Scatter Map/Stream

David Stainton
threebithacker
Masala

Table of Contents

== Glossary

- Provider: the old term we are trying to phase out. It essentially means perimeter node that allows client connections AND can optionally run services that you can interact over the mixnet.
- perimeter node: that clients can connect to and may run mixnet services
- SPOF: single point of failure
- NIKE: non-interactive key exchange

== Introduction

A mixnet can have many possible messaging systems and it's important to think of a mixnet as a transport rather than a messaging system. Here we shall endeavor to replace the naive messaging system which was presented in the "Loopix Anonymity System" paper, where Alice knows the perimeter network node and spool ID that Bob will collect his messages from. And likewise Bob knows the perimeter network node and queue ID from whence Alice may receive messages. This is a terrible design for a lot of reasons one of which: if Alice and Bob are adversarial then they may wish to hide their network and geographical location from one another whereas this design prevents hiding that information.

Therefore our replacement design necessarily has the totality of message paths through the mixnet chosen by more than one client entity (the sender and the receiver). In particular, the retriever of a message chooses the path through the network that the retrieved message takes (via a Sphinx SURB) where as the Loopix paper describes messages retrieval as happening with the wire protocol and a direct conncection from client to network perimeter node.

A final introductory point to made here is that: It is not sufficient to merely solve the problem of having message paths only composed by the sender. The early versions of katzen used (and currently still use) a design where each client selects a perimeter node to queue ALL their received messages. The above problem is solved by the client not directly connecting to this perimeter node of the wire protocol but instead communicating with that node over the mixnet. Clients connect to a randomly selected perimeter node.

This is still a bad design for several important reasons: 1. the client selects a node to queue their messages, that's a SPOF 2. the client's SPOF node gets some leaked metadata.

== Metadata leaked with Old Katzen (catshadow) design

The problem with a messaging system where Alice remotely retrieves all her messages from a specific node in the mixnet, let's call it node A, that node is witness to a lot of metadata about Alice's communication. Let's enumerate some of the information that node A learns: * time Alice retreives messages * number of messages Alice receives

However if we make a few more assumptions, then node A can learn even more: * if both Alice and Bob have message spools on node A then node A can learn when they are chatting, the message sequence, number of messages sent and receives, timing of sent and retrieval.

It is worth stating that in the namenlos network currently there is only one perimeter node running the message spool service.

== Proposed solution: Scattering things

Instead of the central spool for each client, we propose a system where clients receives messages from many perimeter nodes… and this changes over time.

Operators can then monitor the polling on these instead, but it will be much harder to correlate them, and we can solve much of the polling by using an off-the-shelf PIR system later on. In order for Bob to receive a message from Alice, they need to agree where Alice+Bob should be writing, and where they should be polling.

Our proposal is based on the Tor V3 onion/hidden service design where they use key blinding to look up Introduction Points when connecting to hidden services, to thwart efforts from bad people monitoring these lookups to identify which hidden services exist/are getting traffic. The Tor hidden service makes an anonymous connection to the IP server and publishes an ephemeral public key signed with ed25519 (“descriptor”), and their ed25519 public key. The long-term keypair they use for this is blinded with the daily “shared random value”. The client computes the same blinding for the public key and connects to the IP service, requesting a “descriptor”. They check the signature to ensure it was published by the Tor hidden service. The main point here is that there exists an operation SignBlinded(longtermkeypriv, srv, message) such that Verify(BlindPublic(longtermkeypub, srv), message) works, and that someone who doesn’t know “longtermkeypub” cannot perform this Verification for other values of “srv”.

== Cryptographic Scheme for determining spool permutations

https://github.com/katzenpost/katzenpost/blob/debug_wip_add_map_capabilities/core/crypto/eddsa/blinded25519.go

This is a deterministic cryptographic scheme used by communication partner clients to select perimeter nodes and spool IDs to use for exchanging messages.

The scheme they use in Tor is based on ed25519, where the secret keys are random scalars (“a”), and the public keys are those scalars multiplied with the base point (“G”): priv = a pub = scalar(a) * G (mod p) BlindPublic(pub, srv) = a * G * scalar(srv) (mod p) SignBlinded: like normal eddsa, but using BlindSecret to modify the priv key: BlindSecret(priv, srv) = scalar(a) * scalar(srv) (mod p) Since the multiplication is performed in the prime field p, calculating “backwards” from a BlindPublic() value corresponds to breaking EdDSA (like working out “a” given “aG (mod p)”). Kinda sorta, with some caveats.

Our scheme is based on the same principle, but instead of a public SRV we use a CSPRNG to produce cryptographically secure symmetric secrets. In addition we have long-term identity keys for each peer, and they are somehow exchanged securely (PQ handshake or OOB or whatever). shared_secret: some 256-bit secret alice_priv: secret scalar alice_pub: alice_priv * G (mod p) bob_priv: secret scalar bob_pub: bob_priv * G (mod p) Each peer also keeps track of a counter for messages they have sent: alice_sent := 0 bob_sent := 0 When Alice needs to send Bob a message, she computes: alice_sent += 1 bob_inbox_1: BlindPublic(bob_pub, 1==alice_sent) and uses bob_inbox_1 as the mailbox for the message. The next message gets bob_inbox_2: BlindPublic(bob_pub, 2==alice_sent) and so forth. This scheme works to come up with unique mailbox IDs, but it has some unfortunate pitfalls due to the commutativity of the multiplications: - notably bob_inbox_1 is equal to bob_pub <– oops! - bob_inbox_4 is equal to bob_inbox_2 * 2 <– the service operator can track this too, etc so instead our BlindPublicKeyed uses shared_secret to generate a stream of (large) distinct values: BlindPublicKeyed(pub, index, shared_secret) = pub*scalar(hash(shared_secret, index)) (mod p)

==== Cryptographic Capabilities

Here we describe the cryptographic protocol used by spools to authorize actions by the clients.

It's important to note that this design is not exclusive to the messaging part, we could use it for other things in KP where we need capabilities.

https://github.com/katzenpost/katzenpost/blob/debug_wip_add_map_capabilities/map/common/cap.go

Now the problem becomes that if we want to do something funky with these mailboxes, like use the same mailbox for multiple receivers in a group chat, or have write-only mailboxes that multiple people can write to, but only the owner can read, it would be nice to have the service operator only accept writes from the rightful writer. Enter our capability model: By calling BlindPublic() again with a constant string we can derive keys per mailbox like this: mailbox_pk = BlindPublicKeyed(pub, index, shared_secret) read_cap_pk = BlindPublic(mailbox_pk, "read") write_cap_pk = BlindPublic(mailbox_pk, "write") Given a mailbox_pk, the service operator can now verify signatures made with those keys to enforce access control policies. Capabilities can be handed out for a specific index, or "carte blanche": global_read_cap_pk = BlindPublic(pub, "read") read_cap_pk_1 = BlindPublic(global_read_cap, hash(shared_secret, index)) global_read_cap_sk = BlindSecret(priv, "read") read_cap_sk_1 = BlindSecret(global_read_cap_sk_1, hash(shared_secret, index)) ^– because the multiplication is commutative The service operator can then require the clients to produce signatures over the operation they want to perform ("read mailbox XYZ") or ("write this encrypted message to mailbox ABC"), etc.

If you don't pass on the shared_secret, you can thus dole out discrete capabilities for specific operations by handing out the secret keys: - full cap for the whole thing: shared_secret, priv - global read cap: shared_secret, BlindSecret(priv, "read") - global write cap: shared_secret, BlindSecret(priv, "write") - full caps for specific mailbox: BlindSecret(priv, hash(shared_secret, index)) - read cap for specific mailbox: BlindSecret(BlindSecret(priv,hash(shared_secret, index)), "read") - write cap for specific mailbox: BlindSecret(BlindSecret(priv,hash(shared_secret, index)), "write") - we can potentially derive other capabilities like "append" or "delete" or something like that, if needed

==== Sharding The mailbox IDs can be uploaded to a single service operator, or to a DHT by using the mailbox as sharding key. Doing so enables scaling the storage service horizontally (as long as everybody agrees on the partition key mechanism).

Cryptographic End to End protocol

Uses agl's double ratchet from pond. However it is not documented simply because we didn't design it.

For XX Network (David Chaum's cmix based mixnet) they (Rick Carback and Ben Wenger) designed a custom double ratchet made to work with their mixnet. While I worked with them I wrote a design specification for their ratchet, here:

https://git.xx.network/elixxir/docs/-/blob/master/end_to_end.md

If we want to we can design a new double ratchet. I don't think we *need* to do that. However let it be known that the latest version of the double ratchet makes use of the katzenpost NIKE interfaces and thus can in practice work with *any* NIKE. And by the way we can make a hybrid NIKE from any two NIKEs. Currently it uses CSIDH-512-nobs with X25519 but we want to change that to use CTIDH-1024 with X25519.