Public-Key Infrastructure (PKI)

OBJECTIVE

This lab aims to provide students with practical experience in Public Key Infrastructure (PKI).

By the end of the lab, students will have a comprehensive understanding of ***PKI's functionality*** and its role in ***securing web communications***. Additionally, students will explore the concept of trust within PKI, particularly the significance of the root Certificate Authority (CA) and the implications of compromised root trust.

1 LAB ENVIRONMENT

This lab is designed to provide students with hands-on experience in setting up and managing a ***secure web server*** using Transport Layer Security (TLS). The lab will cover various aspects of Public Key Infrastructure (PKI), including DNS setup, becoming a Certificate Authority (CA), generating certificate requests, issuing certificates, deploying certificates in an Apache based HTTPS website.

We will configure a fictional domain name (e.g., www.icthub.edu) and map it to the container's IP address in the Docker Host VM's "hosts file". Later we will create a root CA, generate a self-signed certificate, and understand the role of a CA in PKI. we will also configure OpenSSL to manage certificates.

2 LAB TASKS

2.1 DNS SETUP

In this task, we will utilize a fictional domain name in the format www.icthub.edu to demonstrate the setup of an HTTPS web server. Subsequently, add the complete domain name to the Docker Host VM's hosts file to ensure accessibility. This is achieved by mapping the domain name to the container's IP address through the following entries in /etc/hosts on the Docker Host VM.

```
10.9.0.90 www.icthub.edu
```

2.2 BECOMING A CERTIFICATE AUTHORITY (CA)

A ***Certificate Authority (CA)*** is a trusted entity responsible for issuing digital certificates, which verify the ownership of a public key by the named subject of the certificate. Several commercial CAs are recognized as root CAs, with VeriSign being the largest at the time of writing. Users seeking digital certificates from commercial CAs must pay for their services.

In this task, we will create digital certificates without relying on commercial CAs. Instead, we will establish ourselves as a root CA and use this CA to issue certificates for others (e.g., servers). In this task, we will become a root CA and generate a certificate for it. Unlike other certificates, which are typically signed by another CA, root CA certificates are self-signed. These certificates are usually pre-loaded into most operating systems, web browsers, and other software that depend on PKI, and are unconditionally trusted.

The Configuration File “openssl.conf”

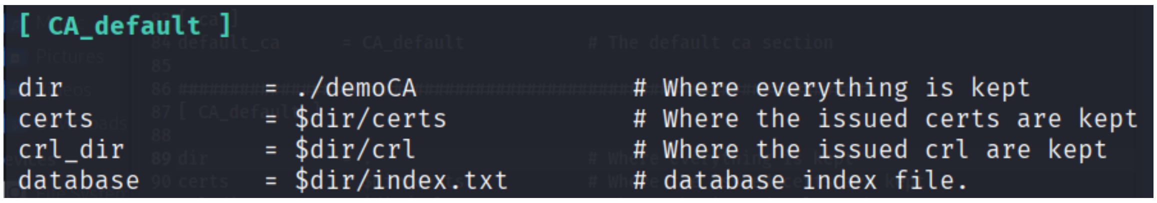
To use OpenSSL for creating certificates, a configuration file is required. This file typically has a ".cnf" extension and is utilized by three OpenSSL commands: **ca**, **req**, and **x509**. By default, OpenSSL uses the configuration file located at "/usr/lib/ssl/openssl.cnf".

Since modifications to this file are necessary, we will copy it to our current directory and instruct OpenSSL to use this copy instead.

Within your home directory, create the following folder structure to store the CA data. Additionally, create a working copy of "/usr/lib/ssl/openssl.cnf" and change the directory to the CA folder.

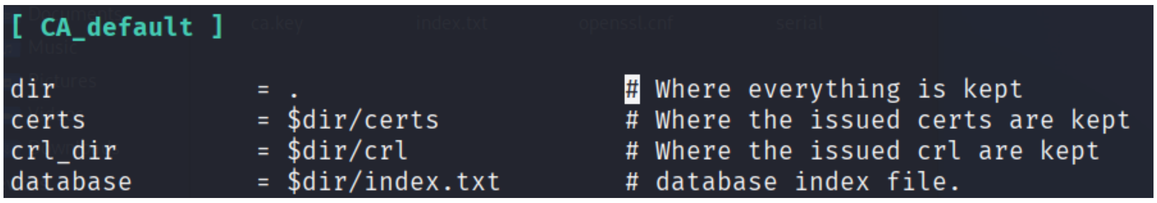
```
# mkdir ~/CA
# mkdir ~/CA/certs
# mkdir ~/CA/crl
# mkdir ~/CA/newcerts
# touch ~/CA/index.txt
# echo 1000 > ~/CA/serial
# cp /usr/lib/ssl/openssl.cnf ~/CA/
# cd ~/CA
```

The [CA_default] section of the configuration file outlines the default settings we need to prepare. We must create several sub-directories.



```
[ 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.
```



```
[ CA_default ]

dir             = .                     # 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.
```

For the index.txt file, simply create an empty file. For the serial file, include a single number in string format (e.g., 1000). Once you have configured the openssl.cnf file, you can proceed to create and issue certificates.

Certificate Authority (CA). As previously mentioned, we need to generate a self-signed certificate for our CA. This ensures that the CA is fully trusted, and its certificate will act as the root certificate. You can use the following command to generate the self-signed certificate for the CA:

```
# openssl req -x509 -newkey rsa:4096 -sha256 -days 365 \
-keyout ca.key -out ca.crt \
-subj '/CN=icthub Root CA/C=EG/ST=Alexandria/L=SG/O=icthub'
```

You will be prompted for a password, use “1234”.

You will need to enter the passphrase each time you use this CA to sign certificates for others. Additionally, you will be prompted to provide subject information, such as the Country Name, Common Name, etc. The command's output is stored in two files: ca.key and ca.crt. The ca.key file contains the CA's private key, while ca.crt holds the public-key certificate. To view the decoded content of the X509 certificate and the RSA key, you can use the following commands. The -text option decodes the content into plain text, and the -noout option prevents the encoded version from being printed:

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

2.3 GENERATING A CERTIFICATE REQUEST FOR YOUR WEB SERVER

The fictional domain name “www.icthub.edu”, which we designated earlier, represents a fictional company seeking a public-key certificate from our CA. To obtain this certificate, the company must first generate a Certificate Signing Request (CSR).

The CSR includes the company's public key and identity information and is sent to the CA for verification. Upon verifying the identity information, the CA will generate a certificate.

The command to generate a CSR is similar to the one used for creating the self-signed certificate for the CA, with the primary difference being the -x509 option. Without this option, the command generates a request; with it, the command generates a self-signed certificate. The following command generates a CSR for “www.icthub.edu”:

```
# openssl req -newkey rsa:2048 -sha256 \
-keyout icthub.key -out icthub.csr \
-subj '/CN=www.icthub.edu/O=icthub/C=EG' \
-passout pass:1234
```

The command will generate a pair of public/private keys and create a Certificate Signing Request (CSR) from the public key. To view the decoded content of the CSR and private key files, you can use the following commands:

```
# openssl req -in icthub.csr -text -noout
# openssl rsa -in icthub.key -text -noout
```

2.4 GENERATING A CERTIFICATE FOR YOUR SERVER

The CSR file requires the CA's signature to form a certificate. In the real world, CSR files are typically sent to a trusted CA for their signature. In this task, we will use our own trusted CA to generate certificates. The following command converts the certificate signing request (icthub.csr) into an X509 certificate (icthub.crt), using the CA's ca.crt and ca.key:

```
# openssl ca -config ~/CA/openssl.cnf -policy policy_anything -md sha256 \
-days 3650 -in icthub.csr -out icthub.crt \
-batch -cert ca.crt -keyfile ca.key
```

In the above command, ~/CA/openssl.cnf refers to the configuration file we copied from /usr/lib/ssl/openssl.cnf and modified it in Task 1. We use the policy_anything policy defined in the configuration file. This is not the default policy; the default policy imposes more restrictions, requiring some of the subject information in the request to match those in the CA's certificate. The policy_anything policy, as its name suggests, does not enforce any matching rules. After signing the certificate, please use the following command to print out the decoded content of the certificate:

```
# openssl x509 -in icthub.crt -text -noout
```

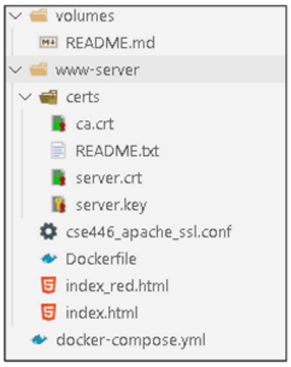
2.5 DEPLOYING CERTIFICATE IN AN APACHE-BASED HTTPS WEBSITE

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 on Apache web server. 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 www.icthub.edu. 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.

Download and extract the “PKI-PoC.zip” file to a folder of your choice and change directory into this folder. This folder is later referred to as the PKI-PoC folder. The folder structure is shown in the image to the right



In our PKI-PoC folder, we have a file called cse446_apache_ssl.conf, which contains the following entry which sets up Apache for both HTTP and HTTPS services. In the Dockerfile, we have already included the commands to copy the certificate and key to the /certs folder of the container.

```
RUN apt-get update && apt-get -y install apache2 openssl curl iproute2

ARG WWWDIR=/var/www/icthub

COPY icthub.crt /etc/apache2/ssl/
COPY icthub.key /etc/apache2/ssl/
COPY cyber_apache_ssl /etc/apache2/sites-available/
COPY index.html /var/www/icthub/index.html
COPY index_red.html /var/www/icthub/index_red.html

RUN  chmod 400 /etc/apache2/ssl/icthub.key \
    && chmod 644 $WWWDIR/index.html \
    && chmod 644 $WWWDIR/index_red.html \
    && a2ensite cyber_apache_ssl
```

In addition to the cse446_apache_ssl.conf file, you will need to modify the Dockerfile where necessary to load the required server certificate and keys, along with the CA public key. Once completed, start the Docker Compose environment and gain shell access to the new container.

2.5.1 Starting the Apache server.

The Apache server does not start automatically in the container due to the requirement to enter the password to unlock the private key. Let us access the container and run the following command to start the server (additional related commands are also listed):

```
// 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. Since our private key is encrypted, Apache will prompt for the password to decrypt it. Inside the container, the password used for the site is your AIU Student ID. Once everything is properly set up, we can browse the website, and all traffic between the browser and the server will be encrypted.

3 Lab Submission

Students must submit:

- Screenshots of key commands & outputs.

Prepared by:

Marwa M. El-Azab
Cybersecurity Research Assistant (MSc Candidate),
Teaching Assistant at AASTMT

Omar M. El-Azab
Cybersecurity Instructor, Technical Advisor,
Adjunct Teaching Assistant at AASTMT