Implementing Textsecure Protocol

Cryptography CS 411 & CS 507 Term Project for Fall 2020

E. Savaş
Computer Science & Engineering
Sabancı University
İstanbul

Abstract

You are required to develop a simplified version of the TextSecure protocol, which provides forward secrecy and deniability. Working in the project will provide you with insight for a practical cryptographic protocol, a variant of which is used in different applications such as WhatsApp.

1 Introduction

The project has three phases:

- **Phase I** Developing software for the Registration and the Station-to-Station (STS) protocols. All coding development will be in Python programming language.
- **Phase II** Developing software for receiving messages from other clients
- TBA

More information about Phase I is given in the subsequent section.

2 Phase I: Developing software for the Registration and Station-to-Station Protocols

In this phase of the project, you are required to upload one file: "Client.py". You will be provided with "Client_basics.py", which includes all required communication codes.

In the Registration protocol (see below) you will register your long term public key with a server. You will also simulate the STS protocol with the same server. The server is accessible via `cryptlygos.pythonanywhere.com`. You may find the connection details in "Client_basics.py". `JSON` format should be used in all communication steps.

In the Station-to-Station (STS) protocol, each party must have a long term public and private key pair to sign messages and verify signatures. A variant of Elliptic Curve Digital Signature Algorithm (ECDSA) will be used with NIST-256 curve in this project. See Section 2.3 for the description of the variant of the ECDSA algorithm that will be used in the project. You should select "`secp256k1`" for the elliptic curve in your python code. **Note that you will NOT use the ECDSA algorithm in the slides.**

2.1 Registration

The long term public key of the server Q_{SL} is given below.

```
X:0xc1bc6c9063b6985fe4b93be9b8f9d9149c353ae83c34a434ac91c85f61ddd1e9
Y:0x931bd623cf52ee6009ed3f50f6b4f92c564431306d284be7e97af8e443e69a8c
```

In this part, firstly you are required to generate a long-term private and public key pair s_L and Q_L for yourself. The key generation is described in "Key generation" algorithm in Section 2.3. Then, you are required to register with the server. The registration operation consists of four steps:

1. After you generate your long-term key pair, you should sign your ID (e.g. 18007). The details of the signature scheme is given in the signature generation algorithm in Section 2.3. Then, you will send a message, which contains your student ID, the signature tuple and your long-term public key, to the server. The message format is

   ```
   {'ID': stuID, 'H': h, 'S': s, 'LKEY.X': lkey.x, 'LKEY.Y': lkey.y}
   ```

   where `stuID` is your student ID, `h` and `s` are signature tuple and `lkey.x` and `lkey.y` are the x and y and coordinates of your long-term public key, respectively. A sample message is given in 'samples.txt'.

2. If your message is verified by the server successfully, you will receive an e-mail, which includes your ID, your public key and a verification code: `code`.

3. If your public key is correct in the verification e-mail, you will send another message to the server to authenticate yourself. The message format is "`{'ID': stuID, 'CODE': code}`", where `code` is verification code which, you have received in the previous step. A sample message is given below.

   ```
   {'ID': 18007, 'CODE': 209682}
   ```

4. If you send the correct verification code, you will receive an acknowledgement message via e-mail, which states that you are registered with the server successfully.

Once you register with the server successfully, you are not required to perform registration step again as the server stores your long-term public key to identify you. **You need to store your long-term key pair as you will use them until the end of the project.**

2.2 Station-to-Station Protocol

Here, you will develop a python code to implement the STS protocol. For the protocol, you will need the elliptic curve digital signature algorithm described in Section 2.3.

The protocol has seven steps as explained below.

1. You are required to generate an ephemeral key pair s_A and Q_A, which denote private and public keys, respectively. The key generation is described in "Key generation" algorithm in Section 2.3

   Then, you will send a message, which includes your student ID and the ephemeral public key, to the server. The message format is "{'ID': stuID, 'EKEY.X': ekey.x, 'EKEY.Y': ekey.y", where `ekey.x` and `ekey.y` are the x the y coordinates of your ephemeral public key, respectively. A sample message is given in 'samples.txt'.

2. After you send your ephemeral public key, you will receive the ephemeral public key Q_B of the server. The message format is "{'SKEY.X': skey.x, 'SKEY.Y: skey.y}", where `skey.x` and `skey.y` denote the x and y coordinates of Q_B, respectively. A sample message is given in 'samples.txt'.

3. After you receive Q_B, you are required to compute the session key K as follows.

   - $T = s_A Q_B$
   - $U = \{T.x||T.y||\text{"}BeYourselfNoMatterWhatTheySay\text{"}\}$[1], where $T.x$ and $T.y$ denote the x and y coordinates of T, respectively.
   - $K = \text{SHA3\_256}(U)$

   A sample for this step is provided in 'samples.txt'.

4. After you compute K, you should create a message W_1, which includes your and the server's ephemeral public keys, generate a signature Sig_A using s_L for the message W_1. After that, you should encrypt the signature using AES in the Counter Mode (AES-CTR). The required operations are listed below.

   - $W_1 = Q_A.x||Q_A.y||Q_B.x||Q_B.y$, where $Q_A.x$, $Q_A.y$, $Q_B.x$ and $Q_B.y$ are the x and y coordinates of Q_A and Q_B, respectively.
   - $(Sig_A.s, Sig_A.h) = \text{SIGN}_{s_L}(W_1)$
   - $Y_1 = E_K(\text{"}s\text{"}||Sig_A.s||\text{"}h\text{"}||Sig_A.h)$

   Then, you should concatenate the 8-byte nonce NONCE_L and Y_1 and send $\{\text{NONCE}_L||Y_1\}$ to the server. Note that, you should convert the ciphertext from byte array to integer. A sample for this step is provided in 'samples.txt'.

5. If your signature is valid, the server will perform the same operations which are explained in step 4. It creates a message W_2, which includes the server's and your ephemeral public keys, generate a signature Sig_B using s_{SL}, where s_{SL} is the long-term private key of the server. After that, it will encrypt the signature using AES-CTR.

   - $W_2 = Q_B.x||Q_B.y||Q_A.x||Q_A.y$. (**Note that, W_1 and W_2 are different.**)

[1] `https://www.youtube.com/watch?v=d27gTrPPAyk`

- $(Sig_B.s, Sig_B.h) = \text{SIGN}_{s_L}(W_2)$
- $Y_2 = E_K(\text{“}s\text{”}||Sig_B.s||\text{“}h\text{”}||Sig_B.h)$

Finally, it will concatenate the 8-byte nonce NONCE_{SL} to Y_2 and send $\{\text{NONCE}_{SL}||Y_2\}$ to you. After you receive the message, you should decrypt it and verify the signature. The signature verification algorithm is explained in Section 2.3. A sample for this step is provided in 'samples.txt'.

6. Then, the server will send to you another message $E_K(W_3)$ [2] where, $W_3 = \{\text{RAND}||\text{MESSAGE}\}$. Here, RAND and MESSAGE denote an 8-byte random number and a meaningful message, respectively. You need to decrypt the message, and obtain the meaningful message and the random number RAND. A sample for this step is provided in 'samples.txt'.

7. Finally, you will prepare a message $W_4 = \{(\text{RAND}+1)||\text{MESSAGE}\}$ and send $E_K(W_4)$ [3] to the server. Sample messages for this step is given below.

   W_4 : 86987
   $MessagetoServer$: 86987

8. If your message is valid, the server will send a response message as
   $E_K(\text{“}SUCCESSFUL\text{”}||\text{RAND} + 2)$ [2]

2.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

Here, you will develop a Python code that includes functions for signing given any message and verifying the signature. For ECDSA, you will use an algorithm, which consists of three functions as follows:

- **Key generation:** A user picks a random secret key $0 < s_A < n - 1$ and computes the public key $Q_A = s_A P$.
- **Signature generation:** Let m be an arbitrary length message. The signature is computed as follows:
  1. $k \leftarrow \mathbb{Z}_n$, (i.e., k is a random integer in $[1, n-2]$).
  2. $R = k \cdot P$
  3. $r = R.x \pmod n$, where $R.x$ is the x coordinate of R
  4. $h = \text{SHA3\_256}(m + r) \pmod n$
  5. $s = (s_A \cdot h + k) \pmod n$
  6. The signature for m is the tuple (h, s).

[2] The first 8-byte of message, which you receive in this step, is nonce.

[3] All encrypted messages must be generated using unique nonce values. For each encryption, you must use `AES.new()`. Then, you must concatenate nonce and ciphertext.

- **Signature verification:** Let m be a message and the tuple (s, h) is a signature for m. The verification proceeds as follows:
  - $V = sP - hQ_A$
  - $v = V.x \pmod{n}$, where $V.x$ is x coordinate of V
  - $h' = \text{SHA3\_256}(m + v) \pmod{n}$
  - Accept the signature only if $h = h'$
  - Reject it otherwise.

Note that the signature generation and verification of this ECDSA are different from the one discussed in the lecture.

3 Phase II: Developing software for receiving messages from other clients

In this phase of the project, you are required to upload one file: "Client_phase2.py". You will be provided with "Client_basic_phase2.py", which includes all required communication codes. Moreover, you will find sample outputs for this phase in 'sample_vector.txt', which is also provided on SUCourse.

You are required to develop a software for downloading 5 messages from the server, which were uploaded to the server originally by a pseudo-client, which is implemented by us, in this phase[4]. The details are given below.

In Phase I, you have already implemented the registration protocol. The details are explained in Section 2.1. If you have not implemented yet, you must implement the protocol before Phase II and register your long-term public key with the server.

3.1 Registration of ephemeral keys

Before communicating with other clients, you must generate 10 ephemeral public and private key pairs, namely $(s_{A_0}, Q_{A_0}), (s_{A_1}, Q_{A_1}), (s_{A_2}, Q_{A_2}), \ldots, (s_{A_9}, Q_{A_9})$,, where s_{A_i} and Q_{A_i} denote your i^{th} private and public ephemeral keys, respectively. The key generation is described in "Key generation" algorithm in Section 2.3.

Then, you must sign each of your public ephemeral key using your long-term private key. The signatures must be generated for concatenated form of the ephemeral public keys $(Q_{A_i}.x||Q_{A_i}.y)$. Finally, you must sent your ephemeral public keys to the server in the form of

```
{'ID': stuID, 'KEYID':i , 'QAI.X': QAi.x, 'QAI.Y': QAi.y, 'SI': si, 'HI': hi},
```

where `i` is the ID of your ephemeral key. You must start generating your ephemeral keys with IDs from 0 and follow the order. Moreover, you **have to store** your ephemeral private keys with their IDs.

[4]This is indeed an asynchronous messaging application, whereby other users can send you a message even if you are not online. Yes, exactly like WhatsApp application.

3.1.1 Resetting the ephemeral keys

If you forget to store your ephemeral private keys or require to get new messages sent by the pseudo-client, you must sign your ID using your long-term private key and send a message to the server in the form of

{'ID': stuID, 'S': s, 'H': h }.

When the server receives your message, your ephemeral keys will be deleted. After you produce as new set of ephemeral keys and register with the server again, pseudo client will produce a new set of 5 messages for you.

3.2 Receiving messages

As mentioned above, you will download 5 messages from the server. In order to download one message from the server, you must sign your ID using your long-term private key and send a message to the server in the form of

{'ID': stuID, 'S': s, 'H': h }

to download one message from the server as follows

{'IDB': stuIDB, 'KEYID': i, 'MSG': msg , 'QBJ.X': QBj.x ,'QBJ.Y': QBj.y },

where `stuIDB` is the ID of the sender, `i` is the ID of your ephemeral key, which is used to generate session keys, `msg` contains both the encrypted message and its MAC, and `QBj.x` and `QBj.y` are x and y coordinates of the ephemeral public key of the server, respectively.

3.2.1 Session Key and `msg` Generation

As mentioned above, the message that you received includes the ciphertext as well as its MAC, which is concatenated to the end of the ciphertext. In order to create a message in this way, the pseudo-client will compute two session keys K_{AB}^{ENC} and K_{AB}^{MAC} using your and its ephemeral keys which are Q_{A_i} and Q_{B_j}, respectively. Before the computation, the pseudo-client requests your ephemeral key from the server and the server sends your ephemeral key Q_{A_i} with your key ID i. Then, it computes the session keys as follows:

- $T = s_{B_j} Q_{A_i}$, where s_{B_j} is the j^{th} secret ephemeral key of the pseudo-client.
- $U = \{T.x||T.y||\text{“}NoNeedToRunAndHide\text{”}\}$ [5]
- $K_{AB}^{ENC} = \text{SHA3\_256}(U)$
- $K_{AB}^{MAC} = \text{SHA3\_256}(K_{AB}^{ENC})$

After it computes the session keys, it encrypts the message with K_{AB}^{ENC} using AES-CTR[6] and computes HMAC of the ciphertext with K_{AB}^{MAC} [7].

[5] `https://www.youtube.com/watch?v=u1ZoHfJZACA`

[6] For encryption, AES-CTR will be used in this project

[7] For all hash computations, SHA3_256 will be used in this project

3.2.2 Decrypting the messages

After you download a message, which is sent by the pseudo-client, from the server, you must generate session keys firstly. As mentioned above, Q_{B_j} and i are given to you in the message. Therefore, you must compute K_{AB}^{ENC} and K_{AB}^{MAC} as $s_{A_i}Q_{B_j}$ and SHA3_256(K_{AB}^{ENC}), respectively. Then, you must verify the HMAC code and decrypt the message. Finally, you must send the decrypted message with your ID as follows

{'ID': stuID, 'DECMSG': decmsg}.

where `decmsg` is the decrypted message.

4 Appendix I: Timeline & Deliverables & Weight & Policies etc.

Project Phases	**Deliverables**	**Due Date**	**Weight**
Project announcement		18/12/2020	
First Phase	File: Client.py	27/12/2020	30%
Second Phase		03/01/2021	30%
Third Phase		08/01/2021	40%

4.1 Policies

- You may work in groups of two.
- Submit all deliverables in the zip file "cs411_507_tp1_yourname.zip".
- You may be asked to demonstrate a project phase to a TA or the instructor.
- In every phase, we will provide you with a validation software in Python language that can be used to check your implementation for correctness. We will also use it to check your implementation. If your implementation in a project phase fails to pass the validation, you will get no credit for that phase.
- Your codes will be checked for their similarity to other students' codes; and if the similarity score exceeds a certain threshold you will not get any credit for your work.