Preventing Metadata Leakage in Communication over Public Channels

Introduction

With zero-knowledge proofs and the like becoming ever-more prevalent in the modern, web3-ified Internet, there arises the need for a new type of private messaging protocol: one that preserves the privacy of its users to the fullest extent possible without the need to trust any third party.

Current implementations of privacy-focused messaging protocols suffer from a number of problems, ranging from metadata leakage (whether the identities of correspondents, the timestamp of message transmission or receipt) to usability issues (such as the need to be online to send and receive messages).

The protocol described in this paper primarily attempts to address two main areas:

1. Metadata leakage. As described below, the protocol attempts to conceal as much metadata as possible.
2. Efficiency. The protocol supports $O(1)$ space complexity for broadcast transmissions.

Metadata

The goal of this paper is to build upon the work of previous protocols to hide even more metadata about conversations. In particular, we will hide the following information, in addition to hiding of the payload itself:

- Sender's identity
- Sender's location (geographical and network)
- Timestamp of transmission
- Receiver's identity
- Receiver's location (geographical and network)
- Timestamp of receipt
- Payload size
- Conversation history
- Forward secrecy

However, privacy of these data is not sufficient to make a usable protocol. We will also ensure the following features:

- Multi-device messaging per-user, with message synchronization across multiple devices.

- Scalable group messaging.
- Inexpensive to run.

Protocol

The key insight of this paper is the channel and sequence hash construction.

A channel is a total-ordered stream of messages. It consists of membership list (e.g. a sender and a receiver, or multiple members of a group) and a shared secret (derivable by Diffie-Hellman or any other method of establishing a shared secret).

A channel has a deterministic sequence hash generator. Hashing the fields of the channel, together with a nonce, produces a "sequence hash." This sequence hash can only be recreated by parties privy to the channel's shared secret, yet it does not reveal the shared secret. The sequence hash serves as the identifying key of a message sent to the message repository.

The message repository is a simple construct, consisting only of a key-value store from which anyone can read, and to which anyone can write. The only restriction is that existing keys cannot be overwritten.

Using a series of abstractions, the channel construct can be composed in a number of different ways, supporting bidirectional $1 \leftrightarrow 1$ messaging, $1 \leftrightarrow N$ messaging, and $N \leftrightarrow M$ messaging.

Of particular interest is $N \leftrightarrow M$ messaging. If all members of a particular group share the same secret, they can all generate the same sequence of hashes, and decrypt all of the messages in the channel. This allows for posting a single message, encrypted with the shared secret, to the public message repository, and all of the members of the group will be able to read it, regardless of the number of members in the group. Thus, we have $O(1)$ space complexity for broadcast transmissions.

Channels

Consider a unidirectional channel c, with sender s and receiver r, where the public keys of r and s are mutually known. Using Diffie-Hellman, or any other method of establishing a shared secret, s and r can establish a shared secret k. In combination with other, optional, static metadata, a similarly secret channel identifier $i_{s \to r}$ can be derived. An example of this process is demonstrated below:

$$i_{s \to r} = H(k, s \parallel r)$$

where H is a keyed cryptographic hash function (such as HMAC-SHA3-512), s is the public key of the sender, r is the public key of the receiver, and k is the shared secret.

The channel identifier $i_{s \to r}$ is used to identify the channel c. However, this identifier must remain secret, as it can be used to derive the sequence hashes for the channel before they have been posted to a public message repository.

Generating sequence hashes

Once a channel has been established, the sender and receiver can generate a sequence hash $h^n_{s \to r}$ for the nth message in the channel. This sequence hash is used to identify the message in the message repository. The sequence hash is generated as follows:

$$h^n_{s \to r} = H(i_{s \to r}, n)$$

where n is the sequence number of the message.

Posting and reading messages

Once a sequence hash has been generated, the sender can post a payload $(h^n_{s \to r}, c)$ to the message repository, where c is the ciphertext of the message, encrypted with the shared secret k. Once the message has posted, the receiver, knowing the shared secret k and the index of last message he saw $n - 1$, can find the message keyed by $h^n_{s \to r}$ in the message repository, and decrypt it.

Development

A reference implementation for this paper is currently in development. The following is a proposal for the development of the protocol.

Phase 1: Proof of Concept

The basic proof of concept has already been completed and shown to work in a limited capacity. The proof of concept is a simple command-line application that allows for $1 : 1$ messaging between two parties. The proof of concept is written in Rust, and uses the *Dalek* libraries for cryptographic primitives, as well as *ChaCha20-Poly1305* for encryption.

Phase 2: Multi-Device & Group Messaging

The next phase of development will focus on multi-device messaging and group messaging. This will be implemented in the proof of concept, and will be written in Rust.

Phase 3: Library

The final phase of development will be to create a library that can be used by other applications. This will be written in Rust, and will be published to crates.io.