CSCI-UA.0480-63: Intro to Computer Security Spring 2023

Project 1: Chatterbox

due: Thursday 2023-04-06 22:00 EDT via classes.nyu.edu

In this assignment you'll implement the cryptographic logic for a secure messaging client. The security goals are integrity and confidentiality of all messages transmitted, deniable session authentication, forward secrecy at the granularity of individual messages, and recovery from compromise. To make the protocol usable you'll also need to handle messages which are corrupted or delivered out-of-order by the network.

We'll build this up in a few stages. You'll also be given some simple libraries to handle many low-level cryptographic details for you as well as testing code to guide your development. The total amount of code you'll need to write is only a few hundred lines. However almost every line will take significant thinking, as is typically the case for cryptographic code.

Comparison to Signal

The end result will be similar to the protocol pioneered by Signal and since adopted by others like WhatsApp and Skype, currently under standardization as MLS. If you're curious, you can read Signal's protocol documentation, though it is not necessary to complete this assignment. Note that the protocol you'll develop won't be exactly like Signal:

- Some complexities such as prekeys have been removed for simplicity.
- Signal uses the curve25519 elliptic curve and AES-CBC encryption with HMAC. We'll use NIST's P-256 curve and AES-GCM authenticated encryption (largely because these are available in Go's standard libraries). You shouldn't need to touch this code directly.
- The data being sent with each message will be slightly different.
- A slightly different key ratchet.

There are many open-source implementations of Signal available. You can refer to these if you find it helpful, but keep in mind that it is an honor code violation to copy another's code. In any case, the effort required in porting an existing implementation to the test framework for this assignment will likely be far higher than doing the assignment yourself.

Starter code

You can download the starter code from Github:

https://github.com/jcb82/chatterbox

If bugs are found, patches may be pushed to the repository and announced on Piazza.

HONOR CODE WARNING: *Do not fork the starter code into a public github repo where other students can easily find your code.*

Programming environment

The supported language for this assignment is Go, an increasingly popular choice for cryptographic code as it provides good performance while avoiding many types of memory corruption problems. If you haven't programmed in Go before, don't worry. It is quite similar to the imperative style in languages like C++, Java or Javascript. There are many resources to help you get up and running with Go for the first time:

- If you haven't used Go, start with the tour here: https://tour.golang.org/welcome/1
- Go by example: https://gobyexample.com/
- Installation: https://golang.org/doc/install
- IDES: https://golang.org/doc/editors.html
- Detailed language specification https://golang.org/ref/spec (you will not need to know Go to this level of detail)

You only need to edit one file: `chatter.go`. You may of course split your code into multiple files if you like. You may want to add additional tests to the provided test code. You should not edit the cryptographic libraries too much, your code will be run with the starter versions for grading.

The most important commands you'll want (if you're working on the command line) are:

- `go mod init chatterbox`: you may need to run this once to create a local project
- `go test`: run all test cases
  - `go test -v` prints more detailed output. There is also a `VERBOSE` constant in the `chat_test.go` file.
  - `go test -short` skips the longer extended test cases
- `go fmt *.go`: automatically format your code according to the language standard.

Part 1: The Triple Diffie-Hellman handshake (3DH)

A chat between Alice and Bob starts with a *handshake* where they establish a shared session key. First, Alice and Bob must learn each other's public keys. In the real world this is a significant challenge. For this assignment we'll assume they get them from a trusted key server.

The classic approach is to do a Diffie-Hellman (hereafter DH) exchange to establish a session key, with Alice and Bob using their private keys to sign their DH values g^a, g^b. These DH shares are called *ephemeral* since they last for one session only, compared to the *long-term* or *identity* public keys which identify Alice and Bob permanently.

Signal does a more interesting handshake to achieve deniability. No signatures are involved. Instead, three DH exchanges are combined to authenticate both parties and produce a fresh shared secret. Alice starts with identity key g^A and ephemeral key g^a (her secrets are A and a). Similarly Bob has identity key g^B and ephemeral key g^b (his secrets are B and b). Alice sends Bob g^A and g^a and he sends back g^B and g^b. Their initial shared secret is:

$$k_{root1} = \text{KDF}(g^{A \cdot b}, g^{a \cdot B}, g^{a \cdot b})$$

Alice is convinced she's talking to Bob if he can derive the same k_{root1}, because this requires knowing his long-term private key *B*. Similarly Bob is convinced he's talking to Alice. But it's also possible for anybody to *simulate* this handshake without the involvement of either party at all by choosing *a* and *b*, so either party can deny they ever participated in the conversation.

Order matters! Note that Alice and Bob will get different results if they don't agree on the ordering of the shares $g^{A \cdot b}$, $g^{a \cdot B}$, $g^{a \cdot b}$ when they compute the KDF. We'll use the following simple convention: whoever sends the first message of the handshake (the *initiator*) is "Alice" in the above formula and whoever sends the second (the *responder*) is "Bob." Both parties will sort the three shares according to their role in the protocol.

Implementation notes: The handshake requires that two messages are exchanged, which are implemented as three methods for a `Chatter` object:

- `InitiateHandshake()`: Set up Alice's state for a session with Bob and return Alice's ephemeral public key.
- `ReturnHandshake()`: Receive Alice's ephemeral key and set up Bob's state for his session with her. Return Bob's ephemeral public key. This function also derives the initial root key and returns a key derived from it (for authentication checks).
- `FinalizeHandshake()`: Alice receives Bob's ephemeral key. She derives the initial root key and returns a hash of it (for authentication checks).

To compute a root key, both sides will call `CombineKeys()` with the outputs $g^{A \cdot b}$, $g^{a \cdot B}$, $g^{a \cdot b}$ in order. Note that `CombineKeys()` is a *variadic* function which can take any number of arguments.

Checking the handshake: Both Alice and Bob return a special check key derived from the root key. This won't be used for any encryption, but can be used by both parties to verify that the handshake was successful (for example, by displaying on screen for an in-person verification). In your implementation, use the `DeriveKey` method on the root key with the label `HANDSHAKE_CHECK_LABEL`. The testing code will require that you derive the check key this way.

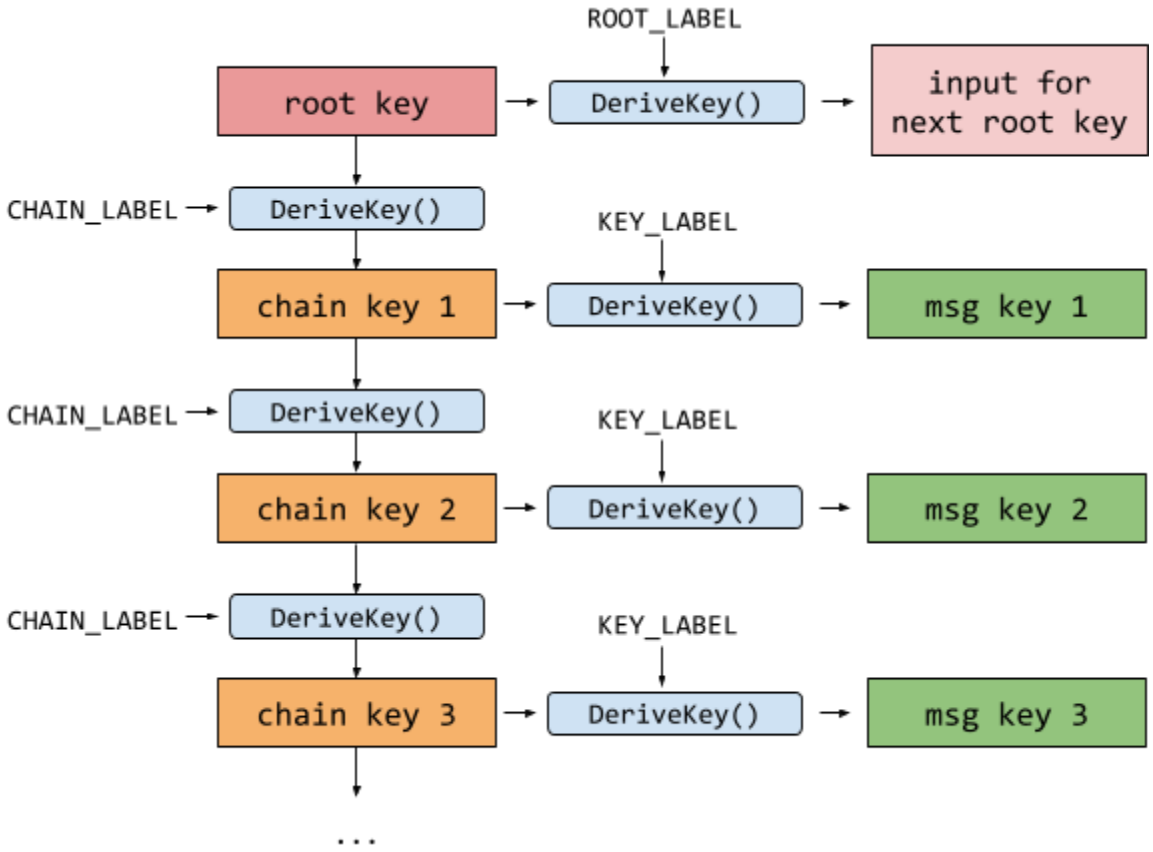
Testing: When you've implemented the handshake correctly, your code should pass the `TestHandshake` test and `TestHandshakeVector` tests. The second of these tests contains a precise expected value based on a fixed random number generator. Until you pass the basic handshake test the remaining tests will be skipped.

Part 2: Deriving forward-secure message keys with a double ratchet

After their handshake, Alice and Bob are ready to chat. They chat through the `SendMessage` and `ReceiveMessage` methods, which is where the bulk of the code you need to write will go (you may of course want to split it off into some helper functions).

From the root key, Alice and Bob need to derive symmetric keys for authenticated encryption. They'll derive these from the root key using the `DeriveKey()` method with the label `CHAIN_LABEL`. To achieve forward secrecy, after every message the sender and receiver *ratchet* the chain key (using the `DeriveKey()` KDF with `CHAIN_LABEL`) to derive a new key. They must also delete the old root key value to ensure that it can't later leak and allow an adversary to decrypt an intercepted ciphertext.

A simple ratchet wouldn't support receiving out-of-order messages though: the old value would need to be kept around if a particular message wasn't received on time, and that could be used to derive all future keys in the chain. So Signal introduced the *double ratchet*:



From each chain key value, a message key is derived by again calling `DeriveKey`. Each message key is used only once: message key 1 is used to send/decrypt message 1 and is then deleted. The advantage of the double ratchet is that, if needed, an old message key can be cached to decrypt an out-of-order message, but keeping this value around does not allow deriving any other message keys in the chain.

These keys should be used to encrypt each message using the provided AES-GCM `AuthenticatedEncrypt()` function. You'll want to create a random initialization vector for each message. In this application, each key is only used once so it would be okay to use a fixed IV, but it is good practice to generate a fresh IV for each message.

Testing: When you've implemented the symmetric ratchet correctly, your code should pass the `TestOneWayChat` test, a simple conversation in which only one party (Alice) sends messages.

Part 3: Adding resiliency with a Diffie-Hellman ratchet

The symmetric double ratchet enables good forward secrecy, as key material can be deleted quickly after it's used. However, if this was the only ratcheting, the protocol would not be resilient to a temporary compromise. If an attacker learns any of the chain key values, they could compute all of the following values indefinitely.

To address this, Signal adds another layer of ratcheting. The root key is continuously updated by new Diffie-Hellman computations. In fact, before Bob (the responder) ever sends a message, he generates a new secret *b2* and ephemeral DH key g^{b2}. He then computes the DH value $g^{a1 \cdot b2}$ (where Alice's initial ephemeral DH value is g^{a1}) and uses this to update his root key. In your implementation, Bob will first ratchet the root key once (with `ROOT_LABEL`) and then call `CombineKeys()` with the new key derived from $g^{a1 \cdot b2}$ (in that order). He'll then derive a new sending key chain as before and use it to encrypt the first message he sends.

For Alice to be able to decrypt, Bob will need to send his new DH value g^{b2} (a *DH ratchet key*) along with his encrypted message. Alice can then derive the same value $g^{a1 \cdot b2}$ and update her copy of the root key to derive Bob's current sending key chain. She can then use this to decrypt Bob's message. As long as Bob is the only one sending messages, she'll keep updating her receiving chain using the symmetric ratchet (the double ratchet implemented above). Alice should also make sure to zeroize her secret *a1* at this point, since she'll no longer need it.

When Alice has a message to send back, it's her turn to:

1. Pick a new DH ratchet key g^{a2}
2. Update her root key by combining with $g^{a2 \cdot b2}$
3. Derive a new sending key chain. Use this to encrypt her message and send it to Bob (along with g^{a2}) so he can update his root key in the same way and decrypt.
4. Ratchet the root key (using `ROOT_LABEL`) to avoid keeping the old one around

All this work to keep Eve out of the conversation! In general, the DH ratcheting proceeds in turns. At first, it's Bob's turn to send a new DH ratchet key and update his sending key chain. He'll then use these keys for all messages he sends until it's his turn again. Note that this process ensures that Alice and Bob are never using the same chain of keys to send messages as each other. The sequence of root keys and derived chains will go like this:

Root key version	Derivation	Sender who uses this chain
k_{root1}	KDF(Triple DH output)	Alice

k_{root2}	Combine(Ratchet(k_{root1}), $g^{a1 \cdot b2}$)	Bob
k_{root3}	Combine(Ratchet(k_{root2}), $g^{a2 \cdot b2}$)	Alice
k_{root4}	Combine(Ratchet(k_{root3}), $g^{a2 \cdot b3}$)	Bob
...	...	...

The two sides must agree on who updates first. By convention we'll say that Bob (the responder) updates first (before sending his first message) and sends a new DH value. If Alice (the initiator) sends the first message, she'll use the sending chain derived directly from the first root key derived from the initial handshake.

Testing: When you've implemented the DH ratchet logic correctly, your code should pass:

- `TestAlternatingChat` (a simple case where both sides send one message at a time)
- `TestSynchronousChat` (both sides may send more than one message at a time)
- `TestSynchronousChatExtended` (a longer, parameterized version with multiple senders randomly sending messages). This is a stress test to identify bugs. You can increase the parameters `EXTENDED_TEST_ROUNDS` and `EXTENDED_TEST_PARTICIPANTS` to make this test more thorough. Your code should handle chat indefinitely without errors.

Part 4: Handling out-of-order messages

So far we've assumed every message is delivered as soon as it is sent. With real networks, messages can be delivered out-of-order. Each message has a `counter` value which the sender uses to signify which order the message was sent. Handling out-of-order messages can be split into two cases:

Handling "early" messages: When an out-of-order message is received with a higher counter value than the receiver has yet seen, they will need to advance their receiving key chain as needed to derive the key needed to decrypt this message. They'll also need to store any intermediate message keys from the chain for messages not yet received in the `Session` member variable `CachedReceiveKeys` (a Go `map`), indexed by sequence number.

Note that sometimes the out-of-order message will also include a new DH ratchet value, requiring the receiving chain to be updated. Only, at what point did the sender update? Consider the following scenario.

1. Alice sends messages 1-3 using her initial sending chain, but Bob doesn't receive them.
2. Alice receives a message from Bob with a new DH ratchet value, making it her turn to pick a new DH ratchet key. She does so and updates her sending chain.
3. Alice sends messages 4-6 using her new sending chain, but Bob doesn't receive them.
4. Finally, Bob receives message 6.

At this point Bob will need to derive and store the keys needed for messages 1-3 using Alice's old sending chain, then derive Alice's new sending chain using her new DH ratchet key, then derive and store the keys needed for messages 4-5, then finally decrypt message 6. To help Bob do this, each message from Alice contains a `lastUpdate` value in addition to a sequence number, indicating which message number was the first sent with the newly updated sending chain (4 in the above example).

Handling "late" messages: Assuming the logic for the above is implemented, handling a late message is easy. Just look up the stored value of the key needed, use it and zeroize it.

Testing: When you've implemented out-of-order message handling, your code should pass:

- `TestAsynchronousChat` test (a simple case of 8 messages)
- `TestAsynchronousChatExtended` (a longer, parameterized version with multiple senders randomly sending messages). You should set `EXTENDED_TEST_ERROR_RATE` to zero until you implement Part 5.You can again up the `EXTENDED_TEST_ROUNDS` and `EXTENDED_TEST_PARTICIPANTS` to make this test more thorough.

Part 5: Handling errors

What happens if a message is received that has been tampered with in transit? The simple answer is nothing: the receiver should reject it and raise an error. The authenticated encryption library should raise an error if the ciphertext has been modified at all. Note that several pieces of important data cannot be encrypted though, since the receiver needs them to figure out which keys to decrypt with! This includes:

- The sender and receiver's identity key fingerprints
- The sender's DH ratchet value
- The sequence number and last update number

All of these values should be added to the additional data portion of the authenticated encryption. Remember that AEAD (authenticated encryption with additional data) is useful to handle data like this which cannot be encrypted but you want to verify the integrity of. A function `EncodeAdditionalData()` has been provided for you which will encode this data to binary.

Avoiding corrupted state: You'll need to make sure not to corrupt your state if a tampered message is received. If you update your root key only to realize later that there's an error, you'll be in trouble if you didn't store the old value to revert back to. You'll need to think carefully about error handling and only update state after confirming the message's integrity.

Testing: When you've implemented error handling correctly, your code should pass the small `TestErrorRecovery` test as well as the `TestAsynchronousChatExtended` test with `EXTENDED_TEST_ERROR_RATE` set to a non-zero value, as well `TestSynchronousChatVector` (a precise expected value using a fixed RNG).

Part 6: Zeroizing key material

It's critical for cryptographic implementations to erase keys when they're no longer needed. It's not enough to just delete your reference to the key and wait for the garbage collector to overwrite this value, you need to actively call the provided `Zeroize()` method on every key as soon as it is no longer needed to overwrite the bits in RAM.[1] This applies to both symmetric keys/chain values and DH private keys. Messages should only be decrypted once, after which their key should be zeroized. If a message is retransmitted with the same counter value, the second attempt should produce an error.

Beware of course, that zeroizing keys too early could prevent you from decrypting legitimate messages, so you'll need to think carefully about when to zeroize. You'll also need to be sure not to make unintended copies of keys in memory. If you pass them by value (instead of by reference) a copy will be made that also needs to be zeroized. It's safer to pass by reference and try to only have one copy that needs to be zeroized.

There is also an `EndSession` method to implement, which should completely delete all remaining key material shared with a designated partner. After calling `EndSession`, it should be possible to open another session with that partner by running the handshake again.

Tip: The Go built-in `defer` schedules a task to be executed when a function exits (no matter what code path is taken). This can be useful to ensure a key is zeroized. Note that unlike freeing allocated memory, it's not a problem if you zeroize the same key more than once.

Testing: There is a `TestTeardown` test which checks that you can call `EndSession` and then perform a second handshake. However, no test code is provided to actually check that you are deleting all key material (not just in `EndSession`, but in `ReceiveMessage` every time a message is received). You'll need to audit your own code. We will evaluate in the grading process that you have correctly zeroized every key no longer needed.

Grading criteria

You're advised to work in the order presented here. Credit will be alloted as following:

Part 1	Part 2	Part 3	Part 4	Part 5	Part 6
15	15	20	20	15	15

Please remove all printout statements from your code prior to submission.

[1] Actually ensuring that every copy where the key might have been stored is destroyed is very challenging due to issues like virtual memory. For this assignment we'll just overwrite the bits.