Symmetric-key, Public-key Crypto and PKI

Due April 14 2017, 11:59PM

1 Overview

The overall learning objective of this lab is to get familiar with the concepts of symmetric-key encryption, hashing, and Public-Key Infrastructure (PKI). You will be getting first-hand experience with using the OpenSSL commandline tool and library to encrypt/decrypt messages under different encryption modes, digital signature, public-key certificate, certificate authority and to construct message digests with hash functions.

The tasks are set up in such a way that you not only gain experience with the *use* of these various mechanisms, but also with their *properties*. You are encouraged to think about what these tools are doing, and why you get the results you get. Some tasks ask you to write up these observations.

You will be using the same VM that you used for the previous projects. Make this a habit: standards, interfaces, requirements, and recommended uses change over time.

2 Symmetric key encryption and decryption

2.1 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, and to see the various cipher modes that OpenSSL supports, you can type `man openssl` and `man enc`.

```
% openssl enc ciphertype -e  -in plain.txt -out cipher.bin \
              -K  00112233445566778899aabbccddeeff \
              -iv 0102030405060708
```

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. 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)
```

However, you will very rarely be typing in the key and initialization vector yourself: that is very prone to error, and is kind of a waste of time. Instead, we can use OpenSSL itself to help us generate random symmetric keys. Really, all we want from a symmetric key is that it be the right size and that it be random, so we generate them with OpenSSL's `rand` command:

```
% openssl rand -base64 16 > symm_key
```

This will generate a 16 byte (128 bit) random value in base 64 encoding. We can use this to encrypt as follows:

```
% openssl enc ciphertype -e -in plain.txt -out cipher.bin \
   -pass file:symm_key -salt
```

Note that the key we generated, `symm_key`, is being used instead of specifying the key by hand. Strictly speaking, we will not use this as our key, but rather as a sort of "password" that will be used to help generate the key. To see this, you can attach the `-P` option to view the key that is actually being used. In any event, decryption also uses the same `-pass file:symm_key` argument (it does not need the `-salt` argument — technically, neither does encryption, as it is the default, but this is just to reinforce that you should never use no salt).

Submission: Please encrypt a file containing your DID by using the `openssl` command and a randomly generated key file as shown in the example commands. Name the key file `task1.key`. Use AES-256 in CBC mode for the encryption. Submit `task1.bin` and `task1.key`.

2.2 Task 2: ECB vs. CBC

The file `terps.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 by using a pass file as described in Section 2.1, 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 a `.bmp` file, the first set bytes contain header information about the picture. We have to set the header correctly so that the encrypted file can be treated as a legitimate `.bmp` file in order for us to view it. We have provided a script (`make_bmp.py`) that you can use to make your encrypted binary a viewable bmp image. You can use the command:

   ```
   % python make_bmp.py terps.bmp your_encrypted_file.bin \
           -o terps-enc-{cbc,ecb}.bmp
   ```

   The python script has a dependency `argparse` which can be downloaded at https://pypi.python.org/pypi/argparse

2. Display the encrypted picture using any picture viewing software.

Submission: Please submit the encrypted versions of the image `terps-enc-cbc.bmp` (for CBC) and `terps-enc-ecb.bmp` (for ECB). Along with the images please include `task2.txt` containing:

1. A description of what the encrypted images look like.
2. Why is there a difference between the two?
3. What if any useful data can you learn by just looking at either picture?

2.3 Task 3: Encryption using the Data Encryption Standard (DES)

In this task you will compute the output of the **first** round of DES with the input and key being **your** Directory ID. If your DID is less than 8 characters long, pad it with ASCII NULLs or if it is more than 8 characters, truncate it to 8. Please show all work.

Documentation for DES can be found here: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
Another resource is available here: http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm

Example input and key: `ngrodrig`

```
ngrodrig: 01101110 01100111 01110010 01101111 01100100 01110010 01101001 01100111
```

Show all steps in the encryption round. You should show these steps in particular:

- Initial permutation
- Key generation:
  - C_0 and D_0
  - Left shift operations on C_0 and D_0
  - Key K_1
- L_0 and R_0
- Mangler function:
  - $K_1 \oplus R_{expanded}$
  - S-Box output
  - Permutation after S-Box (call it P_S)
- $L_0 \oplus P_S$
- Final 64 bit output (L_1R_1)

Note: DES is insecure due to its short key length. This exercise is only meant to introduce you to the working of the algorithm. For most practical purposes, AES with either 128-bit or 256-bit keys is the preferred cipher.

Submission: Please submit a file called `task3.[pdf/txt]`. You may show your work on a piece of paper and scan it for submission (`task3.pdf`). If you prefer to type your answer, please submit `task3.txt` or `task3.pdf` (if you typeset it).

3 Task 4: Hashing

In this task, we give you a list of hashed passwords encoded in base64 encoding. Your task is to crack as many passwords as you can. There are 100 salted and peppered passwords hashed using the SHA3-256 hash algorithm. The salt and pepper used are `CMSC414` and `Spring17` respectively. You will use password dictionaries that either you create or obtain online to crack these passwords.

Hint: You may have to use multiple password dictionaries to crack all passwords.

Example password: `abcd`

- Adding the salt at the front and pepper at the end gives us `CMSC414abcdSpring17`.
- Hashing this value using SHA3-256 and representing the digest in hex gives us

  ```
  2f032af8cbbfba507e47d99a53771c6d0bb8942ada1258661451e734ab5dba80
  ```

- Encoding the hash digest (not the hex representation) using `base64` encoding gives us

  ```
  LwMq+Mu/ulB+R9maU3ccbQu4lCraElhmFFHnNKtduoA=
  ```

The file `hashes.txt` contains 100 hashed passwords. Write a program in either Java, Python or C to crack the passwords. The input to your program must be a password dictionary, and output should be `cracked.txt` with the hashes replaced by the actual corresponding passwords. If you are not able to crack a specific password, leave the hash for that password in the output file as is. To get full credit for this task, you only need to crack 90 out of the 100 passwords.

Extra credit (+5 points): Crack all 100 passwords.

Your program (name it `Task4.[java/py/c]`) will be run as

```
% ./Task4 <pass_dictionary.txt>
```

Submission:

- Your program: `Task4.[java/py/c]`
- The output of running your program (i.e. the file with the cracked passwords): `cracked.txt`.
- You are not required to submit a password dictionary. We will check your code against our password dictionary for correctness.

4 PKI Tasks

4.1 Task 5: 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 6: 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 7: 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 `ss71.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 `ss72.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: Submit your responses to the above questions in `task7.txt`. As mentioned earlier, you also need to submit `server.pem`, `ss71.png` and `ss72.png`.

5 Task 8: Bitcoin

Watch Khan Academy's lectures on Bitcoin here:
`https://www.khanacademy.org/economics-finance-domain/core-finance/money-and-banking#bitcoin`.

For an optional additional resource you may use this ebook:
`https://umaryland.on.worldcat.org/search?databaseList=&queryString=understanding+bitcoin&`

umdlib=

Answer the following questions

1. What is Bitcoin?
2. Explain how Bitcoin uses symmetric encryption, hashing and asymmetric encryption.

Extra credit (+5 points): Explain an attack against Bitcoin.

Submission instructions: Submit a text file called `task8.txt` with your answers.

6 Summary of Submitted Files

Make a gzipped tarball named `<firstname>.<lastname>.tgz` that includes the following files:

- `task1a.bin`
- `task1.key`
- `task1b.bin`
- `task2.txt`
- `terps-enc-ecb.bmp`
- `terps-enc-cbc.bmp`
- `task3.[pdf/txt]`
- `Task4.[java/py/c]`
- `cracked.txt`
- `server.pem`
- `ss71.png`
- `ss72.png`
- `task7.txt`
- `task8.txt`

So Jo Smith would do:

```
tar cvfz jo.smith.tgz task1b.bin task2.txt ...  task7.txt
```

and submit `jo.smith.tgz` via the submit server.

Note: Only the latest submission counts.

7 Points for individual tasks

Problem	Points
1	10
2	10
3	25
4	25
5,6,7	25
8	15
Total	110