```
BIP: 322
Layer: Applications
Title: Generic Signed Message Format
Author: Karl-Johan Alm <karljohan-alm@garage.co.jp>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0322
Status: Draft
Type: Standards Track
Created: 2018-09-10
License: CC0-1.0
```

Abstract

A standard for interoperable signed messages based on the Bitcoin Script format, either for proving fund availability, or committing to a message as the intended recipient of funds sent to the invoice address.

Motivation

The current message signing standard only works for P2PKH (1...) invoice addresses. We propose to extend and generalize the standard by using a Bitcoin Script based approach. This ensures that any coins, no matter what script they are controlled by, can in-principle be signed for. For easy interoperability with existing signing hardware, we also define a signature message format which resembles a Bitcoin transaction (except that it contains an invalid input, so it cannot be spent on any real network).

Additionally, the current message signature format uses ECDSA signatures which do not commit to the public key, meaning that they do not actually prove knowledge of any secret keys. (Indeed, valid signatures can be tweaked by 3rd parties to become valid signatures on certain related keys.)

Ultimately no message signing protocol can actually prove control of funds, both because a signature is obsolete as soon as it is created, and because the possessor of a secret key may be willing to sign messages on others' behalf even if it would not sign actual transactions. No signmessage protocol can fix these limitations.

Types of Signatures

This BIP specifies three formats for signing messages: *legacy*, *simple* and *full*. Additionally, a variant of the *full* format can be used to demonstrate control over a set of UTXOs.

Legacy

New proofs should use the new format for all invoice address formats, including P2PKH.

The legacy format MAY be used, but must be restricted to the legacy P2PKH invoice address format.

Simple

A *simple* signature consists of a witness stack, consensus encoded as a vector of vectors of bytes, and base64-encoded. Validators should construct `to_spend` and `to_sign` as defined below, with default values for all fields except that

- `message_hash` is a BIP340-tagged hash of the message, as specified below
- `message_challenge` in `to_spend` is set to the scriptPubKey being signed with
- `message_signature` in `to_sign` is set to the provided simple signature.

and then proceed as they would for a full signature.

Full

Full signatures follow an analogous specification to the BIP-325 challenges and solutions used by Signet.

Let there be two virtual transactions `to_spend` and `to_sign`.

The `to_spend` transaction is:

```
   nVersion = 0
nLockTime = 0
vin[0].prevout.hash = 0000...000
vin[0].prevout.n = 0xFFFFFFFF
vin[0].nSequence = 0
vin[0].scriptSig = OP_0 PUSH32[ message_hash ]
vin[0].scriptWitness = []
vout[0].nValue = 0
vout[0].scriptPubKey = message_challenge
```

where `message_hash` is a BIP340-tagged hash of the message, i.e. sha256_tag(m), where tag = `BIP0322-signed-message` and `m` is the message as is without length prefix or null terminator, and `message_challenge` is the to be proven (public) key script.

The `to_sign` transaction is:

```
   nVersion = 0 or (FULL format only) as appropriate (e.g. 2, for time locks)
nLockTime = 0 or (FULL format only) as appropriate (for time locks)
vin[0].prevout.hash = to_spend.txid
vin[0].prevout.n = 0
vin[0].nSequence = 0 or (FULL format only) as appropriate (for time locks)
vin[0].scriptWitness = message_signature
vout[0].nValue = 0
vout[0].scriptPubKey = OP_RETURN
```

A full signature consists of the base64-encoding of the `to_sign` transaction in standard network serialisation once it has been signed.

Full (Proof of Funds)

A signer may construct a proof of funds, demonstrating control of a set of UTXOs, by constructing a full signature as above, with the following modifications.

- `message_challenge` is unused and shall be set to `OP_TRUE`
- Similarly, `message_signature` is then empty.
- All outputs that the signer wishes to demonstrate control of are included as additional inputs of `to_sign`, and their witness and scriptSig data should be set as though these outputs were actually being spent.

Unlike an ordinary signature, validators of a proof of funds need access to the current UTXO set, to learn that the claimed inputs exist on the blockchain, and to learn their scriptPubKeys.

Detailed Specification

For all signature types, except legacy, the `to_spend` and `to_sign` transactions must be valid transactions which pass all consensus checks, except of course that the output with prevout `000...000:FFFFFFFF` does not exist.

Verification

A validator is given as input an address A (which may be omitted in a proof-of-funds), signature s and message m, and outputs one of three states

- *valid at time T and age S* indicates that the signature has set timelocks but is otherwise valid
- *inconclusive* means the validator was unable to check the scripts
- *invalid* means that some check failed

Verification Process Validation consists of the following steps:

1. Basic validation
   (a) Compute the transaction `to_spend` from m and A
   (b) Decode s as the transaction `to_sign`
   (c) If s was a full transaction, confirm all fields are set as specified above; in particular that
      - `to_sign` has at least one input and its first input spends the output of to_spend
      - `to_sign` has exactly one output, as specified above
   (d) Confirm that the two transactions together satisfy all consensus rules, except for `to_spend`'s missing input, and except that *nSequence* of `to_sign`'s first input and *nLockTime* of `to_sign` are not checked.

2. (Optional) If the validator does not have a full script interpreter, it should check that it understands all scripts being satisfied. If not, it should stop here and output *inconclusive*.
3. Check the **required rules**:
   (a) All signatures must use the SIGHASH_ALL flag.
   (b) The use of `CODESEPARATOR` or `FindAndDelete` is forbidden.
   (c) `LOW_S`, `STRICTENC` and `NULLFAIL`: valid ECDSA signatures must be strictly DER-encoded and have a low-S value; invalid ECDSA signature must be the empty push
   (d) `MINIMALDATA`: all pushes must be minimally encoded
   (e) `CLEANSTACK`: require that only a single stack element remains after evaluation
   (f) `MINIMALIF`: the argument of `IF/NOTIF` must be exactly 0x01 or empty push
   (g) If any of the above steps failed, the validator should stop and output the *invalid* state.
4. Check the **upgradeable rules**
   (a) The version of `to_sign` must be 0 or 2.
   (b) The use of NOPs reserved for upgrades is forbidden.
   (c) The use of segwit versions greater than 1 are forbidden.
   (d) If any of the above steps failed, the validator should stop and output the *inconclusive* state.
5. Let T by the nLockTime of `to_sign` and S be the nSequence of the first input of `to_sign`. Output the state *valid at time T and age S*.

Signing

Signers who control an address A who wish to sign a message m act as follows:

1. They construct `to_spend` and `to_sign` as specified above, using the scriptPubKey of A for `message_challenge` and tagged hash of m as `message_hash`.
2. Optionally, they may set nLockTime of `to_sign` or nSequence of its first input.
3. Optionally, they may add any additional outputs to `to_sign` that they wish to prove control of.
4. They satisfy `to_sign` as they would any other transaction.

They then encode their signature, choosing either *simple* or *full* as follows:

- If they added no inputs to `to_sign`, left nSequence and nLockTime at 0, and A is a Segwit address (either pure or P2SH-wrapped), then they may base64-encode `message_signature`
- Otherwise they must base64-encode `to_sign`.

Compatibility

This specification is backwards compatible with the legacy signmessage/verifymessage specification through the special case as described above.

Reference implementation

- Bitcoin Core pull request (basic support) at: https://github.com/bitcoin/bitcoin/pull/24058

Acknowledgements

Thanks to David Harding, Jim Posen, Kalle Rosenbaum, Pieter Wuille, Andrew Poelstra, and many others for their feedback on the specification.

References

1. Original mailing list thread: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-March/015818.html

Copyright

Test vectors

Message hashing

Message hashes are BIP340-tagged hashes of a message, i.e. sha256_tag(m), where tag = `BIP0322-signed-message`, and m is the message as is without length prefix or null terminator:

- Message = "" (empty string): `c90c269c4f8fcbe6880f72a721ddfbf1914268a794cbb21cfafee13770ae19f`
- Message = "Hello World": `f0eb03b1a75ac6d9847f55c624a99169b5dccba2a31f5b23bea77ba270de0a7a`

Message signing

Given below parameters:

- private key `L3VFeEujGtevx9w18HD1fhRbCH67Az2dpCymeRE1SoPK6XQtaN2k`
- corresponding address `bc1q9vza2e8x573nczrlzms0wvx3gsqjx7vavgkx0l`

Produce signatures:

- Message = "" (empty string): `AkcwRAIgM2gBAQqvZX15ZiysmKmQpDrG83avLIT492QBzLnQIxYCIBaTpOaD20q`
- Message = "Hello World": `AkcwRAIgZRfIY3p7/DoVTty6YZbWS71bc5Vct9p9Fia83eRmw2QCICK/ENGfwLtptF`

Transaction Hashes

to_spend:

- Message = "" (empty string): `c5680aa69bb8d860bf82d4e9cd3504b55dde018de765a91bb566283c545a99a`
- Message = "Hello World": `b79d196740ad5217771c1098fc4a4b51e0535c32236c71f1ea4d61a2d603352b`

to_sign:

- Message = "" (empty string): `1e9654e951a5ba44c8604c4de6c67fd78a27e81dcadcfe1edf638ba3aaebaed`
- Message = "Hello World": `88737ae86f2077145f93cc4b153ae9a1cb8d56afa511988c149c5c8c9d93bddf`