Report

Ethereum Eclipse Attacks

Author(s):
Wüst, Karl; Gervais, Arthur

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010724205 →

Ethereum Eclipse Attacks

Karl Wüst
ETH Zurich
8092 Zurich, Switzerland
kwuest@ethz.ch

Arthur Gervais
ETH Zurich
8092 Zurich, Switzerland
arthur.gervais@inf.ethz.ch

Abstract

In this technical report, we present three vulnerabilities affecting the Ethereum blockchain network and client. First, we outline an eclipse attack that allows an adversary to partition the peer-to-peer network without monopolizing the connections of the victim. This is attack is possible by exploiting the block propagation design of Ethereum. Second, we present an exploit to force a node to accept a longer chain with lower total difficulty than the main chain. Finally, we outline a bug in Ethereum's difficulty calculation. We provide countermeasure proposals for each reported vulnerability.

1 Permanent Eclipse Attack

1.1 Description

In the following, we describe a consensus critical vulnerability in the Ethereum peer-to-peer protocol. The vulnerability can be exploited in an eclipse attack that causes Denial of Service and that can also be used by an adversary to double spend transactions. Due to the low resource requirements of the attack, an attacker with limited capabilities can easily sustain an attack on the whole ethereum network.

1.2 Components

The attack appears to be a vulnerability in the peer-to-peer protocol. However, it was only tested with the `geth` client.

1.3 Background

In the following, any information that is not part of the yellow paper[1] is based on the official golang implementation of Ethereum (`geth`).

1.3.1 Block Propagation

Blocks are propagated in three different ways in the Ethereum network. Firstly, a node B that receives a new block, will directly push the block to $\sqrt{n}$ of its connected peers, where n is the total number of connected peers. Secondly, B will send a NewBlockHashes message to all of its peers, advertising a new block. When a node A receives an advertisement, it will request the block explicitely after 0.5 seconds from a random peer from which it received a corresponding advertisement (unless A received the block from another peer in the meantime) and then forgets about all other advertisements for that block. This means that if A requests the block from B and B fails to answer, it will not request the block from any other peers. If A misses a block, the third method for block propagation is used, which is the block synchronisation explained below in section 1.3.2.

1.3.2 Block Synchronisation

A node will only synchronise with one other node at a time. A node A starts a block synchronisation in the following cases:

- A starts a connection to a new peer with higher advertised total difficulty (e.g. after joining or rejoining the network).
- A node advertising a higher total difficulty than A connects to A.
- A receives a block with higher total difficulty than the head of its current blockchain and is missing some of the blocks ancestors.

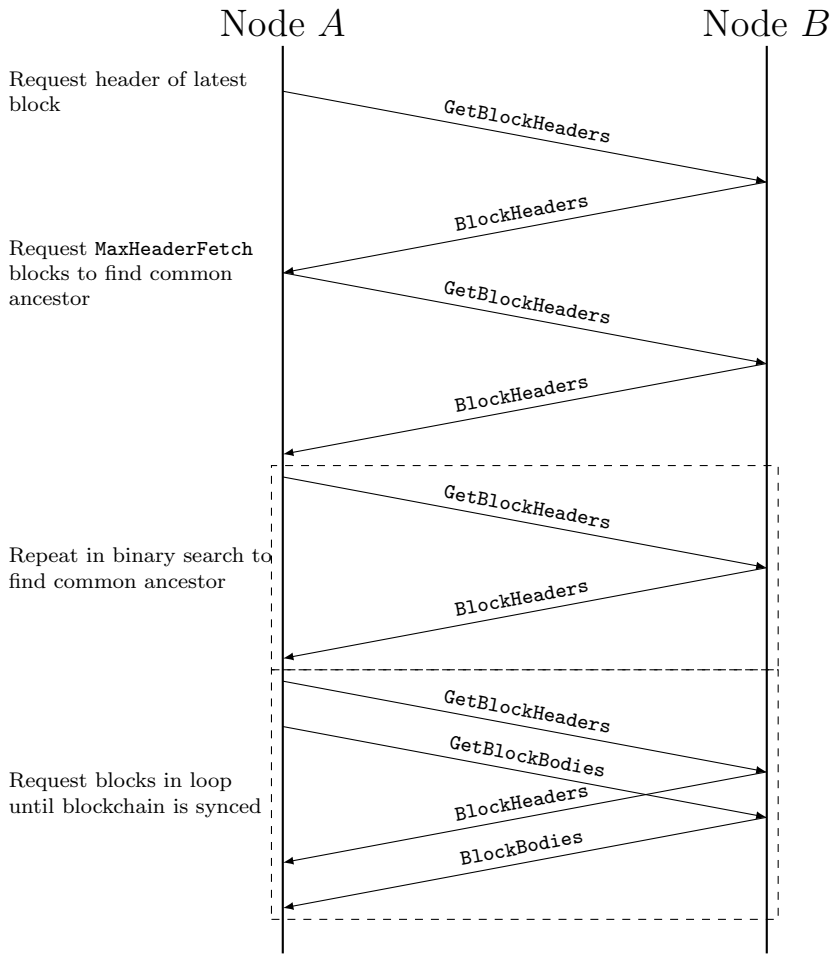


Figure 1: Block Synchronisation between two Nodes A and B

The block synchronisation (Fig. 1) works as follows:

1. The Node with lower total difficulty (node A) sends a `GetBlockHeaders` request to the node with higher total difficulty (node B), requesting the header of the latest block of B.
2. B responds with a `BlockHeaders` message containing the block header of the block specified in the received `GetBlockHeaders` message.
3. A requests the `MaxHeaderFetch` (= 256) Blocks starting at `MaxHeaderFetch` blocks below the height of its own blockchain.
4. B sends up to `MaxHeaderFetch` of the requested blocks (but may send fewer)
5. If none of the blocks received from B are in As blockchain, A starts a binary search over its own blockchain to find a common ancestor, requesting one block from B per step in the search.
6. As soon as A finds a common ancestor, A requests block headers and bodies from B starting from the common ancestor. A asks for `MaxHeaderFetch` blocks per request, but B may send fewer.

1.4 Attack Details

The following is a consensus critical flaw in the peer-to-peer protocol. Using this vulnerability, an attacker B can keep a victim A from receiving a block at height $n + 1$ almost indefinitely. While A may receive later blocks from other peers, it will be stalled at height n since it misses a link in the blockchain. An attacker B can work as follows:

1. B creates a long blockchain starting from the genesis block. This can be done by decreasing the difficulty for each block, thus shortening the time needed to generate a valid proof of work. Creating the alternative blockchain takes a relatively large amount of precomputation. However, once the alternative blockchain exists, it does not need to be computed again and can be used for multiple attacks.
2. If Bs chain is shorter than the valid chain, B forges a block with a high block number and valid proof of work, i.e. it creates a block without parent that uses an arbitrary value as parent hash.

3. If the attacker blockchain is shorter than the valid blockchain, B also needs to forge a block with block number `MaxHeaderFetch` below the height of As blockchain and blocks at heights that will be queried in the binary search but are not contained in Bs blockchain.
4. B connects to Node A, advertising a high total difficulty
5. B sends the block header with the highest block number from his chain (or the block created in step 2) to A in response to the first `GetBlockHeaders` request from A
6. In response to the second `GetBlockHeaders` request from A, B sends the block from its seperate blockchain corresponding to the block number specified in As message (or the block forged in step 3).
7. In response to the requests from As binary search, B responds again with block headers from its own blockchain until the genesis block is reached.
8. A will now request blocks starting from the genesis block. B responds with blocks from its own blockchain, while sending only one block in response for each request. For each request, B can introduce a delay of up to 3 seconds.
9. As long as the attack is in progress, A will not be able to synchronise with any other peers.

1.5 Attack Scenario

A node will sometimes miss a block (i.e. receives neither the block directly nor a NewBlockHashes message, or the asked peer does not respond to the request to get the block corresponding to NewBlockHashes) and needs to synchronise with the network. A will start a synchronisation if it receives a NewBlock message containing a Block with total difficulty higher than the current total difficulty of the node plus the difficulty of the block. In this case, the node will try to synchronise with the peer that sent the NewBlock message.

However, a node will never attempt to synchronise with more than one node at once. This means that if an adversary connects to a node A and starts the synchronisation attack, the node will not be able to synchronise with the valid chain once it misses a block, thus keeping it from receiving the missing block as long as the synchronisation attack persists.

Let us assume that A has all blocks up to block number n from the valid chain, but does not receive block $n + 1$ while the synchronisation attack is in progress. A will still receive newly mined blocks from the valid chain (e.g. blocks $n + 2$, $n + 3$), but it will not accept them as part of the blockchain yet, since it cannot connect them to its blockchain due to the missing block.

Normally, A would, when receiving block $n + i$ ($i \geq 2$) from peer P, start a synchronisation with P to receive all intermediate blocks. However, this will not happen, since A is already synchronising with B.

Since the difficulty to mine a block can be reduced with every block by setting the timestamp of each block to the parent timestamp plus 13 (in the frontier phase) until it reaches the minimum difficulty, a chain surpassing the length of the valid chain can be mined with a single consumer grade GPU within a few weeks.

With such a long chain, the attack can be sustained for weeks (or until it is detected). When mining at or close to the minimum difficulty, a block can be mined in fractions of a second. Since we can delay a block for up to 3 seconds, an attacker can even continue to mine blocks during the attack and is thus only has to stop the attack once the timestamp is too far ahead of the current time.

Due to the block propagation mechanism described in Section 1.3.1, an attacker can even influence the rate of missed blocks at a victim node in order to make the attack successful more quickly. The attacker can connect many nodes to his victim that are modified to never push a block directly but always send NewBlockHashes immediately and never answer to a request to get an advertised block. Since the victim only asks one peer for an advertised block, it will not receive the block if it requests the block from one of the attacker nodes.

1.6 Implementation & Reproduction

The attack was implemented and can be reproduced by modifying a `geth` client to mine a low difficulty chain as described above that surpasses the length of the valid chain and by introducing a delay of 2 seconds for responses to peers. The adversarial chain was mined within 3 weeks on a single Radeon R9 280x GPU. After mining the chain and introducing delays in the client, the attack can then be executed as described in section 1.4.

1.7 Impact

The vulnerability can cause Denial of Service and can be used to double spend confirmed transactions. The attack, when executed against multiple victims also causes multiple forks in the blockchain and

severly impacts consensus. In our assessment, the vulnerability is critical with a CVSS score of 9.1[1].

1.7.1 Denial of Service

The attack described above is trivially a denial of service attack. While the attack is in progress, the attacked node A will not append any blocks to its blockchain and thus not update the accepted state.

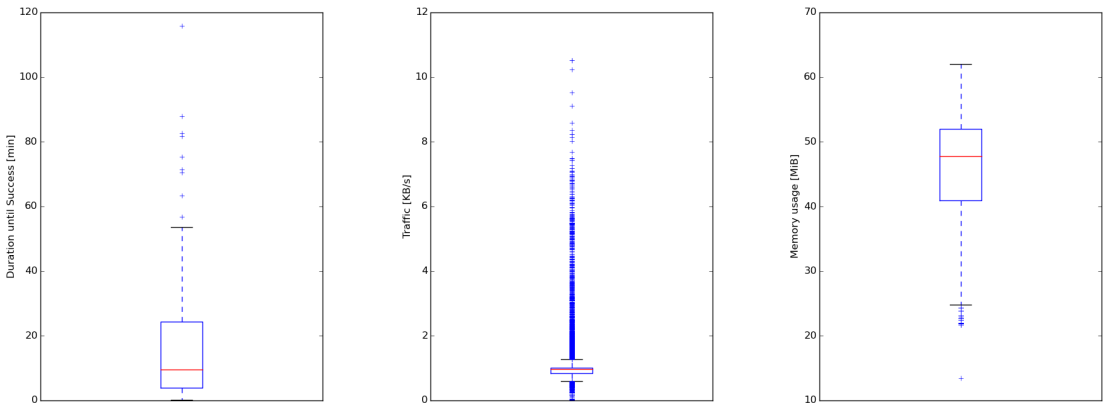
This also has implications on mining. If A is a mining node, it will continue to mine on what A accepts as valid blockchain, i.e. A will mine on top of block n and not receive any rewards for mined blocks (except an uncle reward for the first block, if the attack stops early enough). Thus, the attack is effectively also a denial of service attack on mining.

Since the resource requirements of the attack are very low, an attacker could easily mount a denial of service attack on the whole network (see section 1.7.3).

1.7.2 Double Spending

While the synchronisation attack is in progress and the victim A is stalled at block n, the attacker can connect a second node to A on which he mines a block on top of block n on the valid chain. A will accept this block as new head of his chain and will accept any transaction included in this block. However, since the network already has a chain with higher total difficulty, the rest of the network will not accept the transaction. This allows an attacker to double spend a transaction.

1.7.3 Resource Requirements and Attack Success



(a) Duration until Attack is successful (b) Network Traffic (Upload) (c) Memory Usage

Figure 2: Resource usage for the attacker node

The attack was tested on a node connected to the Ethereum network with the default settings. In our experiments, the attacker node is isolated from the network and only connects to the victim node. We ran the attack 100 times and measured the duration until the attack was successful (i.e. the time until the victim node is stalled at some block), and the network, memory and CPU usage of the attacker node (Fig. 2).

The duration from start of the attack until success has a median value of 9.48 minutes, a mean of 18.68 minutes and a maximum of 115.87 minutes. The attack requires only 0.96 KB/s upload on average with a peak of 10.52 KB/s (download traffic is negligible). The attacker node requires 45 MB of memory on average (max. 62 MB) and has an average CPU usage of 0.07% on an Intel Core i7-4770 where 100% is one core.

Due to the small resource requirements, a Denial of Service attack on the whole network that would cause multiple forks is feasible with constraint resources. The Ethereum network currently consists of approximately 6500 nodes. Assuming the resource requirements scale linearly, attacking the whole network would require an upload bandwidth of approximately 6.5 MB/s, and approximately 307 GB of memory with a CPU usage of 455% (Intel Core i7-4770). Since an attack on the whole network does not require running 6500 full nodes, the memory requirements are likely significantly

[1]CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:H/E:F/RC:C/CR:L/IR:H/AR:H

lower. Attacking the whole network would thus be possible with a few desktop PCs, a fibre connection and a router that is able to sustain 6500 TCP connections. Keep in mind that it would be enough to attack mining nodes for a denial of service attack, i.e. the estimated requirements are an upper bound and the practical requirements are likely a lot lower.

1.8 Possible Countermeasures

- Request block from other peers that have sent the NewBlockHashes message (instead of just 1 random peer), if the first asked peer does not respond. This would decrease the number of missed blocks.
- Allow to synchronise to more than one node. This has to be considered carefully, however, due to the increased load on the network.

2 Synchronising to longer chain with lower total difficulty than the main chain

2.1 Description

A bug in the implementation of the block synchronisation makes it impossible for a node to synchronise to a shorter chain with higher total difficulty than the currently accepted chain.

2.2 Attack Scenario

A node A newly connects to the network and receives a chain that is longer than the valid chain but has a lower total difficulty (e.g. because the adversary advertised a higher total difficulty than honest nodes). A can then no longer synchronise with the valid chain.

2.3 Impact

The attack is a Denial of Service attack on newly joining nodes. Overall severity is low.

2.4 Components

This vulnerability is present in the official golang implementation of the Ethereum client (`geth`), at least up to version 1.4. It is likely also present in other implementations.

2.5 Details

When a node A synchronises the blockchain with another node B, it requests 256 blocks from B, starting at block $n - 256$, where n is the height of As chain (see Section 1.3.2). If A does not receive any blocks in response, it will disconnect from B and will not continue with the synchronisation. This seems reasonable at first, however, since a chain can have a higher difficulty than another longer chain, this could potentially be used in an attack on a newly joining node as follows:

1. B mines a chain that is longer than the valid chain by at least 256 blocks (which is feasible, see section 1.4).
2. B advertises a higher total difficulty than the valid chain.
3. B connects to newly joining node A.
4. A synchronises with node B and receives his chain.
5. Once As chain is at least 256 blocks longer than the valid chain, A will not be able to synchronise with a non malicious node.

2.6 Reproduction

The reproduce the issue, follow the steps outlined in section 2.5.

2.7 Possible Countermeasure

Request blocks starting at block $\min(n, n') - 256$, where n is the length of the chain at node A and n' is the length of the chain at node B.

3 Increasing the MinimumDifficulty Parameter in `geth`

3.1 Description

A bug in the block difficulty calculation, can cause the MinimumDifficulty Parameter to increase.

3.2 Attack Scenario

An attacker sends blocks mines a chain at minimum difficulty and provokes a node to synchronise to this chain in order to increase the MinimumDifficulty parameter to be higher than the difficulty of a block in the valid chain. The victim is then no longer able to synchronise to the valid chain.

3.3 Impact

The attack causes Denial of Service for an attacked node. The attack is, however, difficult to execute in practice and the vulnerability is trivial to fix. Overall severity is low.

3.4 Components

This vulnerability consists of an implementation bug in the official golang implementation of the Ethereum client (`geth`), at least up to version 1.4.

3.5 Details

The bug is part of the function to calculate the difficulty of a block (in package `core`, i.e. the function `CalcDifficulty` in versions $< 1.3.4$, and the functions `calcDifficultyFrontier` and `calcDifficultyHomestead` in versions $\geq 1.3.4$). The functions first calculate the difficulty by subtracting or adding a fraction of the parent difficulty to the parent difficulty, depending on the difference in the timestamp to the parent. If this calculated difficulty is lower than `MinimumDifficulty`, the difficulty is set to `MinimumDifficulty`. Starting at block 200000, an exponential component is added to the difficulty afterwards ($2^{n/100000-2}$, where n is the block number).

The listing below shows the implementation of `calcDifficultyFrontier` (the implementation for `calcDifficultyHomestead` is analogous):

```
1  func calcDifficultyFrontier(time, parentTime uint64, parentNumber, parentDiff *big.Int) *
       big.Int {
2      diff := new(big.Int)
3      adjust := new(big.Int).Div(parentDiff, params.DifficultyBoundDivisor)
4      bigTime := new(big.Int)
5      bigParentTime := new(big.Int)
6
7      bigTime.SetUint64(time)
8      bigParentTime.SetUint64(parentTime)
9
10     if bigTime.Sub(bigTime, bigParentTime).Cmp(params.DurationLimit) < 0 {
11         diff.Add(parentDiff, adjust)
12     } else {
13         diff.Sub(parentDiff, adjust)
14     }
15     if diff.Cmp(params.MinimumDifficulty) < 0 {
16         diff = params.MinimumDifficulty
17     }
18
19     periodCount := new(big.Int).Add(parentNumber, common.Big1)
20     periodCount.Div(periodCount, ExpDiffPeriod)
21     if periodCount.Cmp(common.Big1) > 0 {
22         // diff = diff + 2^(periodCount - 2)
23         expDiff := periodCount.Sub(periodCount, common.Big2)
24         expDiff.Exp(common.Big2, expDiff, nil)
25         diff.Add(diff, expDiff)
26         diff = common.BigMax(diff, params.MinimumDifficulty)
27     }
28
29     return diff
30 }
```

In line 16 of the listing, `params.MinimumDifficulty` is assigned to `diff`. The value is not copied, however. Instead, `diff` and `params.MinimumDifficulty` now reference the same object. In line 25 of the listing, the exponential component is added to `diff`, i.e. the sum is written to the object referenced by `diff` and thus by `params.MinimumDifficulty`.

This could potentially be used for an attack on a node A by an attacker B as follows:

1. B mines a long chain at minimum difficulty (using the flawed implementation to calculate the difficulty).

2. B connects to A, advertising a higher difficulty than As chain.
3. A synchronises with B.
4. Since A checks the difficulty for each block received by B, for every block, `params.MinimumDifficulty` is increased by the exponential component.
5. After syncing enough blocks, As `params.MinimumDifficulty` is larger than the difficulty of the current head of the blockchain.
6. A can no longer synchronise with the valid blockchain.

3.6 Reproduction

The reproduce the issue, follow the steps outlined in section 3.5.

3.7 Possible Countermeasure

Replace `diff.Add(diff, expDiff)` with `new(big.Int).Add(diff, expDiff)` (and analogously for `calcDifficultyHomestead`).

References

[1] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.