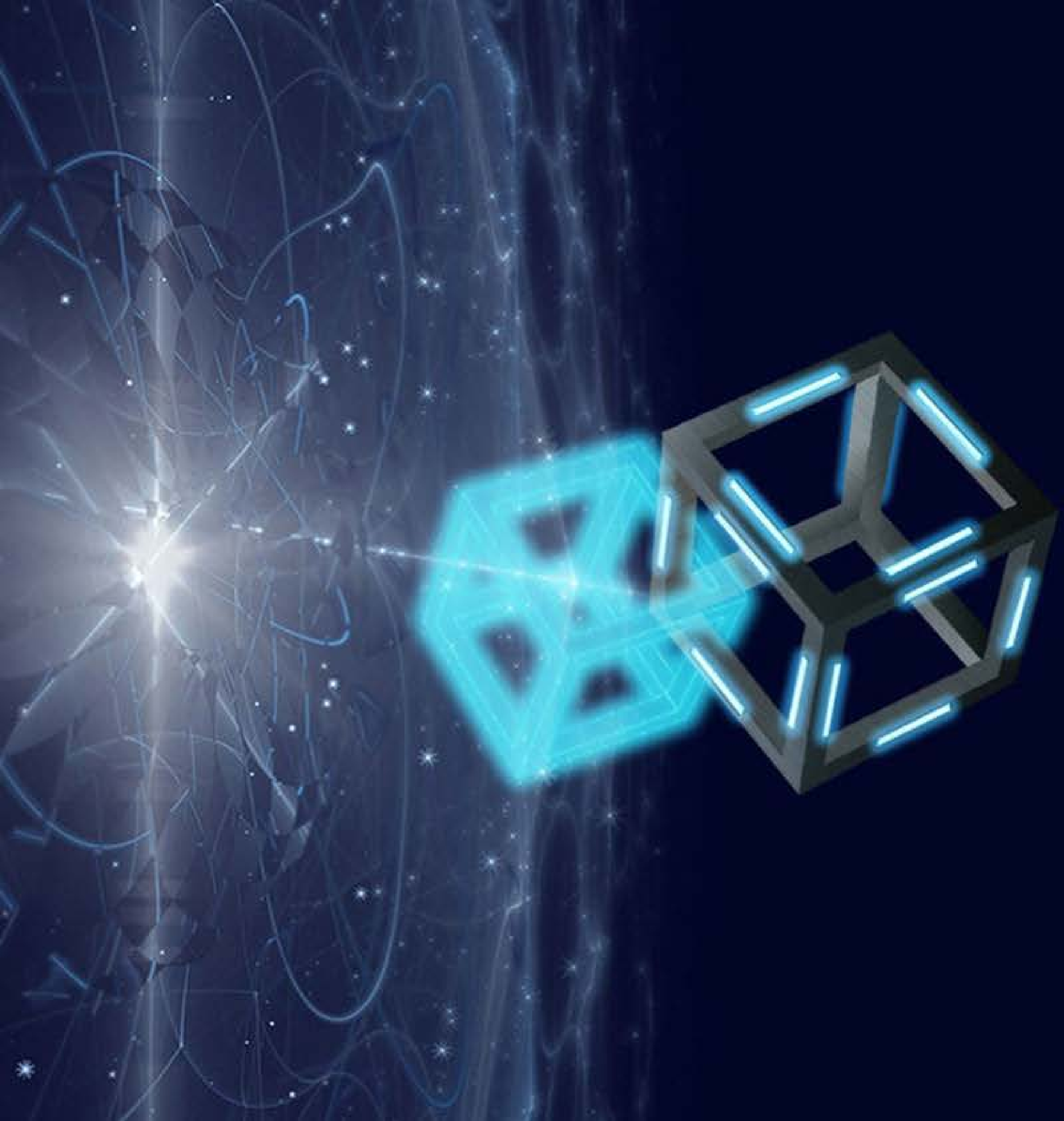
ETHEREUM SMART CONTRACT DEVELOPMENT

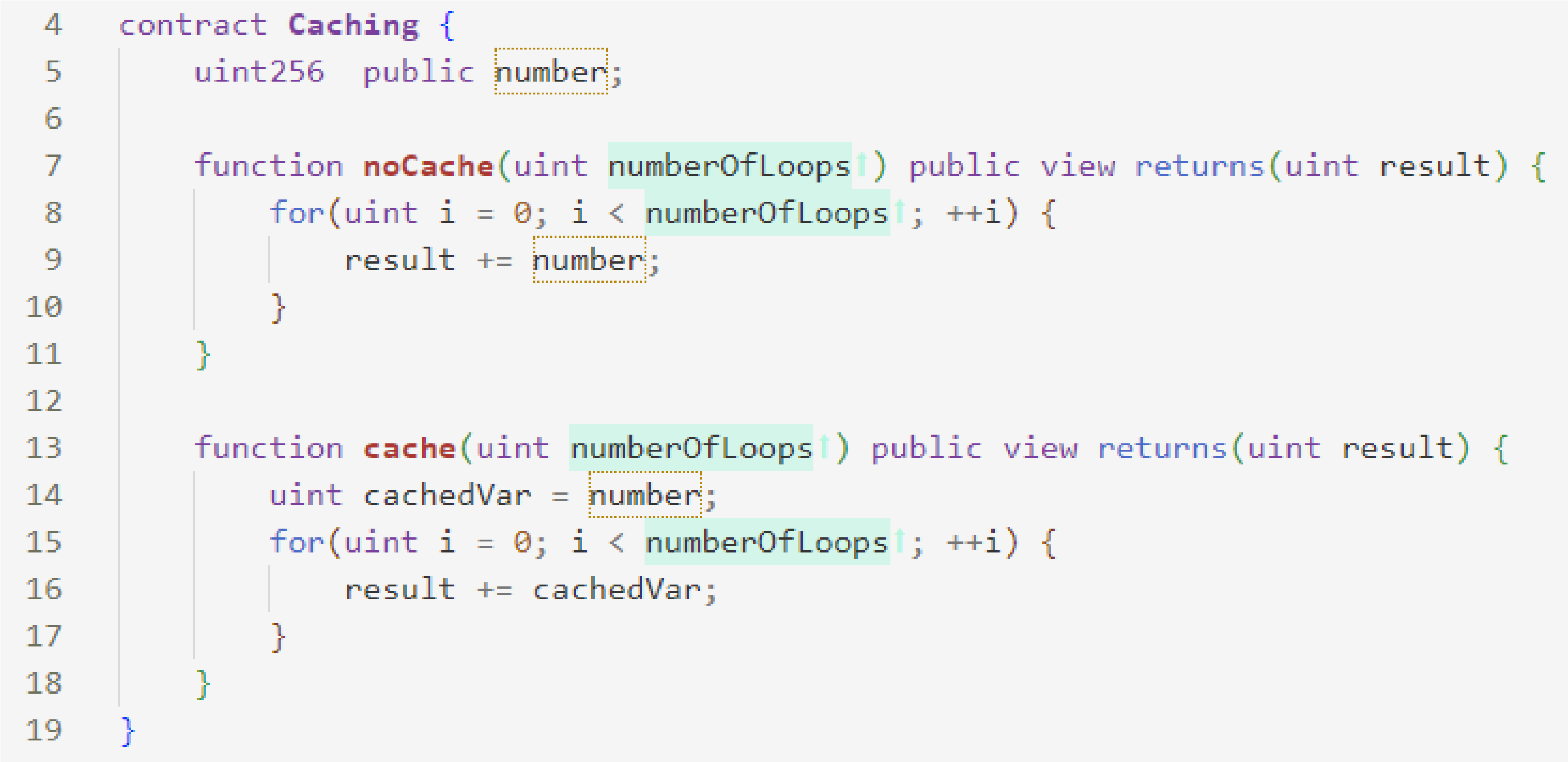
Gas Optimization - Vulnerabilities - Slither



Solidity Gas Optimization

Caching Storage Variables

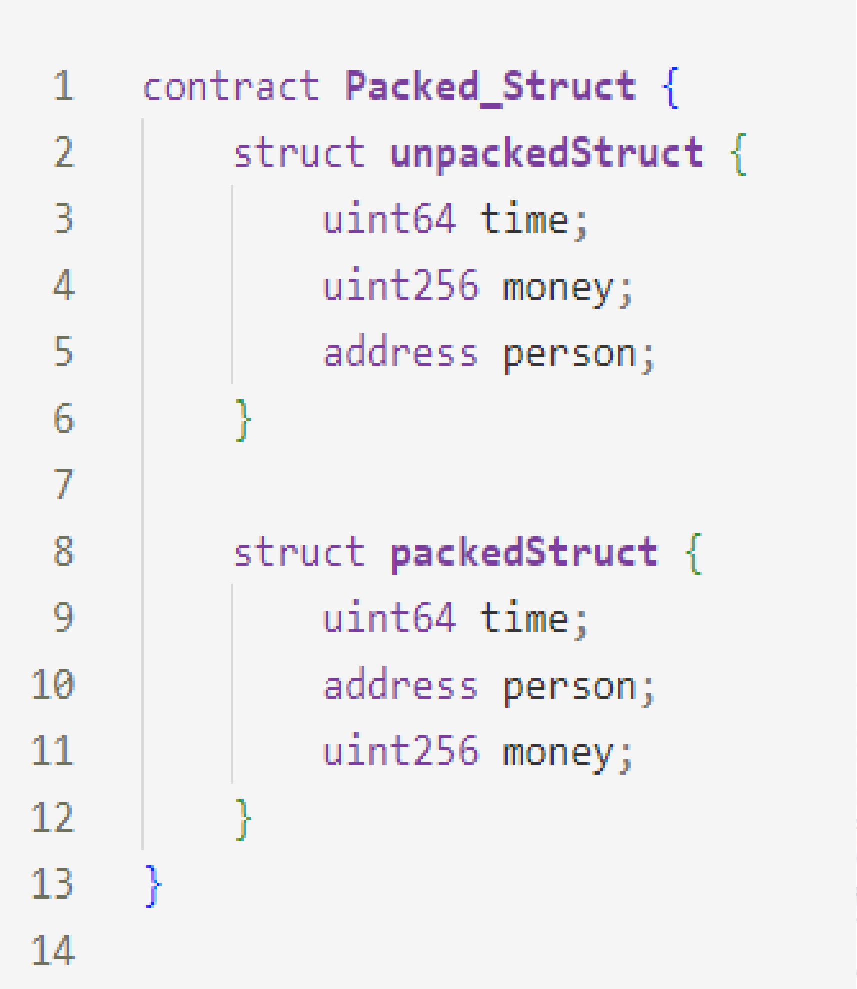
- Reading from a storage variable costs at least 100 gas
- Writing is much more expensive
- Cache storage variables to perform a single read and write operation



```
contract Caching {
    uint256  public number;

    function noCache(uint numberOfLoops) public view returns(uint result) {
        for(uint i = 0; i < numberOfLoops; ++i) {
            result += number;
        }
    }

    function cache(uint numberOfLoops) public view returns(uint result) {
        uint cachedVar = number;
        for(uint i = 0; i < numberOfLoops; ++i) {
            result += cachedVar;
        }
    }
}
```

Pack Structs

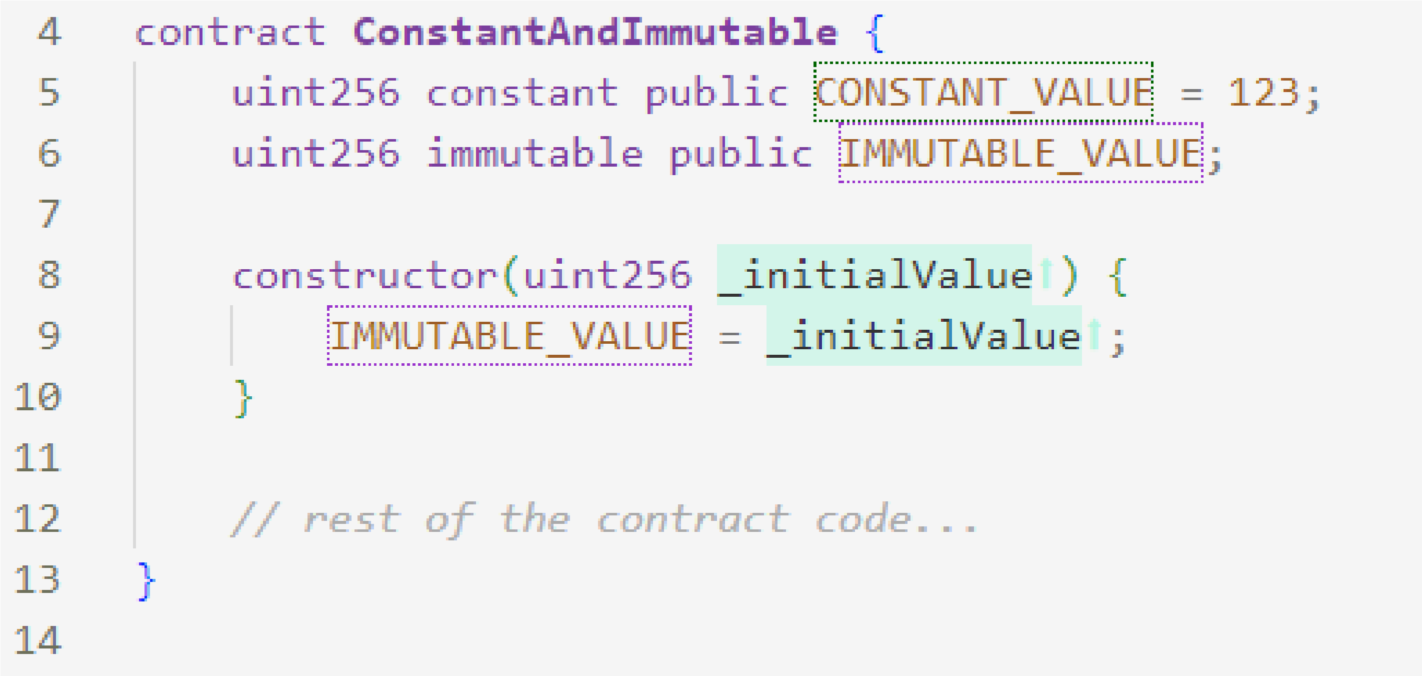
- ➢ Packing state variables into the same slot reduces gas costs by minimizing costly storage related operations
- ➢ Elements of the first structure are stored in three separate slots
- ➢ Elements of the second structure are stored in only two separate slots => cheaper read and write operations



```
contract Packed_Struct {
    struct unpackedStruct {
        uint64 time;
        uint256 money;
        address person;
    }

    struct packedStruct {
        uint64 time;
        address person;
        uint256 money;
    }
}

```

Using Immutable and Constant Variables

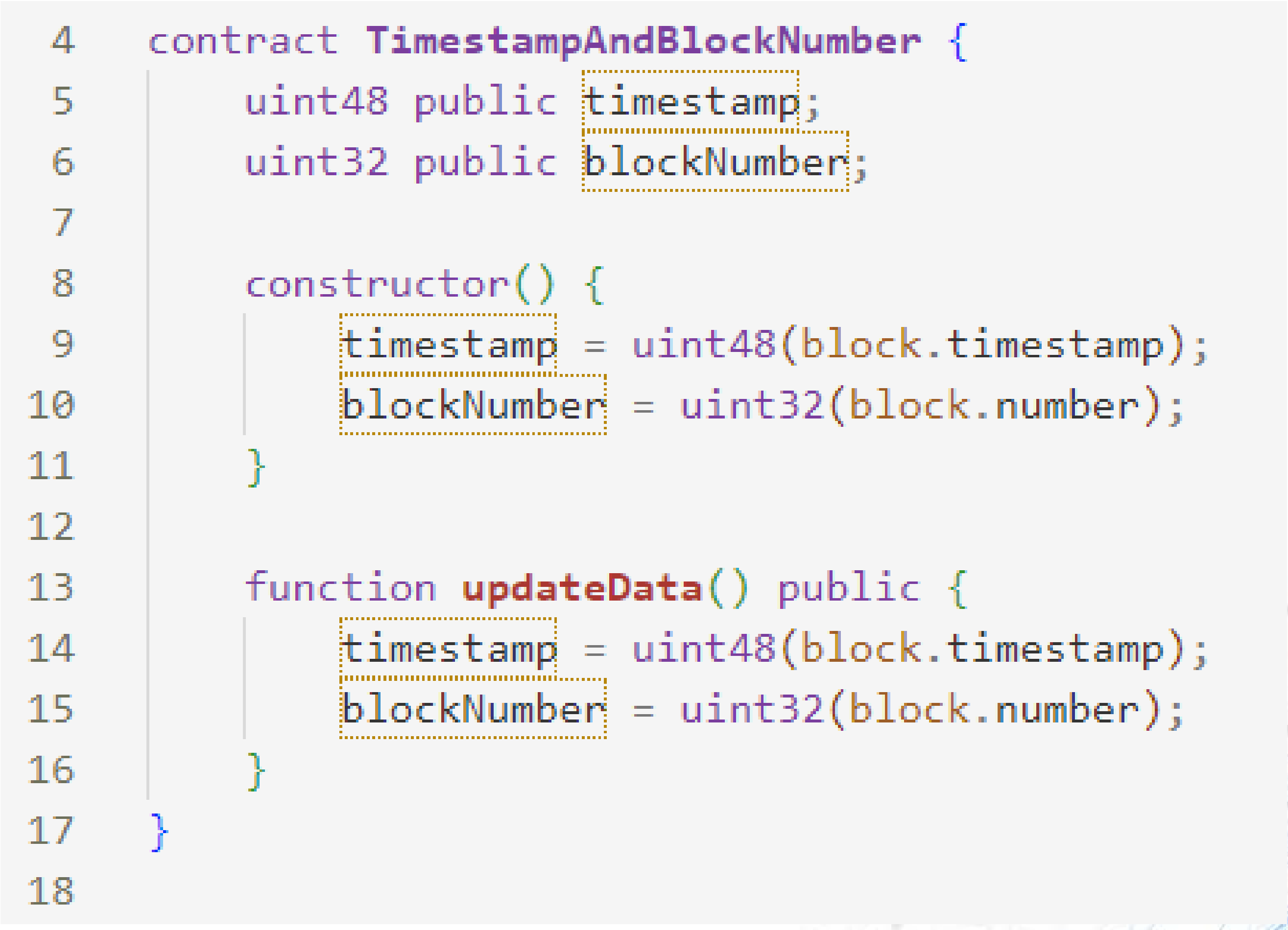
- Variables that are never updated should be immutable or constant
- Constant and Immutable values are integrated directly into the contract bytecode and do not use storage



```
contract ConstantAndImmutable {
    uint256 constant public CONSTANT_VALUE = 123;
    uint256 immutable public IMMUTABLE_VALUE;

    constructor(uint256 _initialValue) {
        IMMUTABLE_VALUE = _initialValue;
    }

    // rest of the contract code...
}

```

Timestamps & Block Numbers in Storage do Not Need to be uint256

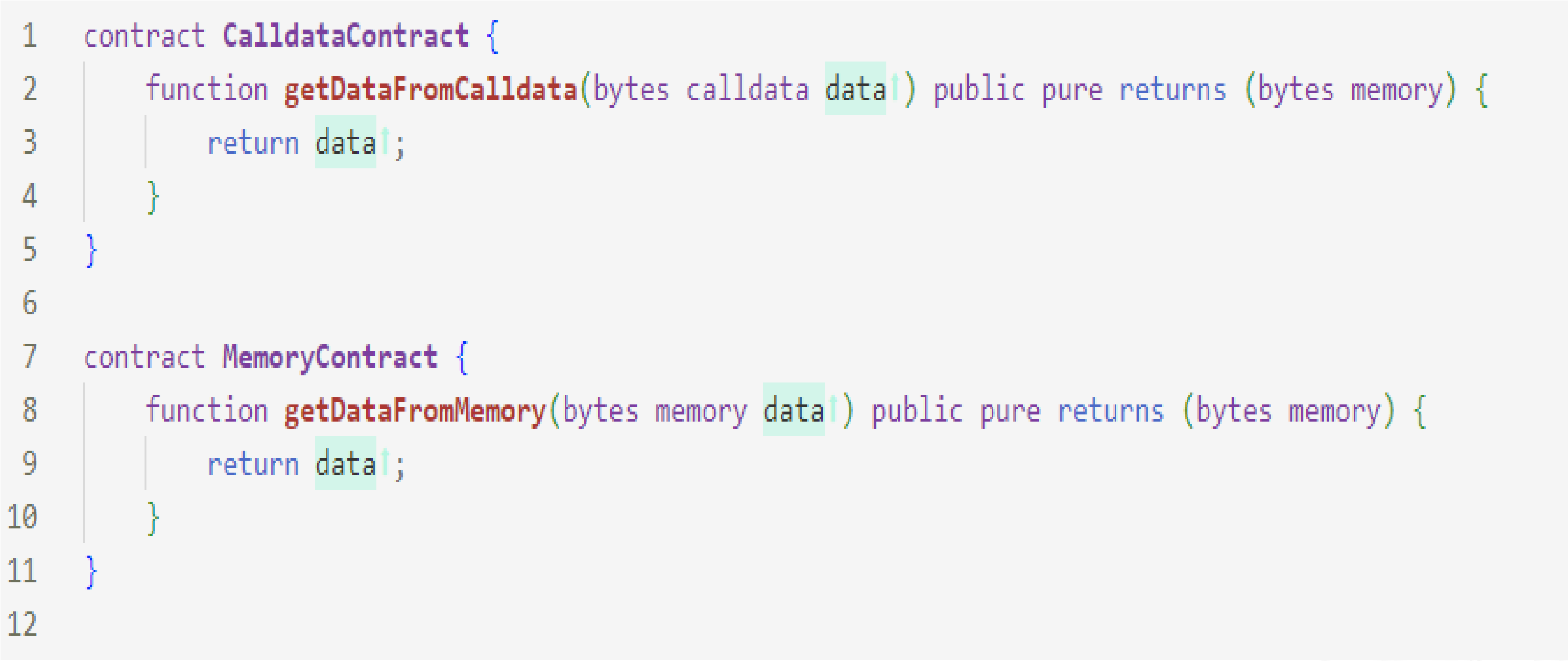
- A timestamp of size **uint48** will work for millions of years into the future
- A block number increments once every 12 secondsn => **uint32** is sufficient



```
contract TimestampAndBlockNumber {
    uint48 public timestamp;
    uint32 public blockNumber;

    constructor() {
        timestamp = uint48(block.timestamp);
        blockNumber = uint32(block.number);
    }

    function updateData() public {
        timestamp = uint48(block.timestamp);
        blockNumber = uint32(block.number);
    }
}

```

Calldata is Cheaper Than Memory

- Only for reference-type arguments of external functions
- Accessing data from calldata requires fewer operations
- Use memory only when data needs to be modified

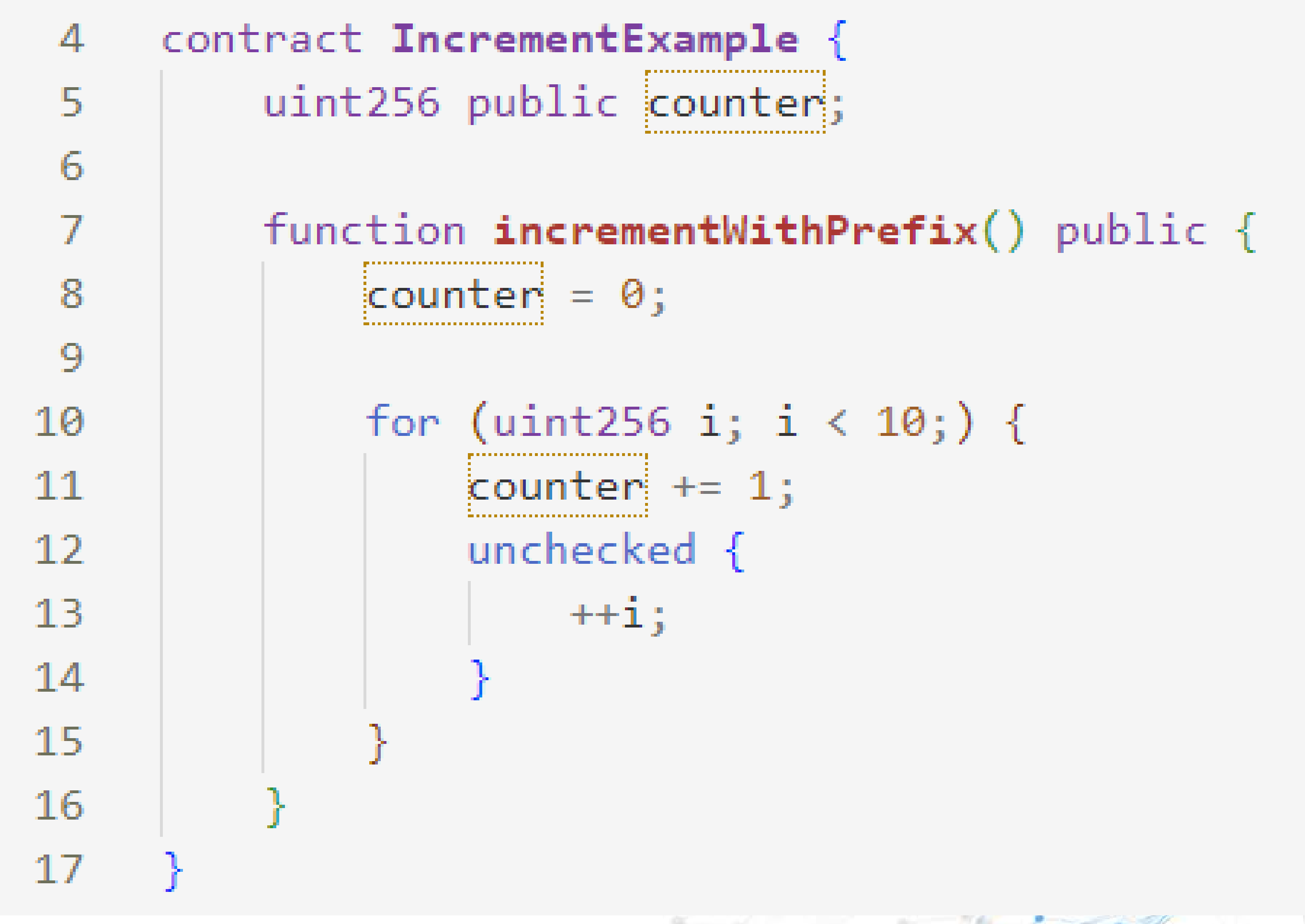


```
contract CalldataContract {
    function getDataFromCalldata(bytes calldata data) public pure returns (bytes memory) {
        return data;
    }
}

contract MemoryContract {
    function getDataFromMemory(bytes memory data) public pure returns (bytes memory) {
        return data;
    }
}

```

Use ++i instead of i++ to Increment

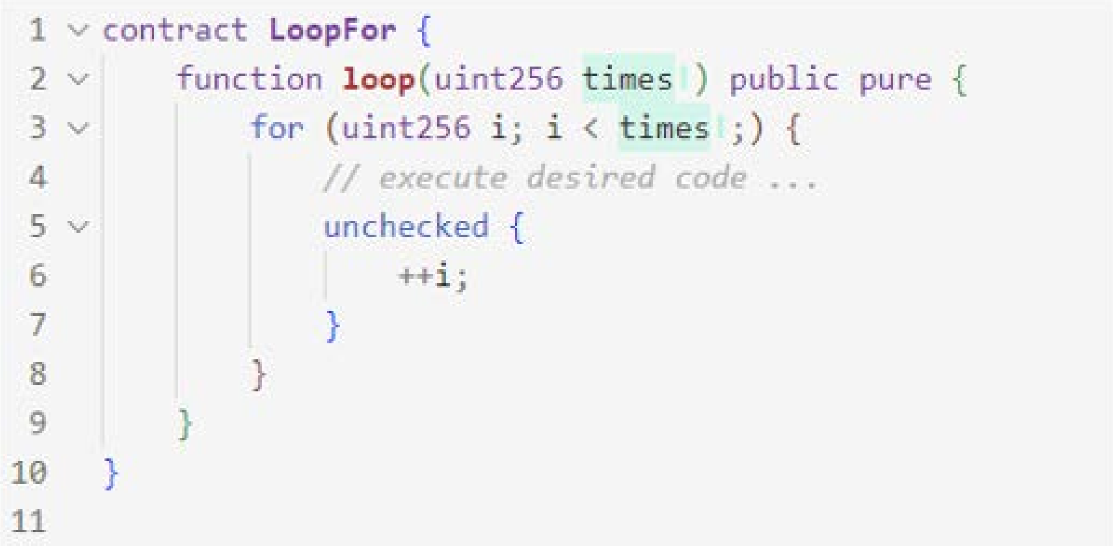
- **i++** returns its old value before incrementing it => 2 values are stored on the stack

- **++i** increments i then returns i => only one item needs to be stored on the stack

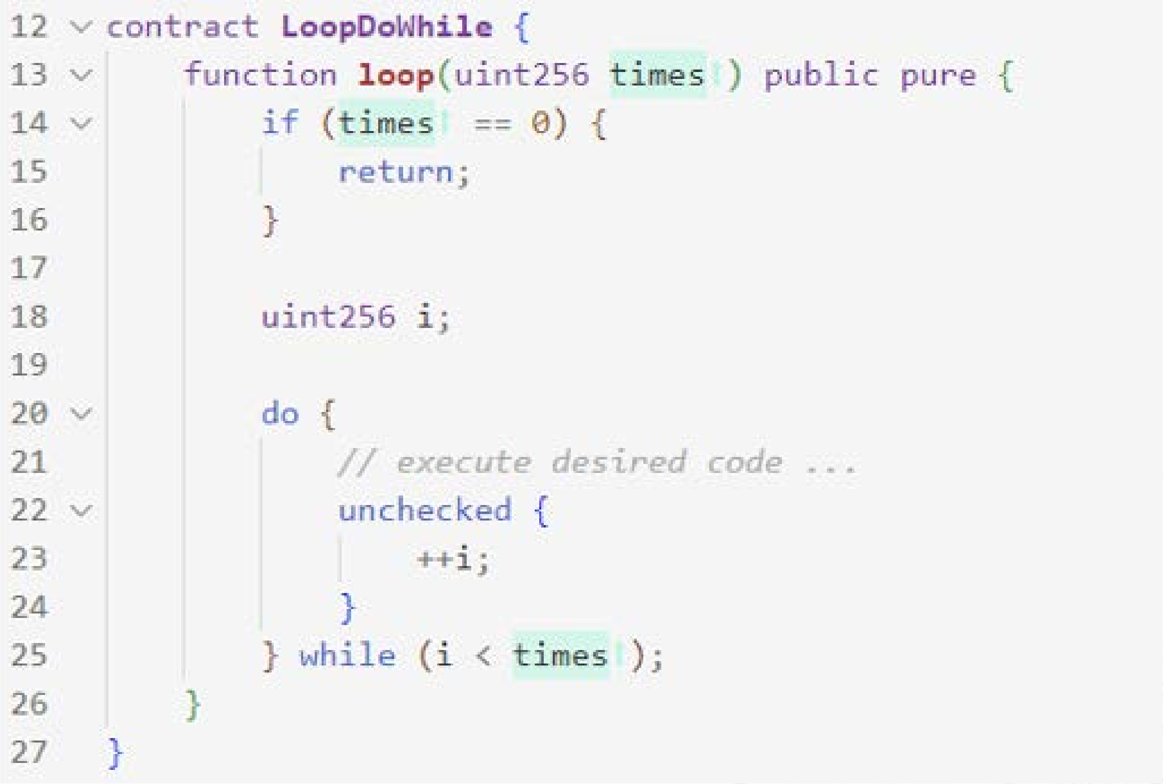


```
contract IncrementExample {
    uint256 public counter;

    function incrementWithPrefix() public {
        counter = 0;

        for (uint256 i; i < 10;) {
            counter += 1;
            unchecked {
                ++i;
            }
        }
    }
}
```

Do-While Loops are Cheaper than For Loops



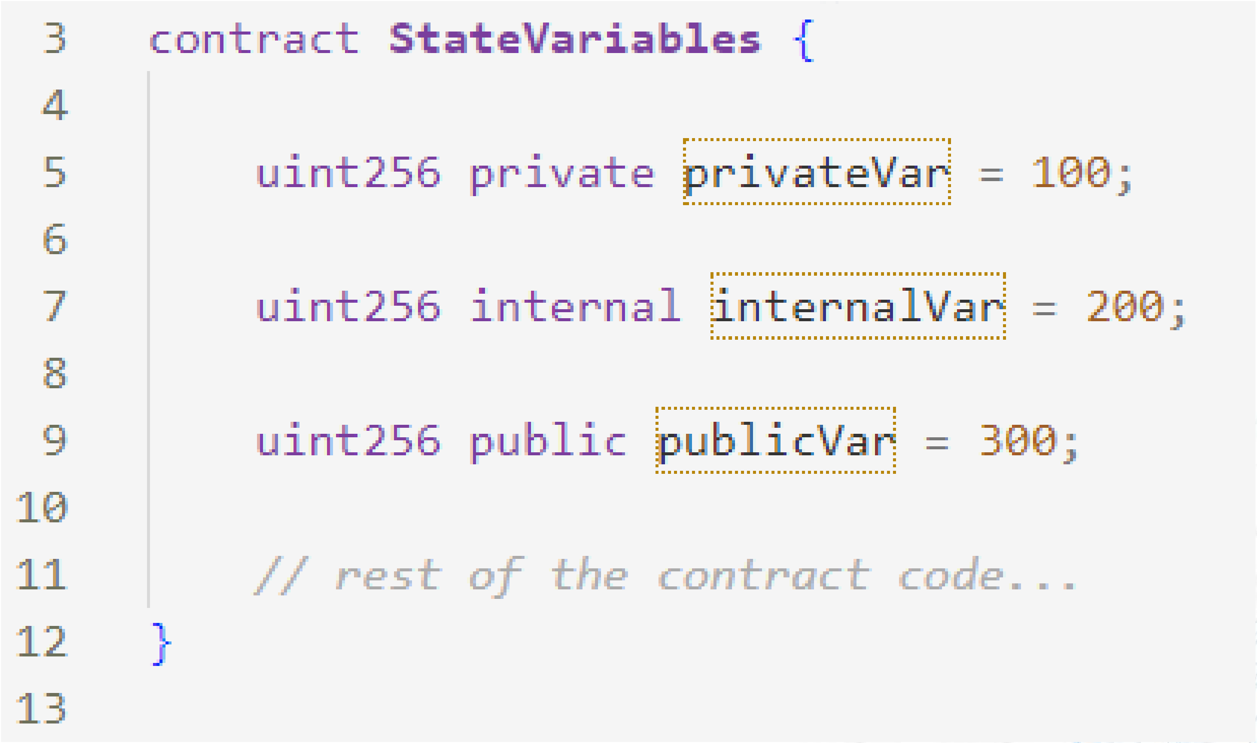
```
contract LoopFor {
    function loop(uint256 times ) public pure {
        for (uint256 i; i < times ;) {
            // execute desired code ...
            unchecked {
                ++i;
            }
        }
    }
}

```



```
contract LoopDoWhile {
    function loop(uint256 times ) public pure {
        if (times  == 0) {
            return;
        }

        uint256 i;

        do {
            // execute desired code ...
            unchecked {
                ++i;
            }
        } while (i < times );
    }
}
```

Don’t Make Variables Public Unless it is Really Necessary

- A public function (getter) is created for public storage variables
- Increases the size of the jump table
- Increases the size of the bytecode
- Makes the contract larger



```
contract StateVariables {

    uint256 private privateVar = 100;

    uint256 internal internalVar = 200;

    uint256 public publicVar = 300;

    // rest of the contract code...
}

```

Additional Resources

- The RareSkills Book of Solidity Gas Optimization :

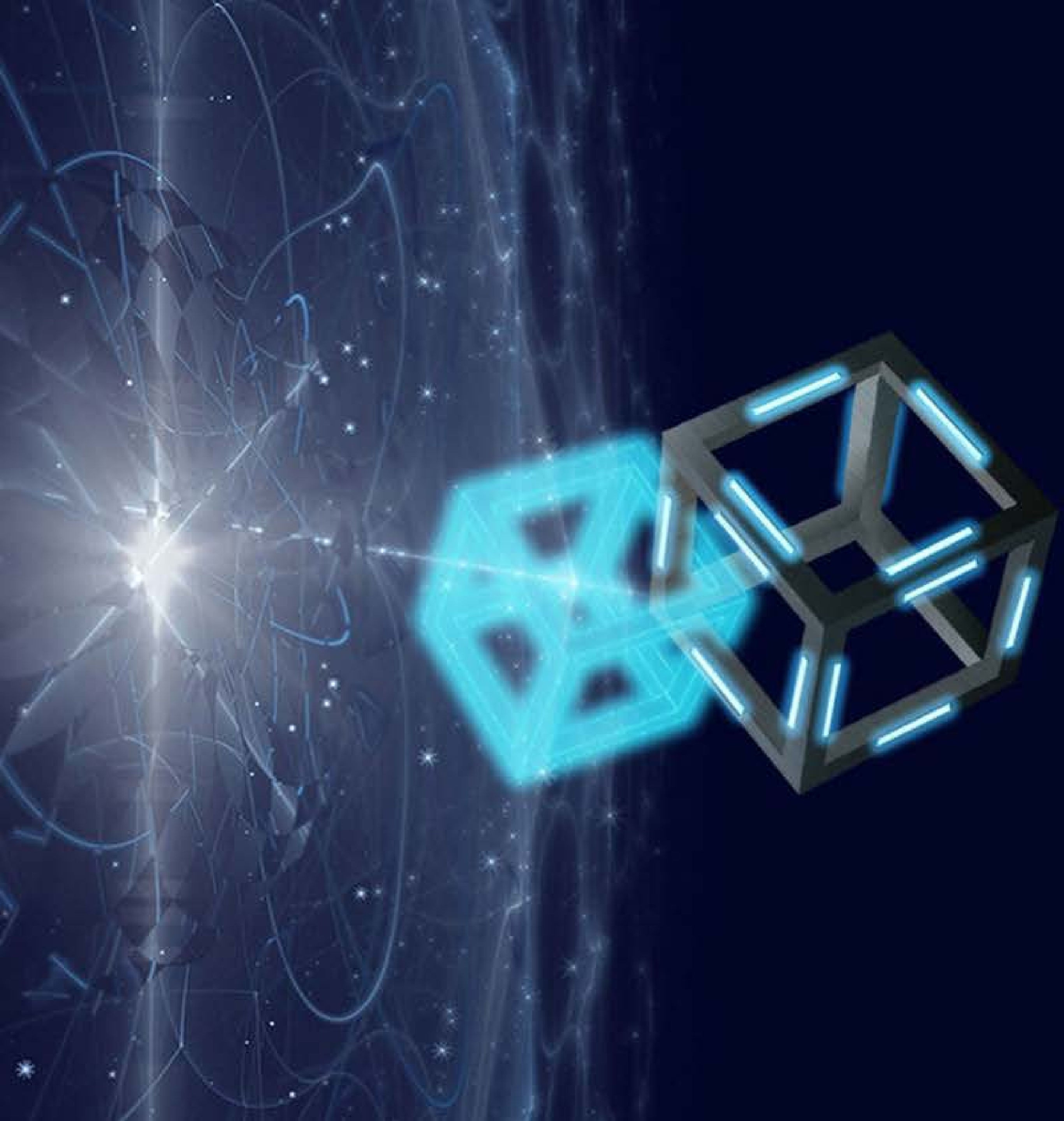
  https://www.rareskills.io/post/gas-optimization

- How to Optimize Smart Contracts in Solidity :

  https://medium.com/@0xkaden/how-to-write-smart-contracts-that-optimize-gas-spent-on-ethereum-30b5e9c5db85

- Solidity Gas Optimizations Cheat Sheet :

  https://0xmacro.com/blog/solidity-gas-optimizations-cheat-sheet



Smart Contract Vulnerabilities

Smart Contract Vulnerabilities

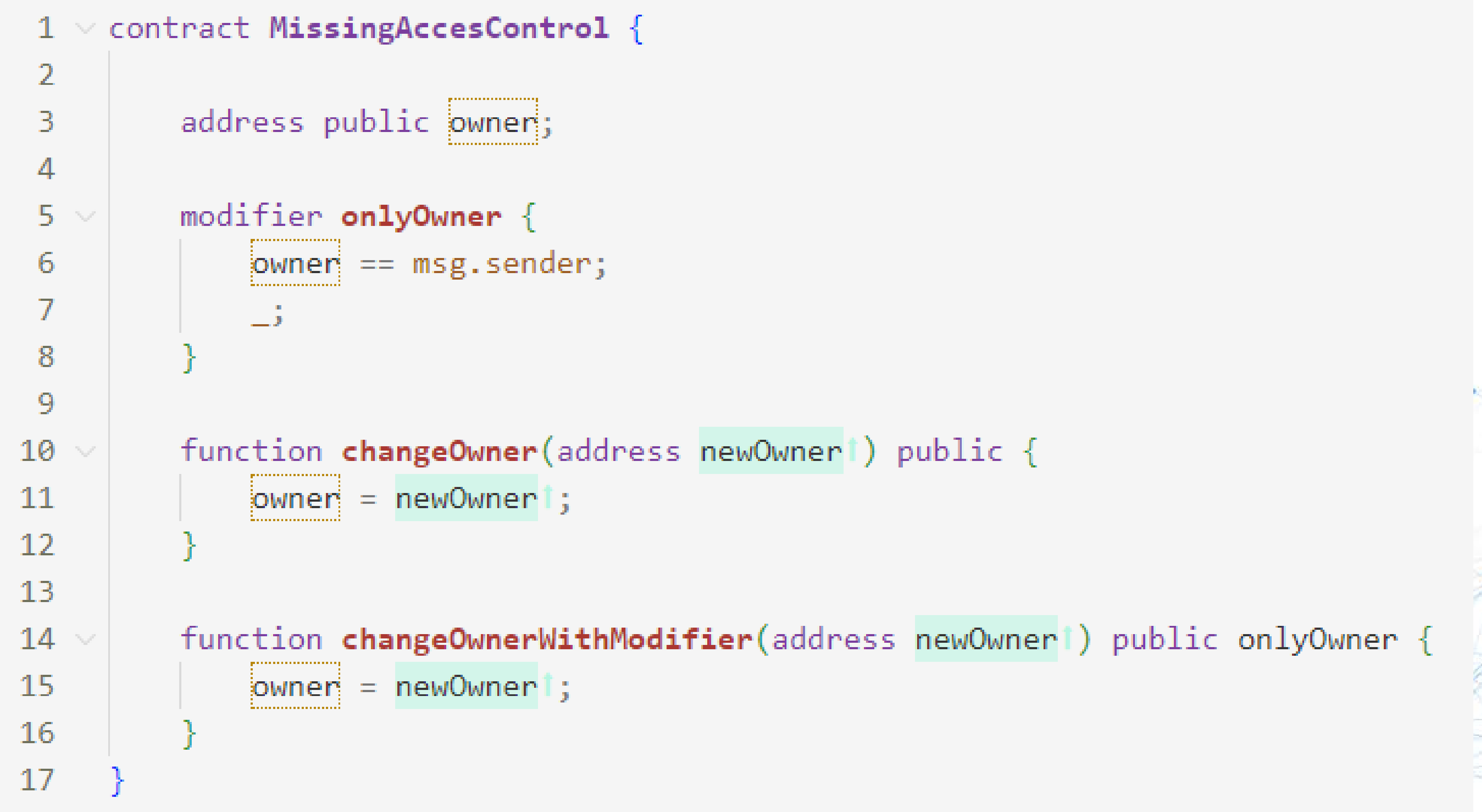
- Vulnerabilities pose significant risks => financial losses, rendering a protocol unusable...

- An audit and thorough testing are essential before deploying smart contracts

Missing Access Control

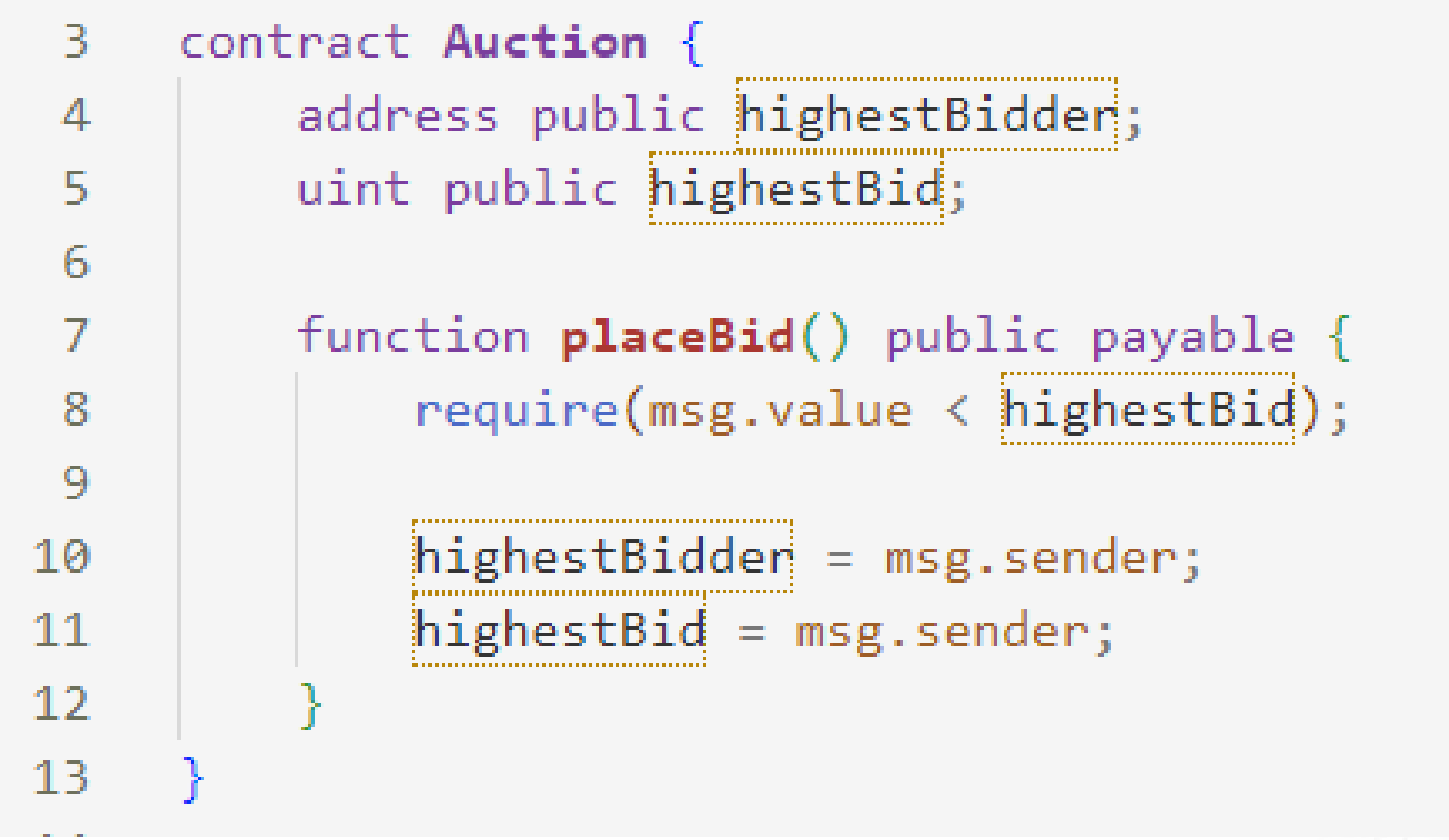
- Placing restrictions on who can call sensitive functions, such as withdrawing Ether, changing the contract owner...

- Even if a modifier is in place, there have been cases where the modifier was not used



```
contract MissingAccesControl {

    address public owner;

    modifier onlyOwner {
        owner == msg.sender;
        _;
    }

    function changeOwner(address newOwner) public {
        owner = newOwner;
    }

    function changeOwnerWithModifier(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

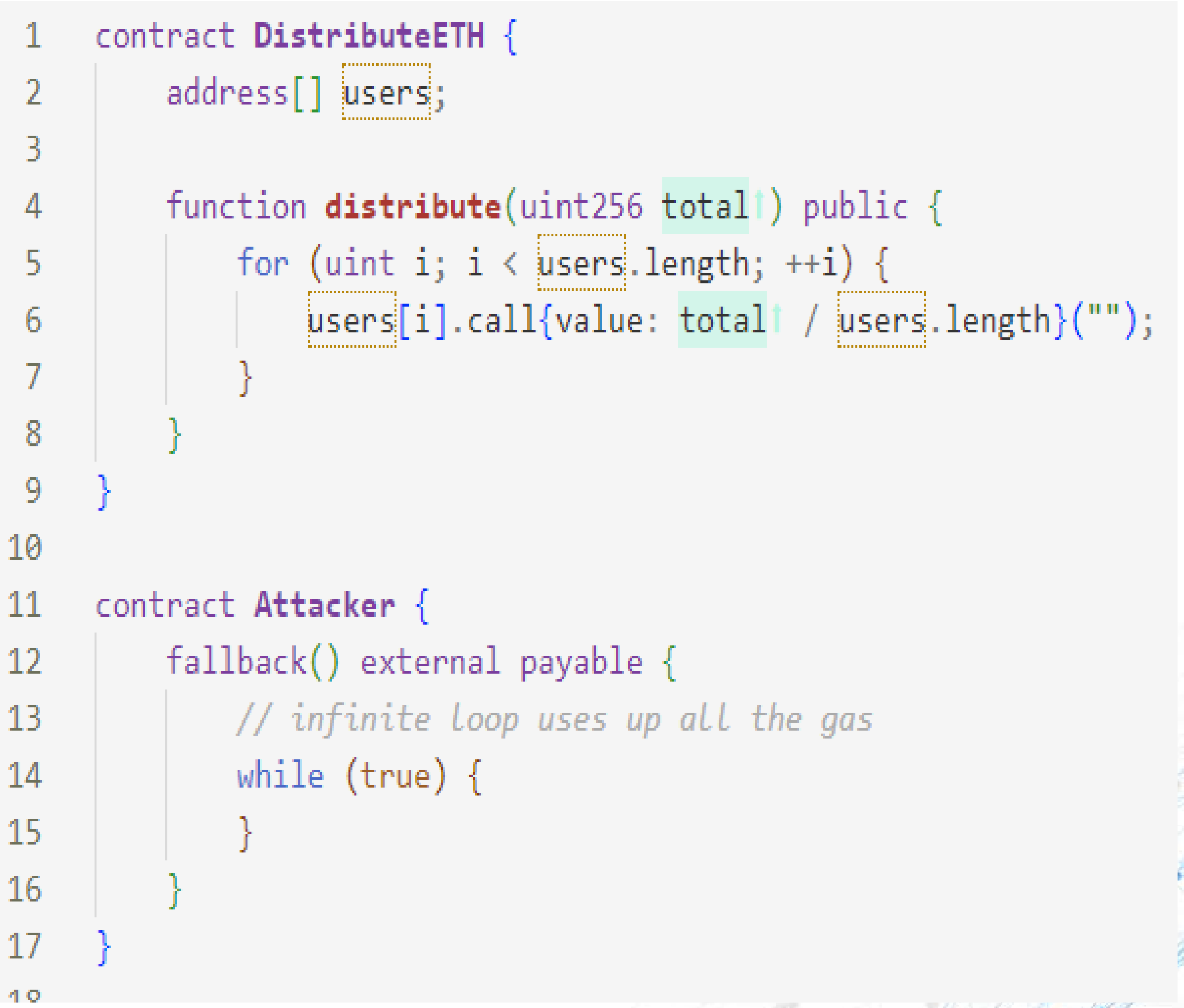
Improper Input Validation



```
contract Auction {
    address public highestBidder;
    uint public highestBid;

    function placeBid() public payable {
        require(msg.value < highestBid);

        highestBidder = msg.sender;
        highestBid = msg.sender;
    }
}
```

Gas Griefing & Denial of Service

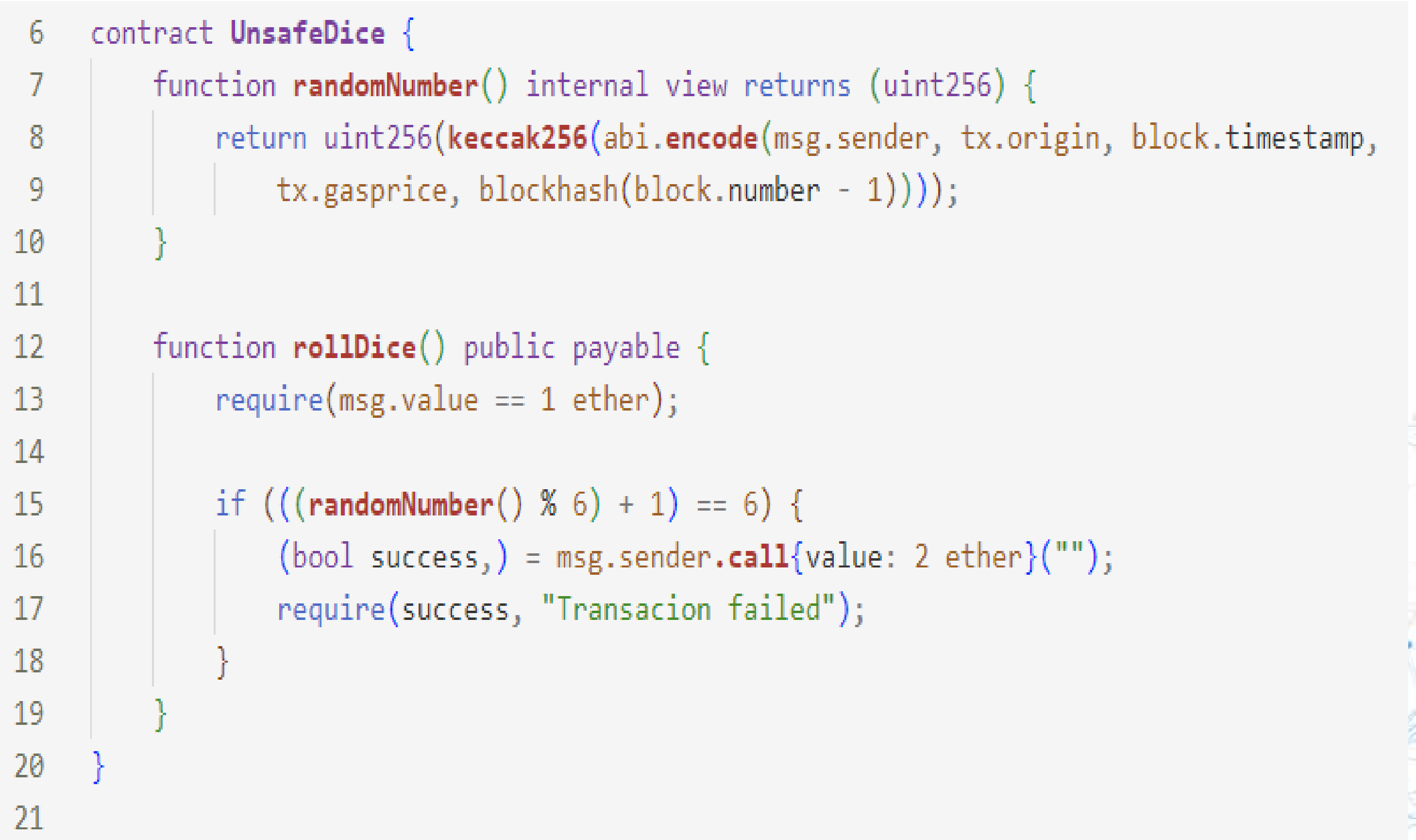
A contract can maliciously consume all the gas by entering an infinite loop



```
contract DistributeETH {
    address[] users;

    function distribute(uint256 total) public {
        for (uint i; i < users.length; ++i) {
            users[i].call{value: total / users.length}("");
        }
    }
}

contract Attacker {
    fallback() external payable {
        // infinite loop uses up all the gas
        while (true) {
        }
    }
}
```

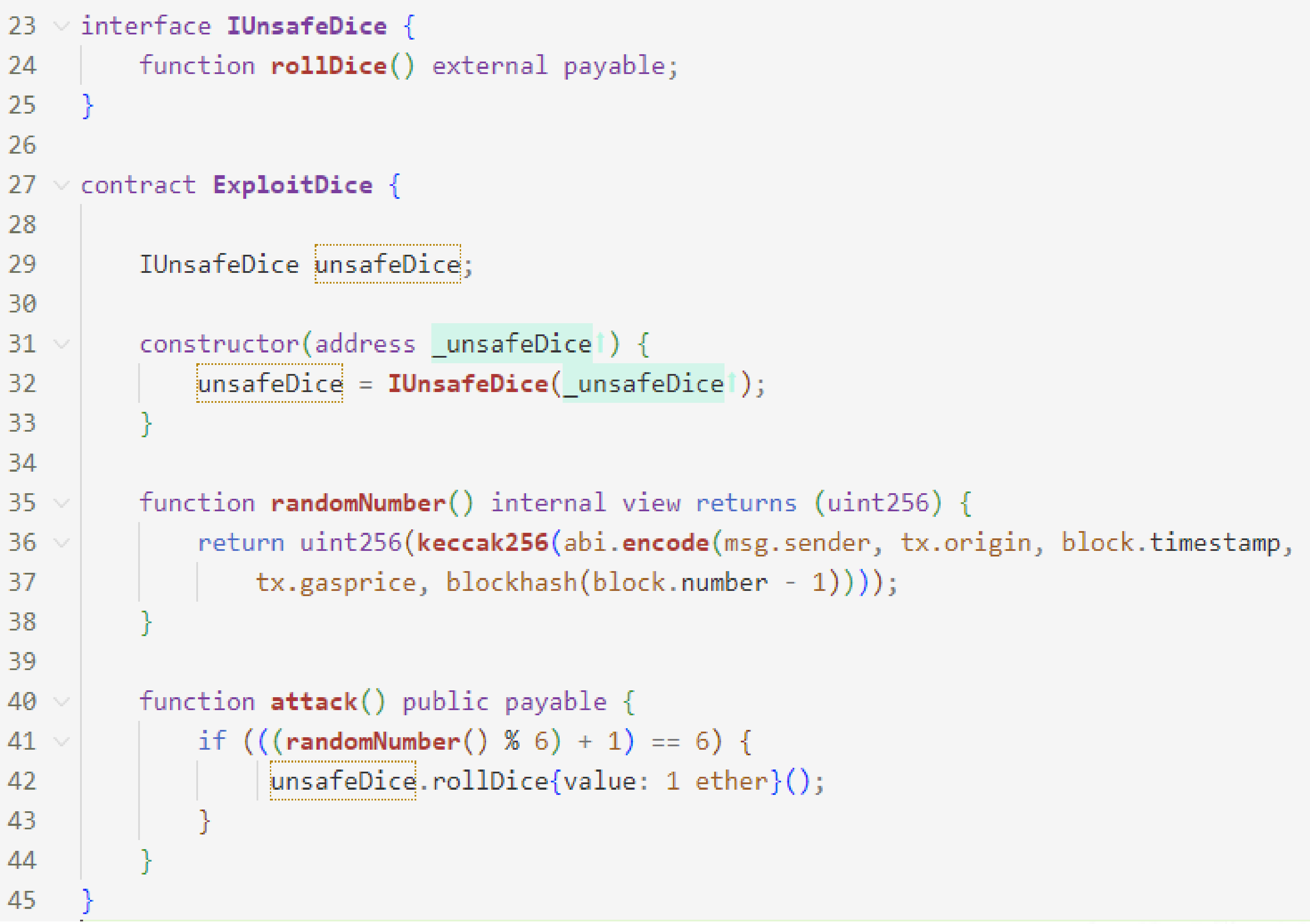
Insecure Randomness

- It is currently not possible to generate randomness securely on the blockchain
- Blockchains must be entirely deterministic, otherwise nodes would not be able to reach consensus about the state
- No matter how randomness is generated, an attacker can always reproduce it



```
contract UnsafeDice {
    function randomNumber() internal view returns (uint256) {
        return uint256(keccak256(abi.encode(msg.sender, tx.origin, block.timestamp,
            tx.gasprice, blockhash(block.number - 1))));
    }

    function rollDice() public payable {
        require(msg.value == 1 ether);

        if (((randomNumber() % 6) + 1) == 6) {
            (bool success,) = msg.sender.call{value: 2 ether}("");
            require(success, "Transacion failed");
        }
    }
}

```

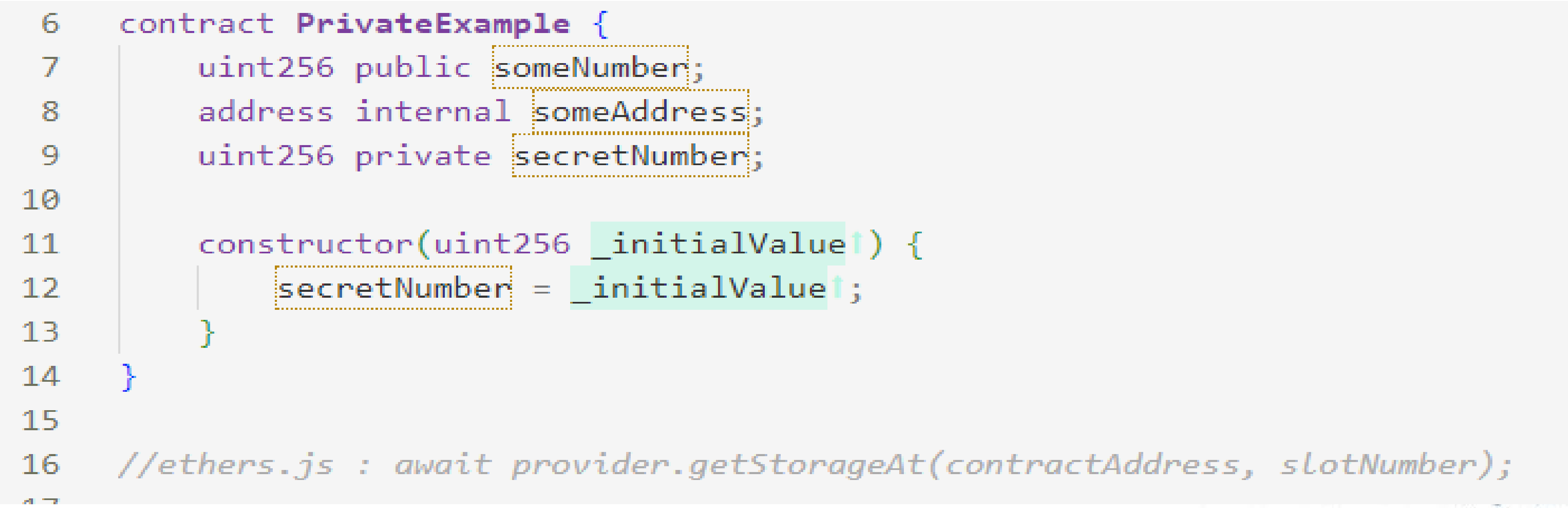
Insecure Randomness



```
interface IUnsafeDice {
    function rollDice() external payable;
}

contract ExploitDice {

    IUnsafeDice unsafeDice;

    constructor(address _unsafeDice) {
        unsafeDice = IUnsafeDice(_unsafeDice);
    }

    function randomNumber() internal view returns (uint256) {
        return uint256(keccak256(abi.encode(msg.sender, tx.origin, block.timestamp,
            tx.gasprice, blockhash(block.number - 1))));
    }

    function attack() public payable {
        if (((randomNumber() % 6) + 1) == 6) {
            unsafeDice.rollDice{value: 1 ether}();
        }
    }
}
```

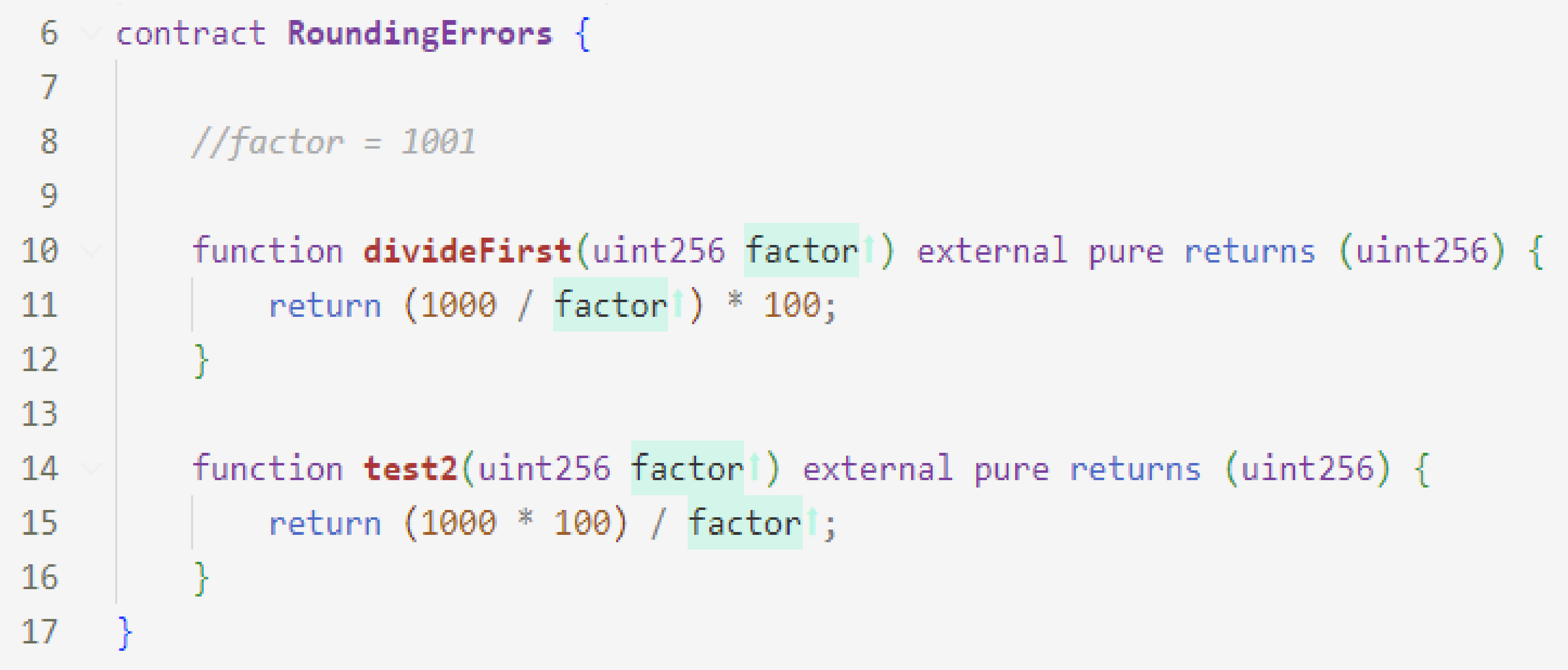
Private Variables

- ➢ Private variables are always visible on the blockchain
- ➢ Never store sensitive information
- ➢ To read a variable, an attacker only needs to know its storage location
- ➢ In the example below, the storage location of 'secretNumber' is 2



```
contract PrivateExample {
    uint256 public someNumber;
    address internal someAddress;
    uint256 private secretNumber;

    constructor(uint256 _initialValue) {
        secretNumber = _initialValue;
    }
}

//ethers.js : await provider.getStorageAt(contractAddress, slotNumber);
```

Rounding Errors - Multiply before Dividing

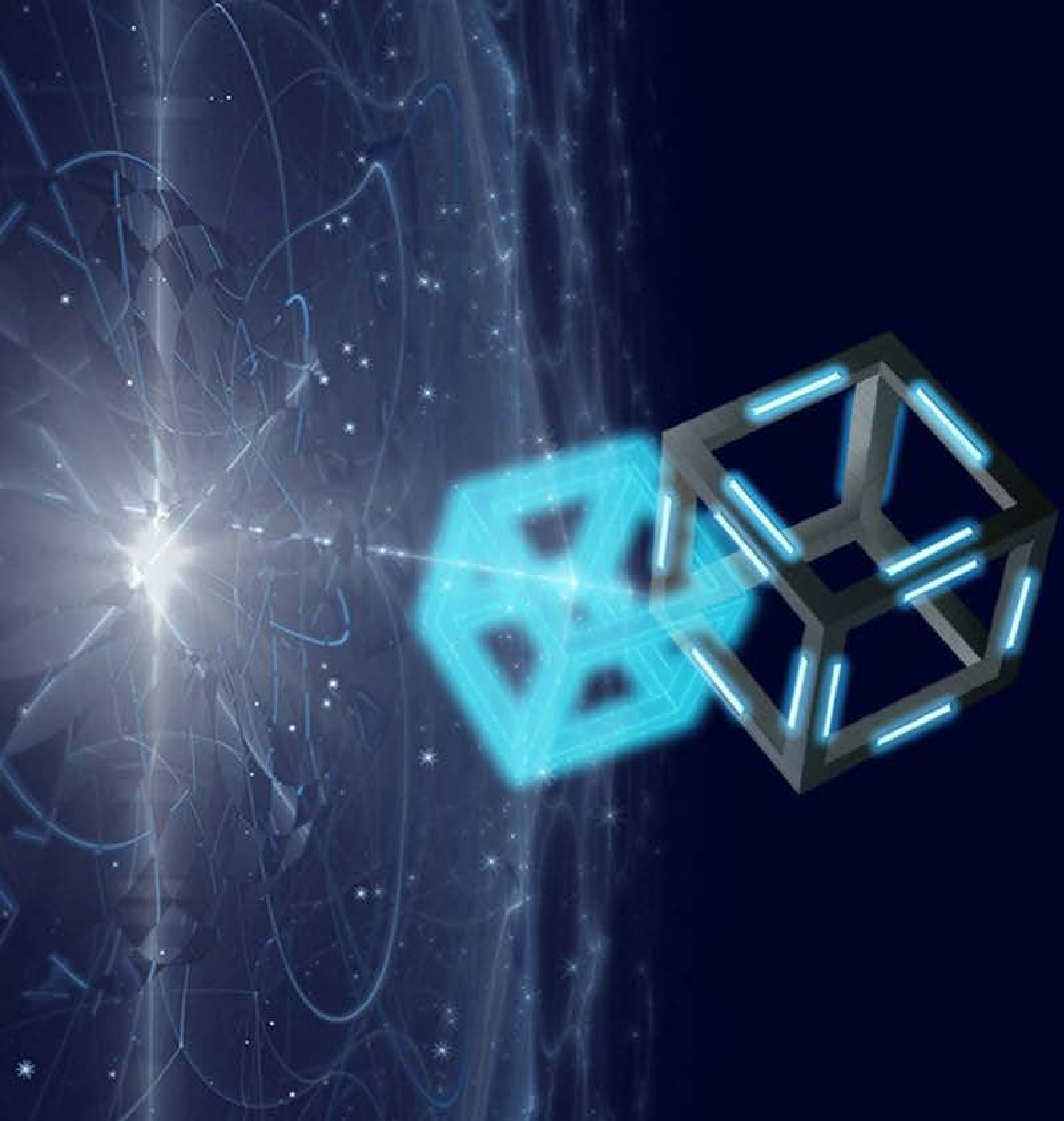


```
contract RoundingErrors {

    //factor = 1001

    function divideFirst(uint256 factor) external pure returns (uint256) {
        return (1000 / factor) * 100;
    }

    function test2(uint256 factor) external pure returns (uint256) {
        return (1000 * 100) / factor;
    }
}
```

Solidity does not have floats, so rounding errors are inevitable. Division should always be performed last.

Additional Resources

- Solidity Smart Contract Attack Vectors : https://github.com/Quillhash/Solidity-Attack-Vectors
- Smart Contract Vulnerabilities : https://github.com/kadenzipfel/smart-contract-vulnerabilities
- Smart Contract Security : https://www.rareskills.io/post/smart-contract-security
- Solidity By Example - Hacks : https://solidity-by-example.org/
- Security Vulnerability Aggregator : https://solodit.xyz/



Slither
Static Analysis

Slither – Trail of Bits

- Open-source static analysis tool
- Specialized in Ethereum smart contract security
- Searches for potential vulnerabilities and bad programming practices in the code
- Provides recommendations to improve code security and quality
- Used by many developers and auditors

- Official Github page: https://github.com/crytic/slither

- **Prerequisites:**
  - - Install Python 3.8+
  - - Install Solc-Select – required if Hardhat, Foundry... is not used

Python 3.8+

- Download Python: https://www.python.org/downloads/
- Install Python: https://www.datacamp.com/blog/how-to-install-python

Solc-Select

- A tool to quickly switch between Solidity compiler versions

- Official Github link: https://github.com/crytic/solc-select

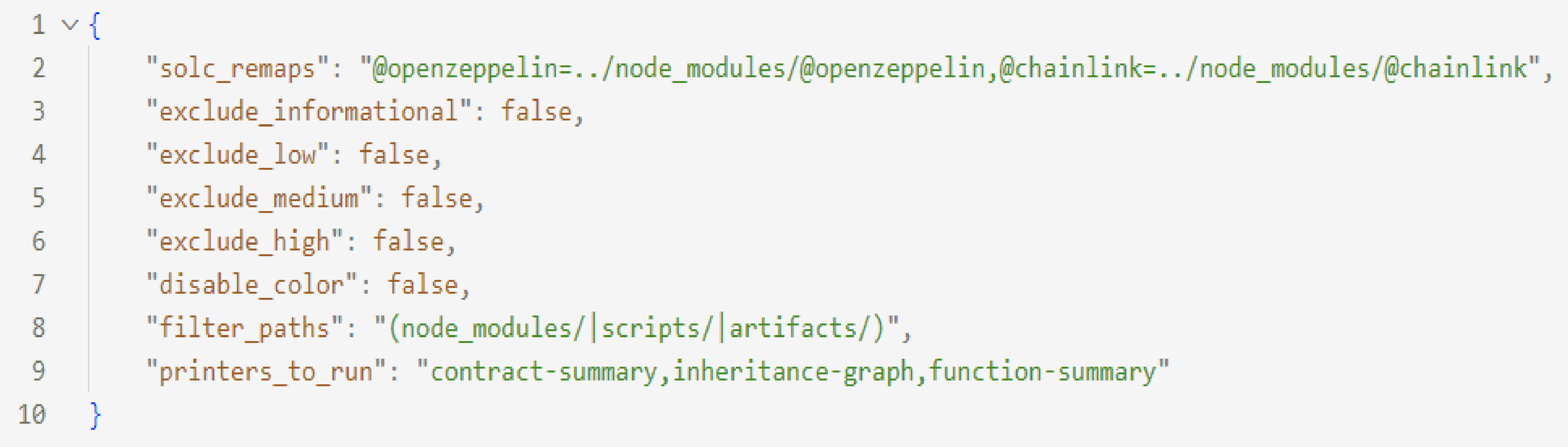
- Installing Solc-Select: ***pip3 install solc-select***

- **Using Solc-Select :**
  - Check the current solc version: ***solc --version***
  - Install a specific solc version: ***solc-select install 0.8.20***
  - Use a specific version: ***solc-select use 0.8.20***

Using Slither

- Installing Slither: ***pip3 install slither-analyzer***

- Add slither.config.json to the project root folder



```json
{
    "solc_remaps": "@openzeppelin=../node_modules/@openzeppelin,@chainlink=../node_modules/@chainlink",
    "exclude_informational": false,
    "exclude_low": false,
    "exclude_medium": false,
    "exclude_high": false,
    "disable_color": false,
    "filter_paths": "(node_modules/|scripts/|artifacts/)",
    "printers_to_run": "contract-summary,inheritance-graph,function-summary"
}
```

- Running Slither : ***slither . or : slither ./contracts/myContract.sol***

Slither Filters

- The results can be filtered:
    - Optimization: --exclude-optimization
    - Informational: --exclude-informational
    - Low findings: --exclude-low

- Using Filters: ***slither . --exclude-informational***

Slither Printers

- By default, no Printers are executed

- Executing Printers:

  slither ./contracts/myContract.sol --print contract-summary,function-summary

- Other useful Printers:
  - inheritance-graph
  - call-graph

- List of all Printers: https://github.com/crytic/slither/wiki/Printer-documentation

Slither Detectors

- By default, all Detectors are executed

- To execute only selected Detectors:
  slither . --detect arbitrary-send,pragma

- To exclude certain Detectors:
  slither . --exclude naming-convention,unused-state

- List of all Detectors: https://github.com/crytic/slither/wiki/Detector-Documentation