```
BIP: 151
Layer: Peer Services
Title: Peer-to-Peer Communication Encryption
Author: Jonas Schnelli <dev@jonasschnelli.ch>
Comments-Summary: Controversial; some recommendation, and some discouragement
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0151
Status: Replaced
Type: Standards Track
Created: 2016-03-23
License: PD
Superseded-By: 324
```

Abstract

This BIP describes an alternative way that a peer can encrypt their communication between a selective subset of remote peers.

Motivation

The Bitcoin network does not encrypt communication between peers today. This opens up security issues (eg: traffic manipulation by others) and allows for mass surveillance / analysis of bitcoin users. Mostly this is negligible because of the nature of Bitcoin's trust model, however, for SPV nodes this can have significant privacy impacts [1] and could reduce the censorship-resistance of a peer.

Encrypting peer traffic will make analysis and specific user targeting much more difficult than it currently is. Today it's trivial for a network provider or any other men-in-the-middle to identify a Bitcoin user and its controlled addresses/keys (and link with his Google profile, etc.). Just created and broadcasted transactions will reveal the amount and the payee to the network provider.

This BIP also describes a way that data manipulation (blocking commands by a intercepting TCP/IP node) would be identifiable by the communicating peers.

Analyzing the type of p2p communication would still be possible because of the characteristics (size, sending-interval, etc.) of the encrypted messages.

Encrypting traffic between peers is already possible with VPN, tor, stunnel, curveCP or any other encryption mechanism on a deeper OSI level, however, most mechanisms are not practical for SPV or other DHCP/NAT environment and will require significant knowhow in how to setup such a secure channel.

Specification

A peer that supports encryption must accept encryption requests from all peers.

An independent ECDH negotiation for both communication directions is required and therefore a bidirectional communication will use two symmetric cipher keys (one per direction).

Both peers must only send encrypted messages after a successful ECDH negotiation in *both directions*.

Encryption initialization must happen before sending any other messages to the responding peer (`encinit` message after a `version` message must be ignored).

Symmetric Encryption Cipher Keys

The symmetric encryption cipher keys will be calculated with ECDH/HKDF by sharing the pubkeys of an ephemeral key. Once the ECDH secret is calculated on each side, the symmetric encryption cipher keys must be derived with HKDF [2] after the following specification:

1. HKDF extraction `PRK = HKDF_EXTRACT(hash=SHA256, salt="bitcoinecdh", ikm=ecdh_secret|cipher-type)`.

2. Derive Key1 `K_1 = HKDF_EXPAND(prk=PRK, hash=SHA256, info="BitcoinK1", L=32)`

3. Derive Key2 `K_2 = HKDF_EXPAND(prk=PRK, hash=SHA256, info="BitcoinK2", L=32)`

It is important to include the cipher-type into the symmetric cipher key derivation to avoid weak-cipher-attacks.

Session ID

Both sides must also calculate the 256bit session-id using `SID = HKDF_EXPAND(prk=PRK, hash=SHA256, info="BitcoinSessionID", L=32)`. The session-id can be used for linking the encryption-session to an identity check.

The `encinit` message type

To request encrypted communication, the requesting peer generates an EC ephemeral-session-keypair and sends an `encinit` message to the responding peer and waits for an `encack` message. The responding node must do the same `encinit`/`encack` interaction for the opposite communication direction.

Field Size	Description	Data type	Comments
33bytes	ephemeral-pubkey	comp.-pubkey	The session pubkey from the requesting peer
1bytes	symmetric key cipher type	int8	symmetric key cipher type to use

Possible symmetric key ciphers types

Number	symmetric key ciphers type
0	chacha20-poly1305@openssh.com

ChaCha20-Poly1305 Cipher Suite

ChaCha20 is a stream cipher designed by Daniel Bernstein [3]. It operates by permuting 128 fixed bits, 128 or 256 bits of key, a 64 bit nonce and a 64 bit counter into 64 bytes of output. This output is used as a keystream, with any unused bytes simply discarded.

Poly1305, also by Daniel Bernstein [4], is a one-time Carter-Wegman MAC that computes a 128 bit integrity tag given a message and a single-use 256 bit secret key.

The chacha20-poly1305@openssh.com specified and defined by openssh [5] combines these two primitives into an authenticated encryption mode. The construction used is based on that proposed for TLS by Adam Langley [6], but differs in the layout of data passed to the MAC and in the addition of encyption of the packet lengths.

`K_1` must be used to only encrypt the payload size of the encrypted message to avoid leaking information by revealing the message size.

`K_2` must be used in conjunction with poly1305 to build an AEAD.

Optimized implementations of ChaCha20-Poly1305 are very fast in general, therefore it is very likely that encrypted messages require less CPU cycles per byte then the current unencrypted p2p message format. A quick analysis by Pieter Wuille of the current *standard implementations* has shown that SHA256 requires more CPU cycles per byte then ChaCha20 & Poly1304.

The `encack` message type

The responding peer accepts the encryption request by sending an `encack` message.

Field Size	Description	Data type	Comments
33bytes	ephemeral-pubkey	comp.-pubkey	The session pubkey from the responding peer

At this point, the shared secret key for the symmetric key cipher must be calculated by using ECDH (own privkey x remote pub key). Private keys will never be transmitted. The shared secret can only be calculated if an attacker knows at least one private key and the remote peer's public key.

- **The `encinit`/`encack` interaction must be done from both sides.**
- Each communication direction uses its own secret key for the symmetric cipher.
- The second `encinit` request (from the responding peer) must use the same symmetric cipher type.
- All unencrypted messages before the second `encack` response (from the responding peer) must be ignored.

- After a successful `encinit/encack` interaction, the "encrypted messages structure" must be used. Non-encrypted messages from the requesting peer must lead to a connection termination.

After a successful `encinit/encack` interaction from both sides, the messages format must use the "encrypted messages structure". Non-encrypted messages from the requesting peer must lead to a connection termination (can be detected by the 4 byte network magic in the unencrypted message structure).

Encrypted Messages Structure

Field Size	Description	Data type	Comments
4	length	uint32_t	Length of ciphertext payload in number of bytes
?	ciphertext payload	?	One or many ciphertext command & message data
16	MAC tag	?	128bit MAC-tag

Encrypted messages do not have the 4byte network magic.

The maximum message length needs to be chosen carefully. The 4 byte length field can lead to a required message buffer of 4 GiB. Processing the message before the authentication succeeds must not be done.

The 4byte sha256 checksum is no longer required because the AEAD.

Both peers need to track the message sequence number (uint32) of sent messages to the remote peer for building a 64 bit symmetric cipher IV. Sequence numbers are allowed to overflow to zero after 4294967295 (2^32-1).

The encrypted payload will result decrypted in one or many unencrypted messages:

Field Size	Description	Data type	Comments
?	command	varlen	ASCII string identifying the packet content, we are using varlen in the
4	length	uint32_t	Length of plaintext payload
?	payload	?	The actual data

If more data is present, another message must be deserialized. There is no explicit amount-of-messages integer.

Re-Keying

A responding peer can inform the requesting peer over a re-keying with an `encack` message containing 33byte of zeros to indicate that all encrypted message following after this `encack` message will be encrypted with *the next symmetric cipher key.*

The new symmetric cipher key will be calculated by `SHA256(SHA256(session_id || old_symmetric_cipher_key))`.

Re-Keying interval is a peer policy with a minimum timespan of 10 seconds.

The Re-Keying must be done after every 1GB of data sent or received (recommended by RFC4253 SSH Transport).

Risks

The encryption does not include an identity authentication scheme. This BIP does not cover a proposal to avoid MITM attacks during the encryption initialization.

Identity authentication will be covered in another BIP and will presume communication encryption after this BIP.

Compatibility

This proposal is backward compatible. Non-supporting peers will ignore the `encinit` messages.

Reference implementation

References

- [1] https://e-collection.library.ethz.ch/eserv/eth:48205/eth-48205-01.pdf
- [2] HKDF (RFC 5869) https://tools.ietf.org/html/rfc5869
- [3] ChaCha20 https://cr.yp.to/chacha/chacha-20080128.pdf
- [4] Poly1305 https://cr.yp.to/mac/poly1305-20050329.pdf
- [5] https://github.com/openssh/openssh-portable/blob/05855bf2ce7d5cd0a6db18bc0b4214ed5ef7516d/PROTOCOL.chacha20poly1305
- [6] "ChaCha20 and Poly1305 based Cipher Suites for TLS", Adam Langley https://tools.ietf.org/html/draft-agl-tls-chacha20poly1305-03

Acknowledgements

- Pieter Wuille and Gregory Maxwell for most of the ideas in this BIP.

Copyright