```
BIP: 137
Layer: Applications
Title: Signatures of Messages using Private Keys
Author: Christopher Gilliard <christopher.gilliard@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0137
Status: Final
Type: Standards Track
Created: 2019-02-16
License: BSD-2-Clause
```

Abstract

This document describes a signature format for signing messages with Bitcoin private keys.

The specification is intended to describe the standard for signatures of messages that can be signed and verfied between different clients that exist in the field today. Note: that a new signature format has been defined which has a number of advantages over this BIP, but to be backwards compatible with existing implementations this BIP will be useful. See BIP 322 [1] for full details on the new signature scheme.

One of the key problems in this area is that there are several different types of Bitcoin addresses and without introducing specific standards it is unclear which type of address format is being used. See [2]. This BIP will attempt to address these issues and define a clear and concise format for Bitcoin signatures.

Copyright

Motivation

Since Bitcoin private keys can not only be used to sign Bitcoin transactions, but also any other message, it has become customary to use them to sign various messages for differing purposes. Some applications of signing messages with a Bitcoin private key are as follows: proof of funds for collateral, credit worthiness, entrance to events, airdrops, audits as well as other applications. While there was no BIP written for how to digitally sign messages with Bitcoin private keys with P2PKH addresses it is a fairly well understood process, however with the introduction of Segwit (both in the form of P2SH and bech32) addresses, it is unclear how to distinguish a P2PKH, P2SH, or bech32 address from one another. This BIP proposes a standard signature format that will allow clients to distinguish between the different address formats.

Specification

Background on ECDSA Signatures

(For readers who already understand how ECDSA signatures work, you can skip this section as this is only intended as background information.) Elliptic Curve Digital Signature Algorithm or ECDSA is a cryptographic algorithm used by Bitcoin to ensure that funds can only be spent by their rightful owners.

A few concepts related to ECDSA:

private key: A secret number, known only to the person that generated it. A private key is essentially a randomly generated number. In Bitcoin, someone with the private key that corresponds to funds on the block chain can spend the funds. In Bitcoin, a private key is a single unsigned 256 bit integer (32 bytes).

public key: A number that corresponds to a private key, but does not need to be kept secret. A public key can be calculated from a private key, but not vice versa. A public key can be used to determine if a signature is genuine (in other words, produced with the proper key) without requiring the private key to be divulged. In Bitcoin, public keys are either compressed or uncompressed. Compressed public keys are 33 bytes, consisting of a prefix either 0x02 or 0x03, and a 256-bit integer called x. The older uncompressed keys are 65 bytes, consisting of constant prefix (0x04), followed by two 256-bit integers called x and y (2 * 32 bytes). The prefix of a compressed key allows for the y value to be derived from the x value.

signature: A number that proves that a signing operation took place. A signature is mathematically generated from a hash of something to be signed, plus a private key. The signature itself is two numbers known as r and s. With the public key, a mathematical algorithm can be used on the signature to determine that it was originally produced from the hash and the private key, without needing to know the private key. Signatures are either 73, 72, or 71 bytes long, with probabilities approximately 25%, 50% and 25% respectively, although sizes even smaller than that are possible with exponentially decreasing probability. Source [3].

Conventions with signatures used in Bitcoin

Bitcoin signatures have the r and s values mentioned above, and a header. The header is a single byte and the r and s are each 32 bytes so a signature's size is 65 bytes. The header is used to specify information about the signature. It can be thought of as a bitmask with each bit in this byte having a meaning. The serialization format of a Bitcoin signature is as follows:

[1 byte of header data][32 bytes for r value][32 bytes for s value]

The header byte has a few components to it. First, it stores something known as the recId. This value is stored in the least significant 2 bits of the header. If the header is between a value of 31 and 34, this indicates that it is a compressed address. If the header value is between 35 and 38 inclusive, it is a p2sh segwit address. If the header value is between 39 and 42, it is a bech32 address.

Procedure for signing/verifying a signature

As noted above the signature is composed of three components, the header, r and s values. r/s can be computed with standard ECDSA library functions. Part of the header includes something called a recId. This is part of every ECDSA signature and should be generated by the ECDSA library. The recId is a number between 0 and 3 inclusive. The header is the recId plus a constant which indicates what type of Bitcoin address this is. For P2PKH address using an uncompressed public key this value is 27. For P2PKH address using compressed public key this value is 31. For P2SH-P2WPKH this value is 35 and for P2WPKH (version 0 witness) address this value is 39. So, you have the following ranges:

- 27-30: P2PKH uncompressed
- 31-34: P2PKH compressed
- 35-38: Segwit P2SH
- 39-42: Segwit Bech32

To verify a signature, the recId is obtained by subtracting this constant from the header value.

Sample Code for processing a signature

Note: this code is a modification of the BitcoinJ code which is written in java.

```
   public static ECKey signedMessageToKey(String message, String signatureBase64) throws Sig
byte[] signatureEncoded;
try {
signatureEncoded = Base64.decode(signatureBase64);
} catch (RuntimeException e) {
// This is what you get back from Bouncy Castle if base64 doesn't decode :(
throw new SignatureException("Could not decode base64", e);
}
// Parse the signature bytes into r/s and the selector value.
if (signatureEncoded.length < 65)
throw new SignatureException("Signature truncated, expected 65 bytes and got " + signatureEn
int header = signatureEncoded[0] & 0xFF;
// The header byte: 0x1B = first key with even y, 0x1C = first key with odd y,
//                  0x1D = second key with even y, 0x1E = second key with odd y
if (header < 27 || header > 42)
throw new SignatureException("Header byte out of range: " + header);
BigInteger r = new BigInteger(1, Arrays.copyOfRange(signatureEncoded, 1, 33));
BigInteger s = new BigInteger(1, Arrays.copyOfRange(signatureEncoded, 33, 65));
ECDSASignature sig = new ECDSASignature(r, s);
byte[] messageBytes = formatMessageForSigning(message);
// Note that the C++ code doesn't actually seem to specify any character encoding. Presumabl
// JSON-SPIRIT hands back. Assume UTF-8 for now.
Sha256Hash messageHash = Sha256Hash.twiceOf(messageBytes);
boolean compressed = false;
// this section is added to support new signature types
if(header>= 39) // this is a bech32 signature
{
header -= 12;
compressed = true;
} // this is a segwit p2sh signature
else if(header >= 35)
{
header -= 8;
compressed = true;
} // this is a compressed key signature
else if (header >= 31) {
compressed = true;
header -= 4;
}
int recId = header - 27;
ECKey key = ECKey.recoverFromSignature(recId, sig, messageHash, compressed);
if (key == null)
throw new SignatureException("Could not recover public key from signature");
return key;
}
```

Backwards Compatibility

Since this format includes P2PKH keys, it is backwards compatible, but keep in mind some software has checks for ranges of headers and will report the newer segwit header types as errors.

Implications

Message signing is an important use case and potentially underused due to the fact that, up until now, there has not been a formal specification for how wallets can sign messages using Bitcoin private keys. Bitcoin wallets should be interoperable and use the same conventions for determing a signature's validity. This BIP can also be updated as new signature formats emerge.

Acknowledgements

- Konstantin Bay - review
- Holly Casaletto - review
- James Bryrer - review

Note that the background on ECDSA signatures was taken from en.bitcoin.it and code sample modified from BitcoinJ.

References

[1] - https://github.com/bitcoin/bips/blob/master/bip-0322.mediawiki

[2] - https://github.com/bitcoin/bitcoin/issues/10542

[3] - https://en.bitcoin.it/wiki/Elliptic_Curve_Digital_Signature_Algorithm