```
BIP: 75
Layer: Applications
Title: Out of Band Address Exchange using Payment Protocol Encryption
Author: Justin Newton <justin@netki.com>
        Matt David <mgd@mgddev.com>
        Aaron Voisine <voisine@gmail.com>
        James MacWhyte <macwhyte@gmail.com>
Comments-Summary: Recommended for implementation (one person)
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0075
Status: Final
Type: Standards Track
Created: 2015-11-20
License: CC-BY-4.0
```

Abstract

This BIP is an extension to BIP 70 that provides two enhancements to the existing Payment Protocol.

1. It allows the requester (Sender) of a PaymentRequest to voluntarily sign the original request and provide a certificate to allow the payee to know the identity of who they are transacting with.

1. It encrypts the PaymentRequest that is returned, before handing it off to the SSL/TLS layer to prevent man in the middle viewing of the Payment Request details.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Copyright

Definitions

Sender	Entity wishing to transfer value that they control
Receiver	Entity receiving a value transfer

Motivation

The motivation for defining this extension to the BIP70 Payment Protocol is to allow two parties to exchange payment information in a permissioned and encrypted way, such that wallet address communication can become a more

automated process. This extension also expands the types of PKI (public-key infrastructure) data that is supported, and allows it to be shared by both parties (with BIP70, only the receiver could provide PKI information). This allows for automated creation of off-blockchain transaction logs that are human readable, now including information about the sender and not just the recipient.

The motivation for this extension to BIP70 is threefold:

1. Ensure that the payment details can only be seen by the participants in the transaction, and not by any third party.

1. Enhance the Payment Protocol to allow for store and forward servers in order to allow, for example, mobile wallets to sign and serve Payment Requests.

1. Allow a sender of funds the option of sharing their identity with the receiver. This information could then be used to:

#* Make Bitcoin logs (wallet transaction history) more human readable

#* Give the user the ability to decide whether or not they share their Bitcoin address and other payment details when requested

#* Allow for an open standards based way for businesses to keep verifiable records of their financial transactions, to better meet the needs of accounting practices or other reporting and statutory requirements

#* Automate the active exchange of payment addresses, so static addresses and BIP32 X-Pubs can be avoided to maintain privacy and convenience

In short we wanted to make Bitcoin more human, while at the same time improving transaction privacy.

Example Use Cases

1. Address Book

A Bitcoin wallet developer would like to offer the ability to store an "address book" of payees, so users could send multiple payments to known entities without having to request an address every time. Static addresses compromise privacy, and address reuse is considered a security risk. BIP32 X-Pubs allow the generation of unique addresses, but watching an X-Pub chain for each person you wish to receive funds from is too resource-intensive for mobile applications, and there is always a risk of unknowingly sending funds to an X-Pub address after the owner has lost access to the corresponding private key.

With this BIP, Bitcoin wallets could maintain an "address book" that only needs to store each payee's public key. Adding an entry to one's address book could be done by using a Wallet Name, scanning a QR code, sending a URI through a text message or e-mail, or searching a public repository. When the user wishes to make a payment, their wallet would do all the work in the background to

communicate with the payee's wallet to receive a unique payment address. If the payee's wallet has been lost, replaced, or destroyed, no communication will be possible, and the sending of funds to a "dead" address is prevented.

2. Individual Permissioned Address Release

A Bitcoin wallet developer would like to allow users to view a potential sending party's identifying information before deciding whether or not to share payment information with them. Currently, BIP70 shares the receiver's payment address and identity information with anyone who requests it.

With this BIP, Bitcoin wallets could use the sender's identifying information to make a determination of whether or not to share their own information. This gives the receiving party more control over who receives their payment and identity information. Additionally, this could be used to automatically provide new payment addresses to whitelisted senders, or to protect users' privacy from unsolicited payment requests.

3. Using Store & Forward Servers

A Bitcoin wallet developer would like to use a public Store & Forward service for an asynchronous address exchange. This is a common case for mobile and offline wallets.

With this BIP, returned payment information is encrypted with an ECDH-computed shared key before sending to a Store & Forward service. In this case, a successful attack against a Store & Forward service would not be able to read or modify wallet address or payment information, only delete encrypted messages.

Modifying BIP70 pki_type

This BIP adds additional possible values for the pki_type variable in the PaymentRequest message. The complete list is now as follows:

pki_type	Description
x509+sha256	A x.509 certificate, as described in BIP70
pgp+sha256	An OpenPGP certificate
ecdsa+sha256	A secp256k1 ECDSA public key

NOTE: Although SHA1 was supported in BIP70, it has been deprecated and BIP75 only supports SHA256. The hashing algorithm is still specified in the values listed above for forward and backwards compatibility.

New Messages

Updated [/bip-0075/paymentrequest.proto paymentrequest.proto] contains the existing PaymentRequest Protocol Buffer messages as well as the messages newly defined in this BIP.

NOTE: Public keys from both parties must be known to each other in order to facilitate encrypted communication. Although including both public keys in every message may get redundant, it provides the most flexibility as each message is completely self-contained.

InvoiceRequest

The **InvoiceRequest** message allows a Sender to send information to the Receiver such that the Receiver can create and return a PaymentRequest.

```
message InvoiceRequest {
        required bytes  sender_public_key = 1;
        optional uint64 amount = 2 [default = 0];
        optional string pki_type = 3 [default = "none"];
        optional bytes  pki_data = 4;
        optional string memo = 5;
        optional string notification_url = 6;
        optional bytes  signature = 7;
}
```

Field Name	Description
sender_public_key	Sender's SEC-encoded EC public key
amount	amount is integer-number-of-satoshis (default: 0)
pki_type	none / x509+sha256 / pgp+sha256 / ecdsa+sha256 (default: "none")
pki_data	Depends on pki_type
memo	Human-readable description of invoice request for the receiver
notification_url	Secure (usually TLS-protected HTTP) location where an EncryptedProtocolMessage SH
signature	PKI-dependent signature

ProtocolMessageType Enum

This enum is used in the newly defined ProtocolMessage and EncryptedProtocolMessage messages to define the serialized message type. The **ProtocolMessageType** enum is defined in an extensible way to allow for new message type additions to the Payment Protocol.

```
enum ProtocolMessageType {
    UNKNOWN_MESSAGE_TYPE = 0;
    INVOICE_REQUEST = 1;
    PAYMENT_REQUEST = 2;
    PAYMENT = 3;
    PAYMENT_ACK = 4;
}
```

ProtocolMessage

The **ProtocolMessage** message is an encapsulating wrapper for any Payment Protocol message. It allows two-way, non-encrypted communication of Payment Protocol messages. The message also includes a status code and a status message that is used for error communication such that the protocol does not rely on transport-layer error handling.

```
message ProtocolMessage {
    required uint64 version = 1
    required uint64 status_code = 2;
    required ProtocolMessageType message_type = 3;
    required bytes serialized_message = 4;
    optional string status_message = 5;
    required bytes identifier = 6;
}
```

Field Name	Description
version	Protocol version number (Currently 1)
status_code	Payment Protocol Status Code
message_type	Message Type of serialized_message
serialized_message	Serialized Payment Protocol Message
status_message	Human-readable Payment Protocol status message
identifier	Unique key to identify this entire exchange on the server. Default value SHOULD be SH

Versioning

This BIP introduces version 1 of this protocol. All messages sent using these base requirements MUST use a value of 1 for the version number. Any future BIPs that modify this protocol (encryption schemes, etc) MUST each increment the version number by 1.

When initiating communication, the version field of the first message SHOULD be set to the highest version number the sender understands. All clients MUST be able to understand all version numbers less than the highest number they support. If a client receives a message with a version number higher than they understand, they MUST send the message back to the sender with a status code of 101 ("version too high") and the version field set to the highest version number the recipient understands. The sender must then resend the original message using the same version number returned by the recipient or abort.

EncryptedProtocolMessage

The **EncryptedProtocolMessage** message is an encapsualting wrapper for any Payment Protocol message. It allows two-way, authenticated and encrypted communication of Payment Protocol messages in order to keep their contents secret. The message also includes a status code and status message that is used

for error communication such that the protocol does not rely on transport-layer error handling.

```
message EncryptedProtocolMessage {
    required uint64 version = 1 [default = 1];
    required uint64 status_code = 2 [default = 1];
    required ProtocolMessageType message_type = 3;
    required bytes encrypted_message = 4;
    required bytes receiver_public_key = 5;
    required bytes sender_public_key = 6;
    required uint64 nonce = 7;
    required bytes identifier = 8;
    optional string status_message = 9;
    optional bytes signature = 10;
}
```

Field Name	Description
version	Protocol version number
status_code	Payment Protocol Status Code
message_type	Message Type of Decrypted encrypted_message
encrypted_message	AES-256-GCM Encrypted (as defined in BIP75) Payment Protocol Message
receiver_public_key	Receiver's SEC-encoded EC Public Key
sender_public_key	Sender's SEC-encoded EC Public Key
nonce	Microseconds since epoch
identifier	Unique key to identify this entire exchange on the server. Default value SHOULD be S
status_message	Human-readable Payment Protocol status message
signature	DER-encoded Signature over the full EncryptedProtocolMessage with EC Key Belongi

Payment Protocol Process with InvoiceRequests

The full process overview for using **InvoiceRequests** in the Payment Protocol is defined below.

All Payment Protocol messages MUST be encapsulated in either a ProtocolMessage or EncryptedProtocolMessage. Once the process begins using EncryptedProtocolMessage messages, all subsequent communications MUST use EncryptedProtocolMessages.

All Payment Protocol messages SHOULD be communicated using EncryptedProtocolMessage encapsulating messages with the exception that an InvoiceRequest MAY be communicated using the ProtocolMessage if the receiver's public key is unknown.

The process of creating encrypted Payment Protocol messages is enumerated in Sending Encrypted Payment Protocol Messages using EncryptedProtocolMes-

sages, and the process of decrypting encrypted messages can be found under Validating and Decrypting Payment Protocol Messages using EncryptedProtocolMessages.

A standard exchange from start to finish would look like the following:

1. Sender creates InvoiceRequest
2. Sender encapsulates InvoiceRequest in (Encrypted)ProtocolMessage
3. Sender sends (Encrypted)ProtocolMessage to Receiver
4. Receiver retrieves InvoiceRequest in (Encrypted)ProtocolMessage from Sender
5. Receiver creates PaymentRequest
6. Receiver encapsulates PaymentRequest in EncryptedProtocolMessage
7. Receiver transmits EncryptedProtocolMessage to Sender
8. Sender validates PaymentRequest retrieved from the EncryptedProtocolMessage
9. The PaymentRequest is processed according to BIP70, including optional Payment and PaymentACK messages encapsulated in EncryptedProtocolMessage messages.

NOTE: See Initial Public Key Retrieval for InvoiceRequest Encryption for possible options to retrieve Receiver's public key.

Message Interaction Details

HTTP Content Types for New Message Types

When communicated via **HTTP**, the listed messages MUST be transmitted via TLS-protected HTTP using the appropriate Content-Type header as defined here per message:
{| class="wikitable" ! Message Type !! Content Type |- | ProtocolMessage || application/bitcoin-paymentprotocol-message |- | EncryptedProtocolMessage || application/bitcoin-encrypted-paymentprotocol-message |}

Payment Protocol Status Communication

Every ProtocolMessage or EncryptedProtocolMessage MUST include a status code which conveys information about the last message received, if any (for the first message sent, use a status of 1 "OK" even though there was no previous message). In the case of an error that causes the Payment Protocol process to be stopped or requires that message be retried, a ProtocolMessage or EncryptedProtocolMessage SHOULD be returned by the party generating the error. The content of the message MUST contain the same **serialized_message** or **encrypted_message** and identifier (if present) and MUST have the status_code set appropriately.

The status_message value SHOULD be set with a human readable explanation of the status code.

Payment Protocol Status Codes

Status Code	Description
1	OK
2	Cancel
100	General / Unknown Error
101	Version Too High
102	Authentication Failed
103	Encrypted Message Required
200	Amount Too High
201	Amount Too Low
202	Amount Invalid
203	Payment Does Not Meet PaymentRequest Requirements
300	Certificate Required
301	Certificate Expired
302	Certificate Invalid for Transaction
303	Certificate Revoked
304	Certificate Not Well Rooted

+==Canceling A Message==+ If a participant to a transaction would like to inform the other party that a previous message should be canceled, they can send the same message with a status code of 2 ("Cancel") and, where applicable, an updated nonce. How recipients make use of the "Cancel" message is up to developers. For example, wallet developers may want to offer users the ability to cancel payment requests they have sent to other users, and have that change reflected in the recipient's UI. Developers using the non-encrypted ProtocolMessage may want to ignore "Cancel" messages, as it may be difficult to authenticate that the message originated from the same user.

Transport Layer Communication Errors

Communication errors MUST be communicated to the party that initiated the communication via the communication layer's existing error messaging faciltiies. In the case of TLS-protected HTTP, this SHOULD be done through standard HTTP Status Code messaging (RFC 7231 Section 6).

Extended Payment Protocol Process Details

This BIP extends the Payment Protocol as defined in BIP70.

For the following we assume the Sender already knows the Receiver's public key, and the exchange is being facilitated by a Store & Forward server which requires valid signatures for authentication.

nonce MUST be set to a non-repeating number **and** MUST be chosen by the encryptor. The current epoch time in microseconds SHOULD be used, unless the creating device doesn't have access to a RTC (in the case of a smart card, for example). The service receiving the message containing the **nonce** MAY use whatever method to make sure that the **nonce** is never repeated.

InvoiceRequest Message Creation

- Create an InvoiceRequest message
- **sender_public_key** MUST be set to the public key of an EC keypair
- **amount** is optional. If the amount is not specified by the InvoiceRequest, the Receiver MAY specify the amount in the returned PaymentRequest. If an amount is specified by the InvoiceRequest and a PaymentRequest cannot be generated for that amount, the InvoiceRequest SHOULD return the same InvoiceRequest in a ProtocolMessage or EncryptedProtocolMessage with the status_code and status_message fields set appropriately.
- **memo** is optional. This MAY be set to a human readable description of the InvoiceRequest
- Set **notification_url** to URL that the Receiver will submit completed PaymentRequest (encapsulated in an EncryptedProtocolMessage) to
- If NOT including certificate, set **pki_type** to "none"
- If including certificate:
  - Set **pki_type** to "x509+sha256"
  - Set **pki_data** as it would be set in BIP-0070 (Certificates)
  - Sign InvoiceRequest with signature = "" using the X509 Certificate's private key
  - Set **signature** value to the computed signature

InvoiceRequest Validation

- Validate **sender_public_key** is a valid EC public key
- Validate **notification_url**, if set, contains characters deemed valid for a URL (avoiding XSS related characters, etc).
- If **pki_type** is None, InvoiceRequest is VALID
- If **pki_type** is x509+sha256 and **signature** is valid for the serialized InvoiceRequest where signature is set to "", InvoiceRequest is VALID

Sending Encrypted Payment Protocol Messages using EncryptedProtocolMessages

- Encrypt the serialized Payment Protocol message using AES-256-GCM setup as described in ECDH Point Generation and AES-256 (GCM Mode) Setup
- Create EncryptedProtocolMessage message
- Set **encrypted_message** to be the encrypted value of the Payment Protocol message

- **version** SHOULD be set to the highest version number the client understands (currently 1)
- **sender_public_key** MUST be set to the public key of the Sender's EC keypair
- **receiver_public_key** MUST be set to the public key of the Receiver's EC keypair
- **nonce** MUST be set to the nonce used in the AES-256-GCM encryption operation
- Set **identifier** to the identifier value received in the originating InvoiceRequest's ProtocolMessage or EncryptedProtocolMessage wrapper message
- Set **signature** to ""
- Sign the serialized EncryptedProtocolMessage message with the communicating party's EC public key
- Set **signature** to the result of the signature operation above

SIGNATURE NOTE: EncryptedProtocolMessage messages are signed with the public keys of the party transmitting the message. This allows a Store & Forward server or other transmission system to prevent spam or other abuses. For those who are privacy conscious and don't want the server to track the interactions between two public keys, the Sender can generate a new public key for each interaction to keep their identity anonymous.

Validating and Decrypting Payment Protocol Messages using EncryptedProtocolMessages

- The **nonce** MUST not be repeated. The service receiving the EncryptedProtocolMessage MAY use whatever method to make sure that the nonce is never repeated.
- Decrypt the serialized Payment Protocol message using AES-256-GCM setup as described in ECDH Point Generation and AES-256 (GCM Mode) Setup
- Deserialize the serialized Payment Protocol message

ECDH Point Generation and AES-256 (GCM Mode) Setup

NOTE: AES-256-GCM is used because it provides authenticated encryption facilities, thus negating the need for a separate message hash for authentication.

- Generate the **secret point** using ECDH using the local entity's private key and the remote entity's public key as inputs
- Initialize HMAC_DRBG
  - Use **SHA512(secret point's X value in Big-Endian bytes)** for Entropy
  - Use the given message's **nonce** field for Nonce, converted to byte string (Big Endian)
- Initialize AES-256 in GCM Mode

- Initialize HMAC_DRBG with Security Strength of 256 bits
- Use HMAC_DRBG.GENERATE(32) as the Encryption Key (256 bits)
- Use HMAC_DRBG.GENERATE(12) as the Initialization Vector (IV) (96 bits)

AES-256 GCM Authentication Tag Use The 16 byte authentication tag resulting from the AES-GCM encrypt operation MUST be prefixed to the returned ciphertext. The decrypt operation will use the first 16 bytes of the ciphertext as the GCM authentication tag and the remainder of the ciphertext as the ciphertext in the decrypt operation.

AES-256 GCM Additional Authenticated Data When either **status_code** OR **status_message** are present, the AES-256 GCM authenticated data used in both the encrypt and decrypt operations MUST be: STRING(status_code) || status_message. Otherwise, there is no additional authenticated data. This provides that, while not encrypted, the status_code and status_message are authenticated.

Initial Public Key Retrieval for InvoiceRequest Encryption

Initial public key retrieval for InvoiceRequest encryption via EncryptedProtocolMessage encapsulation can be done in a number of ways including, but not limited to, the following:

1. Wallet Name public key asset type resolution - DNSSEC-validated name resolution returns Base64 encoded DER-formatted EC public key via TXT Record RFC 5480
2. Key Server lookup - Key Server lookup (similar to PGP's pgp.mit.edu) based on key server identifier (i.e., e-mail address) returns Base64 encoded DER-formatted EC public key RFC 5480
3. QR Code - Use of QR-code to encode SEC-formatted EC public key RFC 5480
4. Address Service Public Key Exposure

Payment / PaymentACK Messages with a HTTP Store & Forward Server

If a Store & Forward server wishes to protect themselves from spam or abuse, they MAY enact whatever rules they deem fit, such as the following:

- Once an InvoiceRequest or PaymentRequest is received, all subsequent messages using the same identifier must use the same Sender and Receiver public keys.
- For each unique identifier, only one message each of type InvoiceRequest, PaymentRequest, and PaymentACK may be submitted. Payment messages

may be submitted/overwritten multiple times. All messages submitted after a PaymentACK is received will be rejected.
- Specific messages are only saved until they have been verifiably received by the intended recipient or a certain amount of time has passed, whichever comes first.

Clients SHOULD keep in mind Receivers can broadcast a transaction without returning an ACK. If a Payment message needs to be updated, it SHOULD include at least one input referenced in the original transaction to prevent the Receiver from broadcasting both transactions and getting paid twice.

Public Key & Signature Encoding

- All x.509 certificates included in any message defined in this BIP MUST be DER [ITU.X690.1994] encoded.
- All EC public keys (**sender_public_key**, **receiver_public_key**) in any message defined in this BIP MUST be http://www.secg.org/sec2-v2.pdf ECDSA Public Key ECPoints encoded using http://www.secg.org/sec1-v2.pdf. Encoding MAY be compressed.
- All ECC signatures included in any message defined in this BIP MUST use the SHA-256 hashing algorithm and MUST be DER [ITU.X690.1994] encoded.
- All OpenPGP certificates must follow RFC4880, sections 5.5 and 12.1.

Implementation

A reference implementation for a Store & Forward server supporting this proposal can be found here:

Addressimo

A reference client implementation can be found in the InvoiceRequest functional testing for Addressimo here:

BIP75 Client Reference Implementation

BIP70 Extension

The following flowchart is borrowed from BIP70 and expanded upon in order to visually describe how this BIP is an extension to BIP70.

Mobile to Mobile Examples

Full Payment Protocol

The following diagram shows a sample flow in which one mobile client is sending value to a second mobile client with the use of an InvoiceRequest, a Store & Forward server, PaymentRequest, Payment and PaymentACK. In this case, the PaymentRequest, Payment and PaymentACK messages are encrypted using EncryptedProtocolMessage **and** the Receiver submits the transaction to the Bitcoin network.

Encrypting Initial InvoiceRequest via EncryptedProtocolMessage

The following diagram shows a sample flow in which one mobile client is sending value to a second mobile client using an EncryptedProtocolMessage to transmit the InvoiceRequest using encryption, Store & Forward server, and PaymentRequest. In this case, all Payment Protocol messages are encrypting using EncryptedProtocolMessage **and** the Sender submits the transaction to the Bitcoin network.

References

- BIP70 - Payment Protocol
- ECDH
- HMAC_DRBG
- NIST Special Publication 800-38D - Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC
- RFC6979
- Address Reuse
- FIPS 180-4 (Secure Hash Standard)