Basic Encryption & Authentication Protocol for RPCh

Authors: Lukas Pohanka, Robert Kiel
Version: 0.2.1

Situation

RPCh is used to route Ethereum transactions from a wallet through the HOPR network to preserve privacy (mostly anonymity through hiding metadata of the transaction sender).

There are the following parties in the current RPCh infrastructure:

- Wallet
- RPCh Client
- HOPR Entry node
- HOPR Intermediate node
- HOPR Exit node
- RPCh Exit node
- Discovery Platform
- Final RPC provider

The Wallet software uses RPCh client API to format the signed transaction as a message that's accepted by the RPCh Exit node. It queries the Discovery platform (public service) for available HOPR Exit nodes which also offers the RPCh Exit node functionality. It also queries the Discovery Platform for available HOPR Entry node. RPCh API then sends the formatted message via the HOPR entry node over HOPR network (where HOPR intermediate nodes act as relays) to the HOPR Exit node. Upon reception by the HOPR exit node, the data is handed over to the RPCh Exit node which reformats the data and sends the transaction to the Final RPC provider. The message sent to the HOPR Entry node is currently not encrypted nor authenticated. The Discovery platform is implemented using an on-chain smart contract, and the data it stores are immutable.

Threat model

- The User **trusts** the Wallet. The user wants to hide the *Information* contained in the transaction from all untrusted nodes in the RPCh infrastructure.
- The Discovery Platform is a third party **trusted** by the User. The *Information* is never reaching it in any form.
- RPCh Client runs in the same environment as the Wallet and must be therefore **trusted** by the User.
- The HOPR Entry nodes nor the HOPR Intermediate nodes **are not to be trusted** by the User.

- The Entry Node chooses the middle node, the node after Entry node and before Exit node, thus it needs to be **trusted** in terms of not revealing the correlation between incoming messages and the chosen path.
- The Final RPC provider executes the transaction based on the *Information* and must be therefore **trusted** by the user.
- The RPCh Exit Node delivers the *Information* to the Final RPC provider, and therefore must be **trusted** by the User.
- The HOPR Exit node runs in the same environment as RPCh Exit node and therefore must be also **trusted** by the User.
- Due to the HOPR protocol already ensuring message confidentiality, integrity and authentication, the *Information* is never disclosed to any HOPR Intermediate nodes while in-transit.

The Problem

As outlined in the above threat model, the only *untrusted party* that can read, re-use, change or otherwise interpret the Information is the HOPR Entry Node. The idea of this document is to propose a protocol so that the Information is hidden from the HOPR Entry node and it cannot act maliciously based on the Information, namely:

- HOPR Entry node cannot read the Information
- HOPR Entry node cannot modify the Information
- HOPR Entry node cannot re-send an Information it has previously received from the User's RPCh Client (replay attack)

The above requirements mean that the designed protocol must establish message confidentiality, integrity and authenticity.

Proposed Protocol

Definitions

The || sign denotes the byte sequence concatenation. Unless explicitly stated otherwise, we use the Big Endian byte ordering (= most significant byte first) when interpreting numbers as bytes.
The `|X|` denotes the size of `X` in bytes (when serialized).
In a special case, constants `|K|` and `|IV|` denote the known fixed key size or the fixed initialization vector size respectively, of the chosen symmetric cipher.

The aim of the protocol is to achieve at least 128-bit security, meaning that breaking the protocol should be at least as hard as e.g. finding a collision (second pre-image attack) of an ideal 256-bit long cryptographic hash function.
Here's the list of the general cryptographic primitives used in the protocol:

- $ENC_K(IV,M) = (C,T)$ denotes authenticated encryption of a plain-text message M, using a secret key K and a unique initialization vector IV. The resulting product is a cipher-text C and an authentication tag T. The additional authenticated data (AAD) that might be supported by such an encryption scheme is omitted for the purpose of this protocol.
- $DEC_K(IV,C,T) = (M, \{0,1\})$ denotes decryption and verification of a given message M, authentication tag T, initialization vector IV and a secret key K. The resulting product is the alleged plain-text message M and an indicator whether the obtained message is authentic (1) or not (0).
- $ECDH(s,P) = Q$ denotes Elliptic Curve Diffe-Hellman step, ie. multiplication of a point P on a common elliptic curve by a secret scalar s, resulting in a new point $Q = sP$.
- $KDF_i(Q,t,n) = s$ denotes a key derivation function with index i, applied to an elliptic curve point Q and additional tag t. The result is a byte string s of length n bytes.
- $GEN() = (s,P)$ generates a random valid keypair on an elliptic curve (compatible with $ECDH$) using a cryptographically secure random number generator. The result is a private key s and a corresponding public key P.
- $COMP(P) = b$ denotes an elliptic curve point serialization function that compresses the given elliptic curve point P into a string of bytes (e.g. this is typically consisting of a sign bit and X coordinate of P).

To meet the minimal security requirements mentioned above, we will assume a secure elliptic curve over a prime field with the size of the prime ~ 2^{256}. We will also assume a symmetric cipher that accepts at least 128-bit secret key and an IV of at least 12 bytes. We assume the symmetric cipher does not require padding of the input message or that it is taken care of automatically in its implementation (in case the instantiation is a block cipher).

The current version byte is denoted as Ver ($|Ver| = 1$) and its value is $0x11$. The second nibble of the byte denotes the ciphersuite version, the first nibble denotes the protocol version. If a different instance of this protocol uses a different combination of cryptographic primitives, it must use a different value of the second nibble, e.g. $0x12$. Protocol modifications which are not backwards compatible must use a different first nibble, e.g. $0x21$.

For concrete examples of instantiations of the above cryptographic primitives, so that the security requirements are fulfilled, see the Instantiation section below.

Initial stage

- Each RPCh Exit Node N generates an elliptic curve keypair, such that:

$$(s_N, P_N) = GEN()$$

and stores the P_n in the Discovery Platform, so it is undeniably linked with some of its identifiers (e.g. peer ID).
- The RPCh Client also maintains a monotonic counter C of messages sent to a given RPCh Exit node. We assume the size of the counter to be at least 64-bits.
- Likewise, the RPCh Exit node maintains the counter value of the last message it received from a given RPCh client.
- We assume that the RPCh Exit node knows the Peer ID of the RPCh Entry node (it is embedded in the message). This due to the current state of the HOPR protocol missing the privacy-preserving return path implementation.

Request construction stage

This stage is executed on a system running the Wallet software with the RPCh client.

A new transaction request is given by the Wallet to the RPCh client and it is pre-formatted in the the format used by RPCh. We will denote this pre-formatted message as M.
Let's assume the RPCh client has already chosen an RPCh Exit node N with an identifier[1] ID_{exit} and verifiably queried its public key P_N from the Discovery Platform.

The RPCh Client then proceeds with generating parameters for the message transformation.

- The RPCh Client generates a new random elliptic curve key pair, computes the Diffie-Hellman step using the RPCh Exit node's public key to generate a per-message secret pre-key.

$$(q_M, Q_M) = GEN()$$
$$S_{pre} = ECDH(q_M, P_N)$$

- Next, it retrieves the counter value C for the selected RPCh Exit node and generates parameters required for a symmetric authenticated encryption of the message M.

$$K_M = KDF_1(S_{pre}, Ver \;||\; ID_{exit} \;||\; \text{"req"}, |K|)$$
$$IV_1 = KDF_2(S_{pre}, Ver \;||\; ID_{exit} \;||\; \text{"req"}, |IV| - |C|)$$
$$IV_M = IV_1 \;||\; C$$

- Then it proceeds with encrypting the actual message M to produce a ciphertext C and authentication tag T.

$$(R, T) = ENC_{KM}(IV_M, M)$$

[1] For example a peer ID of the associated HOPR Exit Node, as they reside on the same system.

- Finally, the RPCh Client formats the final request message `Req` that is then delivered to the HOPR Entry node.

$$W = COMP(Q_M)$$
$$Req = Ver \;||\; W \;||\; C \;||\; R \;||\; T$$

- RPCh client sends `Req` using the HOPR Entry node. Via the HOPR protocol it gets delivered to the destination RPCh Exit node.
- Note, that with the assumed security parameters of the used primitives, the size of the request `Req` is 58 (`|Ver| = 1, |W| = 33, |C| = 8, |T| = 16`) plus the size of the ciphertext `R`. If the underlying symmetric cipher does not require input message padding, the size of `Req` will be exactly equal `58 + |M|`.

Request reception stage

The following stage is executed on the RPCh Exit node, after delivery of the RPCh message `Req`. Since the HOPR protocol at its current state includes the Peer ID of the HOPR Entry node (sender) into the message (missing return path implementation), we assume the RPCh Exit node knows the Peer ID ID_{entry}.

- The RPCh Exit node `N` (with identifier ID_{exit}) decomposes `Req` into the individual parameters (knowing their size is fixed). If `Ver` has a value it supports, it decompresses the EC point Q_M. Then it combines its own private key s_N with the decompressed point to recover the secret per-message pre-key S_{pre}. If the version given by `Ver` is not supported by the RPCh Exit node, the sender is notified via the upper protocol layer.

$$(Ver, W, C, R, T) = Req$$
$$Q_M = COMP^{-1}(W)$$
$$S_{pre} = ECDH(s_N, Q_M)$$

- Then the RPCh Exit node uses S_{pre} to compute the per-message parameters needed for the decryption and verification of the ciphertext. `ID` is an identifier of the RPCh Exit node.

$$K_M = KDF_1(S_{pre}, Ver \;||\; ID_{exit} \;||\; \text{“req”}, |K|)$$
$$IV_1 = KDF_2(S_{pre}, Ver \;||\; ID_{exit} \;||\; \text{“req”}, |IV| - |C|)$$
$$IV_M = IV_1 \;||\; C$$

- Next, it decrypts the ciphertext and obtains the validation result `V` and recovers `M`.

$$(M, V) = DEC_{KM}(IV_M, R, T)$$

- The RPCh Exit Node discards the message M (checks done exactly in the following order) if :
    a. V is equal to 0 (the authentication tag validation failed)
    b. The RPCh Exit node retrieves from its persistent storage the last seen value of the counter C_{last} for the sender with ID_{entry}. If $C <= C_{last}$, the message is discarded (replay attack protection).
- If the message has not been discarded, the RPCh Exit node stores the new value of C by replacing the C_{last}. Then it proceeds with processing the message M as usual, eventually forwarding it to the Final RPC provider.

Response construction phase

After the RPCh receives the result U of the transaction sent to the Final RPC provider, it begins constructing the response which it sends back to the RPC Client (via HOPR network).

We assume that U has already been formatted to the RPCh message format.

- The RPCh Exit node increments the value C of the counter stored for the given sender by 1. It also persists this updated counter value.

- The RPCh Exit Node generates a new set of per-message parameters to encrypt U.

$$K_U = KDF_3(S_{pre}, Ver \,||\, ID_{entry} \,||\, \text{“resp”}, |K|)$$
$$IV_2 = KDF_4(S_{pre}, Ver \,||\, ID_{entry} \,||\, \text{“resp”}, |IV|-|C|)$$
$$IV_U = IV_2 \,||\, C$$

- Then it proceeds with encrypting the response message U to produce a ciphertext C and authentication tag T_2.

$$(R_2, T_2) = ENC_{KU}(IV_2, U)$$

- Lastly, the RPCh Client formats the final request message $Resp$ that is then delivered to the HOPR Entry node and subsequently to the RPCh Client.

$$Resp = C \,||\, R_2 \,||\, T_2$$

- The size of $Resp$ is $24 + |C|$. Note that W int the response was omitted, since the entire session state between RPCh Client and RPC Exit node is kept and not discarded. This could save 33 bytes in the payload (which could be used by R_2) and it is safe to use, since we already need to trust the RPC Exit node. Another 1 byte is saved due to version being omitted (assumed constant per session).

Response reception phase

This is the last phase that is executed on the RPCh Client upon the reception of the response `Resp`.

- After decomposing `Resp` into the individual parameters, the RPCh Client recovers the per-message parameters.

$$(C,\ R_2,\ T_2) = Resp$$
$$K_U = KDF_3(S_{pre}\ ,\ Ver\ ||\ ID_{entry}\ ||\ \text{“resp”},\ |K|)$$
$$IV_2 = KDF_4(S_{pre}\ ,\ Ver\ ||\ ID_{entry}\ ||\ \text{“resp”},\ |IV|-|C|)$$
$$IV_U = IV_2\ ||\ C$$

- Next, it decrypts the ciphertext and obtains the validation result `V2` and recovers `U`.

$$(U,\ V_2) = DEC_{KU}(IV_U,\ R_2,\ T_2)$$

- The RPCh Client discards the message `U` (checks done exactly in the following order) if :
  a. V_2 is equal to 0 (the authentication tag validation failed)
  b. The RPCh Client retrieves from its persistent storage the last seen value of the counter C_{last} for the sender with ID_{exit}. If $C <= C_{last}$, the message is discarded (replay attack protection).
- If the message has not been discarded, the RPCh Client stores the new value of `C` by replacing the C_{last}. Then it proceeds with processing the message `U` as usual, eventually forwarding it to the Wallet.

Edge cases

TBD

- HOPR Exit node public key compromise & revocation
- Counter overflow (pretty hard to achieve if `|C| = 8`)
- …

Instantiation

Here is a concrete and recommended instantiation of the cryptographic primitives:

- `Ver = 0x11`
- `ENC`, `DEC` is Chacha20 with Poly1305. It accepts the key size of 256-bits and IV size is 96-bits. It preserves the size of the plaintext/ciphertext.
- `ECDH` is currently based on secp256k1 due to easier compatibility with the Smart Contracts. However, we would ideally in future want to use X25519, a concrete Elliptic curve Diffie-Hellman instantiation over Curve25519 with the prime field of size $2^{255} - 19$.
- `COMP` and `GEN` are based on Curve25519.

- `KDF` is Hmac-based Key Derivation Function (HKDF) with Blake2s256 as hash function. Alternatively, SHA-3 finalists are also suitable when the input parameters are simply concatenated and given as input into the hash function for a single hashing.