User-Friendly Chat (6.857 final project)

Julian Bangert, Jelle van den Hooff, Tsotne Tabidze

Abstract

Communicating confidentially on the internet is challenging. While strong cryptographic primitives like public-key encryption have been widely available for years, no simple end-to-end encrypted communication apps are widely used. For our 6.857 final project, we designed and implemented a small end-to-end encrypted chat app that builds on a PKI, Keytree, for security. By using an external PKI, our app and protocol are secure yet remain simple.

1 Introduction

Chatting securely on the internet is hard. Existing protocols are either insecure (unencrypted), complicated to use (PGP), or require external authentication (OTR [3, 4]). We believe that most of the complication in these secure protocols comes from the fact that they combine key verification (determining who you are talking to) with the chat protocol.

We propose a design for secure app based on Keytree, an external PKI. Assuming a secure external PKI allows us to build a simpler protocol as we move the hard authentication problem out of the chat protocol. Our app's design lets users securely chat with other users. Our app supports one-on-one conversations between pairs of users, but not group chat. In this paper we will explain and motivate the design of our app.

Our design is federated and does not have a single central server. Instead, each user picks their own chat server (Figure 1). Many users could pick the same chat server, such as a server run by an ISP, or users can run their own server if they prefer to be independent of large providers.

In our app, messages are end-to-end encrypted and only readable on the devices the users use to access the app (such as their laptop or their phone). Crucially, users do not

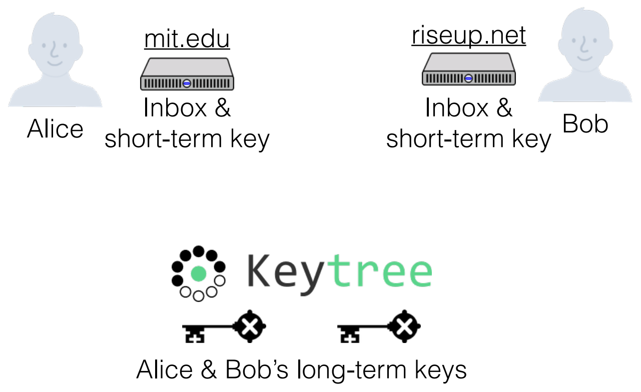


Figure 1: Our app's architecture. Individual users pick a chat server they want to use with their app, such as mit.edu for Alice and riseup.net for Bob. The chat servers store users' inboxes and short-term public-keys. For security, our app relies on Keytree which stores long-term public keys.

have to trust their chat server to protect the confidentiality of their messages. Instead, our app only relies on chat servers to provide availability: A user's chat server must be online and cooperating with our protocol for a user to receive messages. We describe our security goals in depth in §2.

To enable our security goals, our app relies on cryptography as well as users' knowing each others' public keys. To exchange keys, we rely on an external PKI called Keytree. We briefly describe Keytree and its security properties in §3.

Our final app consists of three parts. First, we have a message-exchange protocol (§4) that uses Keytree to let users send messages to each other, supporting offline recipients and forward secrecy. This message-exchange protocol naturally works with multiple servers (§5). Finally, we store a long-term chat history in an efficiently accessible manner (§6).

To test out our design, we built a small prototype of our app in Go and JavaScript (§7).

2 Security goals and assumptions

Our security goals are split in four parts: Confidentiality, integrity, availability, and forward-secrecy. Confidentiality ensures that no one except the intended recipient can read messages. Integrity means that no adversary can fake or modify messages. Availability ensures that our app remains working without relying on a central server. Finally, forward-secrecy protects messages recorded by an adversary even if a key is later recovered.

Confidentiality of contents. Our first goal is to protect the secrecy of messages sent between users. Only the sender and recipient's users devices should be able to see the contents of the messages sent between them. The users' chat servers should not be able to see the message contents, nor should an active adversary than can try to perform a man-in-the-middle on the users. We do not protect metadata or timing information, and instead suggest users use external mechanisms such as Tor.

Integrity. Our second goal is to guarantee that the receiver of a message can determine who sent it. This includes preventing an attacker from modifying a message after it has been sent. A related property is deniability - the receiver should be the only person who is certain of the senders identity. Because strong deniability is hard to enforce, in particular when not securing metadata or timing, we provide deniability only to adversaries who did not log network traffic and only have access to data provided by the receiver. In particular, the receiver should be able to fabricate messages addressed to him that look authentic to third parties.

Availability. Our third goal is to ensure that two users can communicate without relying on third-parties; this is our availability goal. Specifically, as long as the two users' chat servers (and Keytree) are functioning, two users should be able to exchange chat messages. This requirement formalises that we do not rely on a central chat server or other single point-of-failure.

Forward-secrecy. Our fourth and final goal is protect the confidentiality of previously-sent messages if a device later gets compromised by an adversary. Specifically, if two users exchange messages, and then delete them from their devices, an adversary that has recorded the network traffic should be unable to to decrypt those messages even if they get access to the private keys stored on the devices.

Our security assumptions to achieve the above goals rely on standard cryptographic assumptions and the correctness of Keytree. For cryptography, we rely on public-key signing and encryption schemes. Additionally, our clients need secure local storage (where they can store and later delete temporary secret keys), and a secure random-number generator to generate keys.

3 Keytree

To encrypt messages, users need to each other's public keys to encrypt to. Our app uses Keytree, an external PKI to obtain public keys. As Keytree is the foundation of our app's security, we will briefly describe Keytree's design and argue for its security.

Keytree's API has two functions: Register and Lookup. By registering, users can associate public keys with their e-mail address. When a user changes their public keys, they can re-register to update their keys stored in Keytree. To learn someone's public keys, a user can look up their e-mail address in Keytree and retrieve their public keys. Keytree guarantees that, as long as some servers remain honest, users will look up the latest set of keys associated with an e-mail address.[1]

The Keytree systems consists of a number of key servers. Each key server maintains its own independent mapping from names to keys, and is responsible for maintaining the integrity of that mapping. Keytree aims to have all servers store the same mapping, but for simplicity, does not require exact consensus among servers.

When a user registers their e-mail address, the user proves that they own the e-mail address by sending a DKIM-signed e-mail to any key server. DKIM, an anti-spam signature mechanism, is useful because all key servers can verifiy the authenticity of the e-mail using DNS, without having to trust the key server that initially received the e-mail. The DKIM proof is automatically distributed among key servers so that all key servers can update the user's public keys. In the common case where users operate correctly, this ensures that all servers store the same mapping from names to keys.

To look up the keys associated with an e-mail address, a user asks a quorum of servers for those keys. Only if an entire quorum of servers agree and return the same public keys does the client accept the public keys returned by the servers. This ensures that no rogue servers, perhaps compromised or malicious, can trick a user into accepting an illegitimate set of public keys for a user. As a practical optimisation, key servers can act

1. Formally, any two users have the same view of the mapping from names to keys if their locally-configured quorums intersect on at least one server. Once a user confirms they see their keys after registering, they are guaranteed that all other users with overlapping quorums will see those keys.

as (secure, authenticated) mirrors for each other, so that a client can perform a look up with a logical quorum of servers by talking to only a single physical key server.

Keytree has a limited rate of updates, and so can only be used to store rarely changing public keys. Keytree is thus suitable for storing long-term public signing keys, but not useful for storing short-term encryption keys. The rest of this paper builds upon Keytree's long-term public signing key storage to construct a secure chat app with forward-secrecy.

4 Protocol

Building on Keytree, we designed our chat protocol to be as simple as possible while still providing forward secrecy. This section describes our chat protocol.

Our chat protocol has several nice properties: First, it separates providers, who are trusted to provide availability, from the PKI infrastructure, which is trusted to provide confidentiality and integrity. Second, it uses rotating short term keys to provide forward secrecy with less complexity than alternative systems. Finally, we provide repudiability of messages.

To send a message in our chat protocol, a user downloads the recipient's current temporary key, verifies that the key is signed by the recipient's long-term key, and encrypts the message to the short-term key.

Figure 2 describes the complete flow of sending a message. We use NaCl's box algorithm for encryption, which provides message confidentiality and integrity as long as the Alice and Bob know each others' public keys and private keys have not been compromised. The message signature on the short-term key in turn guarantees that the public keys in fact belong to whoever controls the Keytree record and its associated long-term private key. §3 describes how Keytree assures this is in fact Bob and Alice.

Resilience. Our system also tries to minimise the impact if some of the above assumptions are violated. Consider an attacker, Mallory, who might compromise either Alice's long-term or short-term key.

If Mallory compromises Alice's short-term key, he can read messages Bob sent her while that key was current. If Alice's signature on the short-term key has not yet expired, Mallory can also re-use that signature to trick Bob into sending more messages

1. Alice looks up Bob's email address in Keytree, getting his long-term public key and server.
2. Alice asks Bob's server for his last short-term key.
3. Alice validates that the short-term key was signed by Bob and has not expired.
4. Alice encrypts the message with her short term key using NaCl's box algorithm and sends it to Bob's server along with her current key.
5. Bob's server notifies him of the message as soon as Bob comes online. Bob retrieves the message.
6. Bob verifies that the short-term key was signed with Alice's long-term key.
7. Bob decrypts the message and validates its authenticator.

Figure 2: Alice sends a message to Bob

encrypted to the key Mallory has compromised. This attack could be made harder to pull off by having the server validate that older short-term keys cannot overwrite newer ones (in which case Mallory would also have to compromise Alice's chat server). Similarly, as long as the short-term key signature is valid, Mallory can impersonate Alice and send messages on her behalf. The impact of this second attack could be reduced by having a much shorter expiry time for sending messages as opposed to receiving.

While these attacks can break our system, a compromise is limited in time. Typically, clients rotate keys every few minutes and short-term key signatures have lifetimes on the order of hours or days. The lifetime parameters can be adjusted individually by each client based on their security needs.

Forward Secrecy. If Mallory compromises Alice's long-term key, he cannot decrypt any messages he might have previously captured, as the corresponding short-term keys have been deleted. Mallory can impersonate Alice to her peers until he changes her long-term key in the PKI, however unless Mallory also has Bob's long-term private keys he cannot man-in-the-middle their conversation (because whatever short-term key he uses to imitate Bob will not have been signed by Bob).

However, if Alice chose to store her past messages in the encrypted messages in the encrypted block storage as described in §6, the attacker can decrypt Alice's archives with her long-term key. Therefore, Alice should not log sensitive conversations if she does not believe she can keep her long-term private key confidential.

Popular alternative approaches are protocols such as the Axolotl ratchet [6] or OTR [4], which perform a Diffie-Hellman key exchange along with every message exchanged.

Like our protocol, these protocols still rely on a long-term "identity key" to authenticate these key exchange, and compromising this long-term identity key is equivalent to compromising our systems long-term key. The difference between our protocol and these key exchange-based protocols is that our protocol is simpler and does not rely on per-conversation state. This simplicy comes at a cost, an attack has a short time window in which they could decrypt messages and a slightly longer window in which to man-in-the-middle conversations, which we believe is a valid trade-off for relying on just a single, well-vetted cryptographic primitive (NaCl).

Repudiability. The other interesting property we get from NaCl that OTR has to provide through more complicated mechanisms is message repudiation. For example, if Alice is a dissident in a repressive regime and talks to Bob, who turns out to be an informant, it would be preferable if Bob cannot prove to another party, Mallory, what the contents of Alice's messages were. A detailed discussion of message deniability can be found in [5]. NaCl's authenticators are designed to be creatable by the owner of either private key, so both Alice and Bob can compute arbitrary authenticators for any message [2]. This implies that Bob could have forged any message he provides to Mallory. If we assume that Mallory also previously captured the messages, he can of course verify if the authenticated messages provided correspond to those previously sniffed. This loophole is hard to avoid, unless we adopt a scheme that also protects metadata, for example by adding cover traffic. Cover traffic however is very inconvenient for mobile devices, as it constantly drains their battery.

5 Multiple Servers

Keytree typically uses email addresses to identify users, however users should be able to host their chat at another domain than their email. Ideally, users should be able to use their existing identities, such as GMail or Facebook email addresses, without requiring any cooperation on behalf of these providers. To accomplish this, we also publish and sign the location of the chat server in Keytree, so any chat provider can be used for any email address.

To send messages to another user, that users client directly contacts the recipients server, which will queue messages until the recipient comes online. Because we provide end-to-end security, the recipients server does not need to authenticate the sender in any way. The current implementation does not prevent spam or denial-of-service at the server level, however a scheme such as Hashcash [1] or simple rate-limiting could be employed at the server to reduce these. Clients can ignore messages and not display or store them in the message history after decryption.

6 Conversation history storage using a B-tree

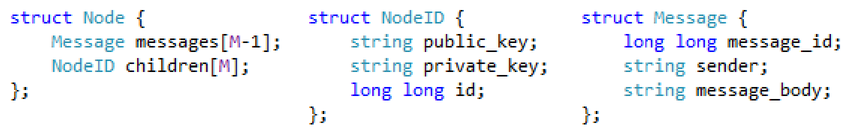
To let users access their conversation history, we store an encrypted B-tree holding a user's chat messages on their own server.

Our message exchange mechanism uses short-term rotating keys to guarantee forward secrecy. While secure, changing and erasing keys means that when the session is over the users themselves no longer have the keys to decipher the messages leaving them unable to read past messages. Yet chat apps are much more useful if they let users access messages sent in the past. We considered storing all messages locally on the user's device, but decided against doing so because clients might not have a lot of storage and users might switch devices.

Instead, we store each user's conversation history on their own chat server, giving us long-term persistence. Currently, this sacrifices some forward secrecy, but we have some ideas on how to restore forward secrecy even with long-term data storage.

The user's conversation history is stored in a key-value database stored in a hierarchical B-tree (Figure 3). This B-tree is implemented entirely on the client, and uses the server merely as a storage back-end for holding the tree's nodes. From the server's viewpoint, it just stores a bunch of key-value pairs. Each key corresponds to an ID of the nodes in the B-tree, with a corresponding value holding an encrypted JSON structure of the node's actual values. We guarantee that, as long as the user's device remains secure, the adversary is retrieve the data stored in the tree, even if the adversary controls the user's chat server.

Each B-tree node is encrypted with a different encryption key. Each node is encrypted with a per-node random key, which is stored in the parent node. The root node, which does not have a parent, has its key stored on the server, encrypted with a long-term key stored user's device. The intuition behind this structure is that the key to the entire tree can be changed by just changing the root's key. From the app's point of view, the B-tree is a key-value store that holds past messages. After receiving a message through the chat protocol, the client writes the chat message to the key-value store. Afterwards, the client can lookup the history by reading from the key-value store.



```
struct Node {
    Message messages[M-1];
    NodeID children[M];
};
```

```
struct NodeID {
    string public_key;
    string private_key;
    long long id;
};
```

```
struct Message {
    long long message_id;
    string sender;
    string message_body;
};
```

Figure 3: Each decrypted B-tree node contains up to M-1 messages and up to M children, where M is the order of the B-tree. Each Message has its own id, the sender and the body of the message. Each child directs us to other node with ID, and it also includes the keys for decrypting and encrypting the child node.

Reading and writing this encrypted B-tree is almost as the same as with a normal unencrypted B-tree. To read a value, the clients starts by decrypting the root, and then depending on the key and the root's children, the client decides what child to fetch next. This process then recursively repeats itself until a leaf holding the value is found. To write a value, the client first uses the previous process to find the leaf holding the key (if any). Then, it modifies the leaf, potentially splits it, and updates all the parents nodes. During this update, the client re-encrypts all parent nodes with different keys. This is done to help with forward secrecy, so that if an adversary obtains a key to a new version of a node, they cannot access the old version.

To access the data in the tree, the client stores a long-term secret key locally. While storing this long-term key lets the client access historic data, but also exposes a forward secrecy problem: An adversary that stores an old copy of the tree, along with an old root, and later captures the long-term key from the user's device can now access the old tree.

We use a long-term key so that the user can write down the key on paper and restore access even if their device fails. An improvement to this design, that we have not yet implemented, is to roll this key forward locally using a one-way function to derive future secret keys from the current secret key. That would allow the user to safely delete nodes of the tree even if their device later gets compromised, while still allowing the user to restore access. An even stronger security guarantee would be to rotate to random keys (so that an adversary who gains access to the written-down key still cannot access the historic data), but that leaves a user vulnerable to data loss if their device fails.

7 Prototype

We implemented a prototype of our system. We will describe the high-level implementation choices, provide an overview of the API the server exposes and discuss how we address message notifications.

Message servers are implemented in Go, and clients in JavaScript. We chose Go because it was designed for building scalable distributed software, and because its typing system provides reasonable guarantees against memory corruption vulnerabilities. We implemented the client in JavaScript, using the React framework and TweetNaCl, to make it readily portable to different platforms, such as web browsers and mobile phones. Currently, the client does not expose a user interface for the Keytree PKI, so users also have to manually publish their long-term public keys and server hostname. Eventually, we plan to include an automated sign-up prompt and interactive Keytree setup assistant.

Currently, the client has only been ported to web browsers, which are a somewhat problematic platform for security-relevant apps. Our app is fetched from a server – which we should not trust to not tamper with the client app, e.g. to make it leak keys. Also, browsers do not offer any good protections for keys in the case of vulnerabilities in our client code. Because key material is just another JavaScript object, an attacker who can execute arbitrary code can retrieve it.

Both of these issues can be mitigated by moving our app into a browser extension or mobile phone app. Both of these offer code-signing and local installation, assuring the user that they are actually running the correct app, as well as black-box cryptographic and keychain APIs that allow the app to perform crypto operations without exposing the key itself to the app.

Our chat server exposes a RESTful API, shown in Figure 7. Note that messages are never deleted by the client, but automatically garbage collected by the server. This means the server does not need an explicit authorisation API for storing and retrieving messages. It is allowed to retrieve another users encrypted messages, but our protocol guarantees that it is impossible to read them.

The API just described is based on polling for new messages, which is quite inefficient, particularly for mobile endpoints. Therefore, we add a web-socket based notification API, which allow the client to open a WebSocket and let the server notify it once new messages are ready. The server-side implementation is pluggable, so that other notification systems, e.g. the proprietary push standards used in mobile phones, can be readily integrated.

GET	`"/block/:key"`	Retrieves a B-tree block
POST	`"/block/:key"`	Stores a B-tree block
GET	`"/message/:address?since=:id"`	Get encrypted messages received by `:address` since message `:id`
POST	`"/message/:address/send"`	Send a message
GET	`"/key/:address"`	Retrieve signed short-term for `:address`
POST	`"/key/:address"`	Publish a short-term key and signature for `:address`
GET	`"/notify/:user/ws"`	Subscribe to notifications when `:address` receives a new message

Figure 4: API exposed by our server

8 Conclusion

We described a chat app based on an external PKI system and rotating short-term keys. Third-party servers are only trusted to maintain availability, whereas confidentiality and integrity are guaranteed by existing end-to-end cryptographic schemes. Our system reduces the complexity of the chat protocol and still provides offline messaging, forward secrecy and strong privacy and integrity guarantees.

References

[1] A. Back et al. Hashcash - a denial of service counter-measure, 2002.

[2] D. J. Bernstein. Subject: Re: crypto flaw in secure mail standards, July 2001. https://groups.google.com/forum/#!msg/sci.crypt/73yb5a9pz2Y/LNgRO7IYXOwJ.

[3] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 Workshop on Privacy in the Electronic Society*, Washington, DC, Oct. 2004.

[4] I. Goldberg and The OTR Development Team. Off-the-record messaging, 2015. https://otr.cypherpunks.ca/.

[5] “greg”. Secure function evaluation vs. deniability in otr and similar protocols, Apr. 2012. http://phrack.org/issues/68/14.html.

[6] T. Perrin and M. Marlinspike. Axolotl ratchet, 2014. https://github.com/trevp/axolotl/wiki.