Cryptocat Protocol Specification
Nadim Kobeissi
Version 2.0 10/09/2012

1. Introduction

Cryptocat (https://crypto.cat) is an open source web application intended to allow secure, encrypted online chatting. Cryptocat encrypts chats on the client side, only trusting the server side with already-encrypted data.

The protocol is developed with the following design goals:

1. Provide a portable encrypted chat environment made for use in web browsers and other media over XMPP.
2. Provide public key cryptography with forward secrecy from chat to chat.
3. Provide the ability for multi-party chat (more than two parties.)
4. Confidentiality and authentication.

This protocol does **not** provide deniability.

Cryptocat deploys various technologies:

1. AES-CTR-256 for encryption and decryption.
2. Curve25519 for Elliptic Curve public key generation.
3. SHA-512 for generating 512-bit message authentication codes, shared secrets and key fingerprints.

This specification is released under the GNU Affero General Public License.[1]

2. Definitions

User: A person n accessing a chat with a specific nickname $nick_n$, a key pair $privateKey_n$ $publicKey_n$, and a fingerprint $fingerprint_n$.

Nickname: A user n's nickname $nick_n$ identifies them within the chat. Nicknames may only be 1 to 16 lowercase alphabetic characters.

Private key: A user n's private key $privateKey_n$ is a randomly generated 32-byte number. It is used to decrypt messages encrypted with n as the intended recipient. Private keys are never re-used once a user leaves a chat.

Public key: A base 64 representation of user n's public key $pubkey_n$ is generated according to §3. It is used by other parties in order to generate the keys for encrypting messages intended for n. Once a user leaves a chat, his public key is removed from the records of other users and it is never re-used.

Chat: A chat session (or "chat room") that contains one or more users, identified by a unique chat name matched by the regular expression `^[a-z0-9]{0,20}$`.

Message: A message being sent to a chat with users a and b would be a JSON object structured as follows:

$$\{nick_a:\{\text{"message"}:ciphertext_a,\text{"hmac"}:hmac_a\}, nick_b:\{\text{"message"}:ciphertext_b,\text{"hmac"}:hmac_b\}\}$$

3. Identification and Key Agreement

Cryptocat relies on an Elliptic Curve 25519 key agreement scheme.[2] Generating the private key *privateKey* relies critically on a strong random number generator, seeded using reliable levels of entropy.

Each user's *privateKey* is stored within their client and is never shared. The user's public key, *publicKey*, is then generated as follows:

$$publicKey = scalarMult(privateKey, basePoint)$$

A user a may demand another user b's public key by sending the following JSON to the chat:

$$\{nick_b:\{\text{"message"}:\text{"?:3multiParty:3?:keyRequest"}\}\}$$

Once user b receives the above key request string, they send a message to user a comprised of the public key reply string concatenated with their public key $publicKey_a$:

$$\{nick_a:\{\text{"message"}:\text{"?:3multiParty:3?:PublicKey:}publicKey_a\text{"}\}\}$$

Both key request and reply messages are not encrypted and do not include an HMAC.

Note: Upon the reception of any public key, the client must check if the key is larger than 2^{192} and smaller than $2^{255} - 19$. If a user's public key is not the proper size, it is discarded.

Note: For convenience and user-friendliness, the client may be configured to hide key request and reply messages from the user interface.

[1] https://www.gnu.org/licenses/agpl-3.0.html

[2] http://cr.yp.to/ecdh/curve25519-20060209.pdf

4. Authentication

Cryptocat clients can generate public key fingerprints for each user using the following technique (all SHA-512 functions produce hexadecimal values:)

$$fingerprint_n = SHA\text{-}512(nick_n + "*" + publicKey_n).substring(0, 40)$$

Users may verify each other's identities simply by confirming their fingerprints over a trusted out-of-band channel (which may be public,) such as a telephone.

5. Shared Secrets

Each two parties use a shared secret to exchange and authenticate messages between themselves. Shared secrets are extremely sensitive and should never be exposed.

In a chat between Alice, Bob and Carol, Alice uses the following formula to establish her shared secret with Bob:

$$secret_{ab} = SHA\text{-}512(base64(scalarMult(privateKey_a, publicKey_b))$$

And the following formula to establish her shared secret with Carol:

$$secret_{ac} = SHA\text{-}512(base64(scalarMult(privateKey_a, publicKey_c))$$

Bob similarly calculates $secret_{ba}$ and $secret_{bc}$, while Carol behaves in a similar fashion in order to obtain $secret_{ca}$ and $secret_{cb}$. Alice uses $secret_{ab}$ to communicate with Bob, while Bob uses $secret_{ba}$. $secret_{ab} = secret_{ba}$ since:

$$scalarMult(privateKey_a, publicKey_b) = scalarMult(privateKey_b, publicKey_a)$$

The first 256 bits of *secret* are used as the encryption key for AES-CTR-256 operations, while the last 256 bits are used as the key for HMAC-SHA-512 operations.

6. Messaging

Messages are formatted in a JSON format according to the definition in §1. AES-CTR-256 is used without padding and with a starting IV of 0. The IV is incremented by 1 for each block. The sender uses the first 256 bits of *secret* as the encryption key. Ciphertext is communicated in Base 64 format.

7. HMAC Generation

Message authentication codes are generated by running the concatenation of all ciphertext arranged by sorting the recipient nicknames lexicographically through HMAC-SHA-512. The sender uses the last 256 bits of *secret* as the HMAC key.

Upon the successful verification of the HMAC, the message is decrypted using the shared secret. If the HMAC check fails, an error is displayed and the message is discarded. HMACs are communicated in Base 64 format.

8. Endnotes

Special thanks to Jacob Appelbaum, Meredith L. Patterson, Marsh Ray, Joseph Bonneau and Arturo Filastò for their helpful comments and insight.