Programming Assignment 2 Winter 2020

CS 255: Intro to Cryptography

Prof. Dan Boneh Due **Monday, March 2nd, 11:59pm**

1 Introduction

In this assignment, you are tasked with implementing a secure and efficient end-to-end encrypted chat client using the Double Ratchet Algorithm, a popular session setup protocol that powers real-world chat systems such as Signal and WhatsApp. As an additional challenge, assume you live in a country with government surveillance. Thereby, all messages sent are required to include the session key encrypted with a fixed public key issued by the government. In your implementation, you will make use of various cryptographic primitives we have discussed in class—notably, key exchange, public key encryption, digital signatures, and authenticated encryption. Because it is ill-advised to implement your own primitives in cryptography, you should use an established library: in this case, the Stanford Javascript Crypto Library (SJCL). We will provide starter code that contains a basic template, which you will be able to fill in to satisfy the functionality and security properties described below.

2 End-to-end Encrypted Chat Client

2.1 Implementation Details

Your chat client will use the Double Ratchet Algorithm to provide end-to-end encrypted communications with other clients. To evaluate your messaging client, we will check that two or more instances of your implementation it can communicate with each other properly.

We feel that it is best to understand the Double Ratchet Algorithm straight from the source, so we ask that you read Sections 1, 2, and 3 of Signal's published specification here: `https://signal.org/docs/specifications/doubleratchet/`. **Your implementation must correctly use the Double Ratchet Algorithm as described in Section 3 of the specification, with the following changes and clarifications:**

- You may use HKDF to ratchet the Diffie-Hellman keys the as described in Section 2.3 of the Signal Specification. Proper usage of HKDF is explained in Section 5.2 of the Signal Specification.
- HKDF is a key derivation function that we've added to lib.js. It is a fast key derivation algorithm that allows the user to specify the desired output key length. Section 5.2 describes how it can be used in your implementation. Read the lib.js comments for how to use our API.
- You may use a variant of HMAC to implement the symmetric key ratchet described in 2.2.

- Use ElGamal key pairs for the Diffie-Hellman key exchange.
- Use AES-GCM as the symmetric encryption algorithm for encrypting messages, using the sending and receiving keys as derived in Section 2.4.
- Disregard the AD byte sequence input for the `ratchetEncrypt` and `ratchetDecrypt` functions in the Signal Specification. Message headers should still be be authenticated.
- The header of all sent messages must include an encryption of the sending key under the government's public key. Use ElGamal public key encryption, with AES-GCM as the symmetric cipher, to encrypt the sending keys. (**Note**: Since we have only provided a 128 bit version of AES-GCM, use the first 128 bits of the Diffie-Hellman output as the AES-GCM key.)
- Every client will a possess an initial ElGamal key pair. These key changes will be used to derive initial root keys for new communication sessions.
- Public keys will be distributed through simple certificates. Each client generates its own certificate upon initialization which contains its ElGamal public key. Assume that there is some trusted central party (e.g. server managed by developers of messaging app), and that this central party can securely receive certificates generated by clients. This central party generates a digital signature on each certificate that it obtains, which serves to endorse the authenticity of the certificate owner's identity and to prevent any tampering of the certificate by an adversary. The signed certificates are then distributed back to the clients, so that every client has the ElGamal public key of every other client in the system.
- We've modified the AES-GCM function in lib.js so that it instantiates a new cipher every time it is called. For this reason, your implementation **must never** call the function with the same key twice. The new API accepts a key (as a hex string) instead of a cipher instance.
- You do not need to handle and recover from dropped or out-of-order messages. You do not need to worry about Section 2.6 of the specification. If you detect that a message has been blocked, dropped, or received out of order, throw an exception.

2.2 Threat model

The goal of the Double-Ratchet algorithm is to provide **Forward Secrecy**: compromise of long term keys or current session key must not compromise past communications.

Specifically, consider a Man-in-the-Middle attacker Eve who sits between Alice and Bob. Eve sees every encrypted message passed between Alice and Bob and writes all of them to persistent storage. Then at some point, Alice's device is compromised and Eve learns Alice's current secret keys. (Assume that Alice has deleted her keys for old messages, as encouraged by the Signal Specification.) Your implementation must ensure that under this scenario, the attacker cannot decrypt **any** of the past messages in her persistent storage despite having full access to Alice's current keys.

After Alice's keys are compromised, the adversary can now launch an active Man-in-the-Middle attack. She impersonates both parties and is able to decrypt all communications between Alice and Bob. However, at some later point, Alice manages to send a single message to Bob without the attacker being able to intercept it. Under this scenario, your implementation must ensure that the

attacker loses all ability to decrypt communications once again. This property is called **Break-in Recovery**.

Implementing the Double-Ratchet algorithm as defined in the Signal documentation is sufficient to ensure these two properties.

Adding the sending key encrypted under the government's public key should not compromise the security properties of the chat client. The government should be able to use its secret key to decrypt any message; however no one else, other than the intended recipient, should be able to learn anything about the message contents.

3 API description

Here are descriptions of the functions you will need to implement.

3.1 `messenger.generateCertificate(username)`

This method should initialize the messenger client for communications with other clients. Generate the necessary ElGamal key pair for key exchanges. Public keys are are placed into a certificate to send to other clients. You are free to design your own certificate object, so long as it has a field called "username".

3.2 `messenger.receiveCertificate(certificate, signature)`

This method takes a certificate from another client and stores it in the messengers internal state, so that the client can now send and receive messages from the owner of that certificate. The second argument is the trusted central party's signature of the certificate. You must verify the validity of the signature (using the trusted central party's public key, provided to you in the messenger class constructor) to ensure that the certificate has not been modified by an adversary. If you detect tampering, immediately throw an exception to end program execution.

3.3 `messenger.sendMessage(name, message)`

Send an encrypted message to the user specified by name. You can assume you already possess their certificate through `messenger.receiveCertificate`, and that they already possess your certificate. If you have not previously communicated, setup the session by generating the necessary double ratchet keys according to the Signal spec. On every send, increment the sending chain (and the root chain if necessary, according to the Signal spec). Create a header including the data necessary for other party to derive the new key, in addition to the new sending key encrypted with the government's public key. The header must include the fields "vGov" and "cGov" which denote the outputs (v, c) of the ElGamal public key encryption. With the new sending key, encrypt the message with the header passed as authenticated data. *Every message must be encrypted with a new sending key.*

3.4 `messenger.receiveMessage(name, [header, ciphertext])`

Receive an encrypted message from the user specified by name. You can assume you already possess their certificate through `messenger.receiveCertificate`, and that they computed the initial root key using the ElGamal key from your certificate. If you have not previously communicated, setup the session by generating necessary double ratchet keys according to the Signal spec. On every receive, increment the receiving chain (and the root chain if necessary, according to the Signal spec) using the information provided in the header, and decrypt with a new receiving key. If tampering is detected in any way, throw and exception to terminate the program (i.e. the adversary has tampered with your ciphertext).

4 Setup Instructions

The setup process is essentially the same as with Project 1, as this project is also based in Node.

Extract the starter code and `cd` into the directory `proj2`. You will need to run `npm install` – this will install the dependencies specified in the `package.json` file locally, into a new directory named `node_modules` under `proj2`. You should now be all set. We have provided a simple test suite, which you can run using the command `npm test` from this directory.

Note that this test suite does not cover all of the properties we will test, and in particular, does not capture many of the security requirements. We will run a more exhaustive set of tests when grading. The autograder will be running Node version 10.

5 Hints and Summary

- All the code you will have to write will be in the file `messenger.js`. Please do **not** write any code in another file.
- Your messaging system will depend on the Stanford Javascript Crypto Library (SJCL) for its underlying crypto implementation. However, you **should not need** to call the SJCL functions directly (and our starter code does not include it directly). We have provided a support code library, `lib.js`, which provides wrappers for any SJCL functions that you should need.
- **We have improved and simplified `lib.js` so that all keys, hash values, ciphertexts, and signatures are represented as hex-encoded strings for your convenience. There should be *no* need to perform any type conversions to bitarrays or other convoluted data types that you may have had to handle in Project 1. You can use the function "hexStringSlice" from `lib.js` in order to retrieve segments of ciphertext strings.**
- You can have a look at the tests being run in the file `test/test-messenger.js`. You are always welcome to write more tests to make sure your implementation satisfies the requirements, but you are not required to, and we will not be grading your tests. The tests are written using the MochaJS framework (https://mochajs.org/) with Chai for assertions (http://chaijs.com/), and should be fairly readable.

- If your application detects tampering with any of its in-transit data (e.g. ciphertexts, signatures, etc.), it should throw an exception (thereby terminating the execution). We will not test what exception is thrown; it is fine to throw a string with an English description of the potential tampering.

6 Extra Credit

We will award 10% extra credit for successfully handling messages that are dropped, delayed or delivered out-of-order. How this can be achieved is described in section 2.6 of the Signal Double-Ratchet specification, and the full algorithm as described in section 3 of the spec includes handling such messages.

Suppose two messages A and B are delivered out-of-order i.e. B arrives before A. A successful implementation will be able decrypt B as soon as it arrives without having to wait for A, yet will still be able decrypt A when it does finally arrive. For full extra-credit, you should be able to decrypt A even if an arbitrary number of messages arrive in between B and A.

We strongly recommend you first implement the project without attempting the extra credit portion.

7 Short-answer Questions

In addition to your implementation, please include answers to the following questions regarding your implementation. Your answers need not be long, but should include important details.

Please submit typed or handwritten answers to the "Project #2 Short-answer Questions" assignment on Gradescope (separate from programming component).

1. In our implementation, Alice and Bob increment their Diffie-Hellman ratchets every time they exchange messages. Could the protocol be modified to have them increment the DH ratchets once every ten messages without compromising confidentiality against an eavesdropper (i.e., semantic security)?

2. What if they never update their DH keys at all? Please explain the security consequences of this change with regards to Forward Secrecy and Break-in Recovery.

3. Consider the following conversation between Alice and Bob, protected via the Double Ratchet Algorithm according to the spec:

   ```
   A: Hey Bob, can you send me the locker combo?
   A: I need to get my laptop
   B: Sure, it's 1234!
   A: Great, thanks! I used it and deleted the previous message.
   B: Did it work?
   ```

   What is the length of the longest sending chain used by Alice? By Bob? Please explain.

4. Unfortunately, in the situation above, Mallory has been monitoring their communications and finally managed to compromise Alice's phone and steal all her keys just before she sent her third message. Mallory will be unable to determine the locker combination. State and describe the relevant security property and justify why double ratchet provides this property.

5. The method of government surveillance is deeply flawed. Why might it not be as effective as intended? What are the major risks involved with this method?