Bitcoin Orphan Transactions and CVE-2012-3789

Dec 14, 2018 • Aleks Kircanski

Cryptocurrency clients ingest and process unauthenticated content, perform cryptographic validations and store large amounts of data. In around 2012, a couple of interesting DoS attack vectors in Bitcoin involving orphan transaction handling were reported by Sergio Demian Lerner. Given that Satoshi-like blockchain clients have been repeatedly reimplemented from scratch since then, it is likely that these type of issues re-occur in other coin implementations. For this reason, in this blog post, we discuss CVE-2012-3789 once again, add some illustrations and further commentary on the issues brought up by this CVE. The bottom-line is: when developing cryptocurrency blockchain client from scratch, surveying previously known blockchain software vulnerabilities (notably, Bitcoin, given the amount of audit it received) is necessary.

Orphan transactions

When a cryptocurrency client processes a new transaction, it must gracefully handle the case in which one (or more) of the new transaction's parent transactions are unknown. Peer to peer network do not preserve the order in which the transactions are broadcasted to nodes. For instance, a peer may broadcast a series of chained transactions in a short period of time, some of which may be delivered before the other ones. The order in which transactions are received is by no means going to be the same as the order transactions were sent. As such, nodes need to keep track of transactions with unknown parent transactions.

Enter orphan transactions. A cryptocurrency client may deal with this by:

- Keeping a list of orphan transactions
- Whenever a new (non-orphan) transaction is processed (either as a standalone transaction, or as a part of a block), the client needs to go through the list of known orphan transactions and decide which ones should be "unorphaned", i.e., reconsidered as a transaction whose all parent transactions are known.
- The previous item needs to be done recursively, as "unorphaning" a transaction is similar to receiving an entirely new transaction
- For each transaction "unorphaning", all entries describing the orphan transactions should be removed

The first requirement in the list mandates having a data structure which keeps a list of orphan transactions: such a list may be indexed by transaction hashes. The remaining requirements dictate a data structure that would somehow facilitate transaction unorphaning. Given a newly received transaction, is this transaction a parent of a known orphan transaction? If yes, is there other unknown parent transactions for this orphan? A data structure that maps orphan transactions

to their parent hashes would be helpful in this case, as, given a new transaction's hash, it would be easy to see if there is any orphans attached to it. In other words, a map that keeps `parent tx -> orphan transaction` relations for all saved orphan transactions is needed.

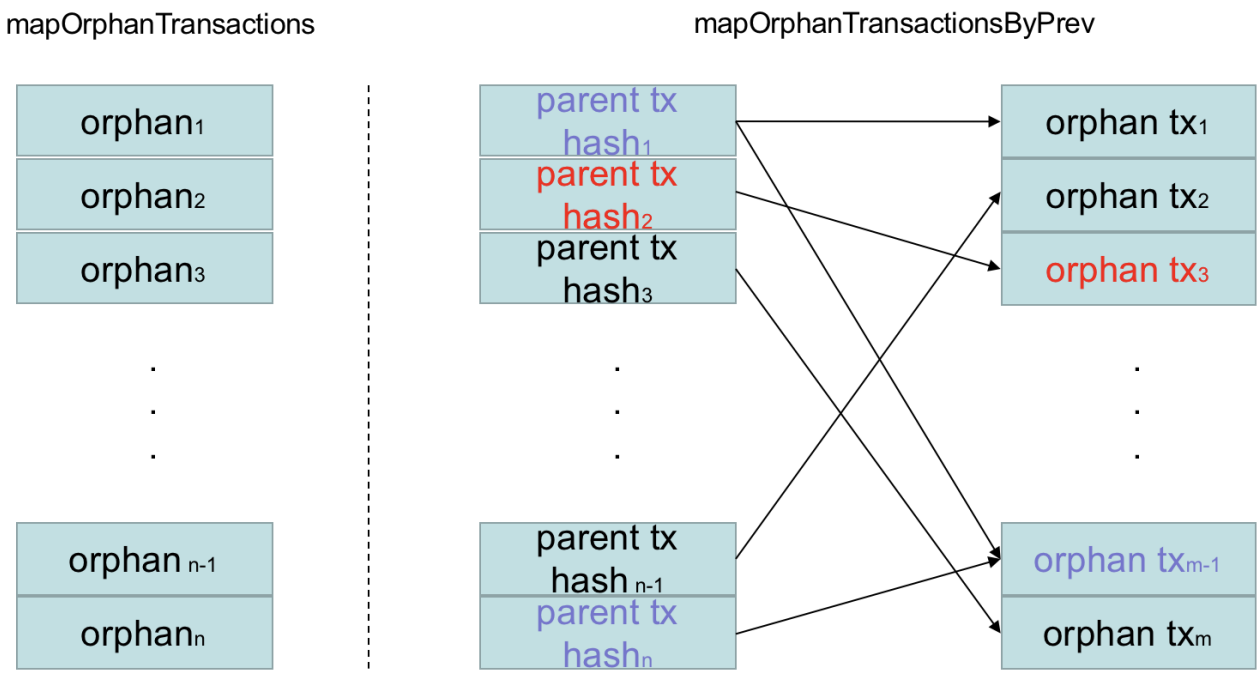
This is what the Bitcoin client's `mapOrphanBlocks` and `mapOrphanBlocksByPrev` maps do, see the source code for Bitcoin 0.6.0 relevant to the attacks discussed in this post. In particular:

```
map<uint256, CDataStream*> mapOrphanTransactions;
multimap<uint256, CDataStream*> mapOrphanTransactionsByPrev;

[...]

void AddOrphanTx(const CDataStream& vMsg)
{
    CTransaction tx;
    CDataStream(vMsg) >> tx;
    uint256 hash = tx.GetHash();
    if (mapOrphanTransactions.count(hash))
        return;

    CDataStream* pvMsg = mapOrphanTransactions[hash] = new CDataStream(v
    BOOST_FOREACH(const CTxIn& txin, tx.vin)
        mapOrphanTransactionsByPrev.insert(make_pair(txin.prevout.hash,
}
```



Given the `mapOrphanTransactionByPrev` map, when a new transaction arrives and is accepted to the mempool, it is now easy to look up what orphan transactions depend on it. Suppose that `orphan-tx-hash_3` (colored in red on the picture) is currently stored as orphan. Now assume its parent transaction `parent-tx-hash_2` arrives and is deemed to be valid.

Since `orphan-tx-hash_3` depends only on `parent-tx-hash_2`, it can be unorphaned and erased from the orphan memory store. Now `orphan-tx-hash_3` is regarded as a new transaction that may unorphan other orphan transactions. The recursive unorphaning algorithm implemented in a form of a loop is here.

With such orphan handling mechanism in place, let's discuss the issues that can arise with it.

DoS via excessive `parent -> orphan child` relations (part of CVE-2012-3789)

Consider a peer to peer network where a peer sends data to the target peer and the target peer stores this data in some form. If there is little or no cost involved on the side of the sending peer and there is no limit on the stored data size on the side of the receiving node, memory/storage exhaustion Denial of Service concerns arise: the receiving peer can simply be made to store more data it can handle. The exploitability of such a DoS attack vector would be determined by factors such as whether the receiving peer stores all the data sent to it (or, say, just the hash of the data), the network throughput between peers, etc.

Limiting the data amount a peer will receive could be achieved by simply refusing to receive new data entries, or by introducing an ejection policy. For instance, entries could be ejected based on age and other factors, or purely randomly. While these approaches do mitigate the previously mentioned memory exhaustion Denial of Service, it is interesting to note that legitimate entries may be blocked from entering the memory store or, legitimate entries may be ejected from the memory store. This by itself could be considered a problem.

Enough digression and back to the orphan transactions issue we're discussing here, the limitless orphan memory store allowed a straightforward memory exhaustion DoS attack in early Bitcoin. As a result, the number/size threshold on the orphan store was added by introducing an orphan ejection policy: an old orphan is randomly chosen and ejected once the size threshold is surpassed.

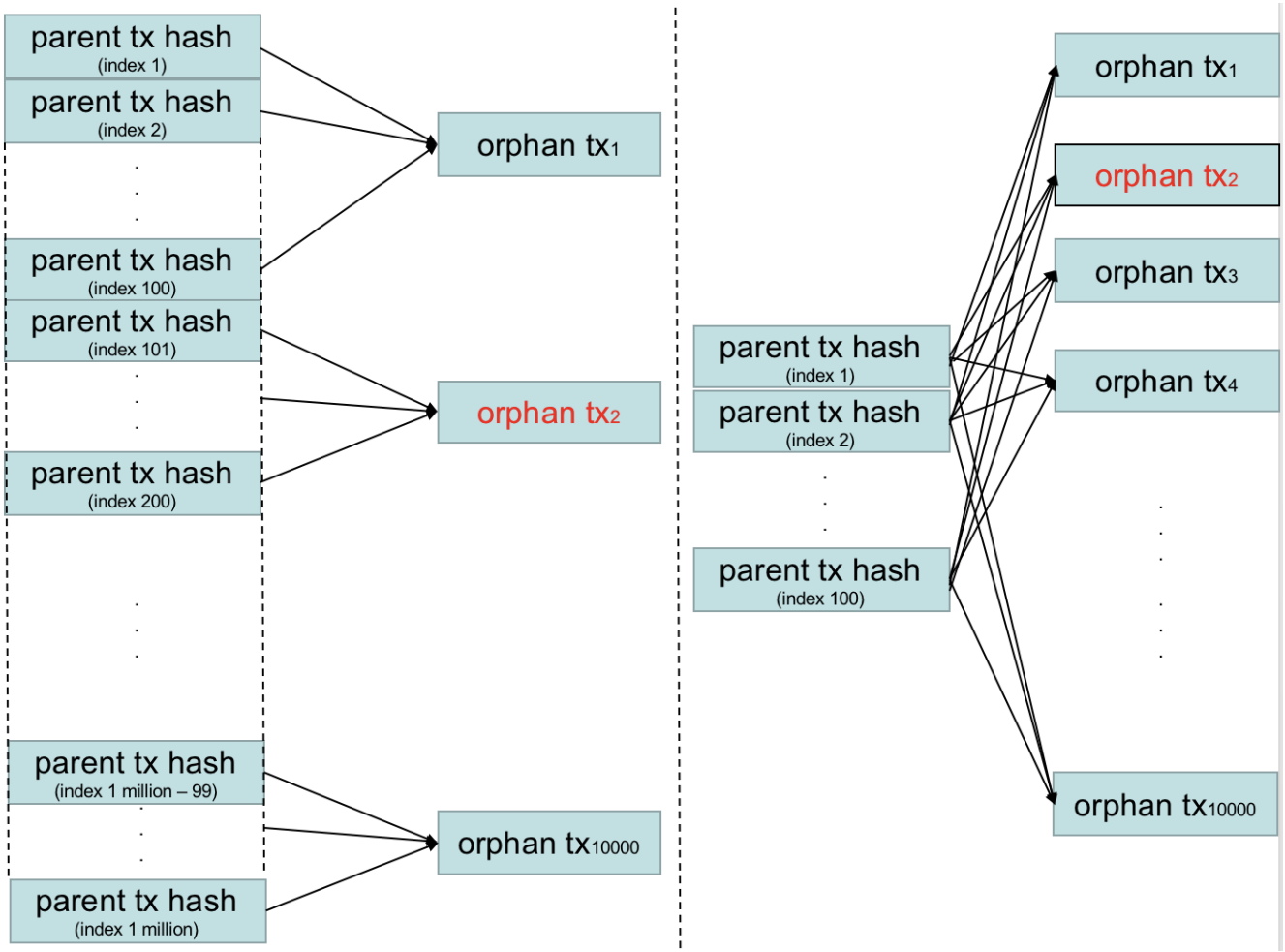
The function that ejects/deletes orphans is here:

```
void static EraseOrphanTx(uint256 hash)
{
    if (!mapOrphanTransactions.count(hash))
        return;
    const CDataStream* pvMsg = mapOrphanTransactions[hash];
    CTransaction tx;
    CDataStream(*pvMsg) >> tx;
    BOOST_FOREACH(const CTxIn& txin, tx.vin)     {
        for (multimap<uint256, CDataStream*>::iterator mi = mapOrphanTra
             mi != mapOrphanTransactionsByPrev.upper_bound(txin.prevout.
        {
            if ((*mi).second == pvMsg)
                mapOrphanTransactionsByPrev.erase(mi++);
            else
```

```
                mi++;
            }
        }
        delete pvMsg;
        mapOrphanTransactions.erase(hash);
    }
```

Before deletion, the corresponding transaction is pulled from `mapOrphanTransactions`. The for loop looks up the entries in `mapOrphanTransactionsByPrev` and decides which pairs to delete. A trivial implementation here would go over all of the map entries. A natural optimization present in the early Bitcoin client is to go only through those `mapOrphanTransactionsByPrev` entries that are indexed by actual inputs to the transaction that we're deleting (see the for loop bounds in the code snippet). Out of such a constrained set of pairs, finally, only the pairs with expected child transaction are matched (see the if condition). It is possible for a parent transaction to be a parent of multiple different transactions and not necessarily those we want to delete. Going back to the illustration above, when deleting `orphan-tx-hash_m-1`, `EraseOrphanTx` iterates over three pairs, but only deletes two of them: `(parent-tx-hash_1, orphan-tx-hash_1)` does not get deleted since `orphan-tx-hash_1` does not match the deleting transaction.

The question that CVE-2012-3789 answers positively is whether it's possible to have a sufficiently high number of iterations in the for loop above to cause CPU exhaustion. The maximum number of orphans is 10000, see (MAX_ORPHAN_TRANSACTIONS). An orphan's parent transaction is identified with a transaction hash and the parent transaction's output index. A transaction can't repeat parent transaction entries, however, an orphan transaction can reference the same parent hash with different output indexes. Consider what happens if the victim client ends up with the `mapOrphanTransactionsByPrev` store in the state described by the *left* side of the picture:



The right column in the left side of the picture is basically one (non-existent) transaction repeated with different output indexes. All of the 10000 orphans point to the same (unknown) transaction. Suppose the client now needs to delete `orphan-tx_2-hash`. The `EraseOrphanTx` function now iterates through all of the edges on the picture, since the parent transaction hash maps to the set of orphan transactions 100*10000 times. Repeated deletion itself can be triggered by sending orphans on top of 10000 orphans, since surpassing the threshold triggers an ejection. As such an attacker can achieve CPU exhaustion on the target node.

A similar DoS setting that one may consider is shown on the right side of the picture. In this case, each of the 10000 orphans depend on the same 100 inputs of the same transaction. A single transactions cannot have duplicate inputs, however if a client does not reject orphans referencing the same inputs, then the number of edges the deletion loop needs to go over is similar to as on the left side of the picture.

TL;DR: Handling orphan transaction in coin clients is fertile ground for memory/CPU exhaustion attacks, since getting victim nodes to store orphan transactions comes at no cost. See this related discussion on DoS vectors on orphan blocks. Care should be exercised when storing and processing data maps coming from peers, especially if the sending peers do not pay fees or do not need a PoW for the data to be processed. Finally, limiting a data store's size for memory exhaustion DoS mitigation can be done by introducing a data entry ejection policy. However, ejecting entries leads to a (possibly) unintended consequence of legitimate data entry being erased. As shown by CVE-2012-3789, Denial of Service attacks easily sneak into apps.

Cryptography Services

Cryptography Services is a dedicated team of consultants from NCC Group focused on cryptographic security assessments, protocol and design reviews, and tracking impactful developments in the space of academia and industry.