CSCI-UA.0480-11: Intro to Computer Security Spring 2019

Project 1: Chatterbox

due: Tuesday 2019-03-26 22:00 EST via classes.nyu.edu

In this assignment you'll implement the cryptographic logic for a secure messaging client. The security goals are to achieve integrity and confidentiality of all messages transmitted, deniable session authentication, forward secrecy at the level of individual messages, and recovery from compromise. To make your protocol usable you'll also want to recover gracefully from errors and handle messages delivered out-of-order.

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 about 100-200 lines. Unfortunately, almost every line will take significant thinking, as is typically the case for cryptographic code.

Comparison to Signal

The end result will quite similar to the protocol pioneered by Signal and since adopted by WhatsApp, Skype, and many other tools which support encrypted communication. 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:

- Many 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, it is 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 patches are needed to fix bugs, they'll be uploaded to the repository.

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 panic. 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:

- Read first if you haven't used Go: A Tour of Go 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 add additional tests or split your code into multiple files if you like. 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 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`: a cool feature of Go: automatically format your code according to the language standard.

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

A chat between Alice and Bob starts with a *handshake* where they exchange cryptographic key material and 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 just assume they get them from some trusted source like a key server.

A 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 session. 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 send Bob g^A and g^a and he sends back g^B and g^b. Their initial shared secret is:

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

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 both parties need to agree on an ordering of the shares $g^{A \cdot b}$, $g^{a \cdot B}$, $g^{a \cdot b}$ when they compute the KDF, or they will get different results. We'll use the following simple convention: whoever sends the first message of the handshake (the *initiator*) is "Alice" 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()`: Alice sets up state for her session with Bob and returns an ephemeral public key.
- `ReturnHandshake()`: Bob receives Alice's ephemeral key. He sets up state for his session with her and returns his own ephemeral public key. He 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. In your implementation, use the `DeriveKey` method on the root key with the label `HANDSHAKE_CHECK_LABEL`. The testing code will assume you derive the returned key this way (and that you combine keys in the order listed above).

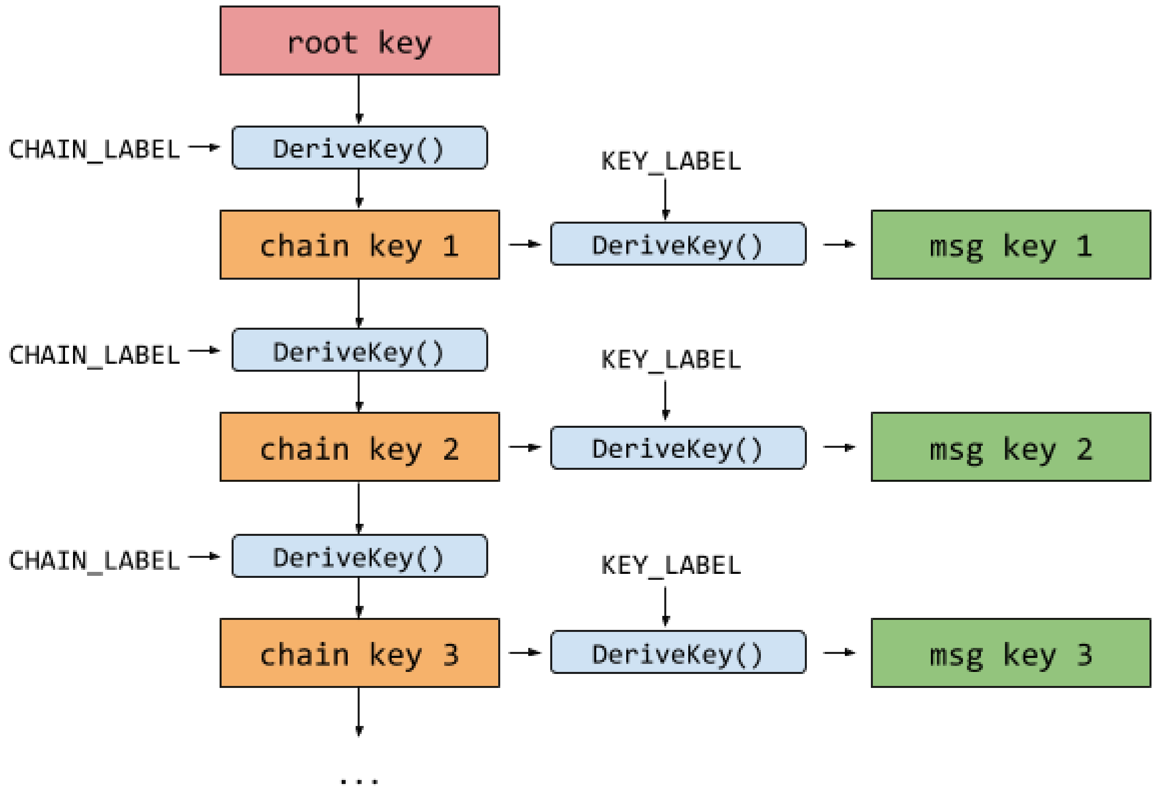
Testing: When you've implemented the handshake correctly, your code should pass the `TestHandshake` test and `TestHandshakeVector` tests. The second of these tests contains a precise expected value based on a fixed random number generator. Until you pass the basic handshake test the remaining tests will 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 are actually the only two additional methods you'll need to implement besides the handshake methods (you may of course want 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 (again using the `DeriveKey()` KDF with `CHAIN_LABEL`) to derive a new key. They should also delete the old 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 instead uses a *double ratchet* as follows:



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 AESGCM `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 doublet ratchet correctly, your code should pass the `TestOneWayChat` test, for a simple conversation in which only one party 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 Alice (the initiator) ever sends a message, she generates a new secret *a*2 and ephemeral DH key g^{a2}. She then computes the DH value $g^{a2 \cdot b1}$ (where *b1* is Bob's initial ephemeral DH key is g^{b1}) and uses this to update her root key. In your implementation, Alice will update the root key by calling `CombineKeys()` with the old root key and the new key derived from $g^{a2 \cdot b1}$ (passed in that order). She'll then derive a new sending key chain as before and use it to encrypt the next message she sends.

For Bob to be able to decrypt, Alice will need to send her new DH value g^{a2} (her DH ratchet key) along with her encrypted message. Bob can then derive the same value $g^{a2 \cdot b1}$ and update his copy of the root key to derive Alice's current sending key chain. He can then use this to decrypt Alice's message. As long as Alice is the only one sending messages, she'll keep updating this sending chain using the symmetric ratchet (the double ratchet implemented above). Bob should also make sure to delete his secret *b1* at this point, since he'll no longer need it.

When Bob has a message to send back, it's his turn to:

1. Pick a new DH ratchet key g^{b2}
2. Update his root key by combining with $g^{a2 \cdot b2}$
3. Derive a new sending key chain
4. Use this to encrypt his message and send it (along with g^{b2}) to Alice so she can update her root key in the same way and derive the keys needed to decrypt Bob's message

All this work to keep Eve out of the conversation! In general, the DH ratcheting proceeds in turns. At first, it's Alice's turn to send a new DH ratchet key and update her sending key chain. She'll use these keys for all messages she sends until it's her 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(Tripartite DH output)	Bob
k_{root2}	Combine(k_{root1}, g^{a2b1})	Alice
k_{root3}	Combine(k_{root2}, g^{a2b2})	Bob
k_{root4}	Combine(k_{root3}, g^{a3b2})	Alice
...	...	...

Note the convention that it's Alice (the initiator)'s turn to send a new DH ratchet value first. If Bob happens to send a message first, he'll use the sending chain derived directly from the first root key derived from the handshake.

Testing: When you've implemented the DH ratchet logic correctly, your code should pass the `TestAlternatingChat` test (a simple case where both sides send one message at a time), the `TestSynchronousChat` test (both sides may send more than one message at a time), the `TestSynchronousChatVector` test (a precise expected value using a fixed RNG), and the `TestSynchronousChatExtended` test (a longer, parameterized version with multiple senders randomly sending messages). The last one should be a good stress test to identify bugs. You can increase the parameters `EXTENDED_TEST_ROUNDS` and `EXTENDED_TEST_PARTICIPANTS` to make this test more thorough.

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. Conceptually, the solution is straightforward. Each message has a `counter` value the sender users to signify which order the message was sent.

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. A good place to store them is in 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. Bob sends messages 1-3 using his initial sending chain, but Alice doesn't receive them.
2. Bob receives a message from Alice with a new DH ratchet value, making it his turn to pick a new DH ratchet key. He does so, and updates his sending chain.

3. Bob sends messages 4-6 using his new sending chain, but Alice doesn't receive them.
4. Finally, Alice receives message 6.

At this point Alice will need to derive and store the keys needed for messages 1-3 using Bob's old sending chain, then derive Bob's new sending chain using his new DH ratchet key, then derive and store the keys needed for messages 4-5, then finally decrypt message 6. To help Alice do this, each message from Bob contains a `lastUpdate` value in addition to a sequence number, indicating at what point he last updated his 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 correctly, your code should pass the `TestAsynchronousChat` test (a simple case of 8 messages) and the `TestAsynchronousChatExtended` test (a longer, parameterized version with multiple senders randomly sending messages). 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 enroute? 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 it's really AEAD (authenticated encryption with additional data) to handle data like this which cannot be encrypted but you want to verify the integrity of. A function `EncodeData()` 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.

Part 6: Deleting key material

It’s critical for cryptographic implementations to delete 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 (both symmetric keys/chain values and DH private keys) as soon as it is no longer needed. Messages need only be decrypted once, after which their key should be zeroized.

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 delete. 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 exist 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.

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 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. If you don’t get all parts working correctly, you will receive partial credit in the following amounts:

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