Report: Security Audit of BTC Relay implementation

Andrew Miller, Feb 7 2015

This is the final report covering a short and preliminary audit (budget: 10 hours) of the BTC Relay design and implementation.

BTC Relay is an Ethereum contract that acts as an SPV client of the Bitcoin blockchain - that is, it provides a service that other Ethereum contracts can use to verify if a given Bitcoin transaction is valid. The primary goal of this audit is to look for major security flaws in design or implementation that could lead to 1) being unable to verify a valid Tx, and 2) falsely reporting an invalid Tx as valid.

To provide this service, BTC Relay keeps track of the Bitcoin headers and proof-of-work chain in the contract's storage. To avoid having to do a lot of special work to handle reorgs, the headers are stored in a skip-list data structure (overlaid on top of Ethereum contract storage) that supports quickly finding ancestors of a given chain.

Since SPV security relies on at least one honest user submitting new Bitcoin block headers in a timely fashion, BTC Relay features an incentive mechanism where the submitter can choose a price, and consumers that call verifyTx must pay it.

Summary:

Overall, I've found the BTC Relay design and implementation to be of excellent quality. The included tests included both units and end-to-end examples that exercise almost all of the Bitcoin blockchain headers and a variety of transactions. The included user interface examples are great for illustrating the intended use of this as a service.

The implementation suffered from a small number of easily-corrected pitfalls, most notably 1) not accounting for an ambiguity in Bitcoin's transaction Merkle tree data format, and 2) forgetting to check the return value of a send.

The ancestor skip-list data structure is a great approach, although I provide a recommendation that could reduce its overall storage and update cost.

My largest complaint is about the incentive mechanism, which I believe is underspecified and

“confusing-by-design”. In a nutshell, this mechanism attempts to charge consumers of the relay service a “fee,” although there are workarounds such that users typically do not need to pay the fee. Evaluating this mechanism is outside the explicit scope of this audit, but I’ll explain why I find this confusing and undesirable. The incentive mechanism also suffers from an easily-corrected mistake, as explained below - arguably, as this is a specification error, even this would have been out of scope! Therefore my recommendation is to rethink and simplify the incentive mechanism.

Methodology

Officially, the audit covers the code at this commit:
https://github.com/ethereum/btcrelay/commit/0c013319ccb3b274cf0e878e21753f5b68933628
although I tried to keep up with the latest versions as updates were made.

The following describes my intended procedure:

- Look for typical errors and pitfallls
    Forgetting to check the return value in send
    Mistakes in data format conversions
    Mistakes in magic offsets (~calldataload) which may depend on data size
    Control flow paths that lose money or apply state changes only partially
    Misinterpretation of serpent/EVM semantics
    Merkle-tree ambiguity

- Check for consistency with Bitcoin spec
    Is it possible to accept invalid transactions or headers?
    Are assumptions about Bitcoin formats that are likely to change?

- Evaluate code quality
    Are style guidelines and conventions applied consistently to mkae code review easy?
    Do the tests cover branches of the code?

Not covered:
    - Analysis of evm bytecode and bugs / under-specification of serpent compiler. The serpent compiler does not have a full specification

Main findings:

The return value of send() is not checked, leading to inconsistent state:
    There is an instance of an unguarded send() when paying out the fee to the recipient. Since a log event is generated after the send, this could result in consumers taking an action

based on the log event, even though the fee may have actually not been sent. The likelihood of harm from this is low, assuming that consumers are unlikely to rely on log events.

This bug was addressed quickly, by throwing an exception (invalid()) after checking the return value of send().
https://github.com/ethereum/btcrelay/commit/26ee2bcc4468329939a3f093023496986c357074

Merkle tree handling is susceptible to ambiguity:

To check if a Tx is contained within a Bitcoin block, the Merkle tree siblings "along the path" must be provided by the caller, and the BTC Relay contract checks it by recomputing the root hash.

In Bitcoin Merkle trees, there's no explicit indication whether a given node is an interior node or a leaf. This creates a possible ambiguity, where an internal node (two 32-byte hashes) could be confused with a 64-byte transaction!

Lets say the merkle tree of a block is:

```
          H
   H0          H1
Tx0  Tx1   Tx2
                  Tx3
```

Then the BTC Relay contract could be convinced to relay a transaction containing H0 even though H0 isn't actually a transaction hash.

Joseph Chow's solution is to exclude 64-byte transactions. It's conceivable that a valid 64-byte transaction could exist in the future, and in that case it would not be able to be verified in btcRelay. This risk seems low, since such a transaction would be much smaller than the average today.

The incentive policy is misleading and may be circumvented:

Whoever submits a block header can set an arbitrary "fee". It appears that the intent is that the fee must be paid in order for "verifyTx" or "relayTx" to work. An obvious DoS vector would be for the first submitter to set this fee very high.

However, there is a separate function by which you can change the fee down to 0 (or any value smaller than the previous fee), for a different price, "gChangeRecipientFee". This makes the incentive scheme complicated to understand, and misleading, since if the cost to change the fee is less than the nominal fee, then the nominal fee no longer applies.

The game could be simplified by automatically capping the fee at the "gChangeRecipientFee". I would recommend this based on a principle of "transparent incentives", or "avoiding incentive-compatibility-through-obscurity".

Another potential DoS vector would be to set the both the initial fee and the "gChangeRecipientFee" to be very high. "gChangeRecipientFee" is typically based on the cost of initially submitting the header (gas used * gas price). So, an obvious attack would be to set the gasPrice very high (especially for a submitter who is also a miner).

There is already a countermeasure against this attack, namely that the effective gas

price is constrained to within a +/- 1/1024 factor of the effective gas price for the previous block header. This is intended to mirror Ethereum's policy for the total gas limit in a block.

If there are not many submitters, then it is plausible that a miner might gradually try to raise this price.

The incentive implementation does not *clamp* properly, leading to a DoS vector

Actually, the current implementation makes this slightly worse, because it does *not* even quite mirror Ethereum's policy. Ethereum's policy for evolving a value is:

```
LO = 1023*prevX/1024
HI = 1025*prevX/1024
if (x < LO) then x := LO
if (x > HI) then x := HI
```

whereas BtcRelay's policy is:

```
if (x < LO or x > HI) then
      x := prevX
```

So, if an Attacker (say, a 10% miner) gets lucky and increases the transaction gas price above 1.001 of the default gas price (e.g., by being the first miner to mine an Ethereum block after a new Bitcoin block is found, two consecutive Bitcoin blocks in a row), and if everyone else only uses the default gas price (50 szabo today), then the miner can *continue to jack the price up* over time.

More Notes:

Conventions for macros are inconsistently applied.

m_Blah indicates a macro.

The distinction between macro and non macro is relevant because macros are inlined, where as function calls require an entire new message, thus incurring more gas and increasing the call stack depth.

High variance attack on the first block:

There are several special cases handling the "first" block (which may not be the genesis block, but rather a contemporary block around the deployment of the). One is that the target difficulty is not enforced, since in general the target difficulty is derived from the target difficulty of the previous block.

The following comment from the code provides a rationale:

> "This allows blocks with arbitrary difficulty [to be] added to the initial parent, but as these forks will have lower score than the main chain, they will not have impact"

It would be a longshot, but a less-than-50% attacker has a much better chance of finding one block with much greater apparent difficulty than a long chain of blocks adding up to the same difficulty.

Recommendation: include an initial difficulty target along with the first parent.

Ancestor data structure is too sparsely documented:

Joseph Chow provided a more comprehensive explanation in private conversation about how the ancestor data structure worked. It would be useful to put this explanation into the comments or design documentation!

The relayTx() function sends all the remaining gas:

After a transaction is verified, when .processTransaction is invoked on the relayee, this can trigger arbitrary contract code. If this code uses up the available gas, then the code following the invocation will not be applied. In particular, there is a log event https://github.com/ethereum/btcrelay/blob/develop/btcrelay.se#L217 that will not be triggered. This could cause problems for a reasonable consumer of BTC Relay, that uses log events to check whether someone has made a “best effort” to relay the transaction to the relayee.

Recommendation: allow relayTx to include a “gas to send relayee” field, and only pass this much gas.

Note that this .processTransaction invocation is *not* susceptible to the call-stack hazard. The reason is that self.verifyTransaction() is called prior to relayee.processTransaction(). This means that if the callstack is already at 1024, then verifyTransaction() will also fail.

Otherwise, I checked that all functions pass only a limited amount of gas to the callee.

Can the call stack be advanced significantly when traversing the blockchain?

Verifying a Tx requires traversing potentially a large number of blocks. Fortunately the traversal function is iterative (fastGetBlockHash) rather than recursive, so this risk isn’t a problem.

Suboptimal skipping data structure

The ancestor data structure is in btcChain.se. The skip distances are 5**0 (the previous block) through 5**7. So the maximum skip is 5**7 = 78125.

Adding a new header requires fetching data for 8 previous headers. In total, the storage cost for N block headers is 8N.

This could be reduced to only requiring a constant amount of work to add each block, although my first attempt at this comes with a tradeoff, that it makes traversals more expensive. The following is a very rough description of the idea:

Each block B has two links, a "back" link to block B-1, and an "up" link, to a block that is more "round"

The "roundness" of a number is the largest power of 2 that divides it

So 6 has roundness 2, 96 has roundness 32…. basically it's

So let's say you want to go from 117 to 1

You'd go 117[1] 116[4] 112[16] 96[32] 64..... 64 is special because it can't go 'up' any further, so 64 just goes back half way to 32.

A worst case example would be from 128 to 65

127 126 124 120 112 96 (too far, goes to 64) 95 94 92 88 80 (too far, goes to 64) 79 78 76 72 (too far, goes to 64) etc...

This is O(log^2 N) for traversing n blocks, rather than O(log N) as it is now.

I don't know if there is any other better way that only uses 2 pointers per block.

Timestamp handling:

Timestamps aren't sanity checked. This is explicitly a cost-saving measure, and I think Joseph has a suspicion about it. Does this lead to any problems?

Casting a timestamp "into the future" can be used to lower the bitcoin difficulty. At first glance, we'd expect this to open up an attack vector where a miner who finds the 2016'th block in an interval, gives a fake timestamp, lowers the difficulty to minimum, and then very quickly finds 6 weak blocks in succession. This would trigger the verifyTx/relayTx behavior with 6 confirms, even though it would shortly be reorged. Bitcoin's defense against this sort of thing is to only allow the difficulty to change by a factor of 4 every 2016 blocks. This behavior is adequately replicated in btcRelay, so it's OK.

Recommendations:

- Test coverage:

I don't know of existing tools for checking test coverage with high level code. The Ethereum virtual machine debug logs provide enough information (e.g., program counter at each step) to debug with the EVM byte-code, but the Serpent compiler forgets most of the structure.

However, a reasonably easy-to-implement approach would be to generate a detailed log file, and look for regions of the bytecode that were never executed (i.e., the program counter never pointed at these).

- Service specification

Make a list of expected / reliable behaviors for the server.

For example, are the log messages intended to be reliable?

This would be a natural place to include caveats about 64-byte transactions not being verifiable, etc.

- Gas calculations

There can be a large variation in the amount of gas required to verify a transaction depending on how far back it goes, how deep the Merkle tree is, etc. Provide a calculator or a table?