CS2107 Self-Exploration Activity 6

Notes:

In this Activity 6 about **Public Key Infrastructure (PKI)**, you will perform the following:

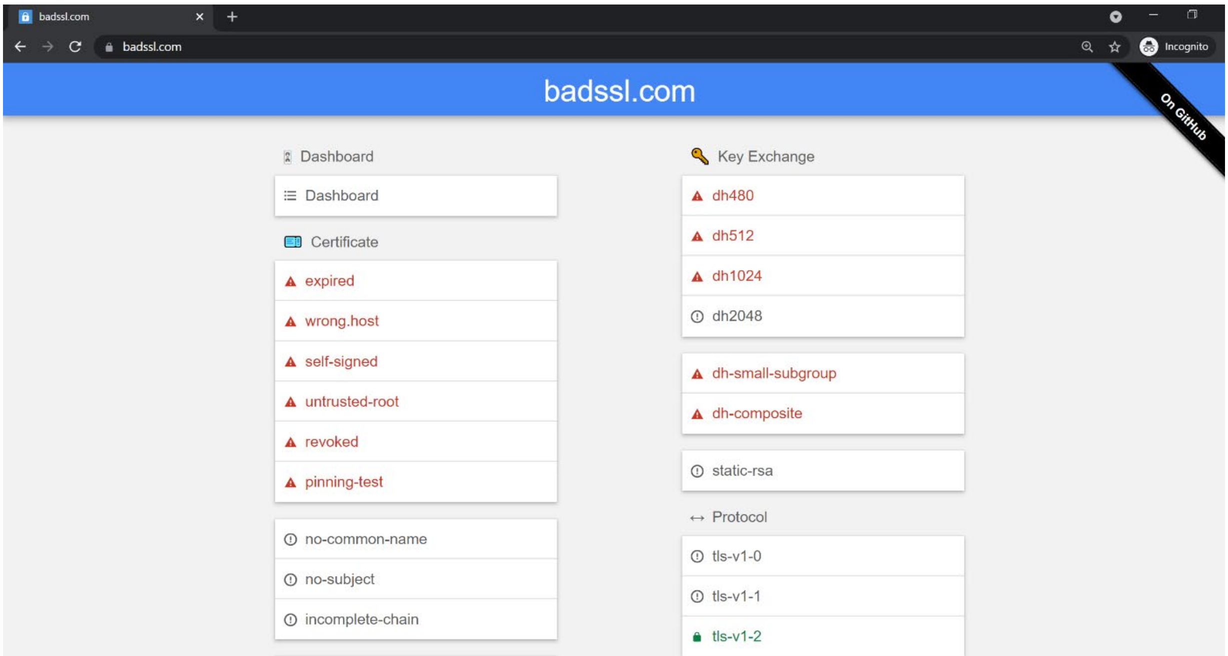
1. To observe how your web browser **reports various certificate problems** of the websites being visited;
2. To check how a commercial CA accepts a **request for a certificate issuance**;
3. To use `openssl` to generate an **RSA public-private key pair** and **Certificate Signing Request (CSR)** for a certification with a CA;
4. To use `openssl` to inspect the **content of an issued certificate**;
5. (*Extra*) To use `openssl` to generate perform **RSA encryption/decryption operations** using the previously generated RSA public-private key pair.

(**Task 1** is a simple task, which can help you understand how a browser deals with certificate errors/problems. **Tasks 2-5** in this particular activity are rather technical; and ***only if you are curious*** on how an RSA private/public key pair can first be generated, and how a certificate can be subsequently requested and inspected.)

Task 1: Observing Browser Reports of Certificate Problems

Using your browser, do visit https://badssl.com/. In the shown webpage, you should notice the "***Certificate***" **section** as shown in the figure below. Please click all the links in this section, which will bring you into *different websites* whose URLs are sub-domains of badssl.com. Observe how your browser **reports all the different**

certificate issues with the visited websites, such as expired, wrong-hostname, self-signed, untrusted root-CA, and revoked certificates.



You can also visit the websites listed in the "***Cipher Suite***" section. The cipher suite of these websites uses **broken/insecure crypto algorithms**, such as rc4-md5, and rc4, 3des, which are not accepted by HTTPS (i.e. the underlying TLS protocol).

Task 2: Checking How a Commercial CA Accepts a Request for Certificate Issuance

As explained in our lecture about a CA and its roles, you can check some widely known **commercial CAs**, including DigiCert (https://www.digicert.com). DigiCert was reported to be the world's largest high-assurance CA, commanding 59% of the Extended Validation SSL certificate market, and 96% of organisation-validated certificates globally (Netcraft SSL Survey, September 2020, https://trends.netcraft.com/www.digicert.com#extended-validation).

You can also take a look at the following webpage about how you can **create a *Certificate Signing Request* (CSR)** for DigiCert: https://www.digicert.com/kb/csr-creation.htm. As explained, a ***CSR*** is an encoded file that provides you with a standardized way to send to a CA **your public key** as well as some information identifying your company and **domain name**, including: *common name* (e.g., www.example.com), *organization name and location* (country, state/province, city/town), *key type* (typically RSA), and *key size* (2048-bit minimum).

Task 3: Using `openssl` to Generate an RSA Key Pair and CSR for a Certification

You can use `openssl` to **create a CSR**. The **easiest/automated** way is utilize this following "***OpenSSL CSR Wizard***" page: https://www.digicert.com/easy-csr/openssl.htm. You can thus easily generate the `openssl` command that you need to run, such as:

```
openssl req -new -newkey rsa:2048 -nodes -keyout
www_mydomain_com.key -out www_mydomain_com.csr
-subj "/C=SG/ST=Singapore/L=Singapore/O=My
Organization/OU=My Dept/CN=www.mydomain.com"
```

The ***single* command** above asks `openssl` to create both your private key and CSR, and saves them to 2 files: `www_mydomain_com.key`, and `www_mydomain_com.csr`.

You can then **submit** the generated CSR file to the CA after verifying the CSR file with the following command:

```
openssl req -text -in www_mydomain_com.csr -noout -verify
```

Alternatively, you can utilize `openssl` to generate your RSA private/public key pair and create a CSR ***in two separate steps***. The steps are given below.
If needed, you can refer to the following articles:

- https://www.digicert.com/kb/ssl-support/openssl-quick-reference-guide.htm
- https://wiki.openssl.org/index.php/Command_Line_Utilities
- https://www.madboa.com/geek/openssl/#how-do-i-generate-a-certificate-request-for-verisign.

Notice that, the `genrsa` sub-command of `openssl` is superseded by the newer **`genpkey`** sub-command. You can display the valid options for this sub-command by running: `openssl genpkey -help`. Likewise, you can use the **`pkey`** sub-command for your public and private key management operations instead of the RSA-specific key management sub-command `rsa`.

First, do generate a (AES-256) **password-protected RSA private/public key pair** with the key size of **2048 bits** by running:

```
openssl genpkey -aes256 -algorithm RSA
-pkeyopt rsa_keygen_bits:2048 -out private-key.key
```

The operation outputs the ***"private-key file"*** `private-key.key`.

[***Note***: This file is stored in the **Privacy-Enhanced Mail (PEM)** file format, which is the **base64-encoded** version of the binary DER format (you can also see: https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail). As explained in the wiki article, the PEM file format is commonly used for storing/sending cryptographic keys and certificates. If you are curious, you can view the outputted base64 characters of the file by running: `less private-key.key`.]

To really see the **details** of the generated key pair, however, you need to run:

```
openssl pkey -in private-key.key -text
```

Notice that, as shown, the outputted `private-key.key` actually contains **both the private key and the public key**. More specifically, it contains **all the selected RSA parameters**, including the modulus (*n*), publicExponent (*e*), privateExponent (*d*), prime1 (*p*), and prime2 (*q*).

Next, you can **extract the public key** from your private-key file, and output it to the `public-key.key` file as follows:

```
openssl pkey -in private-key.key -out public-key.key
-pubout
```

You can optionally view the **details of the public key** in the file by invoking:

```
openssl pkey -in public-key.key -pubin -text
```

Notice that only the modulus (*n*) and publicExponent (*e*) are contained in the public-key file and shown by the command accordingly.

Lastly, to **create a CSR** from your public-key file, you can use the **`req`** sub-command (for CSR management tasks) as follows:

```
openssl req -new -key public-key.key
-out www_mydomain_com.csr
```

After entering the command, you will be asked series of questions about your domain name's information. Your answers will be embedded in the CSR.

Like in the single-command way above, you can then **submit** the generated CSR file to the CA, perhaps after optionally verifying the CSR file using:

```
openssl req -text -in www_mydomain_com.csr -noout
-verify
```

Task 4: Inspecting the Content of an Issued Certificate

After you've received your certificate from your CA, you can inspect, verify, and subsequently install it on your server.

To **inspect your issued certificate**, you can use the `openssl` **`x509`** sub-command as shown at: https://www.digicert.com/kb/ssl-support/openssl-quick-reference-guide.htm#ViewingCertificateInformation.

For instance, you can view the **whole content** of your certificate by running:

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

To extract **specific parts of the certificate**, you can run the following commands for different respective extracted information:

- **Who issued** the certificate:
  ```
  openssl x509 -in www_mydomain_com.crt -issuer -noout
  ```
- The **subject** to whom was it issued:
  ```
  openssl x509 -in www_mydomain_com.crt -subject -noout
  ```
- For **what dates** is it valid:
  ```
  openssl x509 -in www_mydomain_com.crt -dates -noout
  ```
- The above, **all at once**:
  ```
  openssl x509 -in www_mydomain_com.crt -issuer
  -subject -dates -noout
  ```

You can then **verify** the issued certificate by running the `openssl` **`verify`** sub-command. Do refer to its following man page for details: https://www.openssl.org/docs/man1.0.2/man1/openssl-verify.html.

Once your checking on the certificate is done, you can subsequently **install the certificate**, for instance, on your Ubuntu server running Apache2 as described in the following document: https://www.digicert.com/kb/csr-ssl-installation/ubuntu-server-with-apache2-openssl.htm.

Task 5 (*Extra*): Using `openssl` to Perform RSA Encryption/Decryption Operations using the RSA Key Pair

In Task 3 above, you do generate an RSA private/public key pair. You can then additionally use `openssl` to perform **RSA encryption/decryption operations** by using its **`rsautl`** sub-command.

You can perform **an encryption** of the `plaintext.txt` plaintext file as follows:

```
openssl rsautl -encrypt -inkey public-key.key
-pubin -in plaintext.txt -out ciphertext.txt
```

Note that the command above needs **the public-key file** of the recipient in PEM format. You can easily derive the needed public-key file (`public-key.key`) from the recipient's certificate (`recipient.crt`) as follows:

```
openssl x509 -in recipient.crt –pubkey -noout
-out public-key.key
```

An RSA decryption can be done as follows:

```
openssl rsautl -decrypt -inkey private-key.key
-in ciphertext.txt -out recovered-plaintext.txt
```

To use various other options of the `rsautl` sub-command, including the **padding schemes available (e.g. OAEP)**, do check the sub-command's **man page**: https://www.openssl.org/docs/manmaster/man1/openssl-rsautl.html.