Security Design/Analysis of Cryptik

Best Team Ever

Fall 2021

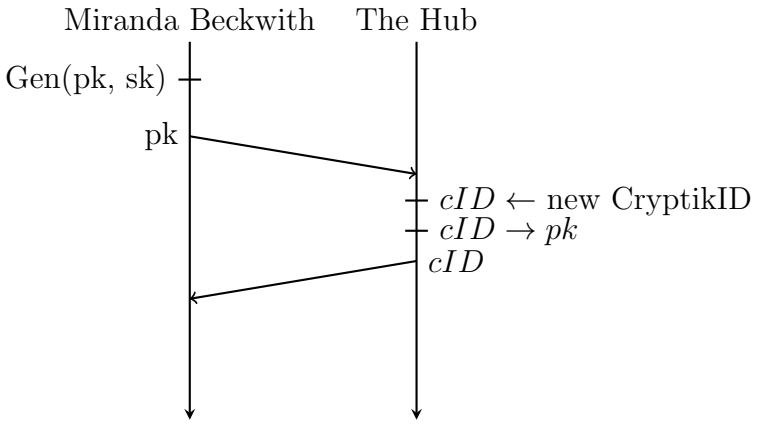
1 Introduction

Suppose we have a family who would like to communicate with utmost secrecy, we can assume they have a secret they are sharing from the world. In order to keep this information secure, they use a system named Cryptik to send and recieve messages to and from their family members. Cryptik ensures the prevention of four attacks; data eavesdropping, data modification, data originator spoofing, and data replay.

2 Authenticating Users

2.1 Creating An Account

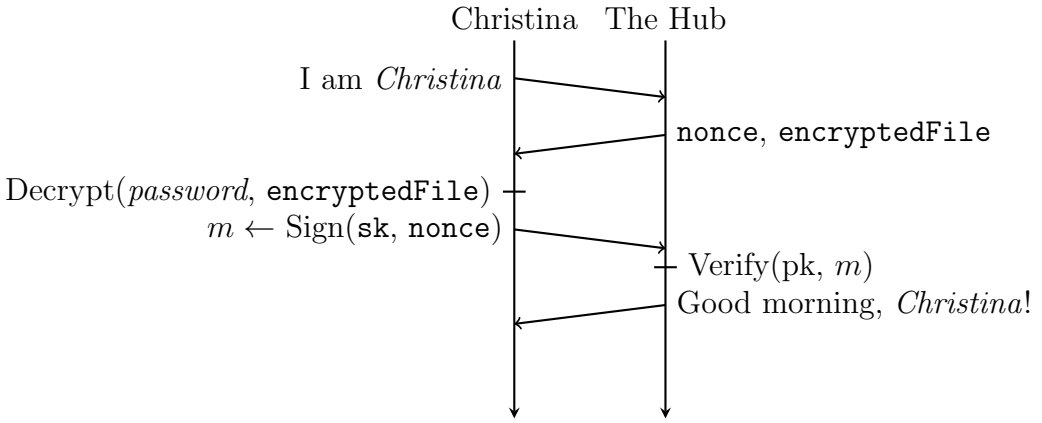
So essentially what we want to do is create a public/secret key pair. To get a CryptikID, send the public key, and get a CryptikID returned. Then since a password was created, we can then encrypt the secret key locally, as well as send it to the server (although this sharing can be done asynchronously).



2.2 Logging In

Rather than use a traditional method for identifying and authenticating users (ie sharing passwords with the server, we use a different mechanism to handle that we need to maintain data that is secret from the server. Instead of having two passwords, one to authenticate with the server and one to decrypt data needed for end-to-end security, and to allow logging in on multiple devices, we reuse the public/secret key pair needed for signatures. Below is a visualization of the authentication

mechanism for users. More than just the secret key of the public/private key pair, it holds all shared keys and other secrets when starting a channel.



Because we need a public/private key pair to sign messages to prevent spoofing and repudiation, we can reuse this for authentication. To authenticate an existing user, they provide their CryptikID which is sent to the server to retrieve a nonce and the user's file holding their secret keys. The user's file is encrypted using a key derived from the password. Because we do not share the key with the server, it cannot decrypt the file. To authenticate the user, a hash of the password cannot be used. Instead the server sends a nonce with the encrypted file, which the user can sign and return, provided they have the correct password to decrypt the file. Since an adversary cannot decrypt the file, they do not have access to the secret key and thus cannot sign and authenticate the nonce.

This approach allows for the server to hold the user's secret keys without knowing them. This has the benefit of maintaining secrecy while also allowing the user to log in on multiple devices, securely sharing and storing sensitive data over an untrusted medium. The one downside is that an attacker can then do an offline attack on the encrypted file for a user. While this is potentially dangerous, given that we are using AES with 256 bit keys, no known adversary can decrypt data unless poor passwords are used. Thus like every other application, it is important to practice good password hygiene.

2.3 Certificates for users

To validate that a certain user sent a specific message, thereby preventing integrity violations with modification or spoofing, each user has a public key kept on the server. A user, say Rosalie, sending a message can sign it by decrypting (using their private key) a hash of the message. This allows the recipients to verify that Rosalie actually sent this message without interference in the middle.

2.4 User IDs

Everyone is identified by a Cryptik ID, a 16-bit number represented as four hexidecimal characters. We chose this because it maintains the aura of secrecy we aim to provide with Cryptik. This also provides security by obscurity as guessing someone's Cryptik ID exists with probability $\frac{1}{2^{16}} = \frac{1}{65{,}536}$. Preventing users from messaging each other is not in the scope of this document, but the use of Cryptik IDs does provide this extra benefit. Our use of a 16-bit number also limits complexity of variable length naming schemes, thereby limiting one class of bugs prevalent in many other designs.

To find friends, we expect users to use out of band methods (such as texting their cryptik IDs).

3 End-To-End Communication

3.1 Generating a Shared Key

To generate a shared key, we use an expanded version of Diffie Hellman, where instead of having two parties agree on a key, n-parties agree, where$n$ is the number of members in the chat. This allows flexibility in the amount of users

As we recall, Diffie Hellman works by having a public n, then generating a public g, $g^x \ mod \ n$ and $g^y \ mod \ n$, with private x and y kept by each of the parties communicating. They then have a shared secret of $g^{xy} \ mod \ n$, which each communicating party can communicate given they have either x or y, but any other party without knowledge of either secret cannot easily compute the shared secret.

We expand on this notion by instead of having two private values, we have m private values, where m is the number parties communicating. While this does require $O(m^2)$ exponentiations among the group to establish the shared secret, similar to two party Diffie Hellman, this provides forward secrecy and symmetric encryption with the shared secret.

This method satisfies correctness (all members will achieve the same shared secret) because Z_n^* is a group with respect to the set of $\{1, 2, ..., n\}$ and the operation multiplication modulo n. We use the group's property of associativity to prove correctness. To prove this, we must show that all orderings of applications of the operation result in the same value.

As an example, if Alice, Esme, and Rosalie wanted to communicate with secrets a, e, and r respectively, they Alice can compute the shared secret by computing $((g^e)^r)^a \ mod \ n$, Esme by computing $((g^a)^r)^e \ mod \ n$, and Rosalie by $((g^a)^e)^r \ mod \ n$. As maintained before, this requires eight exponentiations, which is within our $O(m^2)$ (nine exponentiations in this instance) bound.

3.2 Method for Sharing Values for Keys

To avoid $O(n!)$ exponentiations to create the secret key and get the better upper bound of $O(n^2)$, we use a circular method of exchanging values. This gives us a simple, principled method of exchanging key data while sending the minimal number of exponentiations and messages. The downside to this protocol is that it requres each user to accept the chat in a serialized fashion. We argue this is not a large issue given that we already require every member of a chat to accept before messages can be sent.

Algorithm 1

```
1: function NewChat(members, g, n, exp)
2:     if Deny then
3:         Send deny to all members before you in list
4:         Return                          ▷ We are done and don't have to do anything else
5:     end if
6:     if ∄secret then                                  ▷ Round 1, generate our secret
7:         mySecret ← randVal % n
8:         newExps ← [x^mySecret % n for x ∈ exp] + +[g^mySecret % n]
9:         Send members, g, n, newExps to next member
10:    else                                  ▷ Round 2, we get the shared secret value
11:        sharedSecret ← exp[0]^mySecret % n
12:        if ∃exp[1] then                                  ▷ There is more values
13:            newExps ← [x^mySecret % n for x ∈ exp[1 :]]
14:            Send members g, n, newExps to next member
15:        end if
16:    end if
17: end function
```

From the shared secret, we can hash it to get a key known only by the others in the chat.

3.3 Messaging with a Shared Key

Given that we have a shared key, we can easily encrypt messages destined for the group chat. Given that we have solved the problem of sharing keys, the problem of actually encryption and decryption is trivial. We can use any semmtric key cipher we chose, then encrypt the message and any associated data we would like to keep secret, then pass this to the server with enough information to deliver it to the intended recipients. Because we require the server to be an intermediary to share messages, it can cause denial of services when it is down (either maliciously or not). However, because of the key generation explained above and other mechanisms explained below, the server cannot decrypt or modify messages.

In our implementation, we used AES in GCM mode as our encryption algorithm. We use 256 bits for strong encryption, and detail our decision to use GCM mode below.

3.4 Protections Against Classes of Attacks

Given that we have created a shared secret key, we can now encrypt messages in an end-to-end fashion, where ends are people in the conversation and *no one else*. We still must protect against eavesdropping, modification, spoofing, and replay attacks, something that merely having a shared secret does not protect against. We use AES in GCM mode to provide authenticated encryption and signatures with the public keys stored on the server to prevent spoofing, and vector clocks to prevent replay attacks.

To protect against eavesdropping we use AES encryption with 256 bits. Given that this is the current standard for secure encryption used globally, it works as a solution for us. We use AES in GCM mode to prevent modification attacks, as GCM not only makes the output blocks non-deterministic (preventing against IND-CPA attacks), but also authenticates the messages, preventing intermediate attackers from modifying the message (and also preventing IND-CCA attacks).

To prevent spoofing attacks, we use digital signatures, such that only the person with the private key (the person sending the message) can feasibly generate a valid signature. As mentioned earlier, our server maintains public keys for each user, meaning they have a certficate that can be used to verify their signature.

To prevent replay attacks, we use vector clocks. This guarantees a causal ordering of messages, which is especially useful as many messages can be sent at a similar time, and this maintains some ordering between them. This prevents replay attacks, as messages replayed will be dropped, as the vector clock on the replayed message will be lower than what the user has. In the event that an attacker tries to send a message to a user that has not recieved the message, two actions can happen. If the user is not in the chat, they can simply drop the message. If the user is in the chat however, they can accept the message as if it was just a regular message from the server. Given that the messages are encrypted and authenticated, the attacker cannot read or modify the message.

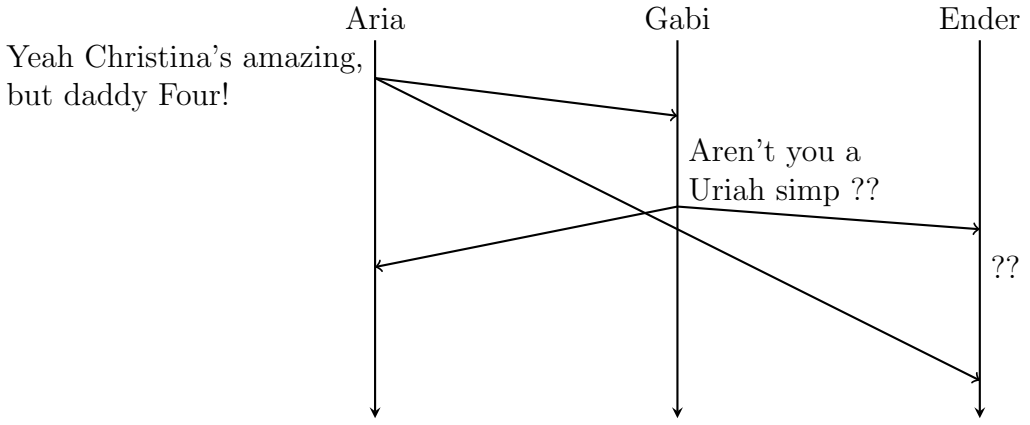
3.5 Limitations

Because we use a centralized server to act as a courier who delivers messages, there is certain information that must be leaked to them in order to ensure correctness. While this does not allow for the server to read or modify messages, it does leak who is sending whom a message, at what time the message is sent and retrieved, and aggregate statistics like how many messages are sent and in what frequency. We argue that this does not leak information about the message itself, and this data does not allow for the server to cause integrity violations.

4 Vector Clocks

While vector clocks do not ensure that all messages are delivered in the same order at each end, they give strong enough ordering guarantees (causal ordering) to maintain an understanding of the conversation. Additionally, unlike a total ordering solution, which would provide each user with the same order, this does not require coordination with the server or other expensive consensus mechanisms (Paxos). This means that some ordering can be gotten without leaking information.

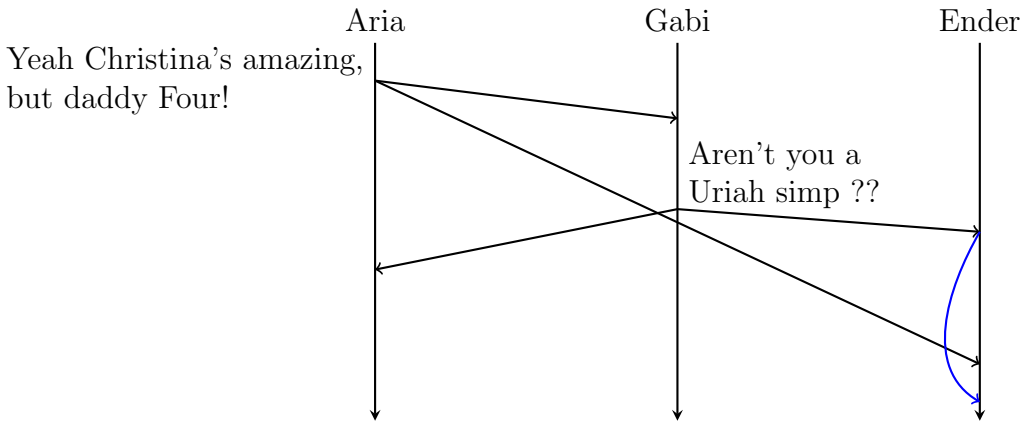
4.1 Causal Ordering: a Definition



The above example demonstrates a movitating example for our choice of causal ordering. Given that Gabi is responding to Aria, Ender is confused when he receives Gabi's message first before Aria's. Causal ordering is necessary for multiple security reasons. First while there are timestamps

on messages, a process can modify their timestamp in ways that would make it seem like a process is responding to messages not actually delivered at the time. Additionally, this prevents someone from modifying a message later by only delivering messages that are causally consistent, which means a later modified message would not be redelivered/updated.

Looking at our same example from above, adding causal consistency includes delaying the delivery of Gabi's message until after Aria's message is delivered. We demonstrate this by not actually delivering the Gabi's message when it is recieved, but using the blue arrow to indicate that delivery is delayed. Given this order of messages Ender sees, he is no longer confused about the conversation.



4.2 Implementing Causal Ordering Using Vector Clocks

We now introduce vector clocks as a means of providing causal ordering. The benefits of vector clocks is that they do not require any mechanisms for coordinating ordering outside of regular messages sent, meaning they are just a small amount of data added to each message. The data we append is a set of counters, with one counter for each party in the chat. In the case of 3 people in a chat, we would have 3 counters as the vector clock. A vector clock is a concise way of explaining the state of a chat that a person sees.

The counters are monotonically increasing values that increment when a message is sent, rather than a traditional clock that counts time. Each person has their own clock in the vector clock, and they are the only one to increment their clock, which they do when they send a message. Receiving processes have 3 responses to a message. We will use the example of Uriah, Marlene, and Lynn in a group chat, with the first clock going to Uriah, the second to Marlene, and the third to Lynn. We use the format $< u, m, l >$ to represent the vector clock.

1. The receiving process's vector clock has a higher value at the sending process's clock, so they can discard the message. This is because the message has already been seen. For example, if Marlene recieves a message from Uriah with vector clock $< 4, 2, 2 >$, but Marlene's local vector clock is at $< 6, 3, 8 >$, she will discard the message because she has already seen the message. In this scenario, Lynn would also receive the message and do a similar calculation, but we only focus on Marlene for simplicity.

2. The receiving process's vector clock is one less at the sender's clock, but the receiving process is otherwise greater than or equal for the other clocks. In this case, the receiver can deliver the message immediately and update their local vector clock. If Marlene sends a message with $< 6, 3, 6 >$ and Uriah has the local clock of $< 6, 2, 7 >$, he can deliver the message immediately and update his clock to $< 6, 3, 7 >$.

3. The receiving process has a vector clock more than 1 larger than the receiving process' vector clock, the receiver must buffer the message and wait until previous messages have been delivered. This can happen in two ways. First say if Lynn sends a flurry of messages that get reordered, and Marlene sees a message with $< 5, 2, 4 >$, but only has a local vector clock of $< 5, 2, 2 >$, Marlene has to wait for and deliver first the message with vector clock $< 5, 2, 3 >$. The other scenario is when Uriah interrupts Lynn with a message vector clock of $< 6, 2, 6 >$. If Marlene gets Uriah's message but only has a vector clock of $< 5, 2, 4 >$, she must wait for Lynn's two messages before delivering Uriah's.

Because each message is broadcasted to the group chat, we have the convenient property that each message we recieve will only increment at most one clock in our vector clock by one. This is convenient because we do not have to have specific rules about skipping over certain clocks as can occur in other instances.

We must also handle concurrent messages, where after a lull where everyone has the same vector clock ($< 6, 3, 8 >$), Uriah and Lynn simultaneously decide to send a message. Uriah's message will have a vector clock of $< 7, 3, 8 >$ and Lynn's of $< 6, 3, 9 >$. Marlene could deliver the two messages in either order, as there is no causal relationship between the two messages.

5 Sources

1. GCM https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf
2. GCM https://eprint.iacr.org/2004/193.pdf