Group chat specification

Threebit Hacker
David Stainton

Abstract

Prerequisites

This design specification is dependent on the BACAP and Pigeonhole protocol designs from our paper Echomix: A Strong Anonymity System with Messaging [https://arxiv.org/abs/2501.02933], which describes

- the BACAP (blinded and capability) in §4 and
- the Pigeonhole protocol in §5.

The source code [https://github.com/katzenpost/hpqc/blob/main/bacap/bacap.go] and API docs [https://github.com/katzenpost/hpqc/blob/main/bacap/bacap.go] for BACAP are available on pkg.go.dev.

Introduction

The Pigeonhole protocol establishes anonymous cryptographic communication channels which have a *readcap* (read capability) and a *writecap* (write capability). For example, if Alice and Bob want to communicate, they can each create their own Pigeonhole/BACAP channels and exchange readcaps on those channels. Now when Bob writes to his channel, Alice can read those messages because she has Bob's readcap. Likewise, when Alice writes to her channel, Bob can read those messages because he has Alice's readcap. This is the most basic construction using BACAP and Pigeonhole.

Here we extend this basic design to work as a minimal group-chat protocol, without key rotation.

BACAP primitives give us two message types:

- `SingleMessage`
- `AllOrNothingMessage` (used for big messages: upload *n* chunks to a temporary stream and then put a pointer to that in your own stream as a single message.)

The group state consists of:

- a `MembershipCap` for each member, containing:
  - a BACAP readcap
  - a nickname
- a `MembershipHash` (a hash over all of the MembershipCaps)

The group chat is completely decentralized. Each member must keep track of every other member.

Group chat message types

All messages are `SingleMessage` if they fit in one BACAP slot, or an `AllOrNothingMessage` if they are too big.

- `Text` type payloads are normal chat text messages.

```
// TextPayload encapsulates a normal text message.
type TextPayload struct {
    // Payload contains a normal UTF-8 text message to be displayed inline.
    Payload []byte
}
```

- `Introduction` type messages introduce new group members.

```
// Introduction introduces a new member to the group.
type Introduction struct {
    // DisplayName is the party's name to be displayed in chat clients.
    DisplayName string

    // UniversalReadCap is the BACAP UniversalReadCap
    // which lets you read all messages posted by this user.
    UniversalReadCap *bacap.UniversalReadCap
}
```

- `FileUpload` type

A `FileUpload` can be used for various purposes such as uploading an image to be displayed inline by the chat client. Likewise, a sound bite could be made visible in the chat along with a play-button. Beyond that, we can support arbitrary file attachments.

```
// FileUpload encapsulates several file types
// which result in different client behaviors.
type FileUpload struct {
    // Payload contains the file payload.
    Payload []byte

    // FileType is the identifier for each file type.
    // Valid file types are:
    // "image"
    // "sound"
    // "arbitrary"
    FileType string
}
```

- `Who` type

The `Who` message type is used to query who is currently in the group.

```
// Who is used to query the group chat to find out the member read capabilities
type Who struct {}
```

- `ReplyWho` type

The `ReplyWho` message answers the Who query with an `AllOrNothingMessage` BACAP stream containing readcaps for all group chat members.

```
type ReplyWho struct {
    Payload *bacap.BacapStream
}
```

- `GroupChatMessage` type

The `GroupChatMessage` message encapsulates all of the above-mentioned message types and is serialized with CBOR.

```
// GroupChatMessage encapsulates all chat message types.
type GroupChatMessage struct {
    // Version is used to ensure we can change this message type in the future.
    Version int

    // MembershipHash is the hash of the user's PleaseAdd message.
    MembershipHash *[32]byte
```

```
    TextPayload *TextPayload
    Introduction *Introduction
    FileUpload *FileUpload
    Who *Who
    ReplyWho *ReplyWho
}
```

Protocol flow

The protocol flow for making a new group from scratch (using whatever authentication protocol) is essentially for everybody to exchange `PleaseAdd` messages.

```
// PleaseAdd is a message used by a client to try and gain access to a chat grou
type PleaseAdd struct {
    // DisplayName is the party's name to be displayed in chat clients.
    DisplayName string

    // UniversalReadCap is the BACAP UniversalReadCap
    // which lets you read all messages posted by this user.
    UniversalReadCap *bacap.UniversalReadCap
}

type SignedPleaseAdd struct {
    // PleaseAdd contains the CBOR serialized PleaseAdd struct.
    PleaseAdd []byte

    // Signature contains the cryptographic signature over the PleaseAdd field.
    Signature []byte
}
```

For introduction to an existing group over an existing channel between an introducer member and new member, an `Invitation` message is used.

```
type Invitation struct {
    GroupName string
}
```

The `Invitation` protocol flow works as follows.

1. There exists a group called YoloGroup. A member of the group invites a potential new member with an `Invitation` message.

2. If the invited party wants to join, then they reply with a `SignedPleaseAdd` message meaning "I want to join your group." This provides the invited party's BACAP universal readcap, their display name, and a cryptographic signature produced by their BACAP writecap.

3. The introducer receives the `SignedPleaseAdd` message.

    a. If the introducer does not like the DisplayName, they reply to the invited party with a `PleaseReviseDisplayName` message that contains the original `SignedPleaseAdd`. Then they wait for a new `SignedPleaseAdd`.

    b. If the introducer approves of the `DisplayName`, then:

    - Because existing members need the new member's readcap, the introducer publishes the `SignedPleaseAdd` to their own BACAP stream for the rest of the group to read.

    - Because the new member needs existing members' readcaps, the introducer replies to the new member with `ReplyWho` message containing readcaps for all existing members.

IMPORTANT: The content of both replies must be sent in the same `AllOrNothingMessage`, despite the `SignedPleaseAdd` being written to the introducer's own BACAP stream for the group and the `ReplyWho` being written to the BACAP stream the introducer is using to communicate with the new member.

NOTE: We could send all of this information as part of the initial `Invitation`, but that would allow silent members to read other members' streams without them knowing it, which is an anti-goal.

Addenda

GOOD QUESTION: If we are adding a lot of people at once,do we really need to upload all of the members *n* times?

FUTURE WORK: Forward secrecy. We can add two extensions that allow transmitting public keys + stuff encrypted under those public keys. We can also refer to the Reunion protocol which is a *n*-way PAKE with strong anonymity properties. Reunion is described in Communication in a world of pervasive surveillance: Sources and methods: Counter-strategies against pervasive surveillance architecture [https://research.tue.nl/en/publications/communication-in-a-world-of-pervasive-surveillance-sources-and-me]. Currently, Python [https://codeberg.org/rendezvous/reunion/] and Go [https://github.com/katzenpost/reunion/] implementations of Reunion are available.