Report: TIC Project

Nguyen Viet Sang, **Bui Thi Thuy Dung**, **Le Tan Nhat Linh**
viet-sang.nguyen@etu.unilim.fr, thi-thuy-dung.bui@etu.unilim.fr, tan-nhat-linh.le@etu.unilim.fr

Master 1 Cryptis, Faculty of Science and Technique, University of Limoges

1 Introduction

In this project, we implement a service called CertiPlus which permits to create attestations and verify attestations using signatures and timestamps. This service applies QRcode and Steganography to hide information. A Frontal Server is also used to manage connections with clients. The demo can be found at https://youtu.be/_Zw2cijWcmE

This report is organised as follows:

- Section 2 presents the generation of keys and certificates. It includes the generation of Certificate Authority (CA) and the generation of Web Server.
- Section 3 presents the creation of an attestation. In this section, we firstly discuss the general flow of creating an attestation. We then show the details of creating signatures and requesting timestamps.
- Section 4 presents the verification for an attestation. As section 3, we begin with the general flow and then, go to the details of verifying a signature and verifying a timestamp.
- Section 5 shows an example of this service. In this example, we present how to establish and run the system, and how to create an attestation as well as verify it.
- Section 6 analyses some risks of the system. We present the assets (biens), services of security, threats (menaces), some scenarios and recommendations (préconisations).
- Section 7 concludes our work.

2 Generate private keys and certificate

2.1 Certificate Authority (CA)

We use Elliptic Curves to create a CA. We firstly generate a private key by the command shown in listing 1.1. After generating the key (line 2), we use the algorithm of Triple DES to protect this private key by a passphrase (line 4).

```
1 echo "Generate CA key..."
2 openssl ecparam -out $PATH_TO_CA_KEY -name prime256v1 -genkey
3 echo "CA private key with password..."
4 openssl ec -in $PATH_TO_CA_KEY -des3 -out $PATH_TO_CA_KEY
```

Listing 1.1: Generating CA private key

We then generate a certificate for this CA by the command shown in listing 1.2. We choose the hash SHA256 for this generation. This certificate contains CA's information as well as the public key.

```
1 echo "Generate CA certificate..."
2 openssl req -config <(printf "[req]\ndistinguished_name=dn\n[dn]\n[ext]\
     nbasicConstraints=CA:TRUE") \
3   -new -nodes -subj "/C=FR/L=Limoges/O=CRYPTIS/OU=SecuTIC/CN=ACSECUTIC" \
4   -x509 -extensions ext -sha256 \
5   -key $PATH_TO_CA_KEY \
6   -text -out $PATH_TO_CA_CERT
```

Listing 1.2: Generating CA certificate

2.2 Certificate of Web Server

For the Web Server, we firstly generate a private key as shown in listing 1.3.

```
echo "Generate web server private key..."
openssl ecparam -out $PATH_TO_SERVER_KEY -name prime256v1 -genkey
```

Listing 1.3: Generating Web Server private key

From the private key and information of the Web Server, we generate a certificate signing request (csr) by the command as shown in listing 1.4 (lines 2-6). Then, we generate the certificate which is authorized by the above CA (lines 8-10).

```
echo "Generate server certificate signing request..."
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=$SERVER_NAME" \
  -reqexts ext -sha256 \
  -key $PATH_TO_SERVER_KEY \
  -text -out $PATH_TO_SERVER_CSR
echo "Generate server certificate..."
openssl x509 -req -days 3650 -CA $PATH_TO_CA_CERT -CAkey $PATH_TO_CA_KEY \
  -CAcreateserial -extfile <(printf "basicConstraints=critical,CA:FALSE") \
  -in $PATH_TO_SERVER_CSR -text -out $PATH_TO_SERVER_CERT
```

Listing 1.4: Generating Web Server certificate

For verification, it is necessary to extract Web Server public key from its certificate. The extraction command is shown in listing 1.5.

```
echo "Extract server public key from certificate..."
openssl x509 -pubkey -noout -in $PATH_TO_SERVER_CERT > $PATH_TO_SERVER_PUBKEY
```

Listing 1.5: Extracting Web Server public key

We create a bundle from the private key and the certificate of Web Server (listing 1.6) in order to be able to use the Frontal Server.

```
echo "Generate bunlde server..."
cat $PATH_TO_SERVER_KEY $PATH_TO_SERVER_CERT > $PATH_TO_BUNDLE_SERVER
```

Listing 1.6: Generating bundle from private key and certificate of server

3 Create attestation

Listing 1.7 shows the flow of creating an attestation. Provided `identite` and `intitule`, we firstly generate the signature of the concatenation of this information (line 2). We then request a timestamp at https://www.freetsa.org with the concatenation (line 3). The obtained signature and timestamp are converted from binary to ASCII with `base64` (lines 5-6 and 13-14). In the next step, we generate the QRcode of signature (line 8), generate the text in attestation (line 9), combine the text and background (fond) (line 10) and combine the QRcode to background (line 11). After that, the hidden message in steganography is expanded (line 15). Finally, we hide the message in the attestation (line 17).

```
def create_attestation(args, identite, intitule):
  is_ok = generate_signature(args, identite+intitule)
  is_ok = is_ok and request_timestamp(args, identite+intitule)

  sig_bin = read_file(args.path_to_signature, 'rb') # signature in binary
  sig_ascii = convert_binary_to_ascii(sig_bin) # signature in ascii

  is_ok = is_ok and generate_qrcode(args, sig_ascii)
  is_ok = is_ok and generate_text_png(args, identite, intitule)
  is_ok = is_ok and combine_text_fond(args)
  is_ok = is_ok and combine_qrcode_fond(args)
```

```
12
13    ts_bin = read_file(args.path_to_ts_response, 'rb') # timestamp in binary
14    ts_ascii = convert_binary_to_ascii(ts_bin) # timestamp in ascii
15    message_stegano = expand_message_stegano(args, identite, intitule, ts_ascii
        )
16    img = Image.open(args.path_to_attestation)
17    is_ok = is_ok and cacher(img, message_stegano)
18    img.save(args.path_to_attestation)
19
20    return is_ok
```

Listing 1.7: Steps of creating attestation

For the hidden message in steganography, we need to concatenate `identite`, `intitule` and `ts_ascii` (timestamp) and expand them to a fixed length (line 15). We know that the length of timestamp is fixed corresponding to a certain algorithm. Therefore, it is only necessary to expand `identite+intitule` to the length of 64. The expansion is shown in listing 1.8. We use the star symbols ($*$) to represent the padding.

```
1  def expand_message_stegano(args, identite, intitule, ts):
2    len_info = len(identite + intitule)
3    if len_info < 64:
4      info = identite + intitule + args.PADDING*(64 - len_info)
5    else:
6      info = identite + intitule
7      info = info[:64]
8    return info + ts
```

Listing 1.8: Expand stegano message

3.1 Create signature

We use algorithm SHA256 to generate a signature with the information of `identite+intitule` and the private key of Web Server. The command of generation is shown in listing 1.9 (lines 3-7).

```
1  def generate_signature(args, info):
2    write_file(args.path_to_info, 'w', info)
3    cmd = subprocess.Popen('openssl dgst -sha256 -sign {} {} > {}'\
4        .format(args.path_to_server_key, \
5            args.path_to_info, \
6            args.path_to_signature), \
7      shell=True, stdout=subprocess.PIPE)
8    (result, ignorer) = cmd.communicate()
9
10   return True if cmd.returncode == 0 else False
```

Listing 1.9: Create signature code

3.2 Request timestamp

To request a timestamp, there are two steps. The first one is to create a query `tsq` (TimeStampQuery) file, which contains a hash of the file we want to sign, as shown on lines 3-5 in listing 1.10. In this case, we use SHA256 as the hash algorithm. This algorithm leads to a certain fixed length of timestamp which will be presented the method of calculation in the section of verification.

The second step is to send the query `tsq` to https://www.freetsa.org and receive a request `tsr` (TimeStampRequest) file. The `tsr` file contains the content of the desired timestamp. This work is shown on lines 9-11 in listing 1.10.

```
1  def request_timestamp(args, data):
2    write_file(args.path_to_ts_data, 'w', data)
3    cmd1 = subprocess.Popen('openssl ts -query -data {} -no_nonce -sha512 -cert
       -out {}' \
```

```
    .format(args.path_to_ts_data, args.path_to_ts_query), \
    shell=True, stdout=subprocess.PIPE)
  (result1, ignorer1) = cmd1.communicate()
    is_ok = True if cmd1.returncode == 0 else False

  cmd2 = subprocess.Popen('curl -H "Content-Type: application/timestamp-query
    " --data-binary "@{}" https://freetsa.org/tsr > {}' \
    .format(args.path_to_ts_query, args.path_to_ts_response), \
    shell=True, stdout=subprocess.PIPE)
  (result2, ignorer2) = cmd2.communicate()
  is_ok = (is_ok and True) if cmd2.returncode == 0 else (is_ok and False)

  return is_ok
```

Listing 1.10: Request timestamp code

4 Verify attestation

Listing 1.11 shows the flow of verification. As mentioned above, the length of timestamp is fixed corresponding to its algorithm. For SHA256, the length is 5493 bytes (`args.TS_LENGTH_SHA512 = 5493`). In `base64`, each character represents 6 bits and hence, 4 characters represent 3 bytes. We calculate the length of timestamp which is converted to ASCII by the equation 1.

$$len\_ts\_ascii = 4 \times \left\lceil \frac{args.TS\_LENGTH\_SHA512}{3} \right\rceil \tag{1}$$

Thanks to the length, we retrieve the stegano message as shown on line 4. From this message, we get the information of `identite+intitule` and timestamp (line 4). Padding is removed from `identite+intitule` (line 6). We then convert the timestamp in ASCII to binary (line 7). Signature in QRcode is also retrieve and converted to binary (line 16-17). Verification of attestation includes verification of signature (line 13) and verification of timestamp (line 19) which are discussed later.

```
def verify_attestation(args):
  img = Image.open(args.path_to_attestation)
  len_ts_ascii = get_len_ts_ascii(args) # length of timestamp in ascii
  message_ascii = recouperer(img, 64 + len_ts_ascii)
  info, ts_ascii = get_info_stegano(message_ascii)
  info = reduce_info(args, info) # remove padding
  ts_bin = convert_ascii_to_binary(ts_ascii) # timestamp in binary

  write_file(args.path_to_ts_response, 'wb', ts_bin)
  write_file(args.path_to_ts_data, 'w', info)
  write_file(args.path_to_info, 'w', info)

  is_ok = verify_timestamp(args)

  is_ok = is_ok and get_qrcode_png(args)
  sig_ascii = get_data_qrcode(args) # signature in ascii
  sig_bin = convert_ascii_to_binary(sig_ascii) # signature in binary
  write_file(args.path_to_signature, 'wb', sig_bin)
  is_ok = is_ok and verify_signature(args)

  return 'Verification OK!' if is_ok else 'Verification Failure!'
```

Listing 1.11: Steps of verifying attestation

4.1 Verify signature

We verify the signature in QRcode with the public key of Web Server and `identite+intitule` retrieved from stegano message. The command of verification is shown in listing 1.12.

```
def verify_signature(args):
  cmd = subprocess.Popen('openssl dgst -sha256 -verify {} -signature {} {}'
    .format(args.path_to_server_pubkey, args.path_to_signature, args.
    path_to_info),
    shell=True, stdout=subprocess.PIPE)
  (result, ignorer) = cmd.communicate()

  return True if (cmd.returncode == 0 and result.decode() == 'Verified OK\n')
     else False
```

Listing 1.12: Verify signature code

4.2 Verify timestamp

We verify the timestamp retrieved from stegano message with the data `identite+intitule`, the two files `cacert.pem` and `tsa.crt` at https://www.freetsa.org. The command of verification is shown in listing 1.13.

```
def verify_timestamp(args):
  cmd = subprocess.Popen('openssl ts -verify -in {} -data {} -CAfile {} -
    untrusted {}'
    .format(args.path_to_ts_response, args.path_to_ts_data, args.
    path_to_ts_cacert, args.path_to_tsa_crt),
    shell=True, stdout=subprocess.PIPE)
  (result, ignorer) = cmd.communicate()

  return True if cmd.returncode == 0 and result.decode() == 'Verification
    else False
```

Listing 1.13: Verify timestamp code

5 Run an example

In this section, we show an example to create and verify an attestation. Figure 1 displays how we run the system in terminal and the responses of commands. The demo of this example can be found at https://youtu.be/_Zw2cijWcmE.

First of all, we run the bash script `run.sh`. This is to create a CA with keys and certificate, then generate Web Server's keys and certificate, and create some necessary folders as well as download files `cacert.pem` and `tsa.crt` from https://www.freetsa.org.

```
./run.sh
```

Listing 1.14: Create folder and download files

Next, we run Web Server (serveur applicatif) with Python script.

```
python3 web_service.py
```

Listing 1.15: Run Web Server

Then, we run Frontal Server (serveur frontal) with `bundle_server.pem` (generated from private key and certificate of Web Server) and certificate of CA `ecc.ca.cert.pem`.

```
socat openssl-listen:9000,fork,cert=CA/bundle_server.pem,\
    cafile=CA/ecc.ca.cert.pem,verify=0 tcp:127.0.0.1:8080
```

Listing 1.16: Run Frontal Server

From now on, all request sent to Frontal Server must be included the argument `--cacert CA/ecc.ca.cert.pem` for verifying the trust of server. In the following command (listing 1.17), we send the information of student, for example, `identite=Sang Nguyen` and `intitule_certif=TIC PROJECT`.

```
curl -X POST -d 'identite=Sang Nguyen' -d 'intitule_certif=TIC PROJECT' \
    --cacert CA/ecc.ca.cert.pem https://localhost:9000/creation
```

Listing 1.17: Request an attestation

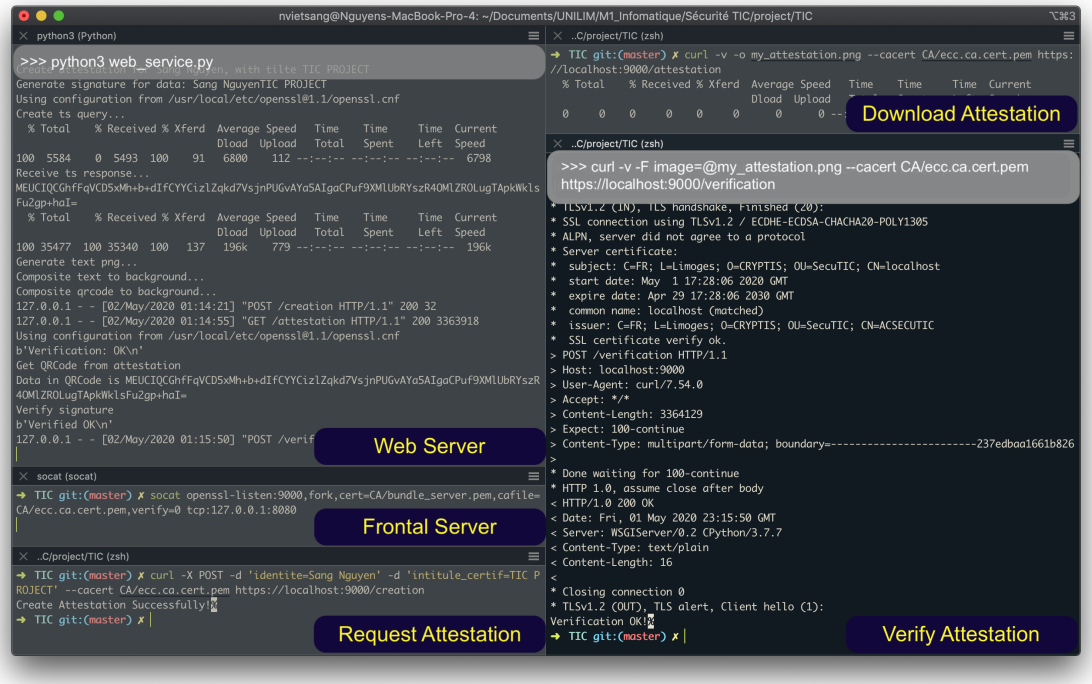

Fig. 1: Example of running system

We then download the attestation and save it as the name `my_attestation.png`

```
curl -v -o my_attestation.png --cacert CA/ecc.ca.cert.pem https://localhost
    :9000/attestation
```

Listing 1.18: Download the attestation

Finally, we verify the attestation which have just been downloaded by the command shown in listing 1.19. If it is successfully verified, we will receive the message `Verification OK!` as shown in figure 1. Otherwise, the received message is `Verification Failure!`.

```
curl -v -F image=@my_attestation.png --cacert CA/ecc.ca.cert.pem \
    https://localhost:9000/verification
```

Listing 1.19: Verify the attestation

Figure 2 shows the result of attestation. As we can see, the information of `identité`, `intitulé` and QRcode are shown in this attestation.

6 Analysis of risks

We could briefly analyse the risks of our system as follows:

- Assets (Biens):
  - Primary assets: Web Server, client computers, private keys of CA and Web Server, people (clients, people who operate servers)
  - Secondary assets: Frontal Server
- Services of security: Frontal Server, protocols of exchanging keys and information, algorithms of generating keys, certificates and signatures
- Threats (Menaces): Attacks of Buffer Overflow, Attacks of Arithmetic Overflow, Attacks of Weak Cryptography, Working time of servers.
- Scenarios:
  - Anyone who can connect to Frontal Server has the right to generate their attestations under the verification of CertiPlus.

Fig. 2: Result of attestation

  - An attacker sends a large number of requests to Frontal Server at the same time which can lead to the death of server.
  - Servers continuously work in many hours which increases their temperatures and leads to fire.
  - Database where stores the private keys of servers and CA can be broken or attacked.
- Recommendations (Préconisations):
  - Clients should be authorized before using the services.
  - Servers should be maintained monthly, for example.
  - A system of recognising different types of attacks (Buffer Overflow, Arithmetic Overflow, etc) should be developed on Frontal Servers.
  - Use a decentralised technique to store private keys
  - etc.

7 Conclusion

In this project, we successfully implement a system called CertiPlus which permits clients to create attestations and verify them. The techniques of signatures and timestamps are utilised in this service. It also uses QRcode and Steganography to hide information. It is so secure from cryptography's point of view. However, we also analyse some possible risks in section 6. It is more ideal if we can avoid these risks.