

Authors: Alan Reiner – Armory Technologies, Inc. Douglas Roark – Armory Technologies, Inc. Scott Hollenbeck – Verisign, Inc. Eric Osterweil – Verisign, Inc. Glen Wiley – Verisign, Inc.

© 2015 Armory Technologies, Inc. *This document is a draft. All contents are subject to change.* 

Version	Date	Author	Notes
0.1	2014/12/09	Alan Reiner	Initial draft.
0.15	2014/12/10	Douglas Roark	Cleaned up some grammar and spruced up various parts of the document.
0.16	2014/12/10	Douglas Roark	Added an appendix section on the applicable math from BIP 32.
0.16.1	2014/12/10	Douglas Roark	Minor adjustments.
0.5	2015/01/25	Douglas Roark	Added the data formats and other important information, updated the appendices, and cleaned up the document formatting.
0.7	2015/01/28	Alan Reiner & Douglas Roark	Various minor revisions.
0.8	2015/02/03	Douglas Roark	Clarify a point about lexicographic sorting of keys in multisig TxOut scripts, and update the SRP and PR data structures.
0.81	2015/02/10	Douglas Roark	Clarify a point regarding EC math.
0.85	2015/02/26	Douglas Roark	Added acknowledgments section and cleaned up various typos.
0.85.1	2015/02/27	Douglas Roark	Formatting changes.

## **Table of Contents**

Section 1 - Introduction	4
Section 2 - Structural Concepts	5
Section 2.1 - Bitcoin addresses	5
Section 3 - Implementation Details	5
Section 3.1 - BTCA components	6
Section 3.2 - Public Key Source Components	7
Section 3.2.1 - Version	8
Section 3.2.2 - isExternalSrc	8
Section 3.2.3 - isUserKey	9
Section 3.2.4 - isStealth	9
Section 3.2.5 - useHash160	9
Section 3.2.6 - useCompressed	10
Section 3.2.7 - isStatic	10
Section 3.2.8 - sourceStr	10
Section 3.2.9 - checksum	10
Section 3.2.10 - isChksumPresent	10
Section 3.3 - Constructed Script Components	.10
Section 3.3.1 - Version	11
Section 3.3.2 - useP2SH	12
Section 3.3.3 - scriptTemplate	12
Section 3.3.3.1 - Multisig scripts	12
Section 3.3.4 - numPKSs	12
Section 3.3.5 - pksEntry	.13
Section 3.3.6 - checksum	13
Section 3.3.7 - isChksumPresent	13
Section 3.4 - Public Key Relationship Proof	.13
Section 3.4.1 - Version	13
Section 3.4.2 - numMults	14
Section 3.4.3 - Multipliers	14
Section 3.5 - Script Relationship Proof	14
Section 3.5.1 - Version	14
Section 3.5.2 - numPKRPs	14
Section 3.5.3 - pkrpObj	.14
Section 3.6 - Payment Request	15
Section 3.6.1 - Version	15
Section 3.6.2 - Flags	15
Section 3.6.3 - numTxOutScripts	15
Section 3.6.4 - reqSize	16
Section 3.6.5 - unvalidatedScripts	16
Section 3.6.6 - recNames	16
Section 3.6.7 - srpLists	16

Section 4 - Usage Concepts	16
Section 4.1 - P2PKH.	16
Section 4.2 - PKRP / Multipliers	16
Section 4.3 - CS	17
Section 4.4 - Payment requests	17
Section 5 - Acknowledgments	18
Section 6 - References	18
Appendix A - BIP 32	20
Appendix A.1 - Hierarchical Deterministic Wallet Overview	20
Appendix A.2 - Mathematical Proofs	21
Appendix B - On-the-wire Example	22
Section 4.3 - CS Section 4.4 - Payment requests Section 5 - Acknowledgments Section 6 - References Appendix A - BIP 32 Appendix A.1 - Hierarchical Deterministic Wallet Overview Appendix A.2 - Mathematical Proofs Appendix B - On-the-wire Example	17 18 18 20 20 21 21

## **1** Introduction

Bitcoin provides strong cryptographic proofs throughout its ecosphere. However, areas remain where authentication can be improved. One example involves proving that an invoice sent to a paying entity contains accurate information. A compromised invoice could cause a payment to be sent to a malicious third party, or an incorrect amount of bitcoins to be sent, or both. As a workaround, a payee could use public key cryptography to sign the invoice. This is not secure enough because a compromised certificate authority can issue invalid certificates in the name of any entity.

This specification proposes a system for giving payees secure tools to procure payments. A payee can send an invoice with cryptographic proofs attached. The payer can then leverage an appropriate verification layer to prove that the payee sent the invoice, with the cryptographic proofs verifying that the invoice is accurate. The following goals apply to this specification.

- Prove a given payment script is related to a wallet.
- Proof of relationship does not reveal other scripts in the wallet.
- Proofs are optional. A payee can provide scripts without proof in order to preserve key chain privacy.
- Provide a mechanism that allows identity information to be embedded in "identity records" found in verifiable authentication chains.
- An identity scheme has a long lifetime. Entities create highly secure offline and/or multisig wallets, generate a master wallet identifier, and sign the identifier one time.
- With a secure cryptographic signature, payment scripts (with proofs) can be safely distributed via an "insecure" web server.
- An attacker compromising a web server cannot replace make payment scripts malicious: Payers can expect a proof that the script is related to the secure ID found in an identity record.
- Provide compatibility with Bitcoin Improvement Protocol (BIP) 32 [BIP32] keypair trees (aka hierarchical deterministic wallets).
- Provide compatibility with complex, non-standard scripts.

One possible solution is to use DNS-based Authentication of Named Entities (DANE) [RFC6394], which is based on DNSSEC [DNSSEC]. An individual DANE record can contain identitybased information. DNSSEC provides an inherent authentication chain that allows the payer to verify that the payee did send the payment request. In addition, if a domain's key is compromised, only that domain and sub-domain is compromised. In other words, payment requests issued by cleansocks.com

will not be compromised because the key for buymorestuff.com has been compromised. This is superior to the certificate authority (CA) system, where a compromised CA can forge keys for any domain.

# 2 Structural Concepts

### 2.1 Bitcoin addresses

The technical side of Bitcoin causes addresses to have ambiguous meanings. Therefore, the word "address" is not used in this specification without giving a specific definition for the address; an address indicates particular use types that are strictly defined.

At a high level, a Bitcoin address provided to receive payment is simply a chunk of data to be used in a particular script template type. Such templates are known as TxOut script templates [TXOUT]. A TxOut script specifies who exactly will receive payment. As of Jan. 2015, there are five standard TxOut types on the Bitcoin network. The TxOut types are not of specific interest except to help explain what this specification is attempting to accomplish.

As of Feb. 2015, the most common TxOut type on the Bitcoin network is known as the pay-topublic-key-hash (P2PKH) template. P2PKH uses addresses starting with 1. In P2PKH, the payer must insert a 20 byte payload, based on a secure hash of a public key, into the accompanying P2PKH script template. The P2PKH template is defined as follows:

#### OP\_DUP OP\_HASH160 <20 byte payload> OP\_EQUALVERIFY OP\_CHECKSIG

Another address example is an address starting with 3. For such addresses, the payer must insert the inner 20 byte payload into what is known as a pay-to-script-hash (P2SH) template, as defined in BIP 16 [BIP16]. The P2SH template is defined as follows:

#### OP\_HASH160 <20 byte payload> OP\_EQUAL

This specification focuses on that fact that the Bitcoin wallet software, at the time of payment, is ultimately constructing a destination, or destinations, for the coins. This specification defines a way for Bitcoin wallet software to communicate both the TxOut script to be paid, as well as a secure authentication chain for verifying that a TxOut script will be spendable only by the intended recipient.

## **3** Implementation Details

Unless otherwise noted, all variables are stored in network order.

The Bitcoin Core source code defines integers with VAR\_INT encoding [VI] and strings with VAR\_STR encoding [VS]. When integers are defined as VAR\_INT, the integers are unsigned and defined as follows.

Value	Variable Size	Format
<= 252 (0xfc)	1 byte	uint8_t integer.
<= 65535 (0xffff)	3 bytes	0xfd followed by the uint16_t (host order) integer.
<= 4294967295 (0xffffffff)	5 bytes	0xfe followed by the uint32_t (host order) integer.
$\leq 2^{64} - 1$ (0xffffffffffffffffffffffffffffffffffff	9 bytes	0xff followed by the uint64_t (host order) integer.

VAR\_STR variables are defined as follows.

Field Description	Data Type	Variable Size	Field Comments
String Length	VAR_INT	1-9 bytes	Length of the string. Length can be 0.
String	Unsigned char array	String Length bytes	Actual string. Not present if length=0.

## **3.1 BTCA components**

There are multiple components of the wallet ID verification system that need to be considered.

- *BTCA Resource Record*: A DANE-like record type, similar to TLSA [RFC6698] or the work-inprogress SMIMEA [SMIME07]. Such records can be signed with a zone operator's zone signing key (ZSK) which can be securely traced back to a trust anchor.
- *Public Key Source (PKS):* Communicates the simplest of ID information, typically about a single BIP32 key tree that would be used as a single signature wallet to manage funds. A PKS can also be used as a placeholder in a more complex, constructed script. A public key source can also be a reference to another resource for getting the public key source. This data structure will typically be signed with a ZSK.
- Constructed Script (CS): This is a data structure which includes a script template and a list of public key sources. The template and list of PKS's will typically be signed with a ZSK.
- Raw Payment Script (RPS): This is the final, complete script expected to receive payment as part of a given payment request. The payer may ignore any attached proofs and simply use this script to pay. The script will usually be provided with a PKRP and/or SRP (see below). However, if a PKS is a stealth address, requires a user-supplied key, or uses an external source, a PKRP or SRP cannot be included. There may be other conditions when a PKRP or SRP cannot be provided.

- Public Key Relationship Proof (PKRP): This is a list of multipliers to apply to a given PKS. If the PKS specifies a root-level public key, but the payment script uses a 3<sup>rd</sup> level public key, then the proof will actually be three 32-byte multipliers.
- Script Relationship Proof (SRP): This is a list of PKRPs to be applied to a Constructed Script, one per public key placeholder in the script template.

# **3.2 Public Key Source Components**

Public key cryptography performs a critical piece of Bitcoin functionality. When coins are sent to another entity, control of the coins is actually transferred to a public key, or public key data. Using a public key controlled by a payee, a payer can use standard TxOut scripts or custom scripts to make payments. In such a case, a signed PKS can be placed in an identity record and downloaded by a paying entity as part of a payment request.

The public key source must contain metadata on top of the public key. The metadata is as follows.

Variable	Description
version	PKS version.
sourceStr	String containing raw key material or a resource (e.g., URL) describing the external location of the raw key material.
isExternalSrc	Boolean indicating whether or not <i>sourceStr</i> is an external key resource.
useCompressed	Boolean indicating if the user should use a compressed version of the public key.
useHash160	Boolean indicating if the user should apply RIPEMD-160(SHA256(PK)) [RIPEMD160] [SHA2] to the public key before usage.
isStatic	Boolean indicating that <i>sourceStr</i> must be used as-is when applied to script templates.
isStealth	Boolean indicating that sourceStr contains a "stealth address." The paying party must generate payment information based on the stealth address and an agreed- upon key exchange method. The paying party must then attach extra information to a transaction so that the payee can determine if the payment information in the transaction actually applies to the payee.
isUserKey	Boolean indicating that the user must supply their own key.
isChksumPresent	Boolean indicating that a checksum is part of the PKS.
checksum	A 4-byte subsection of a double-SHA256() hash of all the other data in the PKS.

(c) 2015 Armory Technologies, Inc.

The PKS data format is as follows. Unless otherwise specified, all data is transmitted in network order.

+	Byte + 0	Byte + 1	Byte + 2	Byte + 3		
Bytes 0-2	Version	Flags				
Bytes 3-(m-1)	sourceStr (VAR_STR)					
Bytes m-(m+3)	Checksum					

The flag field is defined as follows, with reserved bytes set to 0. Bits are listed in network order.

+	Bit + 7	Bit + 6	Bit + 5	Bit + 4	Bit + 3	Bit + 2	Bit + 1	Bit + 0
Bits 15-8		Reserved (set to 0)						
Bits 7-0	Reserved (set to 0)	isChksum Present	isExtSrc	isUsrKey	isStealth	hash160	useComp	isStatic

### 3.2.1 Version

The field is 1 byte large. The current PKS version number is 0.

### 3.2.2 isExternalSrc

If the *isExternalSrc* flag in the PKS is set, the other flags are irrelevant. *sourceStr* will represent one of the following:

- An HTTPS URL, optionally with an attached SSL/CA certificate fingerprint. This indicates that BIP70/BIP72 payment requests [BIP70] [BIP72] can be obtained from the specified URL. If a SSL/CA fingerprint is included, the payee can expect the BIP70/BIP72 payment request to be signed with a certificate that traces back to the specified SSL cert. A PKS that uses an external source must not be provided as part of a Constructed Script. The payment request retrieved from the specified HTTPS URL will provide complete script information for payment.
- A record name or URL to another domain/zone, from which the user can expect to retrieve a PKS. This is a recursive call, allowing chaining of PKS records. This is most useful in escrow situations, where the Constructed Script specifies a PKS controlled by an independent third-party. Instead of initially copying the PKS information into the CS record, the location of the PKS is supplied. This helps the payee identify the PKS, as well as allowing the third-party to update their PKS record without requiring the original CS to be updated.

If this flag is used, no RPS will be provided by the payee, as the payee cannot predict all

components of the final script to be paid.

#### 3.2.3 isUserKey

If the user key flag is enabled, the user must insert data instead of relying on *sourceStr*. An example involves a 2-of-3 multi-signature (multisig) escrow transaction. When coins are sent to a multisig address, every potential signing key must be present in the transaction. The user sending coins may also be a user who can provide a signature authorizing dispersal of the coins. In such a case, the user must supply a public key as part of the multisig transaction. The key must not be affected by any flags or multipliers. The payer is also assumed to understand how to insert appropriate data based on the appropriate script type.

If this flag is used, no RPS will be provided by the payee, as the payee cannot predict all components of the final script to be paid.

#### 3.2.4 isStealth

If the stealth flag is enabled, the supplied source string is a stealth address. It is assumed that the payer and payee will understand how to use the stealth address to generate appropriate payment information.

A core idea of this specification is that a payer does not know exact payment details ahead of time; payment details must be provided by the payee. In particular, the payer must calculate the final payment destination. By default, anybody who sees payment information generated according to this specification can determine where the coins are going. Such viewing means anybody can determine how many coins the payee controls at the given address.

Stealth addresses obfuscate the identity of the receiving party. Using carefully constructed mathematical concepts (e.g., elliptic curve Diffie-Hellman key exchanges [SEC1]), a payee can provide enough information to allow a payer to dynamically generate Bitcoin network payment information understood only by the payer and payee. Such a setup requires the wallet ID verification code to specify that extra metadata is required to determine the final payment destination. Extra metadata may also be required to be sent to the payee.

The concept of Bitcoin "stealth addresses" is relatively new. One example currently in use was developed in 2013 as part of the *sx* program [SX1]. Online documentation that explains the *sx* stealth process [SX2] is available.

If this flag is used, no RPS will be provided by the payee, as the payee cannot predict all components of the final script to be paid.

#### 3.2.5 useHash160

If the useHash160 flag is enabled, the final public key will be hashed using SHA256 and then

RIPEMD-160. (The SHA256-RIPEMD sequence is also known as Hash160().) If enabled, the RIPEMD-160 result is what will be inserted into the script template.

### 3.2.6 useCompressed

If the compression flag is enabled, the final public key will be "point compressed" before being inserted into the script template. Sect. A.3.1.3 of ANSI X9.62:2005 [X962] explains how elliptic curve point compression works. If useCompressed is set, point compression must be applied before hashing the key.

### 3.2.7 isStatic

If the static flag is enabled, the data in *sourceStr* is to be inserted into a script template as-is. No calculations will be performed.

### 3.2.8 sourceStr

The PKS record contains a source string encoded as a VAR\_STR. When *isExternalSrc* is false, the string contains the public key source material required to generate the final payment destination. When *isExternalSrc* is set to true, the string contains a URL pointing to a server that will provide the required public key source material.

Note that *sourceStr* may be a zero-length string; the payee has the option of sending a fully formed payment script.

### 3.2.9 checksum

When a PKS is sent to the entity that will post the PKS in an identity record, the posting entity will require reasonable certainty that the received data is what was intended to be posted. Therefore, a checksum is added to PKS records to be posted in an identity record. The checksum will consist of the first four bytes (network order) of a SHA256(SHA256()), or double-SHA256, hash of the other data in the PKS. The checksum must be present for a PKS embedded by itself in an identity record. The checksum is optional, and not recommended, for a PKS embedded in a CS.

### 3.2.10 isChksumPresent

A CS may use multiple PKS entries. Because a CS will also have a checksum, the usage of a checksum in a PKS would be redundant. The *isChksumPresent* flag is added to indicate whether or not the PKS has a checksum. The checksum must be present for a PKS embedded by itself in an identity record. The checksum is optional, and not recommended, for a PKS embedded in a CS.

## **3.3 Constructed Script Components**

The constructed script is what provides payment information to the payer. A signed CS can be

placed in an identity record and downloaded by a paying entity as part of a payment request.

isChksumPresent Boolean indicating that a checksum is part of the CS.

Variable	Description
version	PKS version.
useP2SH	Boolean indicating whether or not the transaction will use the P2SH transaction method.
scriptTemplate	A TxOut template that contains specially designed opcodes where public key data will be inserted to construct the final TxOut script to be paid.
pksList	A list of PKS entries, one per placeholder in the scriptTemplate

The constructed script source must contain metadata. The metadata is as follows.

The CS data format is as follows. Unless otherwise specified, all data is transmitted in network order.

A 4-byte subsection of a double-SHA256() hash of all the other data in the PKS.

+	Byte + 0	Byte + 1	Byte + 2	Byte + 3	
Bytes 0-2	Version	Fla	ags		
Bytes 3-m		scriptTemplate (VAR_STR)			
Byte m+1	Byte m+1 numPKSs (VAR_INT)				
Bytes (m+2)-n	pksEntry (numPKSs * VAR_STR)				
Bytes (n+1)-(n+4)		Checksum	ı (4 bytes)		

The flag field is defined as follows, with reserved bytes set to 0. Bits are listed in network order.

+	Bit + 7	Bit + 6	Bit + 5	Bit + 4	Bit + 3	Bit + 2	Bit + 1	Bit + 0
Bits 15-8		Reserved (set to 0)						
Bits 7-0			Reserved	(set to 0)			isChksum Present	useP2SH

### 3.3.1 Version

checksum

The field is 1 byte large. The current CS version number is 0.

### 3.3.2 useP2SH

When P2SH is to be used, the payment script will be generated. The script will then be hashed using the Hash160 sequence and placed into a standard P2SH TxOut script. The P2SH TxOut script is what will actually be included in the final transaction.

### 3.3.3 scriptTemplate

*scriptTemplate* will match the final TxOut script with two important exceptions.

First, if the *useP2SH* flag is set, the final *scriptTemplate* output will be processed using standard P2SH rules, as discussed in Sect. 3.3.2.

Second, the byte 0xff becomes a special "escaped" value. Any keys in *scriptTemplate* are replaced with two bytes: 0xff, and the number of keys to go in the escaped location. If only one key should go in a script location, 0xff01 will be used. If two keys are supposed to go together, 0xff02 will be used. If any other part of the script is intended to have a 0xff byte, the 0xff byte will be replaced with 0xff00. Once the final payment script is developed, the escaped values will be replaced appropriately with either 0xff (0xff00) or the appropriate number of public key data entries (0xff01 – 0xffff).

### 3.3.3.1 Multisig scripts

When dealing with multisig transactions, sorting of associated public keys is important. Each possible key order results in a different effective address. In the absence of sorting, wallet software may not be able to recognize relevant TxOut scripts, and different devices would produce different payment requests. For this reason, a community standard has arisen where public keys are always sorted before being inserted into a TxOut script to guarantee that all wallet software generates identical payment scripts.

When an entry in a key list contains multiple keys, the keys must be lexicographically sorted before being inserted into the final TxOut script. If the payee has reasons for keeping the keys in a particular, non-lexicographic order, the payee can simply use multiple 0xff01 entries in the CS instead of one entry for all applicable keys (e.g., 0xff01ff01ff01 instead of 0xff03).

Each key must be derived before being sorted and placed in the script template. There is no way to determine beforehand which PKS will occupy a given position in the final TxOut script. Therefore, any PKS entries in the CS that must be lexicographically sorted in the final TxOut script need not be lexicographically sorted in the CS.

### 3.3.4 numPKSs

*numPKSs* specifies the number of PKS objects in the CS. Note that the number of entries must match the total number of keys specified in *scriptTemplate*.

### 3.3.5 pksEntry

There will be *numPKSs* PKS objects inserted in the CS. The PKS objects must be inserted in the order in which the public keys will be inserted in *scriptTemplate*. If an escaped key entry lists multiple keys, the PKS objects do not need to be in sorted order in the CS. The PKS objects only need to be in an order that allows the proper PKS objects to be used in the proper escaped key entries.

### 3.3.6 checksum

When a PKS is sent to the entity that will post the PKS in an identity record, the posting entity will require reasonable certainty that the received data is what was intended to be posted. Therefore, a checksum is added to PKS records to be posted in an identity record. The checksum will consist of the first four bytes (network order) of a SHA256(SHA256()), or double-SHA256, hash of the other data in the PKS. The checksum must be present for a PKS embedded by itself in an identity record. The checksum is optional, and not recommended, for a PKS embedded in a CS.

### 3.3.7 isChksumPresent

A CS may use multiple PKS entries. Because a CS will also have a checksum, the usage of a checksum in a PKS would be redundant. The *isChksumPresent* flag is added to indicate whether or not the PKS has a checksum. The checksum must be present for a PKS embedded by itself in an identity record. The checksum is optional, and not recommended, for a PKS embedded in a CS.

## 3.4 Public Key Relationship Proof

The PKRP is a struct with 32-byte multipliers meant to be applied to key material in a PKS. The PKRP will be attached, via an SRP, to a payment request sent to a paying entity by the payee. The multipliers must be applied, in order, to the accompanying PKS material.

The PKRP data format is seen below. If no derivation is required for a given key, the accompanying SRP must indicate that there are no PKRPs by setting *numPKRPs* to 0x00. In addition, because *numMults* is intended to be a VAR\_INT but is constrained to one byte, *numMults* cannot be larger than 252.

+	Byte + 0	Byte + 1	Byte + 2	Byte + 3
Bytes 0-1	Version (1 byte)	numMults (1 byte)		
Bytes 2-m		multiplier (numM	lults * VAR_STR)	

### 3.4.1 Version

The field is 1 byte large. The current PKRP version number is 0.

### 3.4.2 numMults

Specifies the number of multipliers to apply to the matching public key. *numMults* must match the number of public key data entries actually contained in the CS.

### 3.4.3 Multipliers

When a child public key is derived, multipliers are used as part of the mathematical process. The PKRP contains a list of multipliers that must be used, in order, when deriving the child public key. Using the multipliers out of order will lead to incorrect results. Appendix A.2 discusses the math in finer detail.

The multipliers must be 32 bytes. This implicitly enforces the idea that this spec currently supports only BIP 32 multipliers. The multipliers must also be used in a setting where the keys are uncompressed. If a compressed key is the final product, an uncompressed public key must be used and then compressed once all the math has been completed.

## 3.5 Script Relationship Proof

The SRP attached to a payment request. The SRP contains the list of PKRPs that to apply to the keys in a CS.

The SRP data format is seen below. The number of PKRPs in the SRP must match the number of keys to be derived in the script template of the CS. For example, the first PKRP in the SRP will be applied to the first PKS entry in the accompanying CS.

+	Byte + 0	Byte + 1	Byte + 2	Byte + 3
Bytes 0-1	Version (1 byte)	numPKRPs (VAR_INT)		
Bytes 2-m	pkrpObj (numPKRPs * VAR_STR)			

### 3.5.1 Version

The field is 1 byte large. The current SRP version number is 0.

### 3.5.2 numPKRPs

Specifies the number of PKRPs to apply to the matching SRP. *numPKRPs* must match the number of keys referenced in the payment request's CS.

### 3.5.3 pkrpObj

SRPs are what are attached to the payment request. The SRPs contain PKRPs that will be used

### (c) 2015 Armory Technologies, Inc.

to apply multipliers to the public key data in the PKS entries in the CS. When keys are created for *scriptTemplate* in the appropriate CS, each PKS intended for *scriptTemplate* has a matching SRP. The PKRPs will be encoded as VAR\_STRs.

## **3.6 Payment Request**

The final payment request sent to a paying entity will have multiple segments and is structured as follows.

+	Byte + 0	Byte + 1	Byte + 2	Byte + 3	
Bytes 0-2	Version (1 byte)	Flags (2 bytes - RESERVED)			
Bytes 3-5	numTxOutScripts (VAR_INT)				
Bytes 6-8	reqSize (VAR_INT)				
Bytes 9-a	unvalidatedScripts (numTxOutScripts * VAR_STR)				
Bytes (a+1)-b	recNames (numTxOutScripts * VAR_STR)				
Bytes (b+1)-c	<pre>srpLists (numTxOutScripts * VAR_STR)</pre>				

It is important to note that the payment request does not have to be validated by the recipient; the payment request is designed to give the recipient options. If the paying entity wishes to blindly trust the fully constructed script in the payment request, the script can be used as-is, with no regard for any other data in the request. If the recipient wishes to validate the payment request, the recipient can obtain the PKS or CS listed in the payment request, validate the PKS or CS, and use the SRPs to generate the keys to be placed in the script template.

The *unvalidatedScripts*, *recNames*, and *srpLists* fields must be in the same order. For example, the first *unvalidatedScript* object can be recreated using the first *recName* and *srpList* objects.

### 3.6.1 Version

The field is 1 byte large. The current payment request version number is 0.

### 3.6.2 Flags

For now, no flags exist. Two bytes are reserved for possible future flags.

### 3.6.3 numTxOutScripts

The number of TxOut scripts is listed in a VAR\_INT that can be up to 3 bytes, implying a maximum of 65,535 TxOut scripts. However, *reqSize*, which lists the number of bytes after *reqSize*, can also be a 3 byte variable. The idea is to prevent DoS attacks by sending requests that require excessive

amounts of work or generate very large TxOut scripts.

#### 3.6.4 reqSize

The number of combined bytes in the *unvalidatedScript*, *recName*, and *srpList* fields.

#### 3.6.5 unvalidatedScripts

The completed TxOut scripts that can be used as-is.

#### 3.6.6 recNames

A list of names of records that contain the TxOut scripts to be validated. The user is assumed to understand how to obtain identity records based off the record names.

#### 3.6.7 srpLists

A list of SRP objects to be applied to the appropriate identity records. The SRP objects will be encoded as VAR\_STRs.

# 4 Usage Concepts

### 4.1 P2PKH

An entity that wishes to use nothing but P2PKH scripts for their identity can embed just a PKS in the BTCA record. The simplest case would be a static address, such as a vanity address that never changes. The output is expected to be put into a P2PKH script. Example BTCA records are seen below.

BTCA Record (PKS): [Flags(static, useHash160) | RawPublicKey]

or

BTCA Record (PKS): [Flags(static) | Hash160(RawPubKey)]

Using only a static PKS is not recommended due to inflexibility if the key is compromised. The recommended use case for a P2PKH payment specifies a raw public key in the PKS object, with PKRPs provided in a CS that uses a P2PKH script.

BTCA Record (PKS): [Flags(useCompressed) | rootPublicKey]

### 4.2 PKRP / Multipliers

When a PKRP is used in a CS, the public key in the PKS is assumed to be a level 0 (root) node in a BIP 32 keypair tree controlled by the payee. In theory, any number of levels/multipliers may be used. This specification requires the use of one multiplier to obtain the final result. It is highly

recommended that the payee distribute multipliers based on level 3 nodes on the tree. That is, usage of three multipliers to create one final multiplier is recommended. The payee must start with the root key in the PKS and derive the public key at the desired location, saving each multiplier (one per level) generated as part of the process. The payer must then apply each multiplier to the root key in the PKS and derive a copy of the resultant public key. The result is sent as proof:

Payment Request: [ScriptToPay(RPS) | URLOrPKSRecordName | mult0]

The payer can pay the RPS without any verification if desired. The payer can also verify the CS record, fetch the PKS record or URL, verify the authentication chain, apply the multiplier to the raw public key in the PKS or URL, and insert the resultant public key into the specified script type. The payer should also check that the result matches the included RPS.

## 4.3 CS

A simple P2PKH could be represented using a CS, though it would be unnecessary to do so:

**BTCA Record (CS):** [ <OP\_DUP OP\_HASH 0xff01 OP\_EQ OP\_CHKSIG> | 0x01 | [Flags(useCompressed) | rootPublicKey] | useP2SH=False ]

A standard 2-of-3 multisig P2SH script where one organization controls all the keys, might use the following:

**BTCA Record (CS)**: [ <OP\_2 0xff03 OP\_3 OP\_CHECKMULTISIG> | PKS1 | PKS2 | PKS3 | useP2SH=True] PubKeySrcList: ([Flags1 | rootPub1], [Flags2 | rootPub2], [Flag3 | rootPub3])

If the owner wanted to deviate from convention and require the final three keys to be always inserted in script-specified order (not sorted before replacing the 0xff03 sequence), then they would use the identical BTCA record but with only 0xff01 bytes.

BTCA Record (CS): [ <OP2 0xff01 0xff01 0xff01 OP\_3 OP\_CHECKMULTISIG> | PKS1 | PKS2 | PKS3 | useP2SH=True]

PubKeySrcList: ([Flags1 | rootPub1], [Flags2 | rootPub2], [Flag3 | rootPub3])

## 4.4 Payment requests

When an organization generates a payment request on their script distribution server, it will calculate three public keys from three different BIP32 public key trees, and will save off the multipliers for each calculation. It will then insert them into a proof to be incldued with the raw payment script.

**Payment Request:** [ScriptToPay(RPS) | RecordNameOrURL | 0x03 | [0x03 | m11 | m12 | m13] | [0x03 | m21 | m22 | m23], [0x03 | m31 | m32 | m33] ]

In both cases, the payer will find the script template and the pubkey source list in the identity record. It will apply the lists of multipliers to each key source, it will end up with 3 public keys, it will sort them lexicographically, it will insert them into the script template in place of the 0xff03 byte sequence, and then it will convert it to a P2SH script. If the result matches the RPS in the payment request, the payer's software will indicate that the payment script has been verified.

Perhaps an organization wishes to specify that payment is to be put into 2-of-2 escrow, using a third-party that the payer should trust. The process requires the escrow service to shift keys constantly, thereby making constant record updates infeasible. One solution is to use an external key source.

BTCA Record (CS): [ <OP\_ 2 0xff02 OP\_2 OP\_CHECKMULTISIG> | PKS1 | PKS2 | useP2SH=False]

**PubKeySrcList**: ([Flags1 | rootPub1], [Flags(isExternal) | btca:escrow.\_btca.armory.com])

**Payment Request:** [ScriptToPay(RPS) | RecordNameOrURL | 0x02 | [0x03 | m11 | m12 | m13] | [0x03 | m21 | m22 | m23]]

In this case, the script-distribution server will actually fetch the script and proof of that script from the remote resource, and send that to the payer. The payer's software, when attempting to compute/verify the second public key source, will fetch an identity record from the specified appropriate URL, and match the proof against the record. The payment request looks the same as if it wasn't external, but the BTCA record indicates this extra step must be performed to do the verification. The benefit of doing this is that it proves:

- The public key being inserted really does belong to the stated third-party.
- The third-party can change their BTCA record without requiring this BTCA record to be changed.

# 5 Acknowledgments

This work was funded and supported by Verisign, Inc. Scott Hollenbeck, Eric Osterweil, and Glen Wiley at Verisign provided feedback and assistance with the specification, including helping with an implementation based on DNSSEC.

Ideas presented in this specification have roots in Dr. Pieter Wuille's work on hierarchical deterministic wallets, as seen in BIP 32 [BIP32].

# 6 References

[BIP16] Andresen, G., "Pay to Script Hash", <u>BIP 16</u>, Jan. 3, 2012.

(c) 2015 Armory Technologies, Inc.

[BIP32]	Wuille, P., "Hierarchical Deterministic Wallets", <u>BIP 32</u> , Jan. 15, 2014.
[BIP70]	Andresen, G. and Hearn, M., "Payment Protocol", <u>BIP 70</u> , July 29, 2013.
[BIP72]	Andresen, G., "bitcoin: uri extensions for Payment Protocol", <u>BIP 72</u> , July 29, 2013.
[CRYPTO]	Stallings, W., "Cryptography and Network Security: Principles and Practice – Sixth Edition", Pearson Education, 2014.
[DNSSEC]	Arends, R., Austein, R., Larson, M., et al, "DNS Security Introduction and Requirements", <u>RFC 4033</u> , Mar. 2005.
[MULTISIG]	"bitcoin.org/guide_transactions.md", Jan. 8, 2015, < <u>https://github.com/bitcoin/bitcoin.org/blob/2d46b41e3e05715306336cbba9e17b4f26441473/_includes/guide_transactions.md</u> >.
[RFC6394]	Barnes, R., "Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)", <u>RFC 6394</u> , Oct. 2011.
[RFC6698]	Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", <u>RFC 6698</u> , Aug. 2012.
[RIPEMD160]	Dobbertin, H., Bosselaers, A., and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD", Apr. 18, 1996, < <u>http://homes.esat.kuleuven.be/~bosselae/ripemd160/pdf/AB-9601/AB-9601.pdf</u> >.
[SEC1]	Certicom Research, "SEC 1: Elliptic Curve Cryptography", <u>SEC 1</u> , Sep. 20, 2000.
[SEC2]	Certicom Research, "SEC 2: Recommended Elliptic Curve Parameters", <u>SEC 2</u> , Sep. 20, 2000.
[SHA2]	National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-2, Aug. 1, 2002.
[SMIME07]	Hoffman, P. and J. Schlyter, "Using Secure DNS to Associate Certificates with Domain Names For S/MIME", <u>draft-ietf-dane-smime-07</u> , Aug. 22, 2014.
[SX1]	"spesmilo/sx", Jan. 11, 2015, < <u>https://github.com/spesmilo/sx/</u> >.
[SX2]	"9. Stealth Payments – sx 1 documentation", < <u>https://sx.dyne.org/stealth.html</u> >.
[TXOUT]	"Transaction – Bitcoin", Sep. 19, 2014, < <u>https://en.bitcoin.it/wiki/Transaction</u> >.
[VI]	"Protocol specification – Bitcoin", Dec. 10, 2014, < <u>https://en.bitcoin.it/wiki/Protocol_specification#Variable_length_integer</u> >.
[VS]	"Protocol specification – Bitcoin", Dec. 10, 2014, < <u>https://en.bitcoin.it/wiki/Protocol_specification#Variable_length_string</u> >.
[X962]	American National Standards Institute, "Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks", <u>ANSI X9.62</u> , Oct. 14, 2012.

# Appendix A - BIP 32

### **Appendix A.1 - Hierarchical Deterministic Wallet Overview**

BIP 32 is a Bitcoin-related specification that provides one method for implementing hierarchical deterministic wallet verification. At a high level, the idea is that a cryptographically secure seed and chain code are developed in a deterministic manner. The seed and chain code are then used, along with a pseudo-random number, to generate a master private key and a master public key. A hierarchical tree of keys, public and/or private, can then be generated in a deterministic manner using the chain code and an index value indicating which leaf is to be generated. Table 1 shows an example of a hierarchical deterministic (HD) tree.



### BIP 32 - Hierarchical Deterministic Wallets

#### Table 1: BIP 32 hierarchy diagram

Generating the child private or public key requires a multiplier based in part on the parent key and chain code. The scheme will work with wallet verification because of the multiplier. Given a root public key and a series of multipliers, a user can generate keys that coincide with the keys generated at a given index point. Any generated keys are the keys that will receive the funds sent by the payer.

All private keys must be kept secret, as is a standard best practice for public key cryptography; private key control will allow a user to spend any coins controlled by the private key. In addition, whenever possible, public keys should not be exposed to anybody other than intended recipients and to parties with specific interests (e.g., auditors). Anybody with a public key will know how many coins are held by the accompanying private key. This represents a potential loss of privacy.

The chain code must also be kept secret. A subtle point of BIP 32 is that, while not explicitly stated in BIP 32, derived chain codes are the same for public keys and non-hardened private keys. (Hardened private keys use a different formula to derive children and, subsequently, are not discussed.) The following equations generate a 64 byte value from

which the multiplier will be generated.

Non-hardened private key derivation: I = HMAC-SHA512(Key =  $c_{par}$ , Data =  $ser_P(point(k_{par})) \parallel ser(i)$ ) Public key derivation: I = HMAC-SHA512(Key =  $c_{par}$ , Data =  $ser_P(K_{par-comp}) \parallel ser(i)$ )

In both cases, *I* actually yields the same value. Sect. A.4.3 of [X962] describes how an elliptic curve public key is generated from the accompanying private key. The result is the following equation.

#### K = point(k)

Because *I* is the same for public key and non-hardened private key derivation,  $I_L$  and  $I_R$  will also be the same. Therefore, the multiplier and the chain code will be the same for the children of non-hardened private and public keys. Distribution of the multiplier is acceptable because the multiplier can be used only to obtain one specific child, and private keys are assumed to be kept safe. Chain code distribution is not acceptable because all child public keys can be generated using a public key's accompanying chain code, along with a non-hardened private key's children should the private key be exposed.

### **Appendix A.2 - Mathematical Proofs**

In BIP 32, the section "Public Parent Key  $\rightarrow$  public child key" explains the math behind deriving a child public key from a parent public key. The math is what makes the wallet identity verification specification possible. If the index value is non-hardened, a private key is not required to derive a public key. (Hardened index values require a private key to be derived and then converted into a public key.)

The following math is applied to obtain child keys with non-hardened index values. Unless otherwise noted, all keys in the mathematical equations must be uncompressed.

 $\mathbf{K}_{par}$  = Parent public key. (65 bytes)  $\mathbf{c}_{par}$  = Parent chain code. (32 bytes)

 $\mathbf{K}_{i}$  = Child public key. (65 bytes)

 $\mathbf{i} = \text{Index} (32 \text{-bit unsigned integer}) \text{ of the child key. (4 bytes)}$ 

I = Result from an HMAC-512 [SHA512] instantiation. (64 bytes)

 $I_L$  = Left 32 bytes of the result from an HMAC-512 instantiation.

**point**(*X*) = Multiplication of the base point of the secp256k1 elliptic curve by scalar X. The result is an uncompressed elliptic curve point. (65 bytes)

 $ser_P(K)$  = Serialized (most significant byte first) version of a compressed public key. (33 bytes)

**parse(***X***)** = Interpret serialized data (most significant byte first) as an integer.

+ = Elliptic curve point addition

(c) 2015 Armory Technologies, Inc.

|| = Concatenation

 $I = HMAC-SHA512(Key = c_{par}, Data = ser_P(K_{par}) || ser(i))$ K<sub>i</sub> = point(parse(I<sub>L</sub>)) + K<sub>par</sub>

Because the math is standardized,  $I_L$  can be distributed as a base point multiplier. This is acceptable because HMAC-SHA512 is a one-way cryptographic hash function. This means that the result cannot be easily reverse engineered to obtain the hashing key or data. In other words, multiplier  $I_L$  cannot be used to find the parent chain code. Because multiplier  $I_L$  is used in conjunction with a public key to find another public key, public distribution of multiplier  $I_L$  is acceptable.

Keys must be provided in an uncompressed or compressed state, as stated in the equations. Uncompressed keys are 65 bytes. Compressed keys are 33 bytes. Sects. A.3.1.3.1 and A.3.1.3.2 of [X962] discusses how to perform elliptic curve point compression over a prime field, which is the field type used by the secp256k1 curve. Sects. A.5.7 and A.5.8 of [X962] respectively discuss how to convert an elliptic curve point to an octet string and vice versa. Both octet strings and elliptic curve points are network order, as seen in Sects. L.2.2, L.2.3, and L.2.7.1 of [X962].

# **Appendix B - On-the-wire Example**

TO BE WRITTEN USING BIP 32 SAMPLE DATA