This is something we (the dark crystal team) are going to be working on over the next couple of months and we would love to hear people's ideas and/or criticism.

Note:

As a new user i have had to remove the links, you can finda version of this doc with links here

Feasibility study of implementing secp256k1 on Secure Scuttlebutt (SSB)

Introduction

Secure-Scuttlebutt has been built with future compatibility for different crypto-primitives. This includes elliptic curve types for signing and encryption as well as hash functions used for content addressing. Keys and hashes are represented as strings containing the key encoded to base64 followed by a delimiter, '.'

, followed by suffix which describes the primitive used, for example 'ed25519'

or 'sha256'

. This makes introducing new kinds of addresses much easier, but there are still a number of issues to address.

Being able to use secp256k1 keys on Scuttlebutt would mean that Scuttlebutt identities could have a corresponding Ethereum address. This would create a bridge between the Scuttlebutt ecosystem and the Ethereum ecosystem, and allow developers to build applications which utilise both networks.

Aspects of Scuttlebutt which are effected

Scuttlebutt uses libsodium to derive Curve25519 keys, used for Diffie-Hellman style encryption, from Ed25519 keys, used for signing and verification. Ed25519 public keys are used to refer to peers on the network and are generally known as 'feed ids' as each peer has their own feed of hash-linked messages.

To implement Secp256k1 keys, we would need to consider how do DH-style scalar multiplication in order to build shared secrets, both for use in the secret handshake and for creating encrypted messages using 'private-box'. The implementations we have looked at give us this without converting the keys but cross-curve encryption presents a problem.

Relevant SSB modules:

secret handshake

This is the mechanism by which peers create a session key for a secure communication channel. It works by means of a mutually authenticating key agreement handshake. The handshake involves a combination of cryptographic signing, ephemeral keys, and Diffie-Hellman (DH) style shared secret derivation.

ssb-ref

This module provides way to validate references and addresses on SSB. It's isFeed

method tests if a given string is a valid feedId using a regular expression. We would need to modify this regular expression to allow for feedIds with the new suffix .secp256k1

.

ssb-keys

Provides key generation, signing and verification of objects, loading keys to or from a file on disk, and encryption using private-box

.

ssb-private

Scuttlebot (the scuttlebutt 'server', or core process) plugin which handles private messaging by calling methods from ssb-keys

which in turn use private-box

.

private-box

Provides encryption of private messages to multiple parties, obfuscating the identities of recipients. Ephemeral keys, DH scalar multiplication and libsodium's 'secretbox' are used.

Secretbox provides symmetric encryption and will work regardless how we choose to generate the keys. The scalar multiplication used to generate a shared secret can be replaced with secp256k1-node

's ecdh

method (although the arguments are taken in a different order).

We have a working proof-of-concept which shows that private-box's functionality can be achieved with secp256k1

keys. More challenging is to get it to work with an array of recipients with a mixture of both curve25519 and secp256k1 keys. The difficulty is that another module, 'ssb-keys' first removes the descriptive suffix (eg: '.ed25519'

) before converting the keys and calling methods from private-box

. So we can't create a drop-in solution without modifying both modules.

ssb-validate

This module provides methods for verifying SSB messages.

Implementing a system of signing and verifying messages with users with varying types of keys is much easier than encrypting messages to multiple parties with differing key types. This is because we only need to 'deal with' one user at time. To create a signature, we need the message, a private key, and to know what type of private key it is. To verify we need the message, signature, public key, and to know what type of public key it is. Unlike DH encryption there are no difficult cases of having a combination of keys of different types. Futhermore, Secp256k1 libraries tend to have well-developed and secure methods for signing and verification because it is the principle security mechanism used in cryptocurrencies.

For these reasons, we feel confident that implementing signature creation and verification on SSB is possible with secp256k1 keys and have rather focused on encryption when researching this document.

secp256k1 implementations in nodejs

secp256k1-node based on bitcoin core's secp256k1 C library

elliptic cryptography library which implements several elliptic curves, including secp256k1. Keys are represented as BigNum (bn.js), but these are easy to convert to Buffers to be consistent with our existing keys.

ethereumjs/keythereum gives us some functions particular to ethereum addresses, uses elliptic.

Deriving ethereum addresses from secp256k1 keys.

Secp256k1 public keys, in their complete form, are 64 bytes produced from concatonating the x and y values of the point on the curve. Typically they are prefixed with an extra byte, 0x04

.

Ethereum addresses are produced by removing the 04 prefix, taking the keccak-256 hash, and then truncating the first 12 bytes, to give us 20 bytes. These are encoded as hex and the '0x' prefix is added.

keythereum's privateKeyToAddress

method makes this easy.

Challenges

Different length of public keys

Scuttlebutt uses Ed25519 and Curve25519 keys, where both public and secret keys have a length of 32 bytes. Secp256k1 public keys, in their compressed form, are 33 bytes. This is the 32 byte x-value prepended by either 0x02

or 0x03

depending on whether the y-value is odd or even. This is a problem because some of our functions (eg: encryption using ‘private-box’) involve concatonating several keys to produce a shared secret. If we have a mix of both key types, it is impossible to know when one ends and the other begins.

To overcome this, we propose that in ‘private-box’ messages, ed25519 public keys are prefixed with an additional byte, 0x00

. This means that all public keys are now 33 bytes, and the first byte determines which crypto primative is used.

Ephemeral keys must be the correct type

Both ‘private-box’ and ‘secret-handshake’ make use of ephemeral keypairs. In order to work with both types of key, we will need to generate and send one keypair for each type. This increases complexity and message size.

DH encryption between users with different key types

This is a serious problem, and while it might be possible to have a way of producing shared secrets between keys on different curves, there may be security risks involved. It is possible to derive an ‘equivalent’ keypair on the other user’s curve using our own secret key as the seed, and then send the corresponding public key together with the encrypted message. But this makes it more difficult to verify who authored the message. Since messages are also signed, this is perhaps not a great problem, but it is not ideal.

A better option is to generate a new address of the alternative key type and automatically publish a public signed message containing the public key, somewhat similar to a TLS certificate. That is to say a user with a secp256k1 address could publish a signed curve25519 public key for encryption. This gives a security enhancement because it is generally regarded as bad practice to use the same keypair for signing and encryption. In fact, there is some discussion on implementing this on SSB anyway, as currently encryption keys are derived from signing keys, which is also not ideal. This would also mean we wouldn’t have to worry about the issues of varying public key lengths, or differing ephemeral key types described above.

Security

The secp256k1-node

module explicitly states in the readme that it is an experimental library. We would naturally want to conduct a security audit before using this module in production.

Social aspects

This document focuses on technical challenges. However, the SSB community is a participative ecosystem where decisions are made together. Making a significant change to the protocol will require discussion and consensus of developers and users. If the change brings security issues or raises ideological questions it may be difficult to reach agreement.

Conclusions

We have focussed on the technical feasibility of implementing a different elliptic curve type on SSB. While we have identified some challenges, we feel they can be overcome, and that building a working proof-of-concept is achievable. But we are aware that there is a long journey between having a proof-of-concept, and a real world working system that works with all the available clients with no bugs or issues.

Furthermore, we have not explored the implications, what would be made possible and what are the potential advantages or issues.

Related work

- Secp256k1 experiments source code of experiments we did as research for this document
- Some SSB links to the discussion on this as well as a copy of Dominics related post

References

- Gavin Wood, ‘Ethereum yellow paper’
- Dominic Tarr, ‘Designing a Secret Handshake, Authenticated Key Exchange as a Capability System’
- Davide De Rosa, Basic blockchain programming - Elliptic Curve Keys
- secp256k1 specification from the ‘Standards for efficient cryptography group’
- A closer look at ethereum signatures - hackernoon

- Generating a usable Ethereum wallet and its corresponding keys - Command line examples using OpenSSL
- Derive a shared secret between different curves
- what does secp256k1 look like?
- secp256k1 article on bitcoin wiki