Basic Encryption & Authentication Protocol for RPCh

Authors: Lukas Pohanka, Robert Kiel
Version: 0.3.0
Last Updated: 12.12.2022

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:

- $\text{ENC}_K(\text{IV},\text{M}) = (\text{C},\text{T})$ 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.
- $\text{DEC}_K(\text{IV},\text{C},\text{T}) = (\text{M}, \{0,1\})$ 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).
- $\text{ECDH}(s,P) = Q$ 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 $Q = sP$.
- $\text{KDF}_i(Q,t,n) = s$ 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.
- $\text{GEN}() = (s,P)$ 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.
- $\text{COMP}(P) = b$ 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 ($|\text{Ver}|$ = 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 maintains a monotonic counter C_{req} of requests sent to a given RPCh Exit node and also C_{resp_last}, which is the counter value last seen on a response from a given RPCh Exit node. Similarly, the RPCh Exit nodes maintains counter C_{req_last} which is the counter value last seen on a request from a given RPCh Client and C_{resp} as a counter of responses sent to a given RPCh Client. We assume the size of the counters to be the same and at least 64-bits, ie. $|C| = |C_{req}| = |C_{resp}| = 8$.
- 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_{req} for the selected RPCh Exit node and generates parameters required for a symmetric authenticated encryption of the message M.

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

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

- 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)
```

- 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 || $C_{req}$ || 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`, $|C_{req}| = 8$, `|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.

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

- 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_{req_last} for the sender with ID_{entry}. If $C_{req} <= C_{req\_last}$, the message is discarded (replay attack protection).
- If the message has not been discarded, the RPCh Exit node stores the new value of C_{req} by replacing the C_{req_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_{resp} 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.

$$
\begin{aligned}
K_U &= KDF_3(S_{pre}, Ver \,||\, ID_{entry} \,||\, \text{"resp"}, |K|) \\
IV_2 &= KDF_4(S_{pre}, Ver \,||\, ID_{entry} \,||\, \text{"resp"}, |IV|-|C|) \\
IV_U &= IV_2 \,||\, C_{resp}
\end{aligned}
$$

- Then it proceeds with encrypting the response message U to produce a ciphertext R_2 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.

$$\mathtt{Resp} = C_{resp} \mid\mid R_2 \mid\mid T_2$$

- The size of `Resp` is $24 + |C_{resp}|$. 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.

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

- 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_{resp_last} for the sender with ID_{exit}. If C_{resp} <= C_{resp_last}, the message is discarded (replay attack protection).
- If the message has not been discarded, the RPCh Client stores the new value of C_{resp} by replacing the C_{resp_last}. Then it proceeds with processing the message `U` as usual, eventually forwarding it to the Wallet.

Use of UTC timestamps as Counters

In general, if time is assumed to be reasonably synchronized and monotonic across Exit and Client nodes, the usage of UTC Timestamp (with millisecond precision) is possible to be used instead of C_{req} and C_{resp}.

The following steps are then different:

- In the Request construction phase, the RPCh Client uses the current UTC timestamp as C_{req} . instead of an incremented C_{last_req} value (it would have retrieved from its persistent storage).
- In the Request reception phase, the RPCh Exit node verifies that received C_{req} is strictly greater than C_{last_req} it has stored and also verifies that the received C_{req} is not in the future compared to its current UTC time (within some reasonable tolerance). If the verification passes, the Exit node updates C_{last_req} with the C_{req} as usual.
- In the Response construction phase, the RPCh Exit node uses the current UTC timestamp as C_{resp} instead of an incremented value C_{last_resp} (it would have retrieved from its persistent storage).
- In the Response reception phase, the RPCh Client node verifies that the received C_{resp} is strictly greater than the C_{last_resp} it has stored and also verifies that the received C_{resp} is not in the future compared to its current UTC time (within some reasonable tolerance).

Edge cases

Loss of the counter

There are situations where the stored counter values C_{last_req} and C_{last_resp} can be lost. This is more likely to happen on the RPCh Client node, rather than on the RPCh Exit node.

Loss on RPCh Exit node

The loss of the C_{last_req} on the RPCh Exit node has greater security implications, since then a malicious Entry node could easily take some past messages it has seen and replay that to the RPCh Exit node.
This is more easily mitigated when Timestamps are being used as Counters. In case of the loss, the Exit node sets the C_{last_req} and C_{last_resp} to the current UTC timestamp. This can have a slightly detrimental effect on RPCh Client requests being rejected in that single point in time, but will prevent the malicious Entry nodes from replaying old recorded requests.

When non-Timestamp based Counters are being used, the security against a replay attack relies on the RPCh Client sending an honest Request first than a replayed Request sent by a malicious Entry node (if C_{last_req} has been lost).

The loss of C_{last_resp} has less security implications, because from the RPCh Client perspective, the request and response are tied together via the unique EC keypair generated during the Request construction phase. Therefore, the Response freshness guarantees are far stronger for the RPCh Client. In case of loss, the RPCh Exit node can set C_{last_resp} to an arbitrarily high value it “knows” (out of band) it has not used before. The selection of such value is left for the implementer to decide.

Loss on RPCh Client node

Since, the received responses are tied to the requests by a unique EC keypair (generated during Request construction phase) and a session is established per request/response roundtrip, the RPCh Client has guaranteed that the received response belongs to the request it has sent earlier.

The loss of either C_{last_req} or C_{last_resp} on the RPCh Client has less detrimental effect for security than on the RPCh Exit node.

In case of loss, the RPCh Client node can set C_{last_req} to an arbitrarily high value it “knows” (out of band) it has not used before. The selection of such value is left for the implementer to decide.

Because of the above mentioned request/response tieing guarantees, the lost value of C_{last_resp} can be set to 0 and will be updated once the next response is received.

The case is mitigated more easily when Timestamps are being used as Counters. In case of loss, the C_{last_req} and C_{last_resp} will be set to the current UTC timestamp value.

Other edge cases

TBD

- HOPR Exit node public key compromise & revocation
- Counter overflow (pretty hard to achieve if $|C| = |C_{resp}| = |C_{req}| = 8$)
- When the return path is added to the HOPR protocol, response construction needs to be adjusted, because the exit node might no longer know the Entry node’s peer ID.
- …

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.