Final Project – Public-Key Ciphers and PKI

This document has been slightly modified by Mirela Damian, Villanova University. The original document can be found at `http://www.cis.syr.edu/~wedu/seed/`.

1 Overview

The learning objective of this project is for students to get familiar with the concepts in the Public-Key encryption and Public-Key Infrastructure (PKI). After finishing the project, students should be able to gain a first-hand experience on public-key encryption, digital signature, public-key certificate, certificate authority, authentication based on PKI. Moreover, students will be able to use tools and write programs to create secure channels using PKI.

2 Project Environment

Cryptography library `OpenSSL`. In this project, we will use `openssl` commands and libraries. Make sure you have `openssl` installed in your VM. It should be noted that if you want to use `openssl` libraries in your programs, you also need to install `libssl-dev`, the development version of ssl, using the following command:

```
sudo apt-get install libssl-dev
```

3 Project Tasks

Switch to supervisor mode to establish the credentials needed to complete the project tasks listed below:

```
sudo su
```

3.1 Task 1: 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 project, 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. Openssl uses the `RANDFILE` environment variable in its config file to specify the location of a random seed, which will later be used in the key generation. You can find your own personal openssl seed in the file `.rnd` in your home directory. Copy it in your current directory with the command

```
cp ~/.rnd .
```

and then change the `RANDFILE` line in `openssl.cnf` to

```
RANDFILE = ./.rnd
```

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.

3.2 Task 2: Create a Certificate for `PKIServer.com`

Now, we become a root CA, we are ready to sign digital certificates for our customers. Our first customer is a company called `PKIServer.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 `PKIServer.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 project, 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".
```

3.3 Task 3: Use PKI for Web Sites

In this project, 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 `PKIServer.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 `PKIServer.com` to our localhost (i.e., 127.0.0.1):

```
127.0.0.1  PKIServer.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
```

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://PKIServer.com:4433/`. Most likely, you will get an error message from the browser. In Firefox, you will see a message like the following: *"pkiserver.com:4433 uses an invalid security certificate. The certificate is not trusted because the issuer certificate is unknown".*

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 `PKIServer.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://PKIServer.com:4433`. Please describe and explain your observations. Also complete the following tasks:

1. Modify a single byte of `server.pem`, and restart the server, and reload the URL. What do you observe? Make sure you restore the original `server.pem` afterward. Note: the server may not be able to restart if certain places of `server.pem` is corrupted; in that case, choose another place to modify.

2. Since `PKIServer.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.

3.4 Task 4: Using PKI to establish secure TCP connections with `PKIServer.com`

In this task, we will implement a TCP client and TCP server, which are connected via a secure TCP connection. The traffic between the client and the server is encrypted using a session key that is known only to the client and the server. Moreover, the client needs to authenticate the server using public-key certificates[1].

We will use `OpenSSL` functions directly to make an SSL connection between the client and the server, in which case, the verification of certificates will be automatically carried out by the SSL functions. There are many online tutorials on these SSL functions, so we will not give another one here. The following are a few tutorials that are useful for this project. These tutorials are also linked in the web page of this project:

- OpenSSL examples: `http://www.rtfm.com/openssl-examples/`

- `http://www.ibm.com/developerworks/linux/library/l-openssl.html`

We provide two example programs, `cli.c` and `serv.c`, in a file `demo_openssl_api.zip`, to help you to understand how to use `OpenSSL` API to build secure TCP connections. The file can be downloaded from the project's web page. The programs demonstrate how to make SSL connections, how to get peer's certificate, how to verify certificates, how to get information out of certificates, etc. To make the program work, you have to unzip it first and run the `make` command. We have included the password for the server and client in the README file. Please complete the following activities:

1. Add comments to `cli.c` and `serv.c`, explaining in detail what each of the TLS/SSL function call accomplishes. Look up the behavior of these functions online. The dynamics of the TLS/SSL client/server application should be clear from your comments.

[1] In practice, the server also needs to authenticate the client. However, for the sake of simplicity, we do not implement the client authentication in this task.

2. Add code to the client program to ensure that the client is talking to the intended server (we use `PKIServer.com` as the intended server), not a spoofed one; namely, the client needs to authenticate the server. This server authentication should be done using the server's certificate: the client should check that the common name in the server's certificate matches the name of the intended server.

3.5 Task 5: Performance Comparison: RSA versus AES

In this task, we will study the performance of public-key algorithms. Please prepare a file (`message.txt`) that contains a 16-byte message. Also generate an 1024-bit RSA public/private key pair. Then, do the following:

1. Encrypt `message.txt` using the public key; save the the output in `message_enc.txt`.
2. Decrypt `message_enc.txt` using the private key.
3. Encrypt `message.txt` using a 128-bit AES key.
4. Compare the time spent on each of the above operations (you may use the `time` Unix command), and describe your observations. If an operation is too fast, you may want to repeat it many times, and then take an average.

After you finish the above exercise, you can now use `OpenSSL`'s `speed` command to do such a benchmarking. Please describe whether your observations are similar to those from the outputs of the `speed` command. The following command shows examples of using `speed` to benchmark `rsa` and `aes`:

```
% openssl speed rsa
% openssl speed aes
```

3.6 Task 6: Create Digital Signature

In this task, we will use `OpenSSL` to generate digital signatures. Please prepare a file (`example.txt`) of any size. Please also prepare an RSA public/private key pair. Do the following:

1. Sign the SHA256 hash of `example.txt`; save the output in `example.sha256`.
2. Verify the digital signature in `example.sha256`.
3. Slightly modify `example.txt`, and verify the digital signature again.

Please describe how you did the above operations (e.g., what commands do you use, etc.). Explain your observations. Please also explain why digital signatures are useful.

4 Submission

You need to submit a detailed project report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this project. Please add your (well-commented) code from Task 4 to the project report as well.