UNIVERSITY OF MARYLAND
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENEE 457/CMSC 498E
Computer Systems Security
Instructor: Dana Dachman-Soled

Programming Project 4: Symmetric Key Cryptography and Public Key Infrastructure

Out: 10/23/17 Due: 11/08/17 10:59am

Instructions

1. Strictly adhere to the University of Maryland Code of Academic Integrity.
2. Submit a .zip file containing the following files: `ECB.bmp`, `CBC.bmp`, `msg.txt`, `ECB.txt`, `CBC.txt`, `CFB.txt`, `OFB.txt`, `Makefile`, `findkey.c`, `server.pem`, `ss81.png` and `ss82.png`. Submit a writeup explaining what you did and responding to the questions as a pdf document at Canvas. Include your full name in the writeup. Name the .zip file as x-project4.zip and the writeup as x-project4.pdf, where x is your first and last name. For example, if your name is "Jane Doe," you should be handing in two files named "JaneDoe-project4.zip" and "JaneDoe-project4.pdf."

Adjusted from SEED labs `http://www.cis.syr.edu/~wedu/seed/`

1 Overview

The learning objective of this lab is for students to get familiar with the concepts in symmetric-key encryption and Public-Key Infrastructure (PKI). After finishing the lab, students should be able to gain a first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV), digital signature, public-key certificate, certificate authority. Moreover, students will be able to use tools and write programs to encrypt/decrypt messages.

2 Lab Environment

Installing OpenSSL. In this lab, we will use `openssl` commands and libraries. We have already installed `openssl` binaries in our VM. It should be noted that if you want to use `openssl` libraries in your programs, you need to install several other things for the programming environment, including the header files, libraries, manuals, etc. We have already downloaded the necessary files under the directory `/home/seed/openssl-1.0.1`. To configure and install `openssl` libraries, go to the `openssl-1.0.1` folder and run the following commands.

```
You should read the INSTALL file first:

  % sudo ./config
  % sudo make
```

```
% sudo make test
% sudo make install
```

Installing a hex editor. In this lab, we need to be able to view and modify files of binary format. We have installed in our VM a hex editor called `GHex`. It allows the user to load data from any file, view and edit it in either hex or ascii. Note: many people told us that another hex editor, called `Bless`, is better; this tool may not be installed in the VM version that you are using, but you can install it yourself using the following command:

```
% sudo apt-get install bless
```

3 Lab Tasks

3.1 Task 1: Encryption using different ciphers and modes

In this task, we will play with various encryption algorithms and modes. You can use the following `openssl enc` command to encrypt/decrypt a file. To see the manuals, you can type `man openssl` and `man enc`.

```
% openssl enc ciphertype -e  -in plain.txt -out cipher.bin \
              -K  00112233445566778889aabbccddeeff \
              -iv 0102030405060708
```

Please replace the `ciphertype` with a specific cipher type, such as `-aes-128-cbc`, `-aes-128-cfb`, `-bf-cbc`, etc. In this task, you should try at least 3 different ciphers and three different modes. You can find the meaning of the command-line options and all the supported cipher types by typing "`man enc`". We include some common options for the `openssl enc` command in the following:

```
-in <file>     input file
-out <file>    output file
-e             encrypt
-d             decrypt
-K/-iv         key/iv in hex is the next argument
-[pP]          print the iv/key (then exit if -P)
```

You do not need to hand in anything for this task.

3.2 Task 2: Encryption Mode – ECB vs. CBC

The file `pic_original.bmp` contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the `.bmp` file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate `.bmp` file. We will replace the header of the encrypted picture with that of the original picture. You can use a hex editor tool (e.g. `ghex` or `Bless`) to directly modify binary files.
2. Display the encrypted picture using any picture viewing software. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Please submit the two resulting picture files `ECB.bmp` and `CBC.bmp`. Include the responses to the above questions in your labreport.

3.3 Task 3: Encryption Mode – Corrupted Cipher Text

To understand the properties of various encryption modes, we would like to do the following exercise:

1. Create a text file `msg.txt` that is at least 64 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 30th byte in the encrypted file got corrupted. You can achieve this corruption using a hex editor.
4. Decrypt the corrupted file (encrypted) using the correct key and IV.

Please answer the following questions: (1) How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. (2) Please explain why. (3) What are the implication of these differences?

Please submit files `msg.txt` `ECB.txt`, `CBC.txt`, `CFB.txt`, `OFB.txt` (corresonding to the decryptions). Include the responses to the above questions in your labreport.

3.4 Task 4: Programming using the Crypto Library

So far, we have learned how to use the tools provided by `openssl` to encrypt and decrypt messages. In this task, we will learn how to use `openssl`'s crypto library to encrypt/decrypt messages in programs.

OpenSSL provides an API called EVP, which is a high-level interface to cryptographic functions. Although OpenSSL also has direct interfaces for each individual encryption algorithm, the EVP library provides a common interface for various encryption algorithms. To ask EVP to use a specific algorithm, we simply need to pass our choice to the EVP interface. A sample code is given in `https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption`. Please get yourself familiar with this program, and then do the following exercise.

You are given a plaintext and a ciphertext, and you know that `aes-128-cbc` is used to generate the ciphertext from the plaintext, and you also know that the numbers in the IV are all zeros (not the ASCII character '0'). Another clue that you have learned is that the key used to encrypt this plaintext is an English word shorter than 16 characters; the word that can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. Your goal is to write a program to find out this key. You can download a English word list from the Internet. We have also linked one on the web page of this lab. The plaintext and ciphertext is in the following:

```
Plaintext (total 21 characters): This is a top secret.
Ciphertext (in hex format): 8d20e5056a8d24d0462ce74e4904c1b5
                            13e10d1df4a2ef2ad4540fae1ca0aaf9
```

Note 1: If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. Some editors may add a special character to the end of the file. If that happens, you can use a hex editor tool to remove the special character.

Note 2: In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the `openssl` commands to do this task.

Note 3: To compile your code, you may need to include the header files in `openssl`, and link to `openssl` libraries. To do that, you need to tell your compiler where those files are. In your `Makefile`, you may want to specify the following:

```
INC=/usr/local/ssl/include/
LIB=/usr/local/ssl/lib/

all:
       gcc -I$(INC) -L$(LIB) -o enc findkey.c -lcrypto -ldl
```

Please hand in your `Makefile` and source code file `findkey.c`.

3.5 Task 5: Pseudo Random Number Generation

Generating random numbers is a quite common task in security software. In many cases, encryption keys are not provided by users, but are instead generated inside the software. Their randomness is extremely important; otherwise, attackers can predict the encryption key, and thus defeat the purpose of encryption. Many developers know how to generate random numbers (e.g. for Monte Carlo simulation) from their prior experiences, so they use the similar methods to generate the random numbers for security purpose. Unfortunately, a sequence of random numbers may be good for Monte Carlo simulation, but they may be bad for encryption keys. Developers need to know how to generate secure random numbers, or they will make mistakes. Similar mistakes have been made in some well-known products, including Netscape and Kerberos.

In this task, students will learn a standard way to generate pseudo random numbers that are good for security purposes.

Task 5.A: Measure the Entropy of Kernel

To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable. Software (i.e. in the virtual world) is not good at creating randomness, so most systems resort to the physical world to gain the randomness. `Linux` gains the randomness from the following physical resources:

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

The first two are quite straightforward to understand: the first one uses inter-keypress timing and scancode, and the second one uses mouse movement and interrupt timing . The third one gathers random numbers using the interrupt timing. Of course, not all interrupts are good sources of randomness. For example, the timer interrupt is not a good choice, because it is predictable. However, disk interrupts are a better measure. The last one measures the finishing time of block device requests.

The randomness is measured using *entropy*, which is different from the meaning of entropy in the information theory. Here, it simply means how many bits of random numbers the system currently has. You can find out how much entropy the kernel has at the current moment using the following command.

```
1  % cat /proc/sys/kernel/random/entropy_avail
```

Please move and click your mouses, type somethings, and run the program again. Please describe your observation in your report and include screenshots to support your observations.

Task 5.B: Get Pseudo Random Numbers from `/dev/random`

`Linux` stores the random data collected from the physical resources into a random pool, and then uses two devices to turn the randomness into pseudo random numbers. These two devices have different behaviors. In this subtask, we study the `/dev/random` device.

You can use the following command to get 16 bytes of pseudo random numbers from `/dev/random`. We pipe the data to `hexdump` to print them out.

```
% head -c 16 /dev/random | hexdump
```

Please run the above command several times, and you will find out that at some point, the program will not print out anything, and instead, it will be waiting. Basically, every time a random number is given out by `/dev/random`, the entropy of the randomness pool will be decreased. When the entropy reaches zero, `/dev/random` will block, until it gains enough randomness. Please explain how you can get `/dev/random` to unblock and to print out random data. Please describe your observation in your report and include screenshots to support your observations.

Task 5.C: Get Random Numbers from `/dev/urandom`

`Linux` provides another way to access the random pool via the `/dev/urandom` device, except that this device will not block, even if the entropy of the pool runs low.

You can use the following command to get 1600 bytes of pseudo random numbers from `/dev/urandom`. You should run it several times, and report whether it will block or not.

```
% head -c 1600 /dev/urandom | hexdump
```

Please describe your observation in your report and include screenshots to support your observations.

Both `/dev/random` and `/dev/urandom` use the random data from the pool to generate pseudo random numbers. When the entropy is not sufficient, `/dev/random` will pause, while `/dev/urandom` will keep generating new numbers. Think of the data in the pool as the "seed", and as we know, you can use a seed to generate as many pseudo random numbers as you want. Theoretically speaking, the `/dev/random` device is more secure, but in practice, there is not much difference, because the "seed" is random and non-predictable. `/dev/urandom` does re-seed whenever new random data become available. The fact that `/dev/random` blocks may lead to denial of service attacks.

It is recommended that you use `/dev/urandom` to get random numbers. To do that in your program, you just need to read directly from this file. The following code snippet shows you how.

```
#define LEN 16  // 128 bits

unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
FILE* random = fopen("/dev/urandom", "r");
fread(key, sizeof(unsigned char)*LEN, 1, random);
fclose(random);
```

4 PKI Tasks

4.1 Task 6: Become a Certificate Authority (CA)

A Certificate Authority (CA) is a trusted entity that issues digital certificates. The digital certificate certifies the ownership of a public key by the named subject of the certificate. A number of commercial CAs are treated as root CAs; VeriSign is the largest CA at the time of writing. Users who want to get digital certificates issued by the commercial CAs need to pay those CAs.

In this lab, we need to create digital certificates, but we are not going to pay any commercial CA. We will become a root CA ourselves, and then use this CA to issue certificate for others (e.g. servers). In this task, we will make ourselves a root CA, and generate a certificate for this CA. Unlike other certificates, which are usually signed by another CA, the root CA's certificates are self-signed. Root CA's certificates are usually pre-loaded into most operating systems, web browsers, and other software that rely on PKI. Root CA's certificates are unconditionally trusted.

The Configuration File `openssl.conf`**.** In order to use `OpenSSL` to create certificates, you have to have a configuration file. The configuration file usually has an extension `.cnf`. It is used by three `OpenSSL` commands: `ca`, `req` and `x509`. The manual page of `openssl.conf` can be found using Google search. You can also get a copy of the configuration file from `/usr/lib/ssl/openssl.cnf`. After copying this file into your current directory, you need to create several sub-directories as specified in the configuration file (look at the `[CA_default]` section):

```
dir             = ./demoCA          # Where everything is kept
certs           = $dir/certs        # Where the issued certs are kept
crl_dir         = $dir/crl          # Where the issued crl are kept
new_certs_dir   = $dir/newcerts     # default place for new certs.

database        = $dir/index.txt    # database index file.
serial          = $dir/serial       # The current serial number
```

For the `index.txt` file, simply create an empty file. For the `serial` file, put a single number in string format (e.g. 1000) in the file. Once you have set up the configuration file `openssl.cnf`, you can create and issue certificates.

Certificate Authority (CA). As we described before, we need to generate a self-signed certificate for our CA. This means that this CA is totally trusted, and its certificate will serve as the root certificate. You can run the following command to generate the self-signed certificate for the CA:

```
$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
```

You will be prompted for information and a password. Do not lose this password, because you will have to type the passphrase each time you want to use this CA to sign certificates for others. You will also be asked to fill in some information, such as the Country Name, Common Name, etc. The output of the command are stored in two files: `ca.key` and `ca.crt`. The file `ca.key` contains the CA's private key, while `ca.crt` contains the public-key certificate.

4.2 Task 7: Create a Certificate for `PKILabServer.com`

Now, we become a root CA, we are ready to sign digital certificates for our customers. Our first customer is a company called `PKILabServer.com`. For this company to get a digital certificate from a CA, it needs to go through three steps.

Step 1: Generate public/private key pair. The company needs to first create its own public/private key pair. We can run the following command to generate an RSA key pair (both private and public keys). You will also be required to provide a password to protect the keys. The keys will be stored in the file `server.key`:

```
$ openssl genrsa -des3 -out server.key 1024
```

Step 2: Generate a Certificate Signing Request (CSR). Once the company has the key file, it should generates a Certificate Signing Request (CSR). The CSR will be sent to the CA, who will generate a certificate for the key (usually after ensuring that identity information in the CSR matches with the server's true identity). Please use `PKILabServer.com` as the common name of the certificate request.

```
$ openssl req -new -key server.key -out server.csr -config openssl.cnf
```

Step 3: Generating Certificates. The CSR file needs to have the CA's signature to form a certificate. In the real world, the CSR files are usually sent to a trusted CA for their signature. In this lab, we will use our own trusted CA to generate certificates:

```
$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key \
             -config openssl.cnf
```

If `OpenSSL` refuses to generate certificates, it is very likely that the names in your requests do not match with those of CA. The matching rules are specified in the configuration file (look at the `[policy_match]` section). You can change the names of your requests to comply with the policy, or you can change the policy. The configuration file also includes another policy (called `policy_anything`), which is less restrictive. You can choose that policy by changing the following line:

```
"policy = policy_match"  change to "policy = policy_anything".
```

4.3 Task 8: Use PKI for Web Sites

In this lab, we will explore how public-key certificates are used by web sites to secure web browsing. First, we need to get our domain name. Let us use `PKILabServer.com` as our domain name. To get our computers recognize this domain name, let us add the following entry to `/etc/hosts`; this entry basically maps the domain name `PKILabServer.com` to our localhost (i.e., 127.0.0.1):

```
127.0.0.1  PKILabServer.com
```

Next, let us launch a simple web server with the certificate generated in the previous task. `OpenSSL` allows us to start a simple web server using the `s_server` command:

```
# Combine the secret key and certificate into one file
% cp server.key server.pem
% cat server.crt >> server.pem

# Launch the web server using server.pem
% openssl s_server -cert server.pem -www
```

Submit `server.pem`. By default, the server will listen on port `4433`. You can alter that using the `-accept` option. Now, you can access the server using the following URL: `https://PKILabServer.com:4433/`. Most likely, you will get an error message from the browser. In Firefox, you will see a message like the following: *"pkilabserver.com:4433 uses an invalid security certificate. The certificate is not trusted because the issuer certificate is unknown"*. Submit a screenshot with Firefox showing this message in `ss81.png`.

Had this certificate been assigned by VeriSign, we will not have such an error message, because VeriSign's certificate is very likely preloaded into Firefox's certificate repository already. Unfortunately, the certificate of `PKILabServer.com` is signed by our own CA (i.e., using `ca.crt`), and this CA is not recognized by Firefox. There are two ways to get Firefox to accept our CA's self-signed certificate.

- We can request Mozilla to include our CA's certificate in its Firefox software, so everybody using Firefox can recognize our CA. This is how the real CAs, such as VeriSign, get their certificates into

Firefox. Unfortunately, our own CA does not have a large enough market for Mozilla to include our certificate, so we will not pursue this direction.

- **Load `ca.crt` into Firefox:** We can manually add our CA's certificate to the Firefox browser by clicking the following menu sequence:

  ```
  Edit -> Preference -> Advanced -> View Certificates.
  ```

  You will see a list of certificates that are already accepted by Firefox. From here, we can "import" our own certifcate. Please import `ca.crt`, and select the following option: "Trust this CA to identify web sites". You will see that our CA's certificate is now in Firefox's list of the accepted certificates.

Now, point the browser to `https://PKILabServer.com:4433`. Submit a screenshot with Firefox showing this message in `ss82.png`. Please describe and explain your observations. Please also do the following tasks:

1. Modify a single byte of `server.pem` in the certificate section (between `---BEGIN CERTIFICATE---` and `---END CERTIFICATE---`), and restart the server, and reload the URL. What do you observe? Make sure you restore the original `server.pem` afterward.
2. Since `PKILabServer.com` points to the localhost, if we use `https://localhost:4433` instead, we will be connecting to the same web server. Please do so, describe and explain your observations.

Submission instructions: Include your responses to the above questions in the labreprot. As mentioned earlier, you also need to submit `server.pem`, `ss81.png` and `ss82.png`.