Larry Huang (lhuan130@asu.edu) [1223797159]
CSE539 Spring 2025 [35406]

Bonus Project – Secure Messaging Demo

NOTE: see page 3 for instructions on the actual demo.

This project serves as a demonstration of many of the security concepts learned in CSE539. As such, each component implemented or planned in this project is described in the following section. The last page contains a screenshot of the project in action.

The Triple Key Exchange

The Triple Diffie-Hellman, capable as a means of end-to-end encryption setup, is used to provide strong anti-eavesdropping within the data exchange necessary for generating further communication keys. The use of an identity key digital signature for the long-term key ensures mutual authentication on the identity key. Furthermore, layering this signature in the middle of a two-part exchange helps prevent the possibility that a man-in-the-middle might try to perform a different exchange with each side.

It is worth noting that once side registers two long-term (one identity and one signed pre) keys and a single-use key, while the other registers only their identity key as long-term and provides an ephemeral key. Furthermore, there are actually FOUR exchanges made with the Elliptic-Curve Diffie-Hellman on each side. The "triple" likely refers to the base version where the one-time key is not required (making only three exchanges as a result), while our version implements four exchanges to incorporate the one-time key.

One interesting problem encountered during writing the full key exchange is that there are two distinct but incompatible key types used in the lecture's description of the triple exchange. The key type for generating a signature, *Ed25519*, is not the same as a *X25519* key, the type that is useful for Diffie-Hellman exchanges. There is not an easy transference between the two (because of being differing curves on the same parameters) for simple handling of two different processes. As such, I chose to use a different sufficiently large elliptic curve type that had capabilities both for the DH exchanges AND signature generation, given our requirements for the identity key. This did not alter any other components of the system significantly, beyond putting all keys used on curve *SECP384R1*.

The next significant security component is the use of Authenticated Encryption with Associated Data; the associated data used to loosely link the two parties is the identity keys' public components. This loose association ensure that it is at least somewhat harder for someone not part of this exchange to access the encrypted data, as in normal circumstances (not in this demo) keys no longer used are deleted from the server (and are only stored if they have not been received by a client).

The AEAD key of this component is generated using *HKDFExpand* with *SHA256*, the input being the Diffie-Hellman exchange data generated from the “triple” key exchange previously. Both parties can generate a series of four DH values from the appropriate public-private pairs; these are fed into the HKDF algorithm and produce a value used in the HKDF scheme. At two stages of the demo, I display the HKDF output to ensure accuracy. The first is at the end of the step where the requesting party performs their side of the handshake; the other when the first-registered party goes back to receive and check the “default first message”. I used AESGCM as our AEAD algorithm to encrypt a default opening message (this is normally application-specific and used to confirm validity); at the receiving stage of this exchange, I then provide a comparison of the decryption output with the input text.

Let’s briefly review the salts and nonces. Default *HKDF* can use an optional salt to provide additional security, but this generally implies that the derivation content is not cryptographically strong and this is assumed to not be the case when the material is the result of DH-exchanges on cryptographically generated EC keys. As such, we use *HKDFExpand* with a longer-than-default hash. Furthermore, with the 50% larger DH-exchange values we get with the key size used, we do have more randomness in the KDF input. On the other hand, *AESGCM* does require a cryptographically random and non-repeating nonce for security, so I prepend the message sent with the nonce (which is generated using python’s *os.urandom* for cryptographic security).

The associated data on each side is stored in a separate file to ensure it can be recollected for use in future exchanges; this is also true of the KDF data. These components will be required for two pieces: the KDF ratchet, and future usage of *AESGCM* (in the context of AEAD). The end state of the demo implementation has not yet implemented an actual KDF ratchet or full messaging exchange system. Only the first message of the demo is sent; however, it does implement correct encryption on the first message using the original Exchange-and-KDF output.

Repository Notes
The GitHub repository contains subfolders used in the code created (each has a README.md inside to ensure population). Repo link: https://github.com/lhuan130/CSE539_Signal_Proto

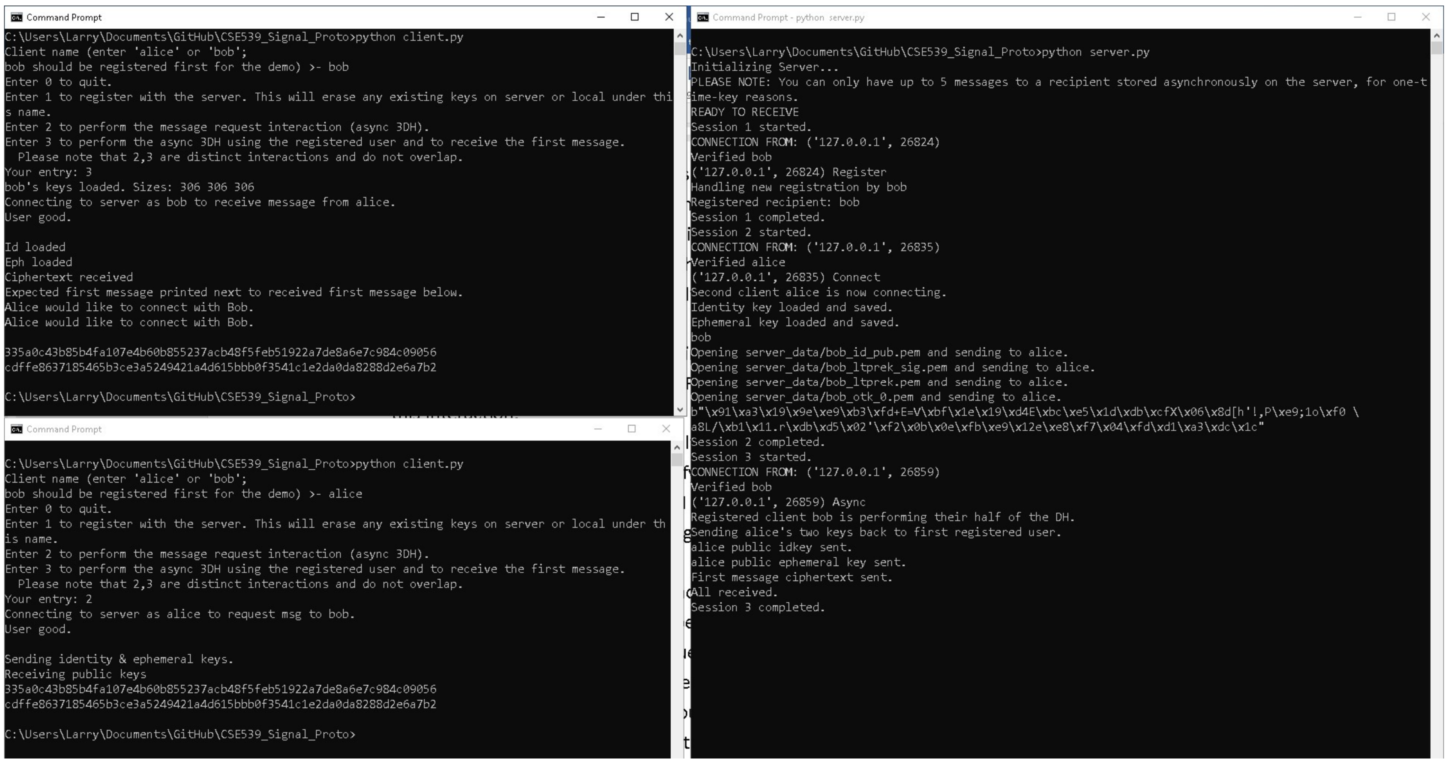
Demo Walkthrough

1. [Implemented] It will likely be most convenient to clone the GitHub repository and use the folder generated as your terminals' starting point. If that is performed, disregard (1a) and (1b).
    a. If this is not desired, create a directory and place all ***.py** into it.
    b. Following (a), create three directories next to the scripts:
        i. alice_data
        ii. bob_data
        iii. server_data
2. [Implemented] Open a terminal window in an environment with the Python cryptography library nicknamed 'hazmat' (https://cryptography.io/en/latest/) installed. Run the Server module with **python server.py**. It will ready before printing a ready to the terminal.
    a. The server will use *localhost:24601* to listen for a connection. If your system does not allow user programs to open ports, the demo will stall here.
3. [Implemented] Once you have "READY TO RECEIVE" on the **server.py** window, run **python client.py** in the same manner within a second terminal window. Within are interaction instructions (for now, type `bob` and hit enter, then `1` and hit enter).
    a. This will load Bob's public keys to the server.
    b. You can do this with both clients, but there is no point because the keys are generated on the client-side runs and will be overwritten if duplicated or not used if not in the protocol.
    c. You *can* use Alice as the registerer and Bob as the requester, but the default opening message might be a bit weird as a result. Try it afterwards if you want.
4. [Implemented] Run another **python client.py** in a third window. Use it to initiate Alice's request with `alice`, enter, `2`, enter. At the end of this interaction, Alice will have a handshake and will have used the derived key to send an interaction check message.
    a. The demo will print the result of KDF(K1,K2,K3,K4) to the console on the last two lines of this interaction.
5. [Implemented] Run client again, this time using `bob` and `3` to do his half of the handshake and to receive the protocol's (default) opening message.
    a. The demo will print the result of KDF(K1,K2,K3,K4) to the console on the last two lines of this interaction. Verify this against step 4.
6. Once both sides start from the shared KDF output, swap back and forth between the two clients and use option `4` to exchange further messages. The ratchet process should automate as each side sends a message.

Here are some concerns regarding the demo's limitations:

1. Notifications are an entirely other beast and are not considered here. To avoid any notification and live connection issues, there is no actually synchronous messaging process. The check for messages is explicit.
2. Try not to run two client-server interactions simultaneously. Changes in server state while the other client is connected may cause an issue not handled in the prototype.
3. You do not technically need a third window for the second client. However, it will likely be easier to differentiate their behavior by doing so.
4. Outstanding issue: killing the server requires going into your tasks/services manager because it fails to unbind from socket without assistance
5. This demo, if run on Windows, may experience WinError 10053. Please note that the socket-send/recv placement and delays within the demo are the result of avoiding this issue, as it seems to appear when a connection tries to send and receive too closely in time.

To reset the demo completely, because files will be created, it is likely easiest to clone again; no files will be (or should be) created outside the repo folder. If there is a lingering OS issue with ports still being held by dead processes due to failure to exit, restarting your computer is the most extreme solution but will absolutely fix this.



```
C:\Users\Larry\Documents\GitHub\CSE539_Signal_Proto>python client.py
Client name (enter 'alice' or 'bob';
bob should be registered first for the demo) >- bob
Enter 0 to quit.
Enter 1 to register with the server. This will erase any existing keys on server or local under thi
s name.
Enter 2 to perform the message request interaction (async 3DH).
Enter 3 to perform the async 3DH using the registered user and to receive the first message.
  Please note that 2,3 are distinct interactions and do not overlap.
Your entry: 3
bob's keys loaded. Sizes: 306 306 306
Connecting to server as bob to receive message from alice.
User good.

Id loaded
Eph loaded
Ciphertext received
Expected first message printed next to received first message below.
Alice would like to connect with Bob.
Alice would like to connect with Bob.

335a0c43b85b4fa107e4b60b855237acb48f5feb51922a7de8a6e7c984c09056
cdffe8637185465b3ce3a5249421a4d615bbb0f3541c1e2da0da8288d2e6a7b2

C:\Users\Larry\Documents\GitHub\CSE539_Signal_Proto>
```

```
C:\Users\Larry\Documents\GitHub\CSE539_Signal_Proto>python client.py
Client name (enter 'alice' or 'bob';
bob should be registered first for the demo) >- alice
Enter 0 to quit.
Enter 1 to register with the server. This will erase any existing keys on server or local under th
is name.
Enter 2 to perform the message request interaction (async 3DH).
Enter 3 to perform the async 3DH using the registered user and to receive the first message.
  Please note that 2,3 are distinct interactions and do not overlap.
Your entry: 2
Connecting to server as alice to request msg to bob.
User good.

Sending identity & ephemeral keys.
Receiving public keys
335a0c43b85b4fa107e4b60b855237acb48f5feb51922a7de8a6e7c984c09056
cdffe8637185465b3ce3a5249421a4d615bbb0f3541c1e2da0da8288d2e6a7b2

C:\Users\Larry\Documents\GitHub\CSE539_Signal_Proto>
```

```
C:\Users\Larry\Documents\GitHub\CSE539_Signal_Proto>python server.py
Initializing Server...
PLEASE NOTE: You can only have up to 5 messages to a recipient stored asynchronously on the server, for one-t
ime-key reasons.
READY TO RECEIVE
Session 1 started.
CONNECTION FROM: ('127.0.0.1', 26824)
Verified bob
('127.0.0.1', 26824) Register
Handling new registration by bob
Registered recipient: bob
Session 1 completed.
Session 2 started.
CONNECTION FROM: ('127.0.0.1', 26835)
Verified alice
('127.0.0.1', 26835) Connect
Second client alice is now connecting.
Identity key loaded and saved.
Ephemeral key loaded and saved.
bob
Opening server_data/bob_id_pub.pem and sending to alice.
Opening server_data/bob_ltprek_sig.pem and sending to alice.
Opening server_data/bob_ltprek.pem and sending to alice.
Opening server_data/bob_otk_0.pem and sending to alice.
b"\x91\xa3\x19\x9e\xe9\xb3\xfd+E=V\xbf\x1e\x19\xd4E\xbc\xe5\x1d\xdb\xcfX\x06\x8d[h'!,P\xe9;1o\xf0 \
a8L/\xb1\x11.r\xdb\xd5\x02'\xf2\x0b\x0e\xfb\xe9\x12e\xe8\xf7\x04\xfd\xd1\xa3\xdc\x1c"
Session 2 completed.
Session 3 started.
CONNECTION FROM: ('127.0.0.1', 26859)
Verified bob
('127.0.0.1', 26859) Async
Registered client bob is performing their half of the DH.
Sending alice's two keys back to first registered user.
alice public idkey sent.
alice public ephemeral key sent.
First message ciphertext sent.
All received.
Session 3 completed.
```