Public-Key Infrastructure (PKI) Lab

1 Overview

Public key cryptography is the foundation of today's secure communication, but it is subject to man-in-the-middle attacks when one side of communication sends its public key to the other side. The fundamental problem is that there is no easy way to verify the ownership of a public key, i.e., given a public key and its claimed owner information, how do we ensure that the public key is indeed owned by the claimed owner? The Public Key Infrastructure (PKI) is a practical solution to this problem.

The learning objective of this lab is for students to gain the first-hand experience on PKI. SEED labs have a series of labs focusing on the public-key cryptography, and this one focuses on PKI. By doing the tasks in this lab, students should be able to gain a better understanding of how PKI works, how PKI is used to protect the Web, and how Man-in-the-middle attacks can be defeated by PKI. Moreover, students will be able to understand the root of the trust in the public-key infrastructure, and what problems will arise if the root trust is broken. This lab covers the following topics:

- Public-key encryption, Public-Key Infrastructure (PKI)
- Certificate Authority (CA), X.509 certificate, and root CA
- Apache, HTTP, and HTTPS
- Man-in-the-middle attacks

Readings. Detailed coverage of PKI can be found in the following:

- Chapter 23 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

Related labs. A topic related to this lab is the Transport Layer Security (TLS), which is based on PKI. We have a separate lab for TLS. In addition, we have a lab called *RSA Public-Key Encryption and Signature Lab*, which focuses on the algorithm part of the public-key cryptography.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Lab Environment

In this lab, we will generate public-key certificates, and then use them to secure web servers. The certificate generation tasks will be conducted on the VM, but we will use a container to host the web server.

Container Setup and Commands. Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

DNS setup. In this document, we use `www.bank32.com` as an example to show how to set up an HTTPS web server with this name. Students need to use a different name for their lab. Unless the name is specified by the instructors, students should include their last name and the current year in the server name. For example, if `John Smith` does this lab in 2020, the server name should be `www.smith2020.com`. You do not need to own this domain; you just need to map this name to the container's IP address by adding the following entries to `/etc/hosts` (the first entry is required, otherwise, the example in this lab description will not work):

```
10.9.0.80   www.bank32.com
10.9.0.80   www.smith2020.com
```

3 Lab Tasks

3.1 Task 1: Becoming 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 from online resources. By default, `OpenSSL` use the configuration file from `/usr/lib/ssl/openssl.cnf`. Since we need to make changes to this file, we will copy it into our current directory, and instruct `OpenSSL` to use this copy instead.

The `[CA_default]` section of the configuration file shows the default setting that we need to prepare. We need to create several sub-directories. Please uncomment the `unique_subject` line to allow creation of certifications with the same subject, because it is very likely that we will do that in the lab.

Listing 1: Default CA setting

```
[ CA_default ]
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
database        = $dir/index.txt    # database index file.
#unique_subject = no                # Set to 'no' to allow creation of
                                    # several certs with same subject.
new_certs_dir   = $dir/newcerts     # default place for new certs.
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 -x509 -newkey rsa:4096 -sha256 -days 3650 \
            -keyout ca.key -out ca.crt
```

You will be prompted for 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 the subject 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.

You can also specify the subject information and password in the command line, so you will not be prompted for any additional information. In the following command, we use `-subj` to set the subject information and we use `-passout pass:dees` to set the password to `dees`.

```
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 \
            -keyout ca.key -out ca.crt  \
            -subj "/CN=www.modelCA.com/O=Model CA LTD./C=US" \
            -passout pass:dees
```

We can use the following commands to look at the decoded content of the X509 certificate and the RSA key (`-text` means decoding the content into plain text; `-noout` means not printing out the encoded version):

```
openssl x509 -in ca.crt -text -noout
openssl rsa  -in ca.key -text -noout
```

Please run the above commands. From the output, please identify the followings:

- What part of the certificate indicates this is a CA's certificate?
- What part of the certificate indicates this is a self-signed certificate?
- In the RSA algorithm, we have a public exponent e, a private exponent d, a modulus n, and two secret numbers p and q, such that $n = pq$. Please identify the values for these elements in your certificate and key files.

3.2 Task 2: Generating a Certificate Request for Your Web Server

A company called `bank32.com` (replace this with the name of your own web server) wants to get a public-key certificate from our CA. First it needs to generate a Certificate Signing Request (CSR), which basically includes the company's public key and identity information. The CSR will be sent to the CA, who will verify the identity information in the request, and then generate a certificate.

The command to generate a CSR is quite similar to the one we used in creating the self-signed certificate for the CA. The only difference is the `-x509` option. Without it, the command generates a request; with it, the command generates a self-signed certificate. The following command generate a CSR for `www.bank32.com` (you should use your own server name):

```
openssl req -newkey rsa:2048 -sha256  \
            -keyout server.key   -out server.csr  \
            -subj "/CN=www.bank32.com/O=Bank32 Inc./C=US" \
            -passout pass:dees
```

The command will generate a pair of public/private key, and then create a certificate signing request from the public key. We can use the following command to look at the decoded content of the CSR and private key files:

```
openssl req -in server.csr -text -noout
openssl rsa -in server.key -text -noout
```

Adding Alternative names. Many websites have different URLs. For example, `www.example.com`, `example.com`, `example.net`, and `example.org` are all pointing to the same web server. Due to the

hostname matching policy enforced by browsers, the common name in a certificate must match with the server's hostname, or browsers will refuse to communicate with the server.

To allow a certificate to have multiple names, the X.509 specification defines extensions to be attached to a certificate. This extension is called Subject Alternative Name (SAN). Using the SAN extension, it's possible to specify several hostnames in the `subjectAltName` field of a certificate.

To generate a certificate signing request with such a field, we can put all the necessary information in a configuration file or at the command line. We will use the command-line approach in this task (the configuration file approach is used in another SEED lab, the TLS lab). We can add the following option to the "`openssl req`" command. It should be noted that the `subjectAltName` extension field must also include the one from the common name field; otherwise, the common name will not be accepted as a valid name.

```
-addext "subjectAltName = DNS:www.bank32.com,  \
                          DNS:www.bank32A.com, \
                          DNS:www.bank32B.com"
```

Please add two alternative names to your certificate signing request. They will be needed in the tasks later.

3.3 Task 3: Generating a Certificate for your server

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. The following command turns the certificate signing request (`server.csr`) into an X509 certificate (`server.crt`), using the CA's `ca.crt` and `ca.key`:

```
openssl ca -config myCA_openssl.cnf -policy policy_anything \
           -md sha256 -days 3650 \
           -in server.csr -out server.crt -batch \
           -cert ca.crt -keyfile ca.key
```

In the above command, `myCA_openssl.cnf` is the configuration file we copied from `/usr/lib/ssl/openssl.cnf` (we also made changes to this file in Task 1). We use the `policy_anything` policy defined in the configuration file. This is not the default policy; the default policy has more restriction, requiring some of the subject information in the request to match those in the CA's certificate. The policy used in the command, as indicated by its name, does not enforce any matching rule.

Copy the extension field. For security reasons, the default setting in `openssl.cnf` does not allow the "`openssl ca`" command to copy the extension field from the request to the final certificate. To enable that, we can go to our copy of the configuration file, uncomment the following line:

```
# Extension copying option: use with caution.
copy_extensions = copy
```

After signing the certificate, please use the following command to print out the decoded content of the certificate, and check whether the alternative names are included.

```
openssl x509 -in server.crt -text -noout
```

3.4 Task 4: Deploying Certificate in an Apache-Based HTTPS Website

In this task, we will see how public-key certificates are used by websites to secure web browsing. We will set up an HTTPS website based Apache. The Apache server, which is already installed in our container, supports the HTTPS protocol. To create an HTTPS website, we just need to configure the Apache server, so it knows where to get the private key and certificates. Inside our container, we have already set up an HTTPS site for `bank32.com`. Students can follow this example to set up their own HTTPS site.

An Apache server can simultaneously host multiple websites. It needs to know the directory where a website's files are stored. This is done via its `VirtualHost` file, located in the `/etc/apache2/sites-available` directory. In our container, we have a file called `bank32_apache_ssl.conf`, which contains the following entry:

```
<VirtualHost *:443>
    DocumentRoot /var/www/bank32
    ServerName www.bank32.com
    ServerAlias www.bank32A.com
    ServerAlias www.bank32B.com
    DirectoryIndex index.html
    SSLEngine On
    SSLCertificateFile    /certs/bank32.crt    ①
    SSLCertificateKeyFile /certs/bank32.key    ②
</VirtualHost>
```

The above example sets up the HTTPS site `https://www.bank32.com` (port `443` is the default HTTPS port). The `ServerName` entry specifies the name of the website, while the `DocumentRoot` entry specifies where the files for the website are stored. Using the `ServerAlias` entries, we allow the website to have different names. You should also provide two alias entries.

We also need to tell Apache where the server certificate (Line ①) and private key (Line ②) are stored. In the `Dockerfile`, we have already included the commands to copy the certificate and key to the `/certs` folder of the container.

In order to make the website work, we need to enable Apache's `ssl` module and then enable this site. They can be done using the following commands, which are already executed when the container is built.

```
# a2enmod ssl                 // Enable the SSL module
# a2ensite bank32_apache_ssl  // Enable the sites described in this file
```

Starting the Apache server. The Apache server is not automatically started in the container, because of the need to type the password to unlock the private key. Let's go to the container and run the following command to start the server (we also list some related commands):

```
// Start the server
# service apache2 start

// Stop the server
# service apache2 stop

// Restart a server
# service apache2 restart
```

When Apache starts, it needs to load the private key for each HTTPS site. Our private key is encrypted, so Apache will ask us to type the password for decryption. Inside the container, the password used for

`bank32` is `dees`. Once everything is set up properly, we can browse the web site, and all the traffic between the browser and the server will be encrypted.

Please use the above example as a guidance to set up an HTTPS server for your website. Please describe the steps that you have taken, the contents that you add to Apache's configuration file, and the screenshots of the final outcome showing that you can successfully browse the HTTPS site.

Shared folder between the VM and container. In this task, we need to copy files from the VM to the container. To avoid repeatedly recreating containers, we have created a shared folder between the VM and container. When you use the Compose file inside the `Labsetup` folder to create containers, the `volumes` sub-folder will be mounted to the container. Anything you put inside this folder will be accessible from inside of the running container.

Browsing the website. Now, point the browser to your web server (note: you should put `https` at the beginning of your URL, instead of using `http`). Please describe and explain your observations. Most likely, you will not be able to succeed, this is because ... (the reasons are omitted here; students should provide the explanation in their lab reports). Please fix the problem and demonstrate that you can successfully visit the HTTPS website.

In the following, we provide instructions on how to load a certificate into Firefox. We intentionally do not explain why and what certificate should be loaded; students need to figure that out. To manually add a certificate to the Firefox browser, type the following URL in the address bar, and click the `View Certificates` button on the page (scroll to the bottom).

```
about:preferences#privacy
```

In the `Authorities` tab, you will see a list of certificates that are already accepted by Firefox. From here, we can import our own certificates. After choosing the certificate file, please select the following option: "Trust this CA to identify web sites". You will see that our certificate is now in Firefox's list of accepted certificates.

3.5 Task 5: Launching a Man-In-The-Middle Attack

In this task, we will show how PKI can defeat Man-In-The-Middle (MITM) attacks. Figure 1 depicts how MITM attacks work. Assume Alice wants to visit `example.com` via the HTTPS protocol. She needs to get the public key from the `example.com` server; Alice will generate a secret, and encrypt the secret using the server's public key, and send it to the server. If an attacker can intercept the communication between Alice and the server, the attacker can replace the server's public key with its own public key. Therefore, Alice's secret is actually encrypted with the attacker's public key, so the attacker will be able to read the secret. The attacker can forward the secret to the server using the server's public key. The secret is used to encrypt the communication between Alice and server, so the attacker can decrypt the encrypted communication.

The goal of this task is to help students understand how PKI can defeat such MITM attacks. In the task, we will emulate an MITM attack, and see how exactly PKI can defeat it. We will select a target website first. In this document, we use `www.example.com` as the target website, but in the task, to make it more meaningful, students should pick a popular website, such as a banking site and social network site.

Step 1: Setting up the malicious website. In Task 4, we have already set up an HTTPS website. We will use the same Apache server to impersonate `www.example.com` (or the site chosen by students). To achieve that, we will follow the instruction in Task 4 to add a `VirtualHost` entry to Apache's SSL configuration file: the `ServerName` should be `www.example.com`, but the rest of the configuration can

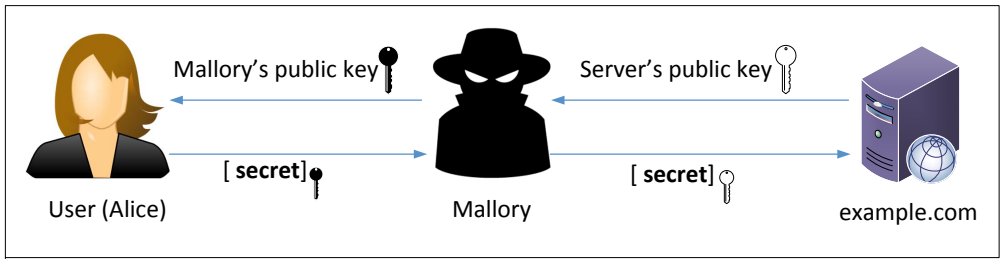

Figure 1: A Man-In-The-Middle (MITM) attack

be the same as that used in Task 4. Obviously, in the real world, you won't be able to get a valid certificate for `www.example.com`, so we will use the same certificate that we used for our own server.

Our goal is the following: when a user tries to visit `www.example.com`, we are going to get the user to land in our server, which hosts a fake website for `www.example.com`. If this were a social network website, The fake site can display a login page similar to the one in the target website. If users cannot tell the difference, they may type their account credentials in the fake webpage, essentially disclosing the credentials to the attacker.

Step 2: Becoming the man in the middle There are several ways to get the user's HTTPS request to land in our web server. One way is to attack the routing, so the user's HTTPS request is routed to our web server. Another way is to attack DNS, so when the victim's machine tries to find out the IP address of the target web server, it gets the IP address of our web server. In this task, we simulate the attack-DNS approach. Instead of launching an actual DNS cache poisoning attack, we simply modify the victim's machine's `/etc/hosts` file to emulate the result of a DNS cache positing attack by mapping the hostname `www.example.com` to our malicious web server.

```
10.9.0.80   www.example.com
```

Step 3: Browse the target website. With everything set up, now visit the target real website, and see what your browser would say. Please explain what you have observed.

3.6 Task 6: Launching a Man-In-The-Middle Attack with a Compromised CA

In this task, we assume that the root CA created in Task 1 is compromised by an attacker, and its private key is stolen. Therefore, the attacker can generate any arbitrary certificate using this CA's private key. In this task, we will see the consequence of such a compromise.

Please design an experiment to show that the attacker can successfully launch MITM attacks on any HTTPS website. You can use the same setting created in Task 5, but this time, you need to demonstrate that the MITM attack is successful, i.e., the browser will not raise any suspicion when the victim tries to visit a website but land in the MITM attacker's fake website.

4 Submission

You need to submit a detailed lab report, with screenshots, 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.

Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.