NOTICE: This document is a work in progress and is not complete, implemented, or otherwise suitable for deployment.

```
BIP: 62
Layer: Consensus (soft fork)
Title: Dealing with malleability
Author: Pieter Wuille <pieter.wuille@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0062
Status: Withdrawn
Type: Standards Track
Created: 2014-03-12
License: BSD-2-Clause
```

Abstract

This document specifies proposed changes to the Bitcoin transaction validity rules in order to make malleability of transactions impossible (at least when the sender doesn't choose to avoid it).

Copyright

Motivation

As of february 2014, Bitcoin transactions are malleable in multiple ways. This means a (valid) transaction can be modified in-flight, without invalidating it, but without access to the relevant private keys.

This is a problem for multiple reasons:

- The sender may not recognize his own transaction after being modified.
- The sender may create transactions that spend change created by the original transaction. In case the modified transaction gets mined, this becomes invalid.
- Modified transactions are effectively double-spends which can be created without malicious intent (of the sender), but can be used to make other attacks easier.

Several sources of malleability are known:

1. **Non-DER encoded ECDSA signatures** Right now, the Bitcoin reference client uses OpenSSL to validate signatures. As OpenSSL accepts more than serializations that strictly adhere to the DER standard, this is a source of malleability. Since v0.8.0, non-DER signatures are no longer relayed already.

2. **Non-push operations in scriptSig** Any sequence of script operations in scriptSig that results in the intended data pushes, but is not just a push of that data, results in an alternative transaction with the same validity.
3. **Push operations in scriptSig of non-standard size type** The Bitcoin scripting language has several push operators (OP_0, single-byte pushes, data pushes of up to 75 bytes, OP_PUSHDATA1, OP_PUSHDATA2, OP_PUSHDATA4). As the later ones have the same result as the former ones, they result in additional possibilities.
4. **Zero-padded number pushes** In cases where scriptPubKey opcodes use inputs that are interpreted as numbers, they can be zero padded.
5. **Inherent ECDSA signature malleability** ECDSA signatures themselves are already malleable: taking the negative of the number S inside (modulo the curve order) does not invalidate it.
6. **Superfluous scriptSig operations** Adding extra data pushes at the start of scripts, which are not consumed by the corresponding scriptPubKey, is also a source of malleability.
7. **Inputs ignored by scripts** If a scriptPubKey starts with an OP_DROP, for example, the last data push of the corresponding scriptSig will always be ignored.
8. **Sighash flags based masking** Sighash flags can be used to ignore certain parts of a script when signing.
9. **New signatures by the sender** The sender (or anyone with access to the relevant private keys) is always able to create new signatures that spend the same inputs to the same outputs.

The first six and part of the seventh can be fixed by extra consensus rules, but the last two can't. Not being able to fix #7 means that even with these new consensus rules, it will always be possible to create outputs whose spending transactions will all be malleable. However, when restricted to using a safe set of output scripts, extra consensus rules can make spending transactions optionally non-malleable (if the spender chooses to; as he can always bypass #8 and #9 himself).

Specification

New rules

Seven extra rules are introduced, to combat exactly the seven first sources of malleability listed above:

1. **Canonically encoded ECDSA signatures** An ECDSA signature passed to OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG or OP_CHECKMULTISIGVERIFY must be encoded using strict DER encoding. To provide a compact way to deliberately create an invalid signature for OP_CHECKSIG and OP_CHECKMULTISIG, an empty byte array (i.e., the result of OP_0) is also allowed. Doing a verification with a non-DER signature makes the entire script evaluate to False (not

just the signature verification). See reference: DER encoding.

2. **Non-push operations in scriptSig** Only data pushes are allowed in scriptSig. Evaluating any other operation makes the script evaluate to false. See reference: Push operators.
3. **Push operations in scriptSig of non-standard size type** The smallest possible push operation must be used when possible. Pushing data using an operation that could be encoded in a shorter way makes the script evaluate to false. See reference: Push operators.
4. **Zero-padded number pushes** Any time a script opcode consumes a stack value that is interpreted as a number, it must be encoded in its shortest possible form. 'Negative zero' is not allowed. See reference: Numbers.
5. **Inherent ECDSA signature malleability** We require that the S value inside ECDSA signatures is at most the curve order divided by 2 (essentially restricting this value to its lower half range). See reference: Low S values in signatures.
6. **Superfluous scriptSig operations** scriptPubKey evaluation will be required to result in a single non-zero value. If any extra data elements remain on the stack, the script evaluates to false.
7. **Inputs ignored by scripts** The (unnecessary) extra stack element consumed by OP_CHECKMULTISIG and OP_CHECKMULTISIGVERIFY must be the empty byte array (the result of OP_0). Anything else makes the script evaluate to false.

Block validity

To introduce these new rules in the network, we add both v3 blocks and v3 transactions. v2 is skipped for transactions to keep the version numbers between transaction and block rules in sync. v2 transactions are treated identically to v1 transactions. The same mechanism as in BIP 0034 is used to introduce v3 blocks. When 75% of the past 1000 blocks are v3, a new consensus rule is activated:

- All transactions in v3 blocks are required to follow rules #1-#2.
- v3 (and higher) transactions in v3 blocks are required to follow rules #3-#7 as well.

When 95% of the past 1000 blocks are v3 or higher, v2 blocks become invalid entirely. Note however that v1 (and v2) transactions remain valid forever.

References

Below is a summary of the effects on signatures, their encoding and data pushes.

Low S values in signatures The value S in signatures must be between 0x1 and 0x7FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 5D576E73 57A4501D DFE92F46 681B20A0 (inclusive). If S is too high, simply replace it by S' = 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141 - S.

Signatures produced by the OpenSSL library are not guaranteed to be consistent with this constraint. Version 0.9.3 of the reference client provides an example for detection and correction.

The constraints on the value R are unchanged w.r.t. ECDSA, and values can be between 0x1 and 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364140 (inclusive).

DER encoding For reference, here is how to encode signatures correctly in DER format.

0x30 [total-length] 0x02 [R-length] [R] 0x02 [S-length] [S] [sighash-type]

- total-length: 1-byte length descriptor of everything that follows, excluding the sighash byte.
- R-length: 1-byte length descriptor of the R value that follows.
- R: arbitrary-length big-endian encoded R value. It cannot start with any 0x00 bytes, unless the first byte that follows is 0x80 or higher, in which case a single 0x00 is required.
- S-length: 1-byte length descriptor of the S value that follows.
- S: arbitrary-length big-endian encoded S value. The same rules apply as for R.
- sighash-type: 1-byte hashtype flag (only 0x01, 0x02, 0x03, 0x81, 0x82 and 0x83 are allowed).

This is already enforced by the reference client as of version 0.8.0 (only as relay policy, not as a consensus rule).

This rule, combined with the low S requirement above, results in S-length being at most 32 (and R-length at most 33), and the total signature size being at most 72 bytes (and on average 71.494 bytes).

Push operators

- Pushing an empty byte sequence must use OP_0.
- Pushing a 1-byte sequence of byte 0x01 through 0x10 must use OP_n.
- Pushing the byte 0x81 must use OP_1NEGATE.
- Pushing any other byte sequence up to 75 bytes must use the normal data push (opcode byte n, with n the number of bytes, followed n bytes of data being pushed).
- Pushing 76 to 255 bytes must use OP_PUSHDATA1.
- Pushing 256 to 520 bytes must use OP_PUSHDATA2.
- OP_PUSHDATA4 can never be used, as pushes over 520 bytes are not allowed, and those below can be done using other operators.
- Any other operation is not considered to be a push.

Numbers The native data type of stack elements is byte arrays, but some operations interpret arguments as integers. The used encoding is little endian

with an explicit sign bit (the highest bit of the last byte). The shortest encodings for numbers are (with the range boundaries encodings given in hex between ()).

- 0: OP_0; (00)
- 1..16: OP_1..OP_16; (51)..(60)
- -1: OP_1NEGATE; (79)
- -127..-2 and 17..127: normal 1-byte data push; (01 FF)..(01 82) and (01 11)..(01 7F)
- -32767..-128 and 128..32767: normal 2-byte data push; (02 FF FF)..(02 80 80) and (02 80 00)..(02 FF 7F)
- -8388607..-32768 and 32768..8388607: normal 3-byte data push; (03 FF FF FF)..(03 00 80 80) and (03 00 80 00)..(03 FF FF 7F)
- -2147483647..-8388608 and 8388608..2147483647: normal 4-byte data push; (04 FF FF FF FF)..(04 00 00 80 80) and (04 00 00 80 00)..(04 FF FF FF 7F)
- Any other numbers cannot be encoded.

In particular, note that zero could be encoded as (01 80) (negative zero) if using the non-shortest form is allowed.

Compatibility

Relay of transactions A new node software version is released which makes v3 transactions standard, and relays them when their scriptSigs satisfy the above rules. Relaying of v1 transactions is unaffected. A v1 transaction spending an output created by a v3 transaction is also unaffected.

Wallet updates As v3 transactions are non-standard currently, it is not possible to start creating them immediately. Software can be checked to confirm to the new rules of course, but using v3 should only start when a significant part of the network's nodes has upgraded to compatible code. Its intended meaning is "I want this transaction protected from malleability", and remains a choice of the wallet software.