Bern University
of Applied Sciences

Informatikseminar

Bitmessage – Communication Without Metadata

Author	Christian Basler
Tutor	Kai Brünnler
Date	May 22, 2015

Contents

1. Synopsis

TODO

2. Basics

While encryption technology like PGP or S/MIME provides a secure way to protect content from prying eyes, ever since Edward Snowdens whistleblowing we learned that metadata — most notably information about who communicates with whom — is equally interesting and much easier to analyze.

With e-mail, we can only prevent this by encrypting the connection to the server as well as between servers. Therefore we can only hope that both our and the recipient's e-mail provider are both trustworthy as well as competent.

With Bitmessage we send a message to a sufficiently large number of participants, with the intended recipient among them. Content is encrypted such as only the person in possesion of the private key can decrypt it. All participants try to do this in order to find their messages.

3. Protocol

3.1. Nomenclature

There are a few terms that are easily mixed up. Here's a list of the most confusing ones:

message	is sent from one node to another, i.e. to announce new objects or to initialize the network connection.
msg	is the object payload containing the actual message written by a user. The term 'message' is never used to describe information exchange between users in this document. 'Content' is mostly used instead.
payload	There are two kinds of payload: message payload for message types, e.g. containing inventory vectors, and object payload, which is distributed throughout the network.
object	is a kind of message whose payload is distributed among all nodes. Somtimes just the payload is meant.

3.2. Process Flow

The newly started node **A** connects to a random node **B** from its node registry and sends a *version* message, announcing the latest supported protocol version. If **B** accepts the version[1], it responds with a *verack* message, followed by a *version* message announcing its own latest supported protocol version. Node **A** then decides whether it supports **B**'s version and sends its *verack* mesage.

If both nodes accept the connection, they both send an *addr* message containing up to 1000 of its known nodes, followed by one or more *inv* messages announcing all valid objects they are aware of. They then send *getobject* request for all objects still missing from their inventory.

Getobject requests are answered by *object* messages containing the requested objects.

A node actively connects to eight other nodes, allowing any number of incoming connections. If a user creates a new object on node **A**, it is offered via *inv* to eight of the connected nodes. They will get the object and distribute it to up to eight of their connections, and so on.

3.3. Messages

The messages, objects and binary format are very well discribed in the Bitmessage wiki [2], the message description is therefore narrowed down to a description of what they do and when they're used.

3.3.1. version / verack

A *version* message contains the latest protocol version supported by a node, as well as the streams it is interested in and which features it supports. If the other node accepts, it acknowledges with a *verack* message. The connection is initialized when both nodes sent a *verack* message.

3.3.2. addr

Contains up to 1000 known nodes with their IP addresses, ports, streams and supported features.

3.3.3. inv

One *inv* message contains the hashes of up to 50000 valid objects. If your inventory is larger, several messages can be sent.

[1]A version is accepted by default if it is higher or equal to a nodes latest supported version. Nodes supporting experimental protocol versions might accept older versions.

3.3.4. getdata

Can request up to 50000 objects by sending their hashes.

3.3.5. object

Contains one requested object, which might be one of:

getpubkey	A request for a public key, which is needed to encrypt a message to a specific user.
pubkey	A public key. See 3.4 Addresses
msg	Content intended to be received by one user.
broadcast	Content sent in a way that the Addresses public key can be used to decrypt it, allowing any subscriber who knows the address to receive the such a message

3.3.6. ping / pong / getbiginv

People looking at the PyBitmessage's source code might be irritated by some other messages that seem to be implemented, but aren't mentioned in the official protocol specification. *ping* does actually cause the node that implements this to send a *pong* message, but this feature isn't actually used anywhere. *getbiginv* seems to be thought for requesting the inventory, but as I understand it can't be used. [3]

3.4. Addresses

BM-2cXxfcSetKnbHJX2Y85rSkaVpsdNUZ5q9h: Addresses start with "BM-" and are, like Bitcoin addresses, Base58 encoded[2].

version	Address version.
stream	Stream number.
ripe	Hash of both public signing and encryption key. Please note that the keys are sent without the leading 0x04 in **pubkey** objects, but for creating the ripe it must be prepended. This is also necessary for most other applications, so it's a good idea to do it by default. `ripemd160(sha512(pubSigKey + pubEncKey))`

[2]Which uses characters 1-9, A-Z and a-z without the easily confused characters I, l, 0 and O.

checksum	First four bytes of a double SHA-512 hash of the above. `sha512(sha512(version + stream + ripe))`

3.5. Encryption

Bitcoin uses Elliptic Curve Cryptography for both signing and encryption. While the mathematics behind elliptic curves is even harder to understand than the older approach of multiplying huge primes, it's based on the same principle of doing some mathematical operation that can be done fast one way but is very hard to reverse. Instead of two very large primes, we multiply a point on the elliptic curve by a very large number[3].

The user, let's call her Alice, needs a key pair, consisting of a private key

$$k$$

which represents a huge random number, and a public key

$$K = Gk$$

which represents a point on the agreed on curve[4]. Please note that this is not a simple multiplication, but the multiplication of a point along an elliptic curve. G is the starting point for all operations on a specific curve.

Another user, Bob, knows the public key. To encrypt a message, Bob creates a temporary key pair

$$r$$

and

$$R = Gr$$

He then calculates

$$Kr$$

uses the resulting Point to encrypt the message[5] and sends K along with the message.

When Alice receives the message, she uses the fact that

$$Kr = Gkr = Grk = Rk$$

so she just uses Rk to decrypt the message.

The exact method used in Bitmessage is called Elliptic Curve Integrated Encryption Scheme or ECIES, which is described in detail on Wikipedia (http://en.wikipedia.org/wiki/Integrated_Encryption_Scheme).

[3]Please don't ask me how to do it. If your're crazy enough, start at http://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication and http://en.wikipedia.org/wiki/Elliptic_curve_cryptography. If you're not that crazy, use a library like Bouncy Casle that does the hard work for you.

[4]Bitmessage uses a curve called *secp256k1*.

[5]A double SHA-512 hash over the x-coordinate is used to create the actual key.

3.5.1. Signature

To sign objects, Bitmessage uses Elliptic Curve Digital Signature Algorithm or ECDSA. This is slightly more complicated, if you want the details, Wikipedia is once again a fine starting point: http://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

4. Issues

4.1. Scalability

Bitmessage doen't really scale. If there are very few users, anonymity isn't given anymore, and with many users traffic and storage use grows quadratically.

4.1.1. Proof of Work

Proof of work has two uses. It helps to protect the network by preventing single nodes from flooding it with objects, and to protect users from spam. There's minimal proof of work required for the network to distribute objects, but users can define higher requirements for their addresses if they get spammed with cheap Viagra™ offers. The proof of work required for an address is defined in the **pubkey**, and senders that are in a user's contacts should not be required to do the higher proof of work.

The difficulty is calculated from both message size as well as time to live, meaning that a message that is larger or stored longer in the network will be more expensive to send.

$$d = \frac{2^{64}}{n(l + \frac{tl}{2^{16}})}$$

d	target difficulty
n	required trials per byte
l	payload length + extra bytes (in order to not make it too easy to send a lot of tiny messages)
t	time to live

To do the proof of work, a nonce must be found such that the first eight bytes of the hash of the object (including the nonce) represent a lower number than the target difficulty.

4.1.2. Message Size Limitation

To prevent malicious users from clogging individual nodes, messages must not be larger than 256 KiB. Because of the proof of work, large objects arent' practical for normal use, but might be used to occupy nodes by sending them garbage.

4.1.3. Streams

The intended solution for this problem is splitting traffic – addresses, more precisely – into streams. A node listens only on the streams that concern its addresses. If it wants to send an object to another stream, it just connects to a node in this stream to send the object, then disconnects. When all active streams are full, a new one is created which should be used for new addresses.

The unsolved problem is to determine when a stream is full. Another issue is the fact that, as the overall network grows, traffic on full streams still grows, as there are more users who might wanto to write someone on the full stream.

4.1.4. Prefix Filtering

TODO[1]

4.2. Forward Secrecy

Obviously it's trivial for an attacker to collect all (encrypted) objects distributed through the Bitmessage network – as long as disk space is not an issue. If this attacker can somehow get the private key of a user, they can decrypt all stored messages intended for that user, as well as impersonate said user[6].

Plausible deniability can, in some scenarios, help against this. This action, called "nuking an address", is done by anonymously publishing the private keys somewhere publicly accessible[7].

Perfect forward secrecy seems impractical to implement, as it requires to exchange messages prior to sending encrypted content. That would in turn need proof of work to protect the network, resulting in twice the work for the sender and three times longer to send — that is, if both clients are online. Exchanging messages would be all but impossible if both users are online sporadically.

5. Discussion

TODO

[6]The latter might be more difficult if they got the key through a brute force attack.
[7]See https://bitmessage.ch/nuked/ for an example.

References

[1] Jonathan Coe. Bitmessage wiki: Scalability through prefix filtering, 2015. https://bitmessage.org/wiki/Scalability_through_Prefix_Filtering.

[2] Jonathan 'Atheros' Warren and Jonathan Coe. Bitmessage wiki: Protocol specification, 2015. https://bitmessage.org/wiki/Protocol_specification.

[3] Jonathan 'Atheros' Warren and ISibbol. Biginv and ping/pong, 2015. https://github.com/Bitmessage/PyBitmessage/issues/112.

Appendix

A. TODO