OpenSSL Intro (1)

OpenSSL is a widely used open-source software library that provides cryptographic functions and protocols to secure communications over computer networks. In this lab, we will gain hands-on experience with OpenSSL by performing cryptographic operations. The goal of this lab is to learn how to generate cryptographic keys, create digital certificates, and perform encryption and decryption operations.

Objectives:

- Understand the purpose and role of SSL Certificate Authorities in web security.
- Create a root Certificate Authority (CA) for local development.
- Generate SSL certificates for localhost domains.
- Configure web servers to use the self-signed SSL certificates for HTTPS.
- Test the secure communication with HTTPS-enabled web servers.

First we install the OpenSSL tool

Use your VM to install
`openssl` with the appropriate command. On Debian that is:

```
$ sudo apt install openssl
```

Alternatively, you could build from the source
https://www.openssl.org/source/ after obtaining and extracting it:

```
$ ./config
$ make
$ make test # This step is optional.
$ sudo make install
```

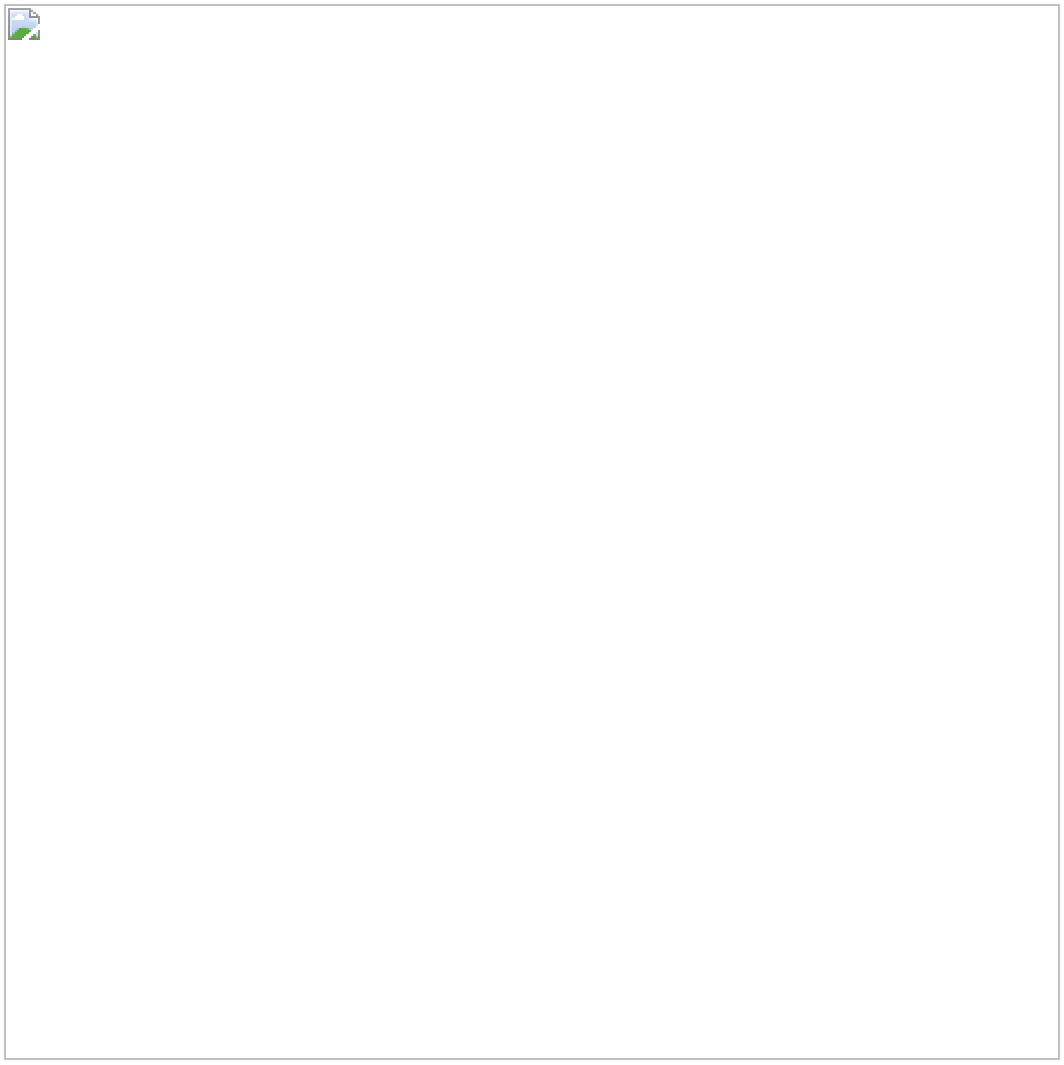
Remember that you can add the
`apt install` line to your Vagrantfile to use OpenSSL on Vagrant.

Use OpenSSL to encrypt files with symmetric encryption

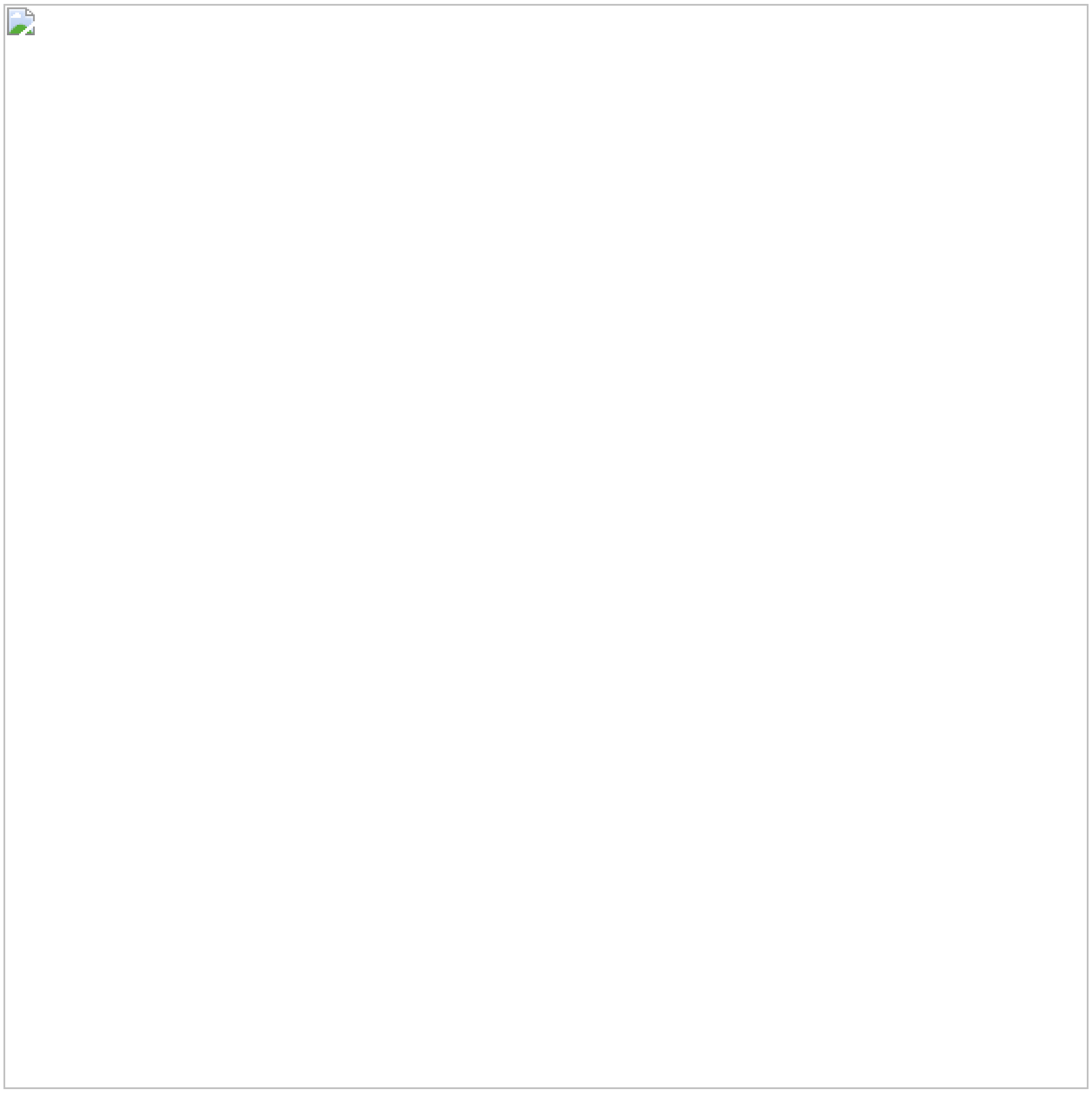
Create a text file called
`mytext.txt` and write a message you want to send inside it. Then, encrypt this file with a symmetric AES 256 key using OpenSSL:

```
openssl aes256 -in mytext.txt -out mytext.enc
```

You'll be asked for a passphrase to create the key. The output will be named
`mytext.enc`. Be sure not to forget your passphrase.



<u>What is the difference between your unencrypted file and the encrypted one? Try and read an ecrypted file.</u>



Alternatively, you can encode the encrypted file with `base64`. This option creates a file with ASCII characters instead of a pure binary stream, making it suitable for sending via email. To do this, encrypt again specifying base64 instead of AES 256, and also passing the `-a` flag:

Command:

```
openssl base64 -a -in mytext.txt -out mytext.enc
```

Output:

<u>Is the file size larger than before? If so, why?</u>

Now, you can send your encrypted sensitive information to the recipient.

How to decrypt using OpenSSL

Use email or another method to swap encrypted files with someone. Utilize the `-d` parameter to decrypt the file, whether it's a binary stream or consists only of ASCII characters. You'll need to use the same passphrase for this as was used to encrypt the file.

Once decrypted, can you read the correct message?

Use OpenSSL to encrypt files with asymmetric encryption

Asymmetric encryption uses two sets of keys, called a key pair. A public key that can be freely distributed with the entity you want to communicate with, and a private key that is never shared.

Generate key pairs

Using the
`openssl` man pages, figure out how to use `genrsa` to generate a 1024-bit public-private key pair.

This generates the private key:

```
openssl genrsa -out rsa.private 1024
```

This generates the public key from the private key:

```
openssl rsa -in rsa.private -pubout -out rsa.public
```

Distribute the public key

The public key is meant to be shared with others. Extract your public key with the OpenSSL tool into a `.pem` file using the `-pubout` parameter:

```
rsa -in rsa.private -pubout -out rsa.public.pem
```

Exchange your public Keys

Exchange public keys with someone else. Next, create an encrypted message using the OpenSSL `-encrypt` command. You will need the following to formulate the command:

- Name of the file
- Other's person public key to encrypt
- The name of the output file

Encrypting the file:

`openssl rsautl -encrypt -inkey receiver_public.pem -pubin -in mytext.txt -out mytext.enc`

Try and view the file, what is the format?

Send the file to the other party using your preferred method. The other person can decrypt the file and read the message using the private key and the parameter `-decrypt`. The recipient can write a response, encrypt it with your public key and send it back.

Decrypting the file:

```
openssl rsautl -decrypt -inkey receiver_private.pem -in mytext.enc -out mytext_decrypted.txt
```

Why do we need HTTPS?

Websites with HTTPS provide assurance to customers that their connections are secure. HTTPS, or Hypertext Transfer Protocol Secure, is a protected version of HTTP, the protocol used for data transmission between your browser and the website you're accessing. Prioritizing the safeguarding of users' privacy and data security is crucial for instilling trust in people connected to the web services we develop online

Why will we use HTTPS locally?
We need to create a development environment that closely resembles production. This practice helps to avoid future issues when deploying your web apps or websites online.

Configure and utilize Nginx with HTTPS for creating a localhost web app and sign a certificate request with OpenSSL.
A certificate signing request (CSR) consists of three entities: the public key, the system that triggers the request, and the signature of the request. The private key will be used to encrypt the certificate. We will use OpenSSL for that.
To create a private key with OpenSSL, we first create a directory and generate the key inside it:

```
$ mkdir ~/store-csr
```

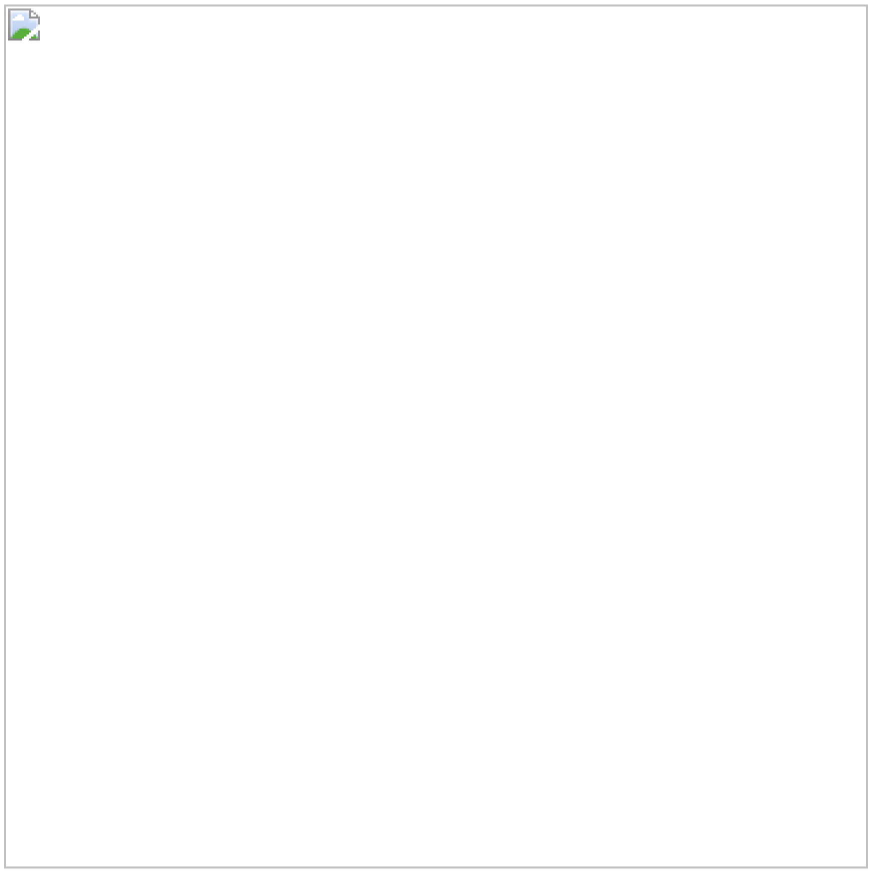
Then, we generate the RSA private key (keep note of the passphrase). Use

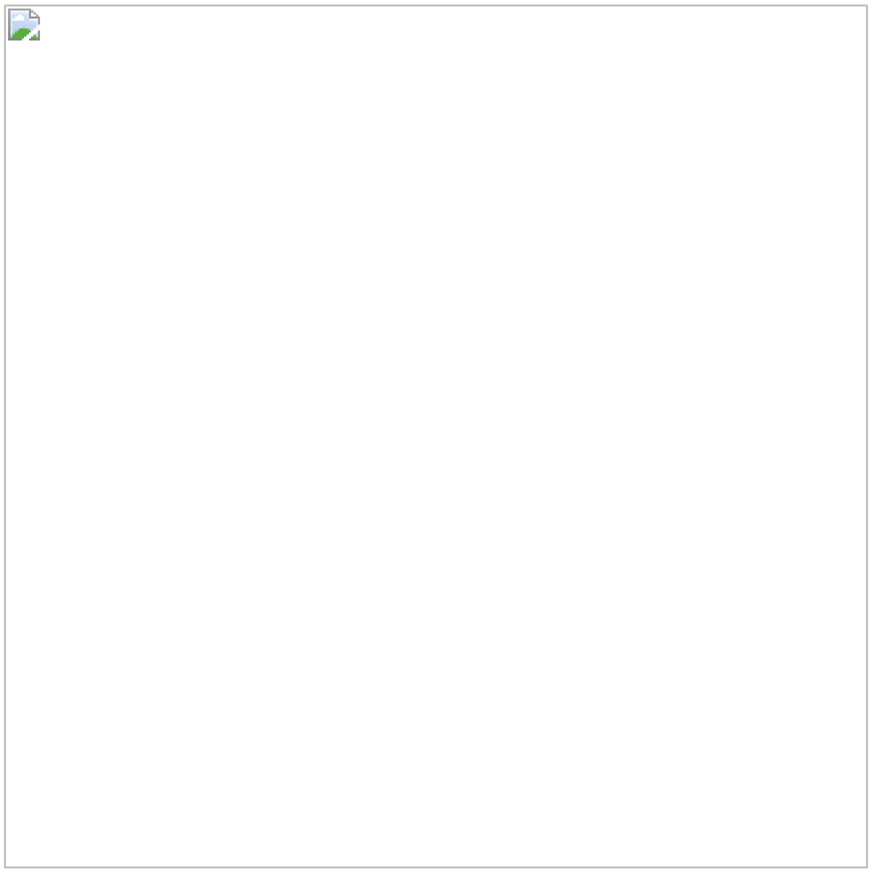
OpenSSL with the
`genrsa` command and the `-aes256 parameter` for a private key with size 2048:

```
openssl genrsa -aes256 -out rsa.private 2048
```

Next, generate a root certificate with OpenSSL using the `req` command and parameters `-x509` for a digital signature and `-sha256` for a hash function.

```
openssl req -x509 -sha256
```





Finally, make it valid for 30 days.

Then, we will make a request for the server we will use later, and we want to authenticate, for example:

```
openssl genrsa -out cert.key 2048
```

As we can see from the output, we generated an RSA private key of 2048 bits. The
`genrsa` command is used to generate RSA private key files, with the **default value being 512 bits**. You can specify different key sizes. Investigate the command options with `-help`.

Now, with the private key, we will create the CSR using the OpenSSL tool. Fill out the fields in a similar way as above, without leaving any value blank. We use the private key to create a certificate signing request as follows:

```
openssl req -new -key cert.key -out cert.csr
```

Create the Configuration File

We create a configuration file named
`openssl.cnf` in the current folder using any text editor of your choice:

```
[#]Extensions to add to a certificate request

basicConstraints       = CA:FALSE

authorityKeyIdentifier = keyid:always, issuer:always

keyUsage               = nonRepudiation, digitalSignature, keyEn

subjectAltName         = @alt_names

[ alt_names ]

DNS.1 = your hostname
```

Sign the certificate request and enter the passphrase. Verify the correctness of the certificate as follows:

```
$ ~/store-csr$ openssl verify -CAfile rootCert.pem -verify_hostname your hostname cert.crt
```

Add the certificate to the browser to avoid receiving warnings about untrusted certificates. For the Mozilla browser, follow these steps:
• Go to Preferences → Privacy → View Certificates.
• Import the rootCert.pem file.
• Click "Next" and then select "Trust this CA to identify websites."
• These steps will ensure that the root certificate is trusted by the browser for identifying websites.

Tip: if you want to name or re-name your machine use: `sudo hostnamectl set-hostname [new-hostname]` *and check with* `hostnamectl`
Move the certificate and key into the folders:

```
$ ~/store-csr$ sudo cp cert.crt /etc/ssl/certs/
$ sudo cp cert.key /etc/ssl/private/
```

Nginx Configuration

Install Nginx:

```
$ sudo apt install nginx
```

Create a configuration file inside `/etc/nginx/sites-available`. For example, create a file `yourhostname.conf`:

```
/etc/nginx/sites-available$ sudo nano myhostname.conf
```

```
server {
    client_max_body_size 100M;
    server_name yourhostname;
    listen 443 ssl;
    ssl_certificate /etc/ssl/certs/cert.crt;
    ssl_certificate_key /etc/ssl/private/cert.key;

    location / {
        proxy_pass http://yourhostname:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto https;
        proxy_read_timeout 60;
        proxy_connect_timeout 60;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Edit your
`bash /etc/hosts` file and add the line `bash 127.0.0.1 “yourhostname”` using `sudo`.
Restart the server:

```
sudo systemctl restart nginx.service
```

Tip: If you encounter any error check with `sudo tail -f /var/log/nginx/error.log`
Create a symbolic link between the sites from nginx and your configuration file:

```
sudo ln -s /etc/nginx/sites-available/yourhostname.conf /etc/nginx/sites-enabled/
```

If everything is configured correctly, you should have access to your local website with your self-signed certificate, i.e.,
https://my-hostname/. This is a good way to test your web application locally before publishing it. Now you have created a local development environment to test your apps using HTTPS.

How to use OpenSSL to test an SSL connections

The OpenSSL tool includes many utilities. We can use one of them to verify a secure connection to a server. To do so, we will use the

`s_client` utility to establish client-to-server communication. Therefore, we can determine whether a port is open, if the server is configured to support SSL, and when the certificate expires.

In the terminal, use

`s_client` against a webserver with HTTPS.

What are the differences compared to other tools such as netcat or telnet?

Try running the command again with the

`-crlf` and `-brief flags`. What is the difference?"