UNIVERSITY OF LIMOGES
Faculty of Science and Technology

<u>Master 1</u>
Cryptology and Information Security (CRYPTIS)
Informatic Course

Security of ICT uses - Semester 2

Authentification, Steganography and Web Security

DUONG Dang-Hung

Professor: Pierre-François BONNEFOI

May 15, 2023

Table of contents

List of Figures

Chapter 1

Project description

The project's objective is to simulate a Certification Authority (called CertifPlus), which is capable of performing the 3 following actions:

- Render a certificate:
  - Step 1: The client submit his name and the title of the certificate
  - Step 2: The server retrieve the information submitted
  - Step 3: The server displays the information on the certificate background
  - Step 4: The server sign the information with his public key to provide authenticity
  - Step 5: The server request the timestamp from `freeTSA.org`, concatenate the information with the timestamp then conceal this information into the certificate
- Return a certificate:
  - Step 1: The client request his certificate (which has been rendered when the client submit his information)
  - Step 2: The server return the certificate (if it has been created) to the client
- Verify a certificate:
  - Step 1: The client submit an image of the certificate to be verified
  - Step 2: The server retrieve the image submitted
  - Step 3: The server extract the data hidden by steganography and verify the timestamp
  - Step 4: The server extract the signature from the QR code and perform authentication by the signature
  - Step 5: The server respond to the client if the certificate is well-verified or was tampered

Then we are required to perform a short risk analysis over the application system.

As there are quite a lot of things to submit for the project, as specified in the course project document, I would like to specify the structure of my project submission as follows:

- The root certificate of the AC: inside `/ecc` directory
- The configuration files of the AC: file `gen_key` and `/ecc` directory
- The certificate of the application generated by the AC: inside `/ecc` directory
- The private key of the AC: inside `/ecc` directory
- The source codes of different Python programs: `backend.py`, `server.py` and `steganography.py`
- The `cURL` requests script: `/client_demo/curl_script`
- A certificate example: inside `/imgs` and `/client_demo`
- The report: This
- Risks analysis: At the end of the report

Chapter 2

Programming Implementation

We begin the project with creating several necessary certificates and keys of the Certificaion Authority and its server. Firstly, we create a private key `ecc.ca.key.pem` for the CA, and create its self-generated certificate `ecc.ca.cert.pem` (the root certificate):

```
openssl ecparam -out ./ecc/ecc.ca.key.pem -name prime256v1 -genkey
openssl req -config <(printf "[req]\ndistinguished_name=dn\n[dn]\n[ext]\
    nbasicConstraints=CA:TRUE") -new -nodes -subj "/C=FR/L=Limoges/O=
    CRYPTIS/OU=SecuTIC/CN=ACSECUTIC" -x509 -extensions ext -sha256 -key ./
    ecc/ecc.ca.key.pem -text -out ./ecc/ecc.ca.cert.pem
```

Secondly, we create a private key `ecc.serveur.key.pem` for the server (the server of Certif-Plus), create its certificate `ecc.serveur.pem` containing its public key which is signed with the certificate of the CA, and extract its public key `ecc.serveur.pubkey.pem` of the server from its certificate:

```
openssl ecparam -out ./ecc/ecc.serveur.key.pem -name prime256v1 -genkey
openssl req -config <(printf "[req]\ndistinguished_name=dn\n[dn]\n[ext]\
    nbasicConstraints=CA:FALSE") -new -subj "/C=FR/L=Limoges/O=CRYPTIS/OU=
    SecuTIC/CN=localhost" -reqexts ext -sha256 -key ./ecc/ecc.serveur.key.
    pem -text -out ./ecc/ecc.csr.pem
openssl x509 -req -days 3650 -CA ./ecc/ecc.ca.cert.pem -CAkey ./ecc/ecc.ca
    .key.pem -CAcreateserial -extfile <(printf "basicConstraints=critical,
    CA:FALSE") -in ./ecc/ecc.csr.pem -text -out ./ecc/ecc.serveur.pem
openssl x509 -in ./ecc/ecc.serveur.pem -noout -pubkey > ./ecc/ecc.serveur.
    pubkey.pem
```

We also create the `bundle_serveur.pem` which is a text file containing both the private key and the server certificate.

```
cat ./ecc/ecc.serveur.key.pem ./ecc/ecc.serveur.pem > ./ecc/bundle_serveur
    .pem
```

These keys and certificates are saved in the directory `/ecc` by default.

2.1 Certificate creation

This behavior of the server is handled by the path `/creation`, which will receive POST requests from clients, containing his personal identity and the title of the certificate.

When receiving the request of the client, we also want to obtain his IP address, hash it and use this to name the files we will create for this client, in order to distinguish his files from those belong to other clients.

```
@route('/creation', method='POST')
def creation():
    contenu_identite = request.params['identite']
    contenu_intitule_certification = request.params['intitule\_certif']
    print('nom prenom :', contenu\_identite, ' intitule de la certification :', contenu\_intitule\_certification)
    # Get the source IP address
    src_ip_address = request.environ.get('REMOTE_ADDR')
    # Hash the source IP address with SHA-256
    src_ip_address_hash = hashlib.sha256(bytes(src_ip_address, "utf-8")).hexdigest()
    # Render the certificate for the IP address requested
    render_cert(identity=contenu_identite, cert_title=contenu_intitule_certification, src_ip_address_hash=src_ip_address_hash)
    response.set_header('Content-type', 'text/plain')
    return 'ok!\r\n'
```

The `render_cert()` performs as follows:

Algorithm 1 render_cert()

Concatenate the identity and certificate title
Create a file `info_fname` named by the source IP address hashed with SHA256 and save the information block to this file
Render the text to display on the certificate using Google API
Resize and integrate the text with the certificate background
`signature` $\leftarrow$ sign(`info_fname`)
Make the QR code from `signature`, resize and integrate it with the certificate
`timestamp_data` $\leftarrow$ `timestamp(info_fname)`
Dissimulate (`info` $\|$ `timestamp_data`) to the certificate by steganography

The `sign()` function takes a file containing the information block as input, say `info_fname`, then use the private key `ecc.serveur.key.pem` of the server to sign and save the signature to a file named texttinfo_fname.sig, using `openssl` command. Additionally, in our program, we choose SHA256 as the hash function:

```
    # Sign the information block with private key
    sign_cmd = ['openssl',
                'dgst',
                '-sha256',
                '-sign',
                './ecc/ecc.serveur.key.pem',
                '-out',
                sig_fname,
                info_fname]
    subprocess.run(sign_cmd)
```

which can be converted to the bash command:

```
openssl dgst -sha256 -sign ./ecc/ecc.serveur.key.pem -out [info_fname].sig
    [info_fname]
```

The `timestamp()` function takes a file containing the information block as input, say `info_fname`, create a `.tsq` (TimeStampRequest), send it to `freeTSA.org/tsr` and return the TimeStampResponse (or timestamp in short) received from that website.

The function to add hide information in the certificate image using steganography is as given in the document.

2.2 Certificate return

This behavior of the server is handled by the path `/fond`, which receive GET requests from the clients, hash the client IP address with SHA-256 to find the filename of the certificate corresponding to that client, and return the certificate for the client (return an error message if the client's certificate is not found):

```
@route('/fond')
def fond():
    src_ip_address = request.environ.get('REMOTE_ADDR')
    src_ip_address_hash = hashlib.sha256(bytes(src_ip_address, "utf-8")).
   hexdigest()
    response.set_header('Content-type', 'image/png')
    try:
        with open(f'./imgs/{src_ip_address_hash}_final.png','rb') as
   descripteur_fichier:
            contenu_fichier = descripteur_fichier.read()
    except Exception:
        return "You have not submit your information yet"
    return contenu_fichier
```

2.3 Certificate verification

This behavior of the server is handled by the path `/verification`, which receive `POST` requests from the clients. The client will submit the certificate to be verified to the server, the server will then save the certificate (named according to the hashed IP address of the client).

```python
@route('/verification', method='POST')
def verification():
    contenu_image = request.files.get('image')
    # Get the source IP address
    src_ip_address = request.environ.get('REMOTE_ADDR')
    # Hash the source IP address with SHA-256
    src_ip_address_hash = hashlib.sha256(bytes(src_ip_address, "utf-8")).hexdigest()
    # Save the image submitted by the client
    contenu_image.save(f'./imgs/{src_ip_address_hash}_verify.png', overwrite=True)
    response.set_header('Content-type', 'text/plain')
    # Return the verification result
    if verify_cert(src_ip_address_hash):
        return 'Certified!\r\n'
    return 'Erroneous Certificate!\r\n'
```

The `verify_cert()` performs as follows:

Algorithm 2 verify_cert(src_ip_address_hash)

`stegano_data` ← data extracted from the image using `extract_stegano()`
`info_block_data` ← `stegano_data[:64]`
`time_stamp_data` ← `stegano_data[64:]`
Verify the `info_block_data` and `time_stamp_data` using Freetsa TSA Certificate and Freetsa CA Certificate
`signature` ← data extracted from the image using `extract_signature()`
Verify the `signature` using server's public key `ecc.serveur.pubkey.pem`

Chapter 3

Results

To test the program, `socat` is used to establish a TCP connection from the client to the server, both on `localhost`. Particularly, we will directly implement the secure TLS/TCP connection between the client and the server, using the CA's certificate `ecc.ca.cert.pem`.

We begin by starting the *application server* on port 8080, as specified in the file `server.py`, and the *front-end server* on port 9876, using the command:

```
$ socat openssl-listen:9876,fork,cert=./ecc/bundle_serveur.pem,cafile=./ecc/ecc.ca.cert.pem,verify=0 tcp:127.0.0.1:8080
```

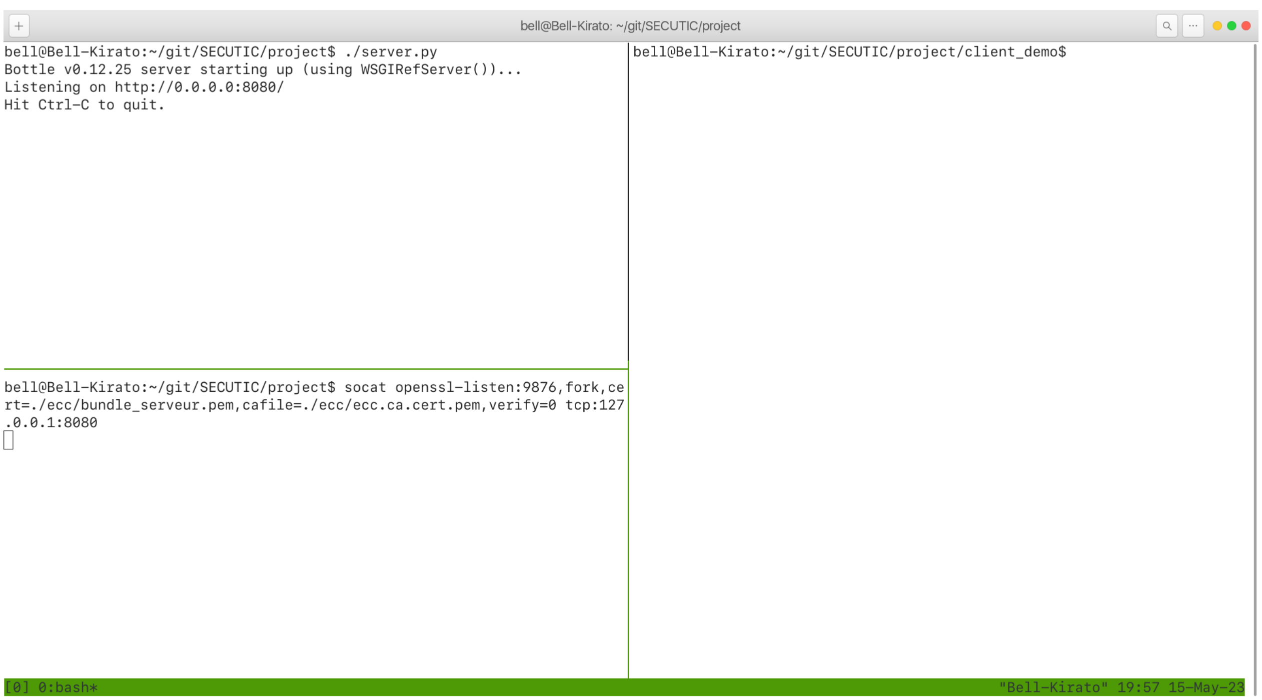

Figure 3.1: Start the application server on port 8080 and the front-end server on port 9876

Next, we try to create a certificate by submitting a POST request containing the identity and the certificate title of the client using `cURL`. Note that when using the TLS/TCP connection, the client should possess the CA's certificate `ecc.ca.cert.pem`:

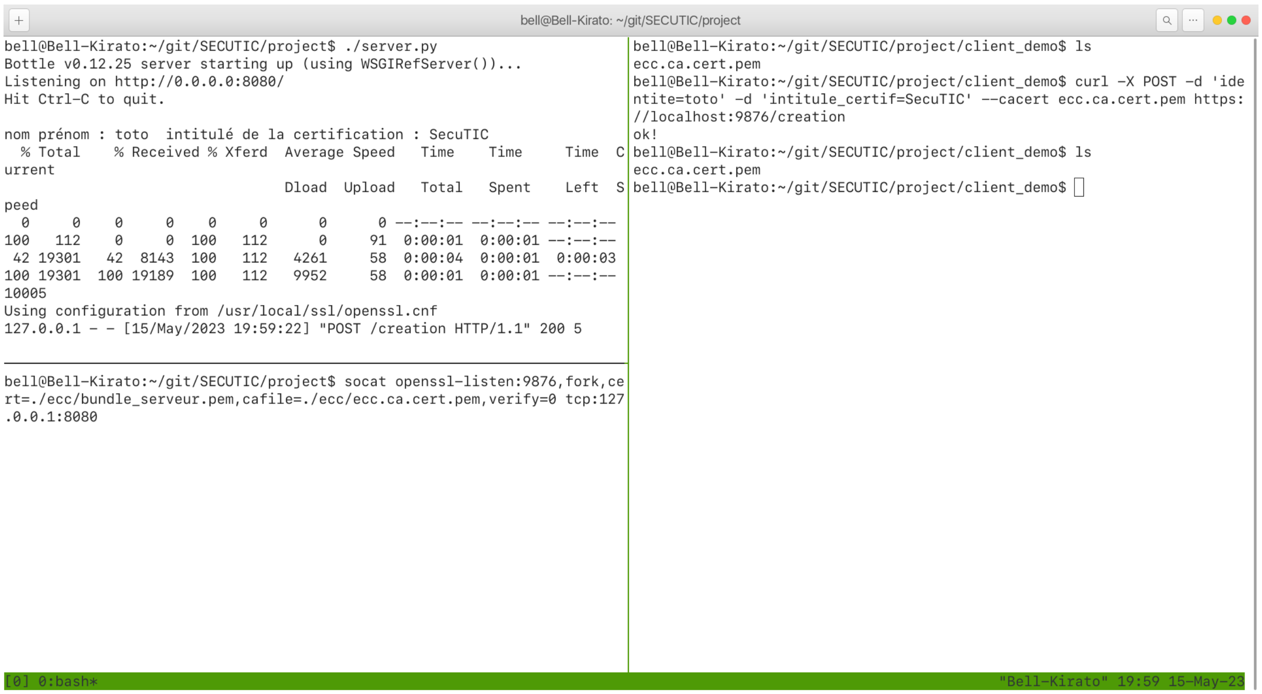

Figure 3.2: The client submit his information to create the certificate through TLS connection

We can see that the server responded with the `ok!` message, which confirms that the POST request has been received and processed properly.

Now we can try to request the certificate from the server, using a GET request, and view the certificate rendered:

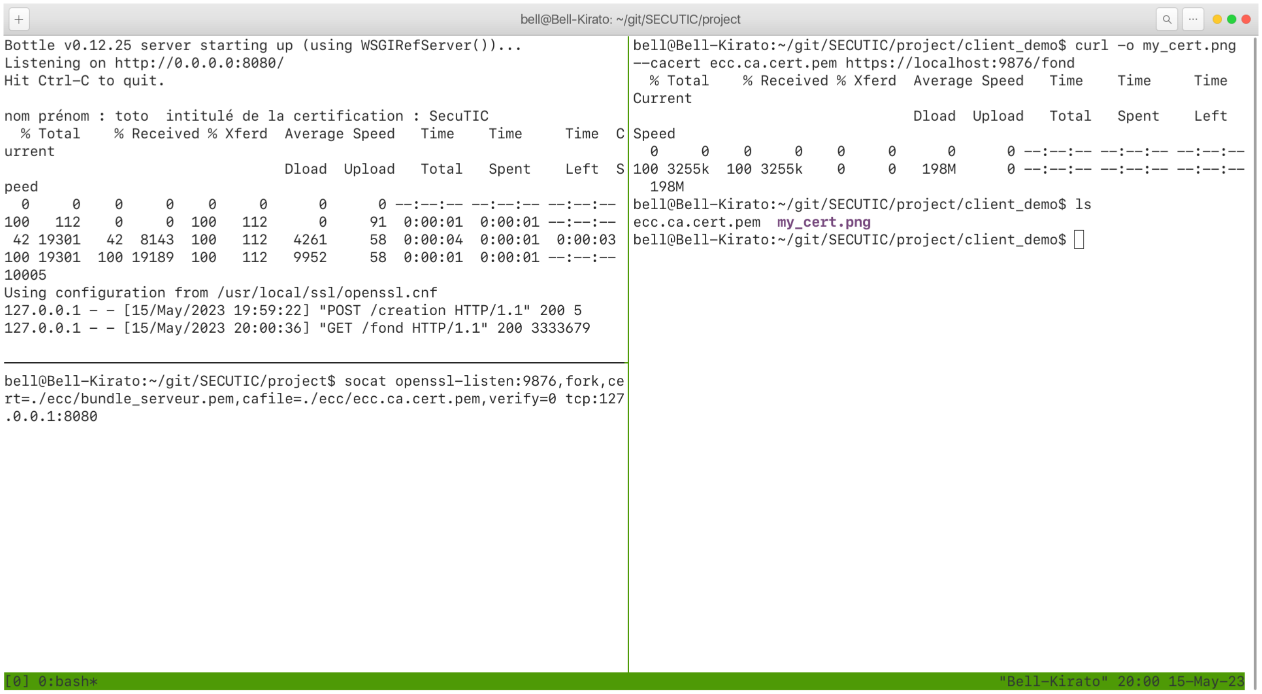

Figure 3.3: The client requests his certificate

Figure 3.4: The client's certificate

Finally, we verify the *authenticity*, *integrity* and *non-repudiation* of this certificate by sending a POST request to the `/verification` path of the server, submitting the exact certificate we just obtained:

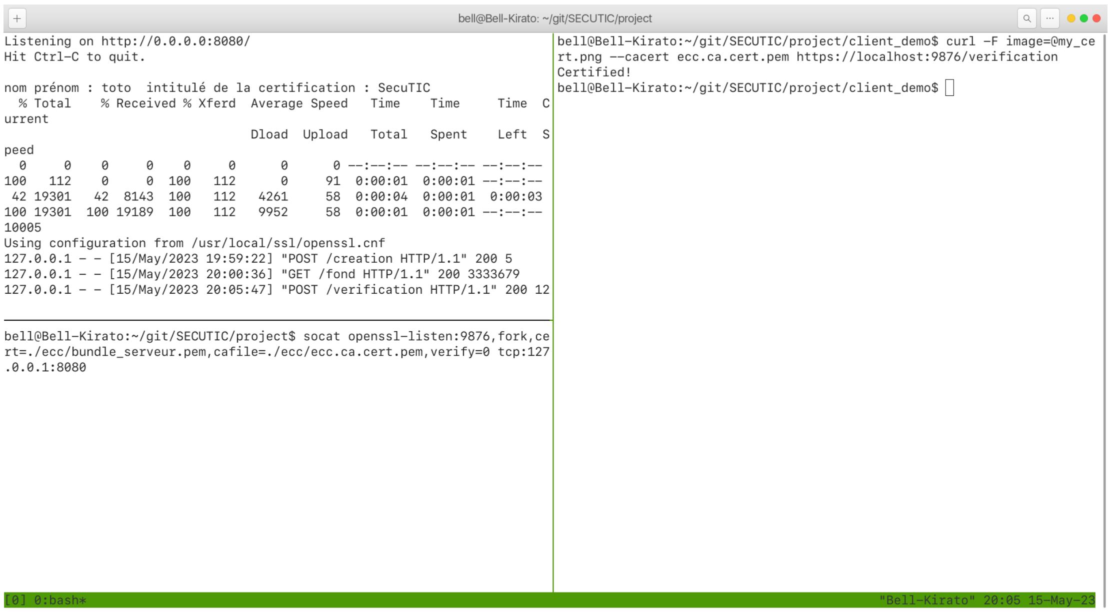

Figure 3.5: The client verifies his certificate

We see that the server responded with "Certified", as this is precisely the certificate that was certified by the server. Now, let's try to edit the image, say, changing the identity from "toto" to "meomeo", and verify it again:

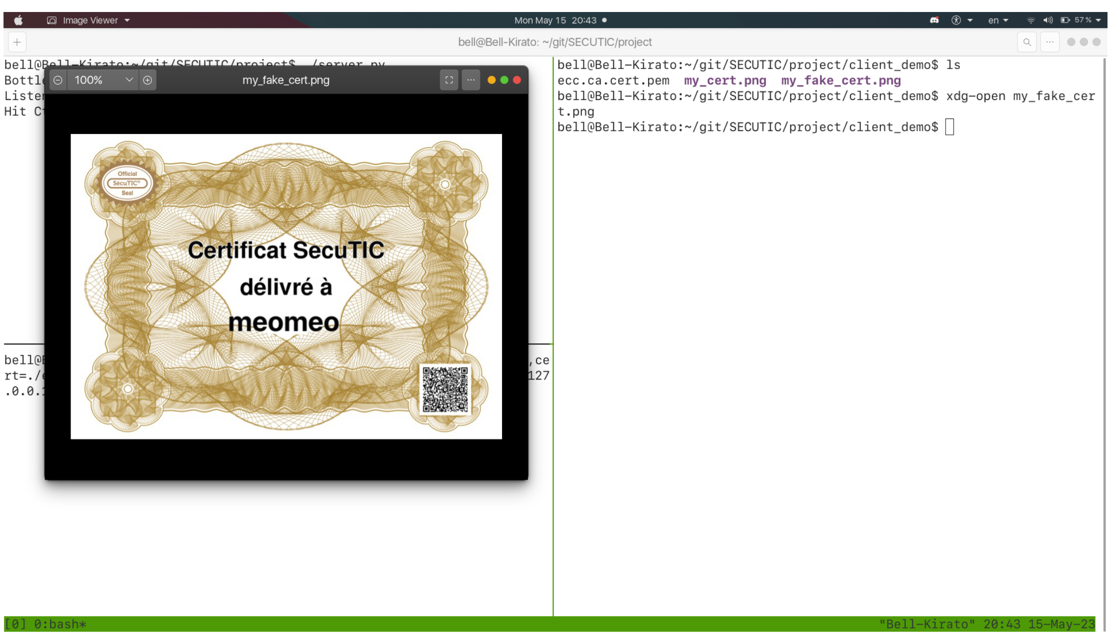

Figure 3.6: The identity in the certificate was modified from "toto" to "meomeo"

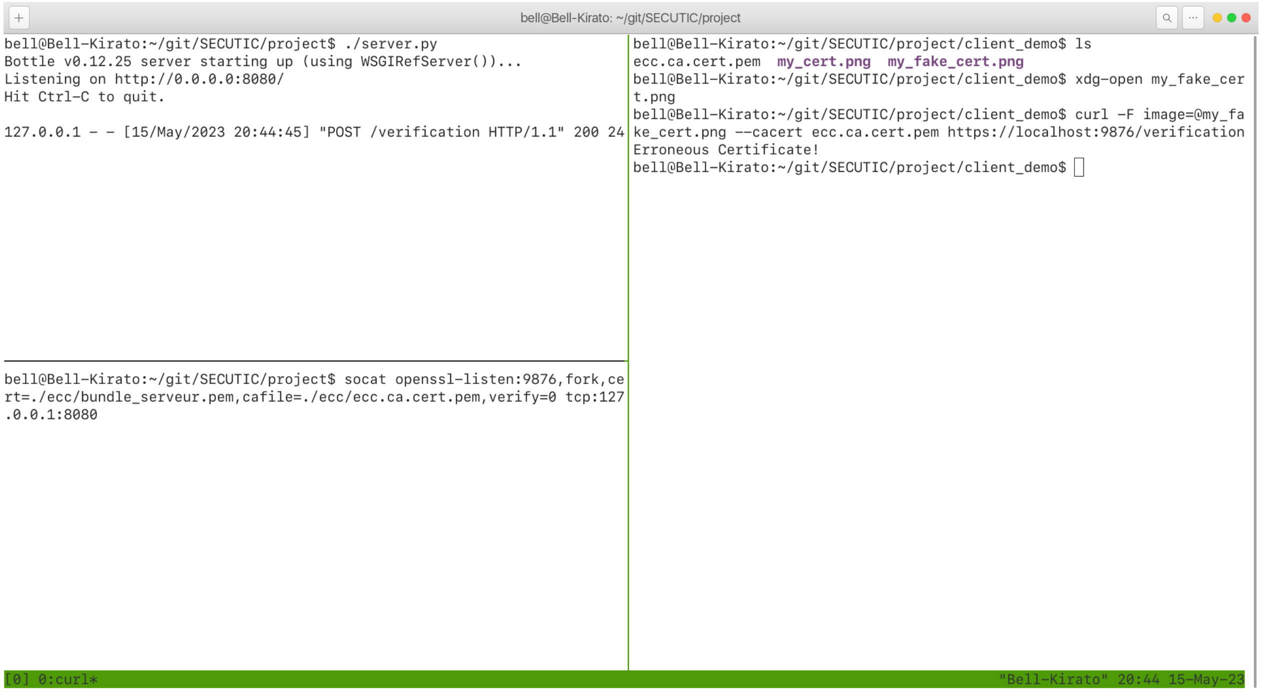

Figure 3.7: The server verify that the certificate was modified and respond with "Errorneous Certificate"

Chapter 4

Risks Analysis

4.1 Primary and secondary assets

In reality, if a company CertifPlus wants to implement this secure electronic certificate distribution systems, here is the list of some primary and secondary assets of such real company, based on different categories:

- Services:
  - Primary assets:
    - WAN Services
    - LAN Services
    - The application server, such as the one that runs the CertifPlus application to provide users with the services we have just done above
    - The data server, such as the one that stores the data of users, and stores the certificates of the clients
    - Common system services
    - Telecommunication services
    - Interface services and terminals made available to users
    - Services for publishing information on an internal or public website (for example the website of the company, or that offers the CertifPlus certificate distribution service, or both)
  - Secondary assets:
    - Services that support hardware equipment
    - Software setups
    - Media software support
    - Service-related security services
    - Accounts and resources required to access the services
- Data:
  - Primary assets:

    * Data files or application databases (such as the information that users submitted to the server and the certificates saved in the server)
    * The certificates and private keys, also saved on the server
    * Data and information published on the website
    * Data exchanged, application screens, individually sensitive data
  - Secondary assets:
    * Logical entities: Files or databases
    * Logical entities: Messages or data packets in transit
    * Physical entities: media and supports
    * Media of access to key data and various means, physical or logical, necessary to access the data

4.2 Threats and their relationships with the assets

Several threats that are currently existing in this system are:

- Insecure storage: The data of users and the certificates are not encrypted, but saved in raw images format in the database. Hence, if an attacker can somehow get access to the database, he can know all the information of all the clients who submitted their information to the application.

- Format string attack and Command injection: In our application, there are many parts where we save the information that was submitted by the clients into temporary files so that we can use them afterwards. If an attacker inject malicious codes when submitting the information to the system, our application could become vulnerable

- Buffer overflow: if the information submitted to the system is to large, for example, the certificate image submitted to be verified exceeds the storage of the database, the application could crash.

- Race condition: In our system, we try to create and save the files of different users based on the hash (using SHA-256 algorithm) of their IP address to avoid conflict in filename. This is in fact a big vulnerability. If the attacker can somehow find out an efficient way to conflict the hash, for example, using Birthday Problem attack, he can create new files with conflict filenames and replace other users' certificates. Or more simply, he can fake IP address or manually forge packets with the victim's IP address using services provided by various means, such as `scapy`. After that, when the users requests for their certificates from the server, the server only checks and find the files with the filenames corresponding to the victim's IP address hashed (which is now replaced by the attacker's malicious file), the server will send that malicious file directly to the victim.

- Dos (Denial of Service): For now, the process of signing the certificates, and requests the signature from freeTSA.org is quite expensive. If too many clients use the service, or an attacker generates a flood of requests, the system and server might get overloaded and goes down. Another possibility is that, since we are relying on freeTSA.org to request for the timestamp signature, if an attacker direct the DoS attack to freeTSA.org, or the website of freeTSA.org is brought down, our system also will not be available by default.

- OWASP (Open Web Application Security Project)