Weaknesses in Bitcoin's Merkle Root Construction

February 25, 2019

Abstract

Bitcoin block headers include a commitment to the set of transactions in a given block, which is implemented by constructing a Merkle tree of transaction id's (double-SHA256 hash of a transaction) and including the root of the tree in the block header. This in turn allows for proving to a Bitcoin light client that a given transaction is in a given block by providing a path through the tree to the transaction. However, Bitcoin's particular construction of the Merkle tree has several security weaknesses, including at least two forms of block malleability that have an impact on the consensus logic of Bitcoin Core, and an attack on light clients, where an invalid transaction could be "proven" to appear in a block by doing substantially less work than a SHA256 hash collision would require.

Credits

Several people have reported some of the issues described in this summary, including Sergio Lerner and Peter Todd on the bitcoin-dev mailing list. The summary presented here is based on private IRC conversations between Johnson Lau and Greg Maxwell. I believe some of the observations discussed in section 3 were originally made by Luke Dashjr.

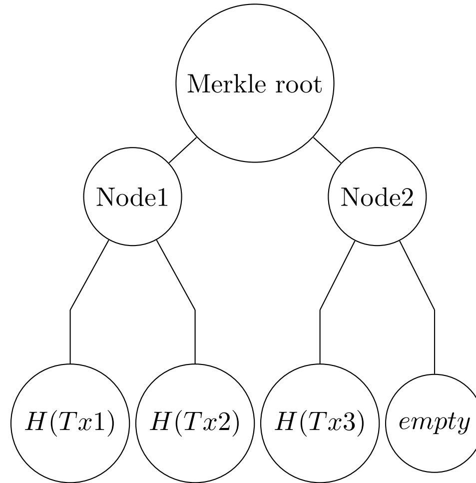
1 Background

1.1 Merkle Root Construction

As a reminder, the Merkle root construction used by Bitcoin involves starting with the double SHA256 hash H of each transaction[1] as the leaf nodes in a tree. We construct a parent node for each pair of consecutive leaves by concatenating the leaves and double-SHA256 hashing the result. Importantly, if there are an odd number of leaves at a level in the tree, the final element is duplicated, so

[1] In this document we will only be discussing the Merkle root value in the block header, for which transactions are serialized for hashing without their witnesses.

that the parent node will be the hash of the child node concatenated with itself. For example, if a block had three transactions, the construction is as follows:



Here, Node1 is calculated as $H(H(Tx1)||H(Tx2))$, and Node2 is calculated as $H(H(Tx3)||H(Tx3))$. Finally, the Merkle root is calculated to be $H(Node1||Node2)$, and that is what appears in the block header.

If a block consists of only a single transaction, then the Merkle root will be the hash of the single transaction. Note that every valid block must have at least one transaction.

1.2 Block validity in Bitcoin Core

Bitcoin Core's consensus logic breaks up block validation into several parts and tracks the validity of a given block through those stages. For example, when a new block header is processed (typically before the block itself is even received), the header is checked for validity under the network's consensus rules. When the block later arrives, the block is also processed in stages, as context-free checks (which don't depend on any other blocks or headers) are checked first, followed by the checks that are dependent on the headers chain that the block is part of, and finally finishing with checks on the transactions and their signatures, if the block appears to be part of the most-work headers chain.

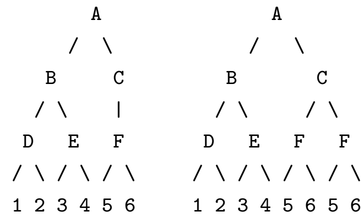
Associated with each stored block header is a cached value that tracks how far through validation the associated block has progressed. If a block is discovered to be invalid, Bitcoin Core stores the invalid status of that block permanently, and will avoid reprocessing that block or any descendants of that block.

In the following sections we'll discuss the concerns around permanently marking blocks as invalid in order to prevent an attacker from using this optimization to split the network.

2 Duplicate transactions, CVE-2012-2459

This is documented in the Bitcoin Core source code:

```
/*     WARNING! If you're reading this because you're learning about crypto
       and/or designing a new system that will use merkle trees, keep in mind
       that the following merkle tree algorithm has a serious flaw related to
       duplicate txids, resulting in a vulnerability (CVE-2012-2459).
       The reason is that if the number of hashes in the list at a given time
       is odd, the last one is duplicated before computing the next level (which
       is unusual in Merkle trees). This results in certain sequences of
       transactions leading to the same merkle root. For example, these two
       trees:
```




```
       for transaction lists [1,2,3,4,5,6] and [1,2,3,4,5,6,5,6] (where 5 and
       6 are repeated) result in the same root hash A (because the hash of both
       of (F) and (F,F) is C).
       The vulnerability results from being able to send a block with such a
       transaction list, with the same merkle root, and the same block hash as
       the original without duplication, resulting in failed validation. If the
       receiving node proceeds to mark that block as permanently invalid
       however, it will fail to accept further unmodified (and thus potentially
       valid) versions of the same block. We defend against this by detecting
       the case where we would hash two identical hashes at the end of the list
       together, and treating that identically to the block having an invalid
       merkle root. Assuming no double-SHA256 collisions, this will detect all
       known ways of changing the transactions without affecting the merkle
       root.
*/
```

In the next section we'll discuss a new form of block malleability that was unknown at the time the above comment was written.

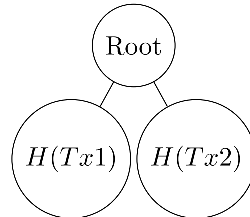
3 Weaknesses resulting from Merkle tree ambiguities

Greg Maxwell has pointed out (in private IRC communications) that a source of weakness arises from a "lack of domain separation between nodes and leaves" in the Merkle tree. A non-leaf node in a Merkle tree is the hash of a 64-byte input, which is the concatenation of the node's children. Meanwhile, a leaf-node

is the hash of a transaction (serialized without its witness). If a transaction's serialization (without witness) can be 64 bytes, the inability to distinguish a leaf-node from a non-leaf node can expose security concerns.

3.1 Block malleability

Consider a block B with 2 transactions. The Merkle root is calculated as follows:



And consider a block with 1 transaction. The Merkle root for that would just be $H(Tx1)$.

Suppose a peer relays the block header for B but claims that B has only one transaction, not two, and that the one transaction T in the block has the serialization $H(Tx1)||H(Tx2)$. If T successfully deserializes into a transaction[2] (and canonically reserializes to $H(Tx1)||H(Tx2)$), then the hash of T will match the Merkle root in the block header. If T is invalid, then the node that receives this version of block B will reject it as invalid. However, if the receiving node permanently marks this block hash as invalid, then it will never process or accept as valid block B, even if transmitted with two transactions from an honest peer. This could be used to cause a node to drop out of consensus.

This generalizes to a block with N transactions by an attacker claiming that the block actually has $N/2^k$ transactions, consisting of a set of 64-byte "transactions" that come from some other row k in the Merkle tree. If those 64-byte values all successfully deserialize as transactions (and hence can be communicated), then a peer receiving the block must not permanently mark the block hash as invalid, even if those 64-byte transactions are invalid.

3.1.1 How much work is required to produce a 2-transaction block such that the Merkle root is the hash of a 64-byte input that deserializes as a transaction?

Transactions are serialized as follows:

$$[version][vin][vout][locktime]$$

The version and locktime are both 4-byte fields, which have no deserialization requirements. The *vin* and *vout* serializations look like this:

$$[|vin|][vin_0]...[vin_n]$$

[2]Note that any block – malleated or not – can only be considered by a node if it can be communicated, which means that the message must be able to be deserialized according to the message processing rules of the network. Messages that cannot be understood are discarded.

and

$$[|vout|][vout_0]...[vout_n]$$

Where the size of vin is encoded using Bitcoin's "compact size". Since each vin contains a 32-byte prevout hash, and we want to create 64 bytes that will deserialize successfully as a transaction, we constrain $|vin|$ to 1, which encodes as the one byte, 0x01. Each vin_i consists of:

$$[hash][index][scriptSig][sequence]$$

where $hash$ is a 32-byte prevout hash, and $index$ is a 4-byte index into the $vout$ vector of the transaction referenced by the prevout hash. Valid coinbases must have a null prevout hash and index set to `0xffffffff`, but invalid coinbase transactions are unconstrained in these values.

The *scriptsig* is encoded with a length followed by the script, so there's a constraint that the length matches the number of bytes read. For now, we assume an empty *scriptSig*, which has a 1-byte encoding of 0x00.

The *sequence* is 4 bytes and not constrained.

The size of the *vout* array is again encoded with a length, which will need to be respected for the serialization to be valid, so we constrain the *vout* size to be 1, which introduces one more fixed byte 0x01. The serialization of the rest of the vout vector is as follows:

$$[amount][scriptPubKey]$$

where *amount* is an 8-byte quantity, and *scriptPubKey*, like *scriptSig*, has a length encoding, which imposes a 1-byte constraint.

Summing this all up, a 64-byte transaction that successfully deserializes could be constructed like this:

```
    ?? ?? ?? ?? 01 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
    A           B  C
    ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

    ?? ?? ?? ?? ?? ?? ?? ?? ?? 00 ?? ?? ?? ?? 01 ??
                   D           E  F           G  H
    ?? ?? ?? ?? ?? ?? ?? 04 ?? ?? ?? ?? ?? ?? ?? ??
                         I  J           K

A = version (4 bytes, unconstrained)
B = #vin (1 byte, constrained)
C = prevout txid (32 bytes, unconstrained)
D = prevout index (4 bytes, unconstrained)
E = scriptSig (1 byte, constrained)
F = sequence (4 bytes, unconstrained)
G = #vout (1 byte, constrained)
H = amount (8 bytes, unconstrained)
```

```
I = length(scriptPubKey) (1 byte, constrained)
J = remainder of scriptPubKey (4 bytes, unconstrained)
K = locktime (4 bytes, unconstrained)
```

Here, we've set the *scriptPubKey* in $vout_0$ to be 5 total bytes, so that the transaction is 64-bytes long, and only one of those bytes is constrained for successful deserialization (the length encoding). But looking at this in total, we can see that we actually just require the sum of the length of the *scriptPubKey* and the length of the *scriptSig* is 4.

Note also that only four of the 64 bytes here are constrained, and they appear in different halves of the transaction. So to produce a block that has a Merkle root which is a hash of a 64-byte quantity that deserializes validly, it's enough to just do 8 bits of work to find a workable coinbase (which will hash to the first 32 bytes), plus another $\approx$ 22 bits of work ($(1/5) * 2^{24}$, so slightly less) to find a workable second transaction which will hash to the second 32 bytes) – a very small amount of computation.

3.1.2 How much work is required to produce a block such that some row of the Merkle tree consists of 64-byte quantities that deserialize as VALID transactions?

Note that the first transaction in a block must be a coinbase, and as discussed above, that largely constrains the first 32 bytes of the first transaction: only the 4 version bytes are unconstrained. So it would take at least 28*8= 224 bits of work to find the first node in a given row of the tree that would match the first half of a coinbase, in addition to the amount of work required to grind the second half of the transaction to something meaningful (which is much easier – only 16 bytes or so are constrained, so approximately 128 bits of work to find a collision). Of course, any of the rows in the Merkle tree could be used, but it nevertheless seems clear that this should be computationally infeasible.

3.2 Attacks on SPV (light) clients

SPV clients expect to be able to receive proofs that a given transaction appears in a given block. The nature of the proof is to provide the client with a path through the Merkle tree, from the root down to the transaction.

Going down the Merkle tree However, suppose a (valid) 64-byte transaction T is included in a block with the property that the second 32 bytes (which are less constrained than the first 32 bytes) are constructed so that they collide with the hash of some other fake, invalid transaction F. Let R be the Merkle root, so that we have:

$$R = H(H(H(...H(T)||H(...)...)$$

$$T = [32 bytes]||[H(F)]$$

If such a construction is possible, then a node could fool an SPV client into thinking that F is in the block by providing a path down to T, and then treating T as an interior node in the tree. Because the number of transactions in a block is not in the block header, SPV clients do not a priori know how many transactions are in the block, or, therefore, the correct depth of the tree.

3.2.1 How much work is required to produce a valid 64-byte transaction such that its lower 32 bytes collide with the hash of some other transaction?

Note Sergio Lerner [1] has published a separate analysis that this can be done in 72 bits of work; Peter Todd [2] has also published an analysis that claims this can be done with 60 bits of work.

From the diagram earlier, we know the first 32 bytes of the transaction are largely constrained, consisting mostly of the prevout being spent, which must be valid. Instead, we focus on the second 32 bytes:

```
    ?? ?? ?? ?? ?? .. .. .. .. 00 ?? ?? ?? ?? 01 ..
    C              D           E  F           G  H
    .. .. .? ?? ?? ?? ?? 04 ?? ?? ?? ?? .? ?? ?? ??
                         I  J           K
C = prevout txid (remaining 5 bytes of the txid, unconstrained)
D = prevout index (4 bytes, 21 bits constrained)
E = scriptSig (1 byte, constrained)
F = sequence (4 bytes, unconstrained if tx version=1)
G = #vout (1 byte, constrained)
H = amount (8 bytes, ~33 bits are constrained, see below)
I = length(scriptPubKey) (1 byte, constrained)
J = remainder of scriptPubKey (4 bytes, unconstrained)
K = locktime (4 bytes, ~29 bits are unconstrained, see below)
```

We can assume the prevout txid bits are unconstrained, because once we have a candidate transaction, we can just separately do 40-bits of work to find an input with the appropriate last 5 bytes of txid. Similarly, we can partially constrain the prevout index, by only requiring it be no more than, say, 2048, as we can assume the ability to create a 2048-output transaction with the appropriate amount in the appropriate output index.

We can assume *sequence* is unconstrained, because that field has no consensus meaning for version 1 transactions. Since we're assuming the high 32 bytes of the transaction are fixed, we can assume that the version is set to 1.

The amount is an 8 byte field. If an attacker has access to a lot of funds, then that permits more values of the amount to be unconstrained, as the consensus constraint is that the output value is less than or equal to the input value. However, as the funds spent in this transaction will be anyone-can-spend (or nobody-can-spend), the funds used in this attack would likely be lost (if the

attacker were also a miner, then they could attempt to recover the funds by respending in the same block, though there is a risk another miner could orphan the block to steal the fees and anyone-can-spend outputs). Moreover, if we assume the funds would be lost, then we have to weigh the cost of this attack with other attacks that could be performed, such as just mining a few invalid blocks and using those to attack a light client (note however that the nature of the attack is different, as this attack would allow fooling a light client about a fake transaction that would appear to be confirmed arbitrarily deep in the chain – this could be much more valuable than mining a few fake blocks that would lack arbitrarily many confirmations for a targeted transaction).

For argument's sake, we'll ballpark the cost that an attacker would be willing to bear in carrying out this attack at 25 BTC, which is currently two block rewards. In this case, the number of constrained bits is 33. (If we were to assume an attacker would risk 687 BTC instead, that would reduce the work required by 5 bits.)

The locktime is a 4 byte integer, interpreted as a time if over 500,000,000, or as a block height if under that value. Current time is roughly 1.4B seconds since the epoch, so we estimate 900M valid values of locktime, which is roughly 29 bits of freedom.

Putting this all together, we have 81 bits of constraint, meaning that an attacker who can do 81 bits of work (followed by another 40 bits of work, to construct the funding transaction whose coins will be spent by this one) is able to fool an SPV client in this way. (Note that this is much less than the 128-bits of work that we would expect to be required to fool an SPV client, by finding 2 transactions with the same hash.)

4 Vulnerabilities and Mitigations

4.1 Block Malleability

The consensus-splitting potential of block malleability arises from the logic around treatment of invalid blocks. As discussed in the Bitcoin Core code comment relating to duplicate transactions, it has been recognized that the consensus logic must not permanently mark any block header as invalid if the block being processed could have been malleated: if some other block that corresponds to the same header might be valid, then marking the header as invalid could result in a consensus split.

Duplicate Transactions The issue with duplicate transactions has been fixed since Bitcoin-Qt version 0.6.1. Then (and now) there were a set of checks (performed in a function called `CheckBlock()`) done on incoming blocks whereby the block would not be stored – and its failure not permanently cached – if any of those checks failed. The new test for duplicate transactions was added to that set, thereby resolving the issue.

Going up the Merkle tree Another check that was also being done in `CheckBlock()` relates to the coinbase transaction: if the first transaction in a block fails the required structure of a coinbase – one input, with previous output hash of all zeros and index of all ones – then the block will fail validation. The side effect of this test being in `CheckBlock()` was that even though the block malleability discussed in section 3.1 was unknown, we were effectively protected against it – as described above, it would take at least 224 bits of work to produce a malleated block that passed the coinbase check.

By Bitcoin Core version 0.13.0, the implementation around tracking block malleability had changed, so that potential Merkle tree malleability was detected in `CheckBlock()`, and a flag tracked whether the block may have been malleated or not. That flag was used by the logic that determined whether a block should be permanently marked as invalid. But `CheckBlock()` was still being invoked twice: once with an early-return feature on failure as discussed, and once again (on success) prior to block storage. Thus even if a block failed in `CheckBlock()` for a reason that was believed to be certainly invalid (and not potentially due to malleation), the logic would not permanently mark the block as invalid.

Thus it appeared redundant to be invoking this function twice. In Bitcoin Core commit dbb89dc, this seemingly redundant call was eliminated, so that when `CheckBlock()` failed due to a reason not specifically known to be potential malleation, the failure was marked as permanent. As a result, Bitcoin Core releases 0.13.0, 0.13.1, and 0.13.2 are all vulnerable to the attack described in 3.1.

This change was reverted in Bitcoin Core commit ba803ef, after the issue became privately known, prior to the 0.14.0 release. The 0.13 releases are the only releases known to be vulnerable to this concern.

Further mitigations are likely to be proposed in the future, such as by never permanently marking a block as invalid if it consists entirely of 64-byte transactions[3], or a consensus change to disallow 64-byte transactions altogether. Proposal of more direct fixes such as these have been delayed pending mitigation and disclosure of the issue facing light clients (described in 3.2).

4.2 Light clients

A number of mitigations are possible to protect light clients from the attack described in 3.2. Starting in Bitcoin Core 0.16.1, 64-byte transactions will no longer be accepted to the mempool. If that policy is adopted by miners, then it may be reasonable to propose a consensus change that would make 64-byte transactions invalid, which if adopted would permanently eliminate the ambiguity between leaves and nodes in the Merkle tree going forward, and also protect

[3]Note that the 64 bytes refers to the transaction length when serialized without witnesses, throughout this document.

existing light clients with no changes required on their part.[4]

Note that a small number of 64-byte transactions have appeared in the blockchain already.

Other mitigations are also available to light clients, such as requiring that proofs always include a path to the coinbase along with the coinbase transaction (for determining Merkle-tree depth, which must be the same for all transactions in a block). This would require a software change for light clients – in particular, to verify that the coinbase transaction has the prescribed form – but would not require any consensus changes.

Sergio Lerner discusses additional mitigations in his report [1] as well.

References

[1] Lerner, Sergio. "Leaf-Node Vulnerability in Bitcoin Merkle Tree Design". August 4, 2017. Announced on the bitcoin-dev mailing list on June 8, 2018 (https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-June/016104.html).

[2] Todd, Peter. Email to the bitcoin-dev mailing list on June 7, 2018 (https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-June/016091.html).

[4]We omitted discussion of attacking an SPV client by going up the Merkle tree and claiming that an interior node is a transaction of interest. Under reasonable assumptions, a light client could be tricked into thinking one of its outputs was spent with 116 bits of work. If a consensus change were made to make 64 byte transactions invalid, then light clients could be completely protected from this if they were modified to reject 64 byte transactions as well.