Cryptocat Protocol Specification

Nadim Kobeissi

Version 1.4a 26/05/2012

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.
2. Provide public key cryptography with forward secrecy from chat to chat.
3. Provide the ability for multi-party chat (more than two parties.)

Cryptocat deploys various technologies:

1. AES-CBC-256 for encryption and decryption with the ISO 10126 padding scheme.
2. Curve25519 for Elliptic Curve public key generation.
3. Whirlpool for generating 512-bit message authentication codes, shared secrets and key fingerprints.
4. Pseudorandom numbers may be generated using either a CSPRNG API internal to the platform. If such an API is not available, a PRNG library such as Fortuna[1] (seeded using mouse movement or keyboard input, including key press/depress timings) may be used.

2. Definitions

User: A person *n* accessing a chat with a specific nickname $nick_n$, a key pair $prikey_n$ $pubkey_n$, and a fingerprint $fprint_n$.

Nickname: A user *n*'s nickname $nick_n$ identifies them within the chat. Nicknames may only be 1 to 12 lowercase alphabetic characters (`^[a-z]{1,12}$`).

Private key: A user *n*'s private key $prikey_n$ 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 Section 3. It is used by other parties in order to generate the keys for encrypting messages intended for *n*. Public keys are never re-used once a user leaves a chat.

Sent sequence number: Every user keeps a record of a received sequence number *seq_s* for every other user. seq_s_n an integer that starts at 1 and increases by 1 after any successful sending of a message to a user *n*. Clients are responsible for preventing sequence number integer overflows.

Received sequence number: Every user keeps a record of a received sequence number *seq_r* for every other user. seq_r_n an integer that starts at 1 and increases by 1 after any successful reception of a message from a user *n*. Clients are responsible for preventing sequence number integer overflows.

Chat: A chat session that contains one or more users, identified by a unique chat name matched by the regular expression `^[a-z0-9]{0,32}$`. After the conversation has no new entries for 60 minutes, all chat data is securely deleted by the server, and the chat name may be later used to instate another chat.

Secure deletion: Secure deletion of a chat session by the server requires that the chat data be overwritten with seven passes:
`0xF6, 0x00, 0xFF, random, 0x00, 0xFF, random`

Line 0: A part of the chat session buffer (be it a text file, database, etc.) dedicated to storing the nicknames and keys of all current users in the chat in the following format (for a chat with *n* participants):
$nick_1$`:`$pubkey_1$`|`$nick_2$`:`$pubkey_2$`|[...]`$nick_n$`:`$pubkey_n$`|`

ISO10126 padding: The last ciphertext block is padded with random bytes, with the last byte defining the padding boundary. For example, the following last block (shown in hexadecimal, with `CC` indicating ciphertext bytes) required padding for 6 bytes:
`CC CC CC CC CC CC CC CC CC CC 6E 61 64 69 6D 06`

3. Identification and Key Agreement

Cryptocat relies on an Elliptic Curve 25519 key agreement scheme[3] with a base point of 9. Generating the private key

[1] Bruce Schneier, Niels Ferguson, and Tadayoshi Kohno. Cryptography Engineering. 1st ed. John Wiley & Sons, 2010.

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

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

prikey relies critically on a strong random number generator, seeded using reliable levels of entropy.

Each user's *prikey* is stored within their client and is never shared. The user's public key, *pubkey*, is then generated as follows: *pubkey = scalarMult(prikey, basePoint)*

After being generated, *pubkey* is communicated to the server in the following format, with *nick* as the user's chosen nickname:

`nick:pubkey|`

The server is responsible of storing the nicknames and keys of all current participants in the chat inside Line 0.

Once a user *n* logs out of the chat, $nick_n$:$pubkey_n$| is deleted from Line 0. The client is responsible for receiving the list of keys and storing them in an accessible manner for use in encryption. The following regular expression may be used to verify any `nick:pubkey|` input:

`^[a-z]{1,12}:(\w|\/|\+|\?|\(|\)|\=)+\|$`

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, clients display a critical warning against communicating with the user.

If a new user chooses the same nickname as an existing user, the server must refuse the user's nickname and public key and return the string `inuse` to the client, which prompts the user to enter a different nickname. Similarly, if the chat contains more users than the server is willing to handle per chat, the string `full` is returned and the user's nickname and public key are refused. If a user's nickname does not match the regular expression `^[a-z]{1,12}$`, the string `error` is returned, and the client prompts the user to enter another nickname.

4. Authentication

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

$fprint_n$ = `Whirlpool(`$nick_n$ + $pubkey_n$`).substr(0, 22);`

The resulting hexadecimal is split into 11 values. Each value is encoded into its ASCII equivalent, and the result is then encoded using Bubble Babble [4] to produce the fingerprint.

Here is an example:

$nick_s$ = `sprite`

$pubkey_s$ = `30Epnoenhda3Di_wILsLWAepnVjK4KGGoOyHIX2Ejo5`

$fprint_s$ = `xipav-kyzek-bydab-veroc-symiv-hogyx`

If a user's public key does not match the regular expression `^(\w|\/|\+|\?|\(|\)|\=)*$`, the client displays a warning to other users and the fingerprint is not calculated. Users may verify each other's identities simply by each user confirming his fingerprint to the other user over a trusted out-of-band channel (which may be public.)

5. Shared Secrets

A shared secret is the single key used to encrypt messages between two parties. It is extremely sensitive and should never be exposed. To calculate the shared secret between Alice and Bob and establish a static Elliptic Curve Diffie-Hellman-Merkle key exchange (with key pairs $prikey_a$, $pubkey_a$ and $prikey_b$, $pubkey_b$ respectively,) Alice uses the following formula:

$$secret_{ab} = Whirlpool(base64(scalarMult(prikey_a, pubkey_b)))$$

Bob uses the following formula:

$$secret_{ab} = Whirlpool(base64(scalarMult(prikey_b, pubkey_a)))$$

Both parties will obtain the same shared secret $secret_{ab}$, since:

$$scalarMult(prikey_a, pubkey_b) = scalarMult(prikey_b, pubkey_a)$$

Before being used, the shared secret is hashed using Whirlpool, producing a 512-bit hexadecimal value. The first 256 bits of $secret_{ab}$ are used as the encryption key for AES-CBC-256 operations. The ciphertext is padded according to ISO 10126.

6. HMAC Generation

Message authentication codes are generated by running the ciphertext through HMAC-WHIRLPOOL. The sender *a* uses the last 256 bits of $secret_{ab}$ (treated as a 64 character string of hexadecimal values) plus seq_s_b as the key for creating the HMAC. The recipient *b* uses the last 256 bits of $secret_{ab}$ plus seq_r_a. as the key for verifying the HMAC.

Upon the successful verification of each HMAC, the message is decrypted using the shared secret. If the HMAC check fails, an error is displayed and the message is discarded.

7. Joining and Leaving

A user *n* may join the chat by sending the chat name, $nick_n$ and $pubkey_n$ to the server. Once a user joins with nickname $nick_n$, the server must append the following line to the chat (join and part notifications are not encrypted):

`> `$nick_n$` has arrived`

[4] http://wiki.yak.net/589/Bubble_Babble_Encoding.txt

Conversely, upon a user leaving, the following line is appended to the chat:

< $nick_n$ has left

The parting notification is followed by a removal of $nick_n$'s entry in Line 0. The following regular expression may be used to identify these lines by the client:

```
^(\&gt\;|\&lt\;)\s[a-z]{1,12}\s(has arrived|has left)$
```

Upon receiving a line that matches the above regular expression, the client requests the new `nick:pubkey|` list from the server (Line 0.) Note that the above regular expression assumes that the characters '>' and '<' have been converted to their HTML entities for safe usage within a web browser. The client does not depend on join/part notifications in order to assess the current users inside a chat, but depends exclusively on the contents of Line 0. If a `nick:pubkey|` is determined to have been removed, the client deletes the removed user's public key, shared secret, *seq_s* and *seq_r* numbers and fingerprint from its records.

8. Messaging

In a conversation with participants Alice, Bob, and Carol, a single message sent by Alice is formatted as such:

alice|N1OlZwFw: [:3](bob)$message_{bob}$|$hmac_{bob}$(carol)$message_{carol}$|$hmac_{carol}$[:3]

1. *alice* is the sender's nickname.
2. *N1OlZwFw* is an example of a randomly generated alphanumeric nonce used to confirm message reception. These nonces must match the regular expression `^\w{8}$`.
3. [:3]s signal the start and end of encrypted content.

(bob) signals that $message_{bob}$|$hmac_{bob}$ are meant for Bob. $message_{bob}$ is encrypted using the shared secret derived from Bob's public key and Alice's private key, while $hmac_{bob}$ is the HMAC intended to verify the authenticity of $message_{bob}$. AES ciphertext is sent in base 64, while the HMAC-WHIRLPOOL hash is sent in hexadecimal. The randomly generated IV for AES-CBC is appended as the first 16 bytes of the ciphertext.

Once Alice sends the message (as formatted above) to the server, she increments both seq_s_{bob} and seq_s_{carol} by 1 (as discussed in Section 5.) The server sanitizes the message using the following regular expression:

```
^[a-z]{1,12}\|\w{8}:\s\[:3\]((\w|\/|\+|\?|\(|\)|\=)*\|(\d|a|
b|c|d|e|f){128})*\[:3\]$
```

The server processes the message and sends the following to Bob...

alice: [:3]$message_{bob}$|$hmac_{bob}$[:3]

...and the following to Carol:

alice: [:3]$message_{carol}$|$hmac_{carol}$[:3]

Upon the reception of the message from Alice, both Bob and Carol increase their seq_r_{alice} by 1. Recipient clients may sanitize messages using the following regular expression:

```
^[a-z]{1,12}\:\s\[:3\](\w|\/|\+|\?|\=)*\|(\d|a|b|c|d|e|f)
{128}\[:3\]$
```

The server finally sends Alice the message nonce (`N1OlZwFw`) in order for her client to validate message reception. It is crucial (from a security standpoint) for the messages to be delivered in the sequence intended by the sender's client. The usage of seq_s_n and seq_r_n numbers attempts to mitigate against delayed, rearranged or dropped messages, and will invalidate HMAC authentications if messages are not received in the right order. Therefore, we recommend that clients be designed to not send further messages until they receive the message nonce for the currently queued message. If a connection error is detected, the currently queued message may be resent.

Cryptocat also supports **private messaging**, which allows for a message to be sent to a single recipient even in a chat with more than two parties. If the sender prefixes their message with an '@' followed by the recipient's nickname *(@carol hey, how's it going?)*, the client automatically encrypts, MACs and sends the message only to the intended recipient (thus formatting the server request as if the chat contained only Carol as a recipient.) The recipient's client then notices the '@' followed by the recipient's nickname upon message reception, and notifies the user that they have been sent a private message.

Files may be sent simply by converting them into base 64 encoded Data URIs and sending the Data URI as a private message. The client detects the Data URI header and interprets the message as a file accordingly. We do not recommend allowing the protocol to send or receive any file except .zip files and certain types of image files. This is to mitigate the possible security concerns related to client access to file extensions with a history of vulnerabilities, such as .pdf and .swf.

9. Endnotes

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