Name, state or province name and organization name may or may not be exact or provided.

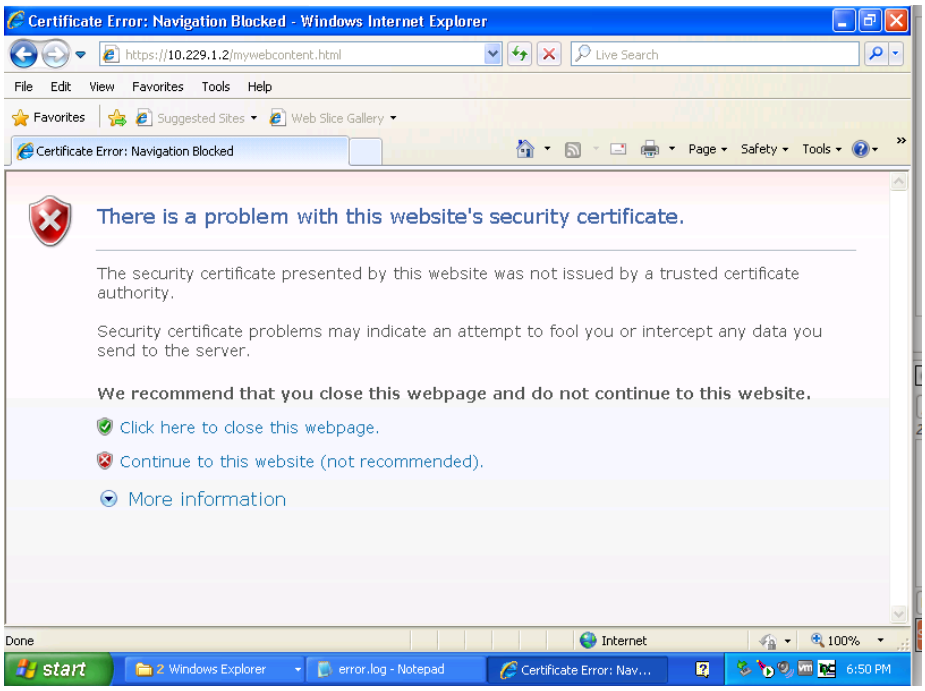
Question 4

Modifying the http-ssl.conf file located in the Apache files, edit the fields such that you include the needed SSL requirements. Look into the apache files to gather correct items.

- We first start by allowing port 443 through the windows firewall.
- In the http.conf file located wherever you have Apache installed, we first add the LoadModule ssl_module modules/mod_substitute.so at the top of the conf file. This initially loads in all the sll information that we need to use to start using Certificate Authorities. We also need to add "Include conf/extra/httpd-ssl.conf" to the http.conf file as this is where the is set up. The path to httpd-ssl.conf is wherever you have that file located.
- We then need to edit so that we can use our certificate. Ensure that this conf file is listening on port 443 and that the header of is in the format of where *default* is the ip address of the host server.
- Mapping the correct certificate files and keys:
- Server Certificate: SSLCertificateFile "C:/Program Files/Apache Software Foundation/Apache2.2/conf/server.crt". This needs to be set to whatever you have as your server.crt.
- Server Private Key: SSLCertificateKeyFile "C:/Program Files/Apache Software Foundation/Apache2.2/conf/server.key". This needs to be set to whatever you have as your server.key.
- Certificate Authority (CA): SSLCACertificateFile "C:/Program Files/Apache Software Foundation/Apache2.2/conf/ca.crt". This needs to be set to whatever you have as the CA.

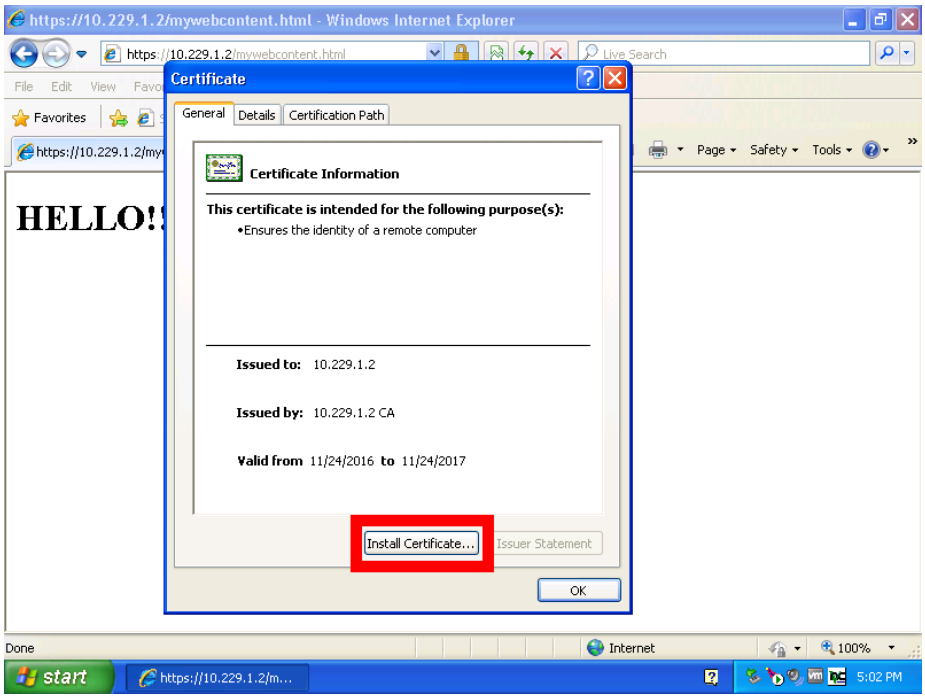
Question 5

Screenshot of web server not recognizing the certificate is called "BeforeCertificate.png".

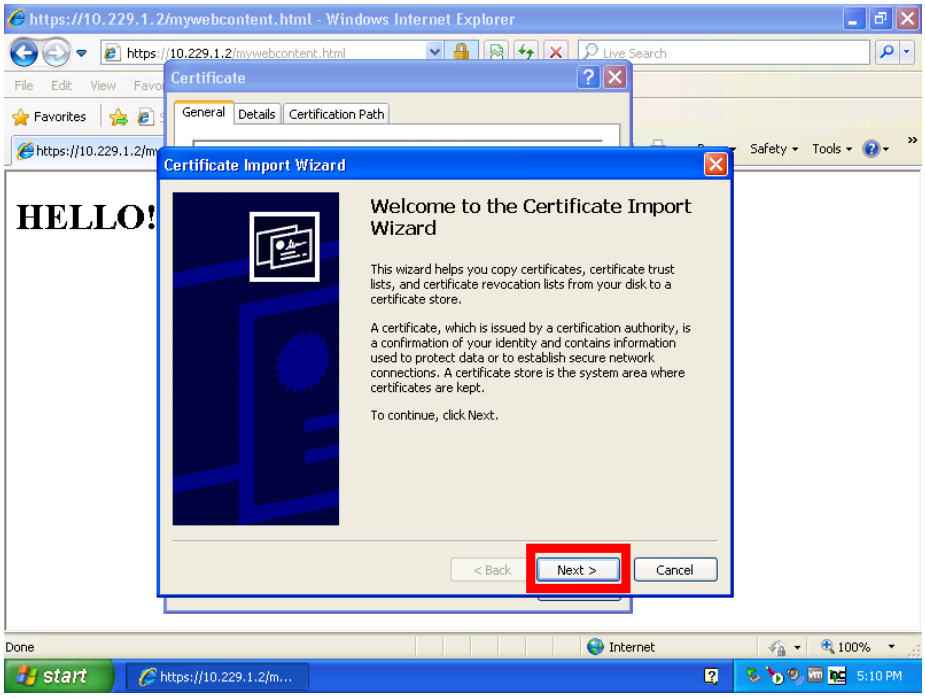


Steps to install certificate:

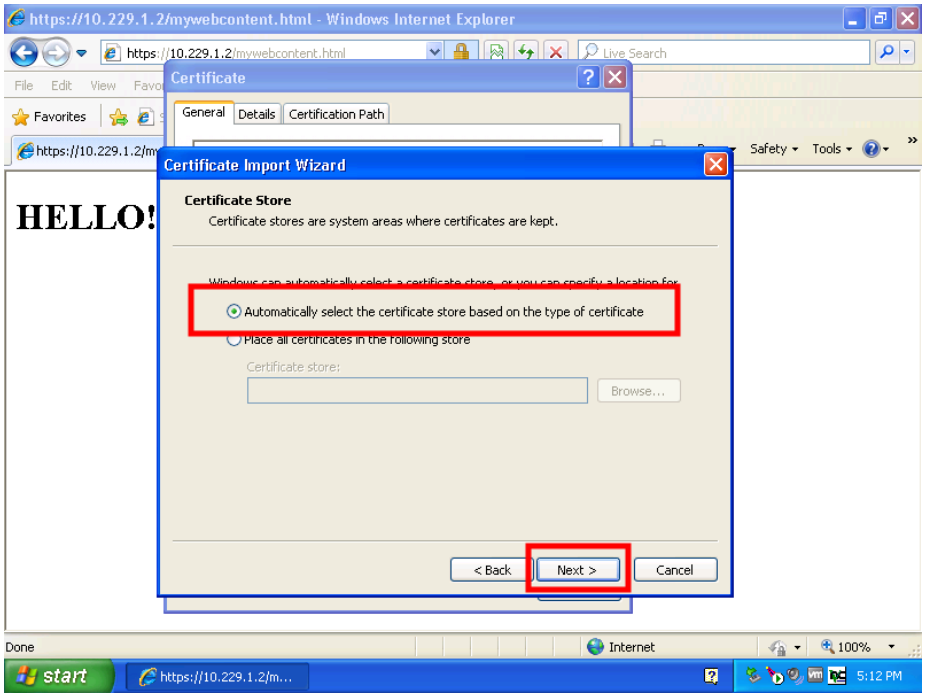
- Step 1: View the certificate that is unrecognized. Select Install Certificate at the bottom of the window after viewing the certificate.



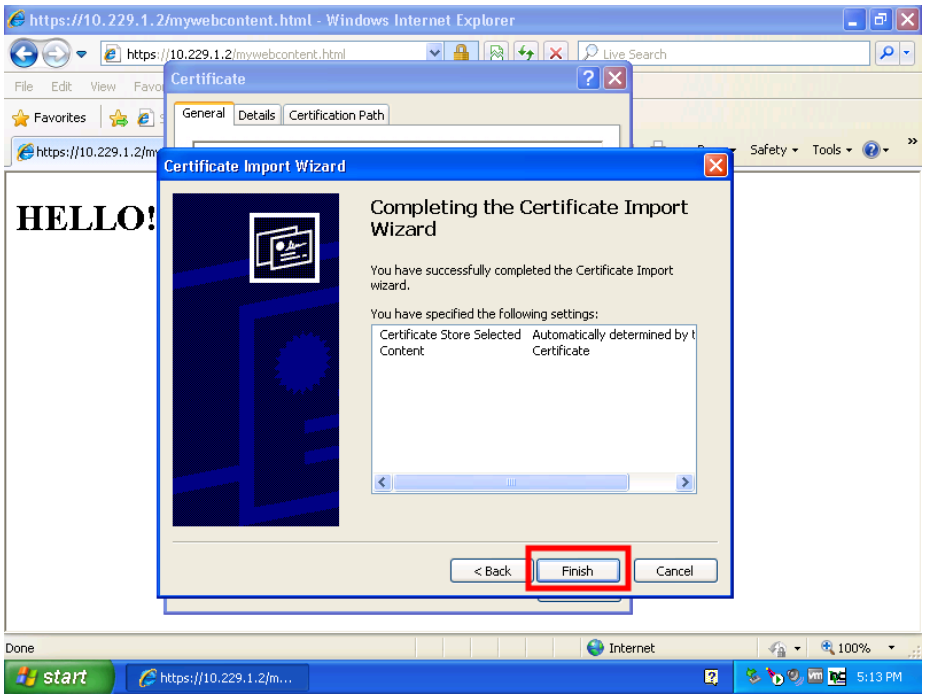
- Step 2: Select the Next button.



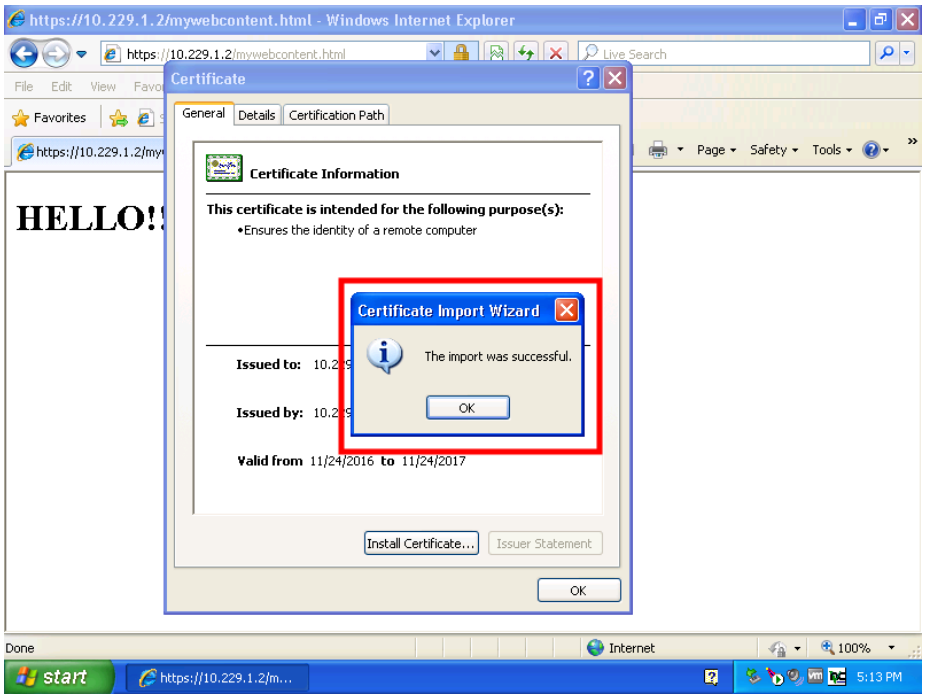
- Step 3: Select the first radio button (Automatically select . . .). Then select the Next button.



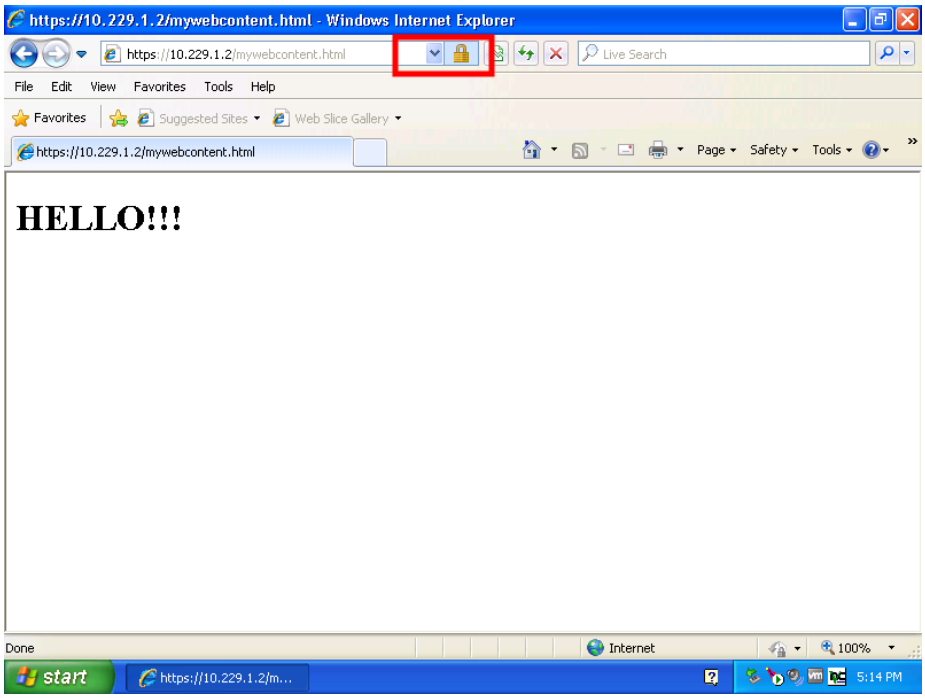
- Step 4: Select the Finish button.



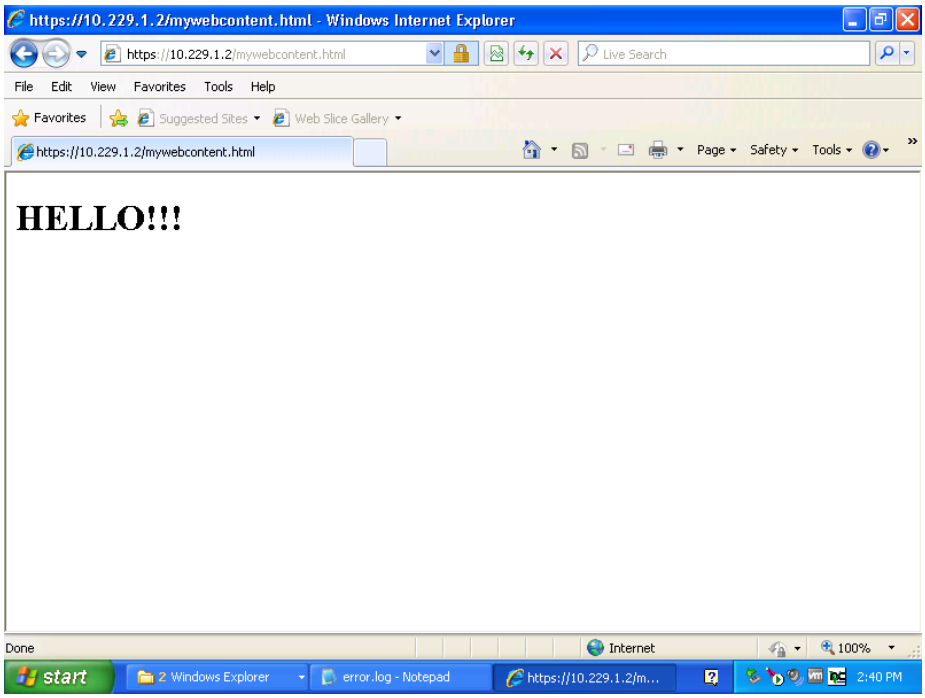
- Step 5: Import should be successful and show a dialog box. Select OK button.



- Step 6: Near the address bar, there should be a lock image to show that the certificate has been successful and approved.



Screenshot of web server successfully loading in the certificate is called "After-Certificate.png"



- Some of the problems that we encountered was after installation of the certificate, it still said that it was an untrusted site even though it accepted the certificate. Closing the explorer and restarting the windows VM seemed to solve that issue though.

Question 6

- Generating signed Java code is very similar to generating a signed certificate authority for a web server.
- We start out by using a command similar to "keytool -genkey -alias server -keyalg RSA -keysize 2048 -keystore keystore.jks".
- This command will create a Java keystore called keystore.jks. This item is similar to the ca.key in that it takes a keysize and a generated random value for RSA encryption.
- We will use this keystore to generate the .csr file.
- We then enter a command similar to "keytool -certreq -alias server -file csr.csr -keystore keystore.jks".
- This command will create a .csr, which is a certificate signing request using the keystore that we created earlier.

- Again this is similar to the web server as it needs a ca.key to create the server.csr.
- This is where some differences start (although there are some differences in the be)

Part 1

Step 1

Question a

The IP addresses of the victim hosts connected to the backbone are *10.229.100.55*, *10.229.100.101*, and *10.229.100.102*.

The procedure used to discover those victim hosts are running the following command on the linux firewall virtual machine (*cs333fw001*):

```
nmap 10.229.100.14-94
nmap 10.229.100.98-255
```

The reason for skipping IP address ranges *10.229.100.1-13* and *10.229.100.95-97* is that those ranges belong to the hosts assigned to each student group and designated machines used by TAs, respectively.

Question b

The services running on the victim host *10.229.100.55* are the following:

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

Port	State	Service	Version
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

The services running on the victim host *10.229.100.101* are:

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)

The services running on the victim host *10.229.100.102* are:

Port	State	Service	Version
22/tcp	open	tcpwrapped	

Port	State	Service	Version
111/tcp	open	rpcbind	2 (rpc #100000)
113/tcp	open	ident	OpenBSD identd
6000/tcp	open	X11	(access denied)

And the *nmap* commands used to determine the above information are:

```
nmap -A 10.229.100.14-94
nmap -A 10.229.100.98-255
```

Question c

The OS information for the victim host *10.229.100.55* are the following:

Running	OS details
Microsoft Windows 95/98/ME/NT/2K/XP	Microsoft Windows Millennium Edition (Me), Windows 2000 Professional or Advanced Server, or Windows XP

The OS information for the victim host *10.229.100.101* are:

Running	OS details
Linux 2.4.X/2.5.X/2.6.X	Linux 2.4.0 - 2.5.20, Linux 2.4.7 - 2.6.11

The OS information for the victim host *10.229.100.101* are:

Running	OS details
Linux 2.4.X/2.5.X/2.6.X	Linux 2.4.0 - 2.5.20, Linux 2.4.7 - 2.6.11

And the *nmap* commands used to obtain the above information are exactly the same as the ones used in the previous question:

```
nmap -A 10.229.100.14-94
nmap -A 10.229.100.98-255
```

Step 2

Question a

The victim hosts can be determined by following the instructions shown in step

one.

To capture the network traffic through ARP poisoning among three victim hosts *10.229.100.55*, *10.229.100.101*, and *10.229.100.102*, issue the following *ettercap* command:

```
ettercap -Q -T -w arp.pcap -M arp /10.229.100.55,101,102/
```

in this case the captured output is recorded in a file named *arp.pcap*; later on either **Wireshark** or **tcpdump** can be used to analyze the network traffic based on this file.

Question b

The connections initiated among three victim hosts are a series of **HTTP** requests, the extracted repeating communication pattern is recorded in the following table (note the *frame number* are for reference only):

Frame Number	Source	Destination	Source Role	Destination Role
457	10.229.100.101	10.229.100.55	HTTP Client	HTTP Server
48590	10.229.100.102	10.229.100.55	HTTP Client	HTTP Server

Within the thirty minutes of the *ettercap* run, host *10.229.100.101* sends **HTTP GET** request to the **HTTP** server at *10.229.100.55* to get a file named */Test2016/sound.mp3* on a regular basis while the other host *10.229.100.102* sends another **HTTP GET** request to the **HTTP** server at *10.229.100.55* to get a different file named */Test2016/image.jpg* from time to time.

Question c

The two client machines with IP addresses 10.229.100.101 and 10.229.100.102 are
connected to each other via ssh on port 22. Both clients are requesting files from the Microsoft IIS WebServer 5.0 to retrieve specific files stated above.

Question d

The fully reconstructed sound file that is retrieved from */Test2016/sound.mp3* is attached in with the deliverables as well as the fully reconstructed image file that is retrieved from */Test2016/image.jpg*. We managed to get enough tcp packets to create a complete copy both files. To get the image.jpg, we had to do a second ARP poisoning for another 30 minutes. We felt that it wouldn't be quality to just dump all of the bytes gathered from all packets. We used

Wireshark mainly to view the packets and to export the objects if all of the packets had been sent from any given **HTTP GET** request.

Part 2

Question a

The *weak* program takes one command line argument and use that to greet the user; then it prompts one-line input, upon receiving an response after the user has hitten the Enter key, it converts the first 12 letters of the input to upper-case if they are lower-case letters; or it converts them to lower-case if the first 12 letters are upper-case; lastly it echos back the input given by the user after conversion and exits.

Question b

The buffer size used by the *weak* program is 27, assuming the ending new line does not count as a special letter.
This size is found by the *exploit.py* program as follows: at start it will give a one-letter (which is of one-byte length) input to the *weak* program and repeatedly double the length of input until a *SIGSEGV* signal is delivered to the *weak* program; the current length of input is set to be the upper bound of the candidate buffer size, then the *exploit.py* program also sets the half of this upper bound to be the lower bound of the buffer size and starts running a binary search algorithm using the upper and lower bound found; lastly it will report the minimum number of letters (or bytes, in this case *28*) that caused a *SIGSEGV* signal to be delivered, which suggests that the actual buffer size is one less than that size.

Question c

To find the function that print the string *Magic cookie found!*, first find the starting address of the *.rodata* section:

```
readelf -S ./weak
```

from the output it shows the starting address of *.rodata* section for the *weak* program is *0809f260* (in the *Addr* column).

Next, find the offset of the string within *.rodata* section:

```
readelf -p .rodata ./weak | grep 'Magic cookie found!'
```

the number shown in square brackets is the offset of the string; in this case it is *8*.

Now the address of the string literal can be calculated by adding the offset to the starting address of *.rodata* section: 8 + 809f260 = 809f268

Lastly search for the address *809f268* in the disassembly output:

```
objdump -S ./weak|less
```

press / followed by *809f268* and Enter to find the address reference in the pager; the only reference is at *804822d* in the *.text* section: *804822d: 68 68 f2 09 08 push $0x809f268*

The next instruction after this is *call*, the address given to *call* is very likely to belong to *printf* function; but in order to get to this point in the program, we need to find the starting address of this subroutine block: scrolling back in the pager we can see that the most recent address that involves a backup of *ebp* register is: *8048224: 55 push %ebp*, the instructions above this one are *nop* and *ret*, so this address is definitely the starting address of this subroutine (function) block.

To call the address *8048224*, or in **Little Endian** byte series form: *0x24 0x82 0x04 0x08*, we need to overflow the buffer in a way such that the return address of the stack frame that containing the buffer is overwritten by this byte series.

Question d

The source code of the exploit is named **exploit.py** and resides under the *part2* sub-directory.

Question e

As explained in question *b*, the **exploit.py** program finds the minimum number of bytes (*28*) that caused the *weak* program to crash via *SIGSEGV* signal through a binary search algorithm implemented in the *buffer_exploit* function; and after it is successfully found, it appends two addresses immediately after the 28 padding bytes (*0* in this case) that point to the beginning location of the secret printing function (procedure to find it is described in question *c*) and the exit function (procedure to find it is described in question *f*) respectively, in order to *try to* overwrite the back-up return address on the stack; turns out in this case after inputting 28 bytes the following bytes that are written after that will actually change the return address of the input-requesting function based on observing the *core dump* produced by the program through the following steps:

```
cd part2
python3 ./exploit.py -e ./weak -o buffer.bin
```

```
cat buffer.bin | ./weak
gdb ./weak core
```

note the binary content writing process is carried out on line *83* in the *exploit.py* script; after finding that simply by appending the *"secret_byte"* right after 28 bytes are written:

```
padding_byte * buffer_size + secret_byte
```

where the multiplication part stands for repeating the *"padding_byte"* *"buffer_size"* number of times.

The instruction pointer changes correspondingly when observing the address printed by *gdb* based on the *core* file; so at this point the byte sequence for *"secret_byte"* is simply changed to the address (*8048224*) found in question *c* in order to make the program jump to this location upon the return of the input-requesting function.

Question f

Based on the observation made in question *c*, the virtual address for the *printf* function (*call 0x8048f10* instruction after *push $0x809f268*) can actually be located within the binary itself, a situation in which suggests that the binary is statically compiled, and this guess is further verified through the *ldd* command:

```
cd part2
ldd ./weak
```

the output of it shows *"not a dynamic executable"*, hence it is a static executable.

Since this is a *statically compiled* executable, the actual C library wrapper for *system calls* can be located within the virtual address space of the executable as well; after looking at the linux kernel documentation on system calls, it becomes clear that the platform-specific system call table (in this case *i386*) can be viewed in the kernel source directory: *"linux/arch/x86/entry/syscalls/syscall_32.tbl"*

The system calls that are of interest are anything that relates to *exit*; after searching for the keyword *exit*, the only two entries that match the keyword are:

System Call Number	ABI	System Call Name	Entry Point
1	i386	exit	sys_exit
252	i386	exit_group	sys_exit_group

Based on the calling convention described in the *"x86 Assembly: Interfacing*

with Linux" article, to locate the above two system calls in the assembly code of *weak* program, we need to find cases where *int $0x80* kernel trap instruction is issued and either one of the two above system call numbers are placed into the *eax* register.

The hexadicimal representation for the system call number *1* and *252* are *1* and *fc*, respectively; turns out the only case involves the above calling pattern is at virtual address beginning at *8050dfc* and ending with *8050e0f*:

```
mov    0x4(%esp),%ebx
mov    $0xfc,%eax
int    $0x80
mov    $0x1,%eax
int    $0x80
hlt
nop
```

To jump to the address *8050dfc* after the secret-printing function returns, all needs to be done is to continue overwriting past the input buffer such that the return address for the secret-printing function becomes *8050dfc*; to achieve this effect *"exit_byte"* is set to the starting address of this block in **Little Endian** form *0xfc 0x0d 0x05 0x08*, so the final binary content that used for buffer overflow is constructed as

```
padding_byte * buffer_size + secret_byte + exit_byte
```

and the program indeed exits without receiving *SIGSEGV* signal.

Reference

- Linux Kernel Documentation on System Calls
- Linux x86 32-bit System Call Table
- x86 Assembly: Interfacing with Linux
- Wireshark User's Guide
- `man 1 ettercap`
- `man 1 nmap`

Division of Workload

The sliding part of assignment 2 was done entirely by Stephen Arychuk, he also did the *ettercap*-based ARP poisoning and *wireshark*-based traffic analysis for part 1 and wrote answers for question *c* and *d* of step 2.
The *exploit.py* program was written by Jiahui Xie, he also wrote the answers for part 2, step 1 of part 1 and question *a* and *b* of step 2.