```
BIP: 66
Layer: Consensus (soft fork)
Title: Strict DER signatures
Author: Pieter Wuille <pieter.wuille@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0066
Status: Final
Type: Standards Track
Created: 2015-01-10
License: BSD-2-Clause
```

Abstract

This document specifies proposed changes to the Bitcoin transaction validity rules to restrict signatures to strict DER encoding.

Copyright

Motivation

Bitcoin's reference implementation currently relies on OpenSSL for signature validation, which means it is implicitly defining Bitcoin's block validity rules. Unfortunately, OpenSSL is not designed for consensus-critical behaviour (it does not guarantee bug-for-bug compatibility between versions), and thus changes to it can - and have - affected Bitcoin software.

One specifically critical area is the encoding of signatures. Until recently, OpenSSL's releases would accept various deviations from the DER standard and accept signatures as valid. When this changed in OpenSSL 1.0.0p and 1.0.1k, it made some nodes reject the chain.

This document proposes to restrict valid signatures to exactly what is mandated by DER, to make the consensus rules not depend on OpenSSL's signature parsing. A change like this is required if implementations would want to remove all of OpenSSL from the consensus code.

Specification

Every signature passed to OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG, or OP_CHECKMULTISIGVERIFY, to which ECDSA verification is applied, must be encoded using strict DER encoding (see further).

These operators all perform ECDSA verifications on pubkey/signature pairs, iterating from the top of the stack backwards. For each such verification, if the signature does not pass the `IsValidSignatureEncoding` check below, the entire

script evaluates to false immediately. If the signature is valid DER, but does not pass ECDSA verification, opcode execution continues as it used to, causing opcode execution to stop and push false on the stack (but not immediately fail the script) in some cases, which potentially skips further signatures (and thus does not subject them to `IsValidSignatureEncoding`).

DER encoding reference

The following code specifies the behaviour of strict DER checking. Note that this function tests a signature byte vector which includes the 1-byte sighash flag that Bitcoin adds, even though that flag falls outside of the DER specification, and is unaffected by this proposal. The function is also not called for cases where the length of sig is 0, in order to provide a simple, short and efficiently-verifiable encoding for deliberately invalid signatures.

DER is specified in https://www.itu.int/rec/T-REC-X.690/en .

```
bool static IsValidSignatureEncoding(const std::vector<unsigned char> &sig) {
    // Format: 0x30 [total-length] 0x02 [R-length] [R] 0x02 [S-length] [S] [sighash]
    // * 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 must use the shortest
    //   possible encoding for a positive integers (which means no null bytes at
    //   the start, except a single one when the next byte has its highest bit set).
    // * S-length: 1-byte length descriptor of the S value that follows.
    // * S: arbitrary-length big-endian encoded S value. The same rules apply.
    // * sighash: 1-byte value indicating what data is hashed (not part of the DER
    //   signature)

    // Minimum and maximum size constraints.
    if (sig.size() < 9) return false;
    if (sig.size() > 73) return false;

    // A signature is of type 0x30 (compound).
    if (sig[0] != 0x30) return false;

    // Make sure the length covers the entire signature.
    if (sig[1] != sig.size() - 3) return false;

    // Extract the length of the R element.
    unsigned int lenR = sig[3];

    // Make sure the length of the S element is still inside the signature.
    if (5 + lenR >= sig.size()) return false;

    // Extract the length of the S element.
```

```
    unsigned int lenS = sig[5 + lenR];

    // Verify that the length of the signature matches the sum of the length
    // of the elements.
    if ((size_t)(lenR + lenS + 7) != sig.size()) return false;

    // Check whether the R element is an integer.
    if (sig[2] != 0x02) return false;

    // Zero-length integers are not allowed for R.
    if (lenR == 0) return false;

    // Negative numbers are not allowed for R.
    if (sig[4] & 0x80) return false;

    // Null bytes at the start of R are not allowed, unless R would
    // otherwise be interpreted as a negative number.
    if (lenR > 1 && (sig[4] == 0x00) && !(sig[5] & 0x80)) return false;

    // Check whether the S element is an integer.
    if (sig[lenR + 4] != 0x02) return false;

    // Zero-length integers are not allowed for S.
    if (lenS == 0) return false;

    // Negative numbers are not allowed for S.
    if (sig[lenR + 6] & 0x80) return false;

    // Null bytes at the start of S are not allowed, unless S would otherwise be
    // interpreted as a negative number.
    if (lenS > 1 && (sig[lenR + 6] == 0x00) && !(sig[lenR + 7] & 0x80)) return false;

    return true;
}
```

Examples

Notation: P1 and P2 are valid, serialized, public keys. S1 and S2 are valid signatures using respective keys P1 and P2. S1' and S2' are non-DER but otherwise valid signatures using those same keys. F is any invalid but DER-compliant signature (including 0, the empty string). F' is any invalid and non-DER-compliant signature.

1. `S1' P1 CHECKSIG` fails (changed)
2. `S1' P1 CHECKSIG NOT` fails (unchanged)
3. `F P1 CHECKSIG` fails (unchanged)

4. `F P1 CHECKSIG NOT` can succeed (unchanged)
5. `F' P1 CHECKSIG` fails (unchanged)
6. `F' P1 CHECKSIG NOT` fails (changed)

1. `0 S1' S2 2 P1 P2 2 CHECKMULTISIG` fails (changed)
2. `0 S1' S2 2 P1 P2 2 CHECKMULTISIG NOT` fails (unchanged)
3. `0 F S2' 2 P1 P2 2 CHECKMULTISIG` fails (unchanged)
4. `0 F S2' 2 P1 P2 2 CHECKMULTISIG NOT` fails (changed)
5. `0 S1' F 2 P1 P2 2 CHECKMULTISIG` fails (unchanged)
6. `0 S1' F 2 P1 P2 2 CHECKMULTISIG NOT` can succeed (unchanged)

Note that the examples above show that only additional failures are required by this change, as required for a soft forking change.

Deployment

We reuse the double-threshold switchover mechanism from BIP 34, with the same thresholds, but for nVersion = 3. The new rules are in effect for every block (at height H) with nVersion = 3 and at least 750 out of 1000 blocks preceding it (with heights H-1000..H-1) also have nVersion = 3. Furthermore, when 950 out of the 1000 blocks preceding a block do have nVersion = 3, nVersion = 2 blocks become invalid, and all further blocks enforce the new rules.

Compatibility

The requirement to have signatures that comply strictly with DER has been enforced as a relay policy by the reference client since v0.8.0, and very few transactions violating it are being added to the chain as of January 2015. In addition, every non-compliant signature can trivially be converted into a compliant one, so there is no loss of functionality by this requirement. This proposal has the added benefit of reducing transaction malleability (see BIP 62).

Implementation

An implementation for the reference client is available at https://github.com/bitcoin/bitcoin/pull/5713

Acknowledgements

This document is extracted from the previous BIP62 proposal, which had input from various people, in particular Greg Maxwell and Peter Todd, who gave feedback about this document as well.

Disclosures

- Subsequent to the network-wide adoption and enforcement of this BIP, the author disclosed that strict DER signatures provided an indirect solution to a consensus bug he had previously discovered.