Chatterbox

Andres Erbsen Asya Bergal David Kaufman

`https://github.com/andres-erbsen/chatterbox`

Overview

Chatterbox constitutes a protocol and set of utilities that provide a private long-haul messaging primitive. It was designed with the following three applications in mind:

1. **Asynchronous messaging:** email, newsletters, server monitoring alerts
2. "**Instant" messaging:** chat, news tickers, low-bandwidth data feeds
3. **Asynchronous control signals:** automatic updates, NAT hole punching.

Under assumptions we deem reasonable, Chatterbox provides the following guarantees for the transmitted messages:

1. **Confidentiality:** neither the network or the servers should be able to learn the message contents.
2. **Authentication:** neither the network or the servers should be able to convince a client B that a message m was received from A unless A actually sent that message to B.
3. **Relationship hiding** in the presence of a pinhole adversary: monitoring any one computer or network connection (including the server, excluding the computers of the communicating parties) should not reveal who is talking to whom.
4. **Forward secrecy**: In the event that a client computer is compromised, all these guarantees should hold for all messages sent more than a constant number of round trips ago. In particular, compromising a client that just sent a message should not enable an attacker to decrypt that message.
5. **Future secrecy**: If an old copy of the client state (a backup) becomes public after a constant number of round-trips since it was fresh, all these guarantees should hold for all future messages.

Furthermore, to not limit the usability of the applications built using chatterbox, we require integrated it with the `dename`[@Dename] PKI system – in the Chatterbox interface, each user is denoted by a human-picked username, not a public key or pre-shared secret. While this makes Chatterbox's security dependent on at least one server in the `dename` quorum behaving correctly, it also removes what we consider the most important adoption bottleneck of existing encrypted messaging systems: manual key management.

Design

Message Security (`ratchet.go`, `client_util.go`)

We use a slight modification of the `axolotl` ratcheting encryption protocol (used in TextSecure and Pond) to provide properties 1, 3, and 4. This protocol attempts to cycle the keys used for encryption as often as possible to ensure Forward Secrecy. A set of private keys held by A and B are cycled every time a roundtrip is completed, and a set of shared keys held by both A and B are cycled every time an individual message is sent. A combination of all these keys is used to encrypt the sent message. We added a new public-key ratchet chain to Axolotl, parallel to the one used to update the "root key" for authentication purposes. Unlike the root key ratchet, our authentication ratchet's keys are kept along until the other party definitely has received the next one. This way, the sender can safely use the Diffie-Hellman shared secret between the recipient's ratchet authentication key and the sender's own authentication public key in Dename to prove that he indeed sent the message without compromising any other property of the protocol. Furthermore, every user of our service signs and publishes 100 pre-keys (Diffie-hellman public keys) that are used to initiate conversations. The Diffie-Hellman shared secret of one of those keys and a Diffie-Hellman key the sender generates is used to bootstrap an Axolotl session without requiring a roundtrip (the second half of the Diffie-Hellman exchange is piggypacked onto sending the first message). It is the server's responsibility to hand out each pre-key a most once, but as long as one use of each key reaches the user (after which the user erases the corresponding secret key), the forward security properties remain intact.

Transport (`transport.go`)

All messages between a client and a server are padded to a constant size, encrypted, expected to be transferred over TOR. While our application does not force itself to use TOR, we expect it to be run inside a wrapper such as `torify`, or ideally from a TOR-only live image or VM such as TAILS or WHONIX. The user authenticates themselves to the server using a variant of the SIGMA-I protocol where signing is replaced with public key authenticated encryption to the signature verifier. Note that breaking this layer would allow an attacker to see when a user receives messages, but not to decrypt them or to directly deanonymize or impersonate the user.

Filesystem interface

Rather than restrict the user to a particular UI, we chose to provide a file system API for sending and receiving messages. Our API permits users to use fancy, feature-filled UIs while still making it easy to view conversations with `cd`, `ls`, and `cat`.

All application state for a user is stored within a root directory. An initialization tool creates this directory and two files in it: `config.pb` and `profile.pb`. `config.pb` contains Chatterbox private account information (i.e. the key signing secret key) and `profile.pb` contains Chatterbox public account information (i.e. the home server tcp address). The rest of the structure is set up and maintained by the client daemon. The daemon creates a `$ROOT/keys` directory to store ratchet keys

and prekeys. `$ROOT/conversations` contains a list of conversations (named "date-sender-recipient-...-recipient" for user readability), each of which contains messages ("date-sender") and a metadata file. For each conversation the daemon maintains a corresponding directory in `$ROOT/outbox` with the same name and metadata file. To send a message in an existing conversation, the user just adds a file to the conversation folder in `$ROOT/outbox`. The daemon will send the message and move it to the `$ROOT/conversations` directory. To start a new conversation, the creates a new directory in `$ROOT/outbox` and a metadata file. The daemon detects that it is a new conversation and handles it appropriately.

The daemon attempts to simulate atomic write operations by using the `$ROOT/tmp` folder as an intermediate destination. For example, when it stores prekeys to disk it first writes the new file to a directory in `$ROOT/tmp`. Once the write is complete it swaps the new and old prekeys files (and securely deletes the old file). This on its own is not quite sufficient to avoid losing data in case of a crash, but it works in some cases. For the rest of the cases we plan to implement a journaling system (and have reserved `$ROOT/journal` for this purpose).

The daemon also creates `$ROOT/ui_info`, which UIs are allowed to use to store their data. UIs are also allowed to make use of the `tmp` folder (within a certain namespace), write messages to the outbox, and read/delete messages from conversations. The hope is that UIs will store all additional information within the `ui_info` folder so users can keep track of where their private information is stored.

Client library (`client_util.go`)

The client library consists of a collection of functions used by the daemon to encrypt and decrypt messages as well as communicate with the server.

Keys uploaded to the server are signed with a special signing key (`SignKeys`). When a key is downloaded from another user's server, the signature is checked against the signing key in their Dename profile (`GetKey`).

Before encrypting and filling in authentication information, the library pads messages to a specified length such that every encrypted message has a fixed length (`EncryptAuth`). This padding is removed upon message decryption.

Since our messages contain no unencrypted metadata that can identify the sender, to decrypt messages (`DecryptAuth`) we iterate through all existing ratchets until we find one that authenticates the message correctly.

Note that encrypting and decrypting the first message from another user (`EncryptAuthFirst` and `DecryptAuthFirst`) work differently since there are no existing ratchets. To decrypt first messages, we must look at the public keys sent at the beginning of the message and decrypt using the secret key corresponding to our public key.

Client daemon (`daemon.go`)

The client daemon forms the interface between the user-facing filesystem API and the server API. The daemon does some initialization, but it primarily consists of an event loop that listens to

file system events (using inotify) and new messages from the server. This event loop triggers message sending (when new messages appear in the outbox) and message receiving (when the server pushes a notification). The daemon calls the methods `sendFirstMessage`, `sendMessage`, `decryptFirstMessage`, and `decryptMessage` to process messages to and from the filesystem.

The daemon is intended to connect to the internet through Tor. Each time it needs to query information from a foreign server it creates a new connection and uses a new set of public/private keys. For the home server it has to use the same public/private keys every time because the server can only identify it by its public key.

Server (`server.go`, `notifier.go`)

The server consists of a main listener thread which forks off connections to clients. The server listens for commands from the client and performs the appropriate action: creating an account, uploading keys, downloading a public key for a user, listing keys and messages, and delivering, downloading, and deleting messages. The server also runs a notifier thread (`notifier.go`) which a client can optionally enable by sending the enableNotify message. The notifier pushes messages to the client as they come in (as opposed to listing and then downloading a specific message).

The server uses levelDB to store all public prekey, registered user, and message information. It performs atomic reads and writes to the database to avoid concurrency problems. The keys section of the database is read-write locked so that no two clients make accidentally receive the same prekey.

Registered users are identified only by a 32-byte public key which is passed to the server through a secure transport handshake. Encrypted messages are associated with their recipients inside the database. The server has no way of decrypting the messages it stores and cannot construct the identity of the user/sender (assuming a Tor connection). This allows the server to be untrusted.

Messages from the client to server and back are constructed in Protobuf (`proto.ClientToServer` and `proto.ServerToClient`) and are padded to a constant size to obfuscate the type of command being sent.

References

The Axolotl protocol: https://whispersystems.org/blog/advanced-ratcheting/ (intro), https://github.com/trevp/axolotl/wiki (detailed pseudocode) https://eprint.iacr.org/2014/904.pdf (analysis of a weakened version).
Pond: https://pond.imperialviolet.org/tech.html
SIGMA-I authenticated Diffie-Hellman protocol: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=F085CC1A89CB1D0C34DC11E34FFDEE5E?doi=10.1.1.377.9124&rep=rep1&type=pdf