Blockchains & Distributed Ledgers

Lecture 04

Aggelos Kiayias

Slide credits: Aggelos Kiayias, Aydin Abadi, Dimitris Karakostas, Christos Nasikas, Dionysis Zindros

Security and fairness of Smart Contracts

Known Attacks

DoS: Unbounded operation

```
// INSECURE
contract NaiveBank {
  struct Account {
      address addr;
      uint balance;
  }

  Account accounts[];
  function applyInterest() returns (uint) {
      for (uint i = 0; i < accounts.length; i++) {
            // apply 5 percent interest
            accounts[i].balance = accounts[i].balance * 105 / 100;
      }
      return accounts.length;
  }

  function openAccount() public returns (uint) { … }
}
```

Source: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., and Smaragdakis Y. OOPSLA ‘18.

DoS: Unbounded operation

```
// INSECURE
contract NaiveBank {
  struct Account {
      address addr;
      uint balance;
  }

  Account accounts[];
  function applyInterest() returns (uint) {
      for (uint i = 0; i < accounts.length; i++) {
            // apply 5 percent interest
            accounts[i].balance = accounts[i].balance * 105 / 100;
      }
      return accounts.length;
  }

  function openAccount() public returns (uint) { … }
}
```

Source: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., and Smaragdakis Y. OOPSLA ‘18.

DoS: Wallet Griefing

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
  if (investors[i].invested == min_investment) {

    if (!(investors[i].addr.send(investors[i].dividendAmount))) {
        revert();
    }

    investors[i] = newInvestor;
  }
}
```

Source: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., and Smaragdakis Y. OOPSLA ‘18.

DoS: Wallet Griefing

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
  if (investors[i].invested == min_investment) {

    if (!(investors[i].addr.send(investors[i].dividendAmount))) {
        revert();
    }

    investors[i] = newInvestor;
  }
}
```

Source: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., and Smaragdakis Y. OOPSLA ‘18.

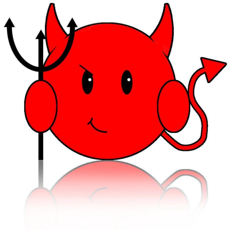
Forcibly Sending Ether to a Contract

- Exploits
  - **misuse** of `this.balance`
  - `Complicated (vulnerable) fallback function`
- How can you **send ether** to a contract **without** firing contact’s **fallback** function ?
  - `selfdestruct(victim)`
  - Contract’s address = hash(sender address, nonce)
  - Anyone can **calculate** a contract’s address **before** it is **created** (contract addresses generation is **deterministic**) and send ether to that address.

Reentrancy

Fallback function

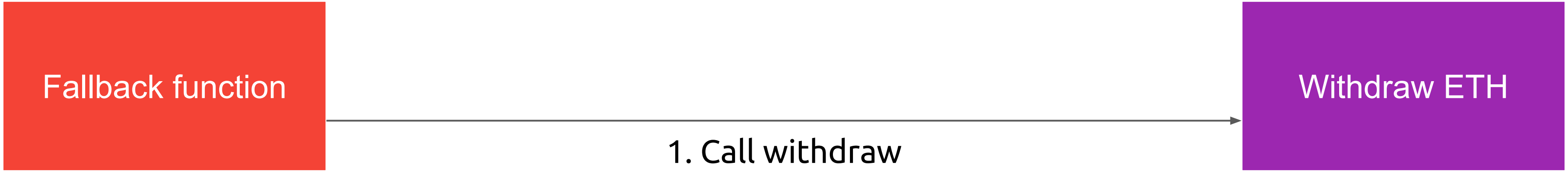
Contract A



Withdraw ETH

Contract B

Reentrancy



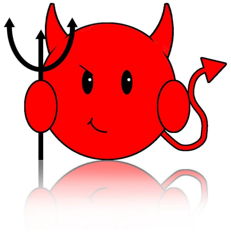
Fallback function

Withdraw ETH

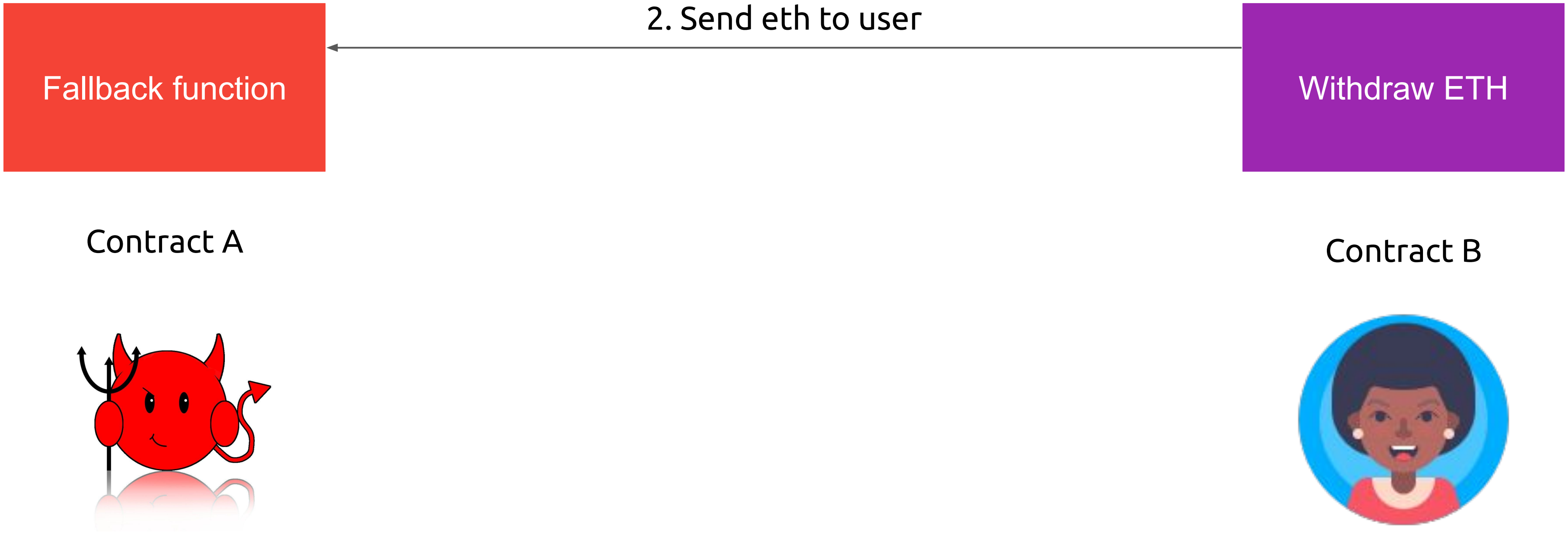
1. Call withdraw

Contract A

Contract B



Reentrancy



2. Send eth to user

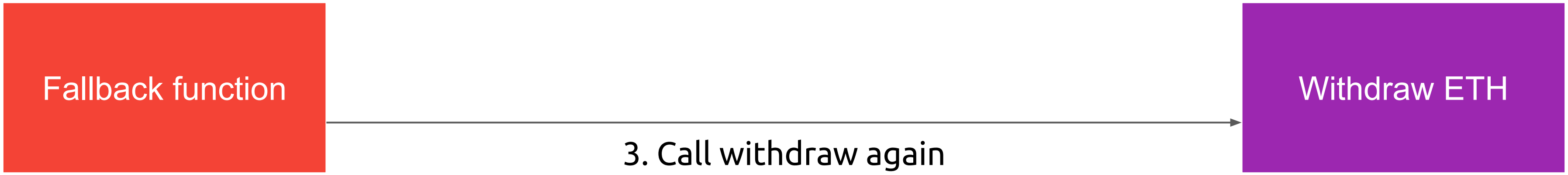
Fallback function

Withdraw ETH

Contract A

Contract B

Reentrancy



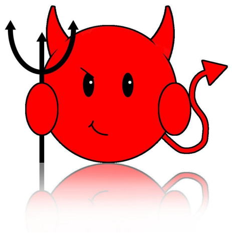
Fallback function

Withdraw ETH

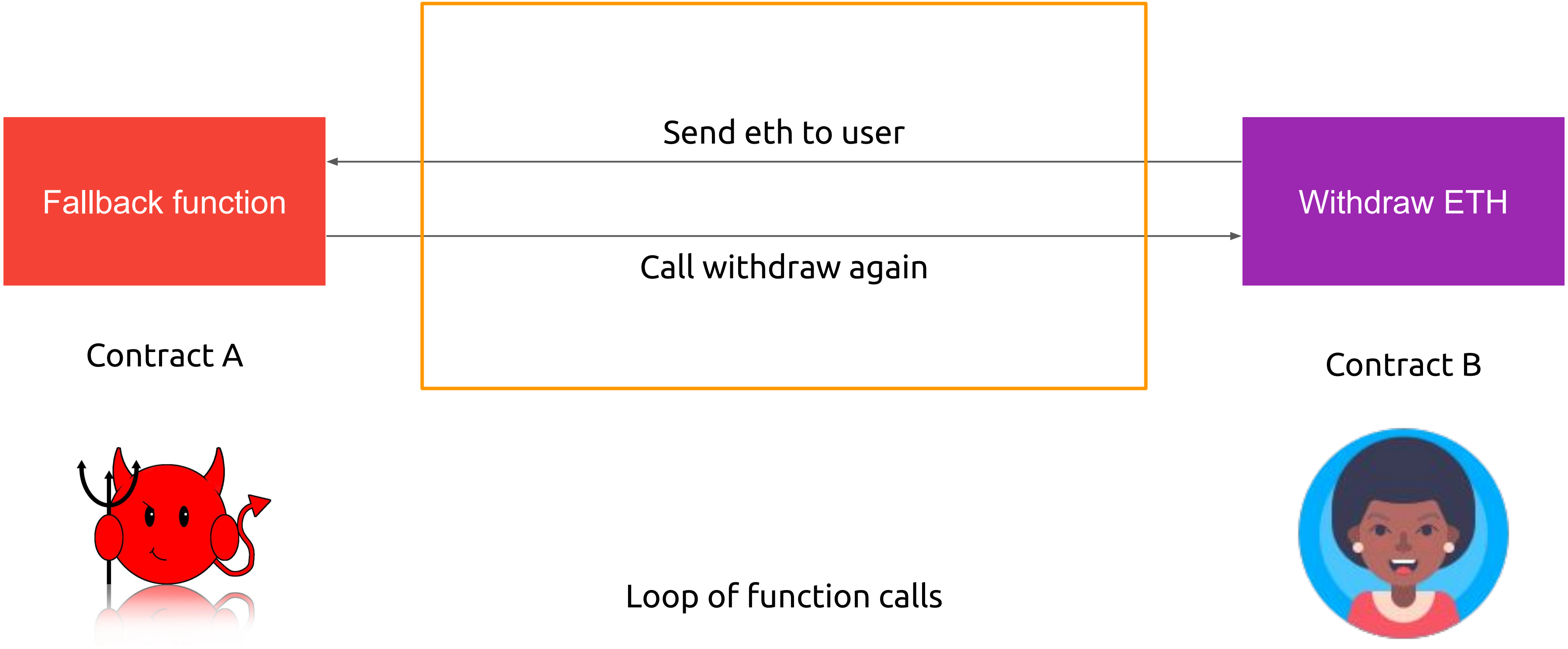
3. Call withdraw again

Contract A

Contract B



Reentrancy



Fallback function

Send eth to user

Withdraw ETH

Call withdraw again

Contract A

Contract B

Loop of function calls

Reentrancy

```
// INSECURE
mapping (address => uint) private userBalances;
function withdrawBalance() public {
      uint amountToWithdraw = userBalances[msg.sender];
      require(msg.sender.call.value(amountToWithdraw)());
      userBalances[msg.sender] = 0;
}
```

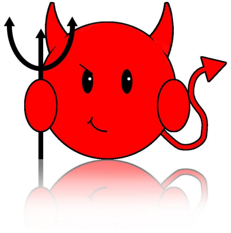
Reentrancy

```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

    uint amountToWithdraw = userBalances[msg.sender];

    require(msg.sender.call.value(amountToWithdraw)());

    userBalances[msg.sender] = 0;

}
```

Reentrancy

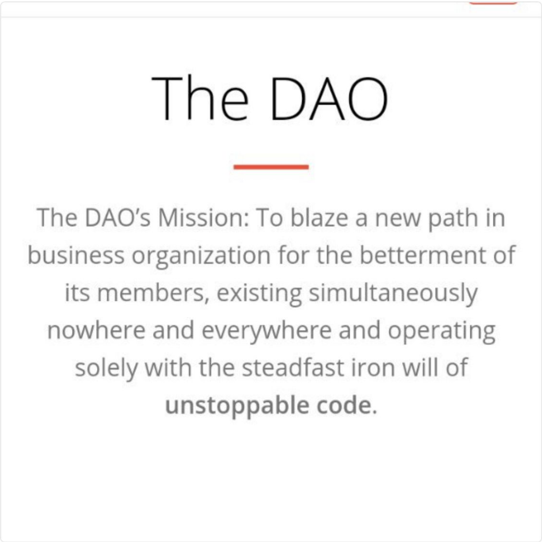
```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

    uint amountToWithdraw = userBalances[msg.sender];

    require(msg.sender.call.value(amountToWithdraw)());

    userBalances[msg.sender] = 0;

}
```

```
function () payable {

    if (victim.balance >= msg.value) {

        victim.withdrawBalance();

    }

  }
```



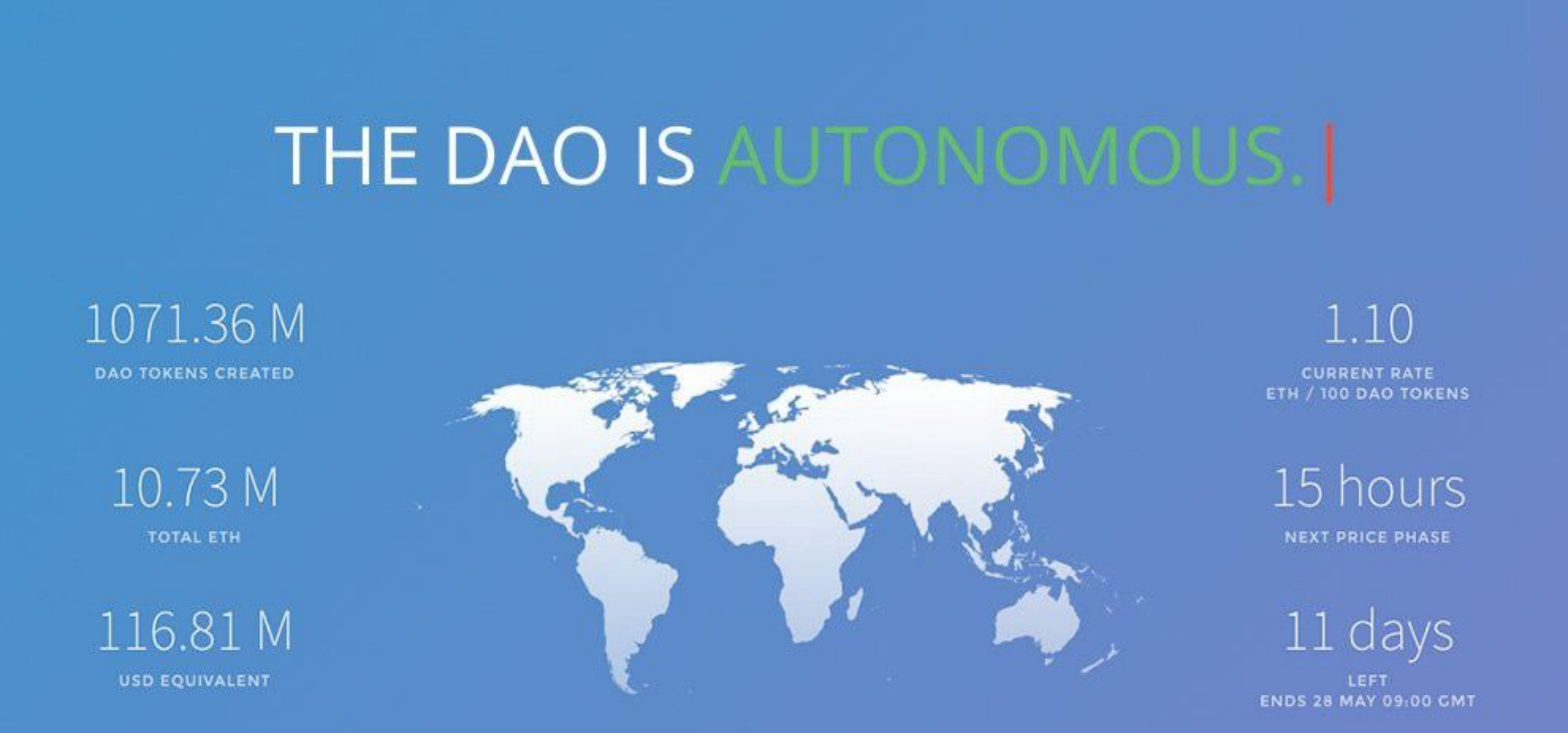
Re-entrancy in the wild: The DAO

- The DAO (distributed autonomous organization)
  - Designed by slock.it in 2016
  - Purpose: Create a population of stakeholders
  - Stake (in the form of DAO tokens) enables them to participate in decision making
  - Decision-making to choose which proposals to fund



The DAO

The DAO's Mission: To blaze a new path in business organization for the betterment of its members, existing simultaneously nowhere and everywhere and operating solely with the steadfast iron will of **unstoppable code.**



THE DAO IS AUTONOMOUS.

1071.36 M
DAO TOKENS CREATED

10.73 M
TOTAL ETH

116.81 M
USD EQUIVALENT

1.10
CURRENT RATE
ETH / 100 DAO TOKENS

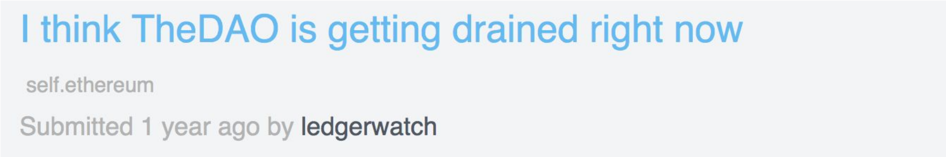
15 hours
NEXT PRICE PHASE

11 days
LEFT
ENDS 28 MAY 09:00 GMT

~150 million USD in ~ 1 month

The DAO Attack

- June 12: The reentrancy bug is identified (but stakeholders are reassured)
- June 17: Attacker exploits it draining ~$50Million at the time of the attack
- July 15: Ethereum Classic manifesto
- July 19: “Hard Fork” neutralizes attacker’s smart contract



I think TheDAO is getting drained right now

self.ethereum

Submitted 1 year ago by ledgerwatch

Reentrancy: solutions

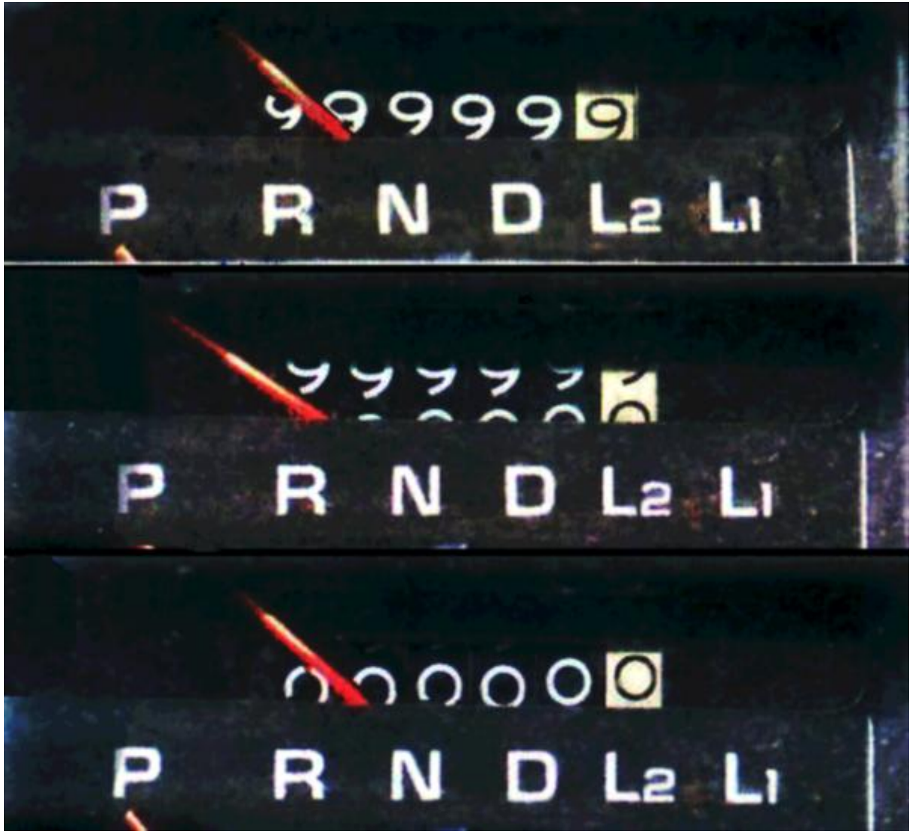
```
// SECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

      userBalances[msg.sender] = 0;

      msg.transfer(amountToWithdraw);

}
```

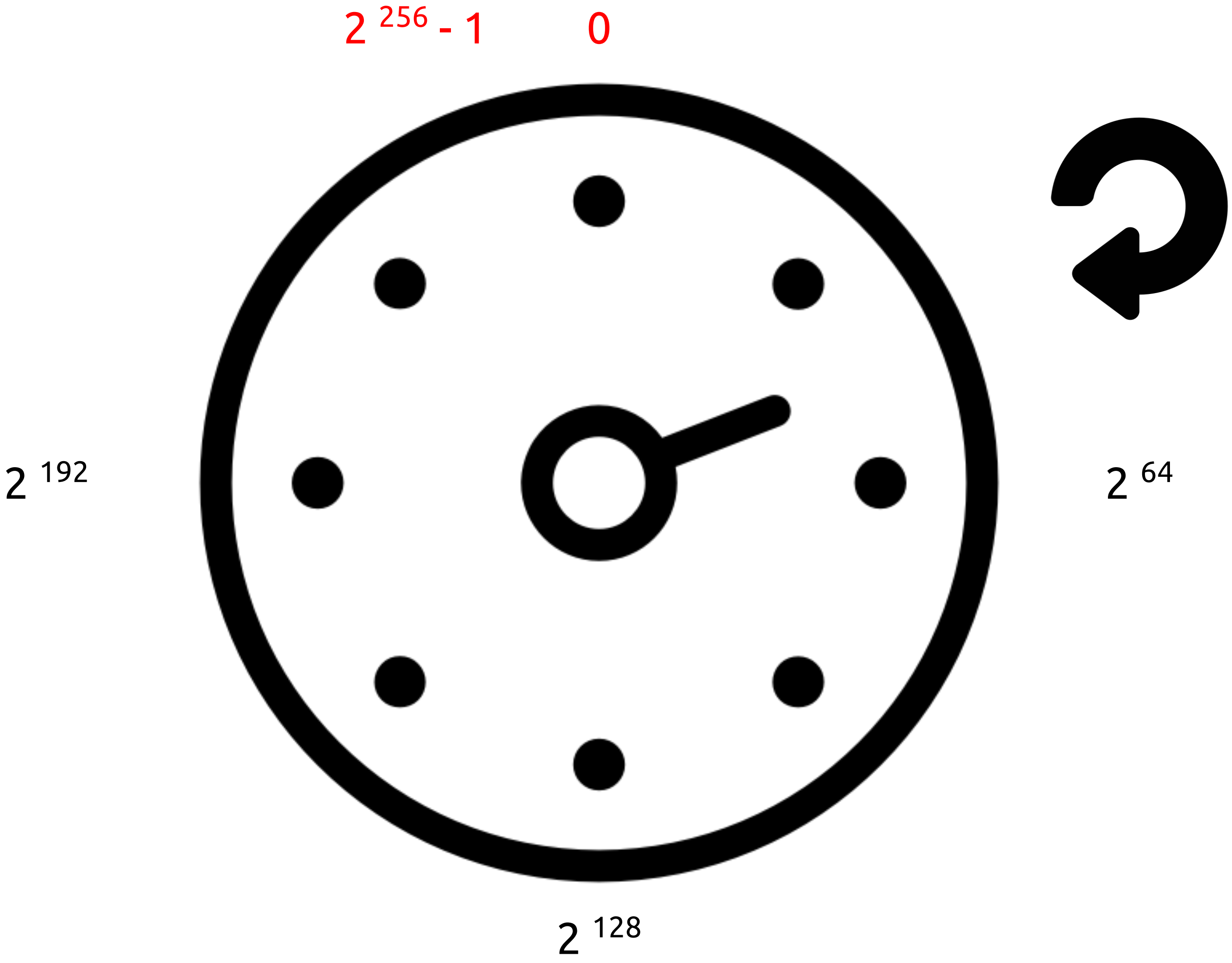
- Use `transfer` or `send` instead of `call`
- Finish all internal work (ie. state changes) and then call external functions
- Checks-Effects-Interactions Pattern
- Mutexes
- Pull-push pattern

Checks-Effects-Interactions Pattern

1. **Perform checks** (e.g sender, value, arguments ect)
2. Update **state**
3. **Interact** with other **contracts** (external function calls or send ether)

Integer Overflow and Underflow





$2^{256} - 1$ 0

2^{192} 2^{64}

2^{128}

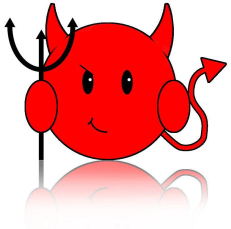
Integer Overflow and Underflow

```
// INSECURE

function withdraw(uint256 _value) {

    require(balanceOf[msg.sender] >= _value);

    msg.sender.call.value(_value)();

    balanceOf[msg.sender] -= _value;

}
```

Integer Overflow and Underflow

```
// INSECURE
function withdraw(uint256 _value) {
    require(balanceOf[msg.sender] >= _value);
    msg.sender.call.value(_value)();
    balanceOf[msg.sender] -= _value;
}
```

Integer Overflow and Underflow



```
// INSECURE

function withdraw(uint256 _value) {

      require(balanceOf[msg.sender] >= _value);

      msg.sender.call.value(_value)();

      balanceOf[msg.sender] -= _value;

}
```

```
function attack() {

      victim.donate.value(1)();

      victim.withdraw(1);

}

function() {

      if (performAttack) {

            performAttack = false;

            victim.withdraw(1);

      }

 }
```

Integer Overflow and Underflow: solutions

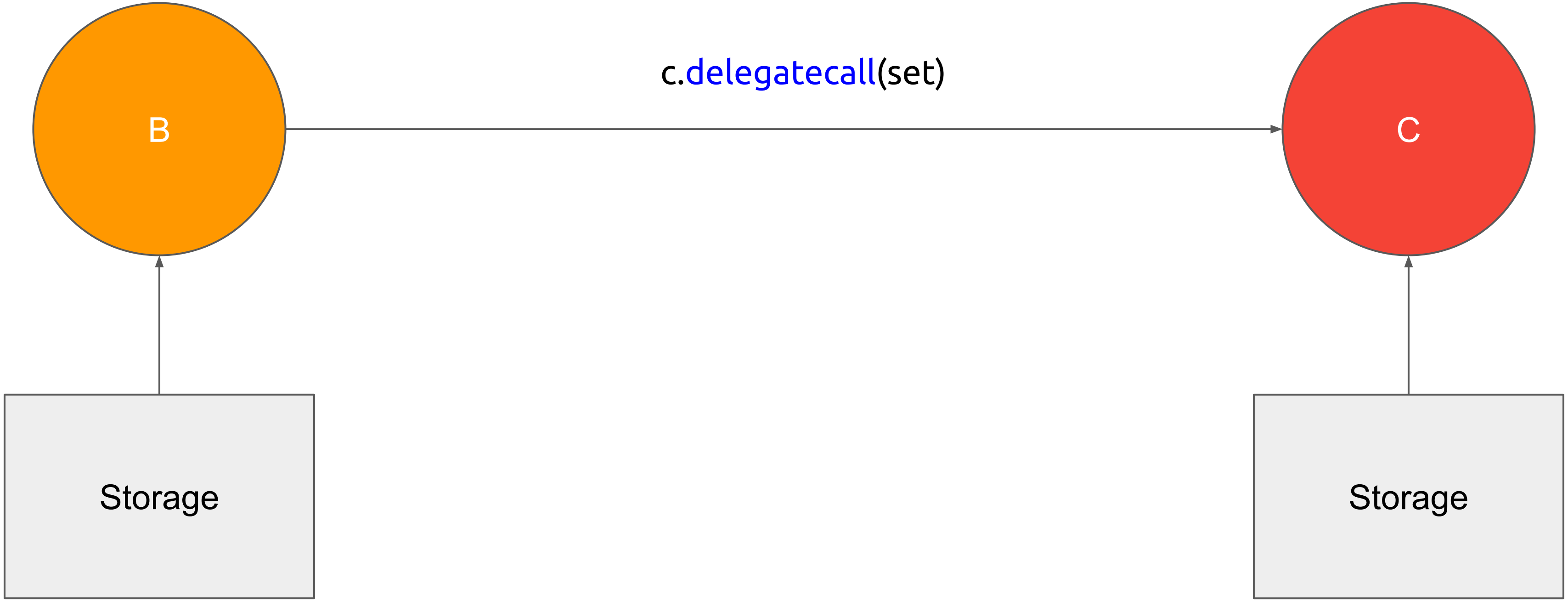
Use OpenZeppelin’s SafeMath library

```
// OpenZeppelin: SafeMath.sol

function add(uint256 a, uint256 b) internal pure returns
(uint256) {
      uint256 c = a + b;
      require(c >= a, "SafeMath: addition overflow");

      return c;
}

function sub(uint256 a, uint256 b) internal pure returns
(uint256) {
      require(b <= a, "SafeMath: subtraction overflow");
      uint256 c = a - b;

      return c;
}
```

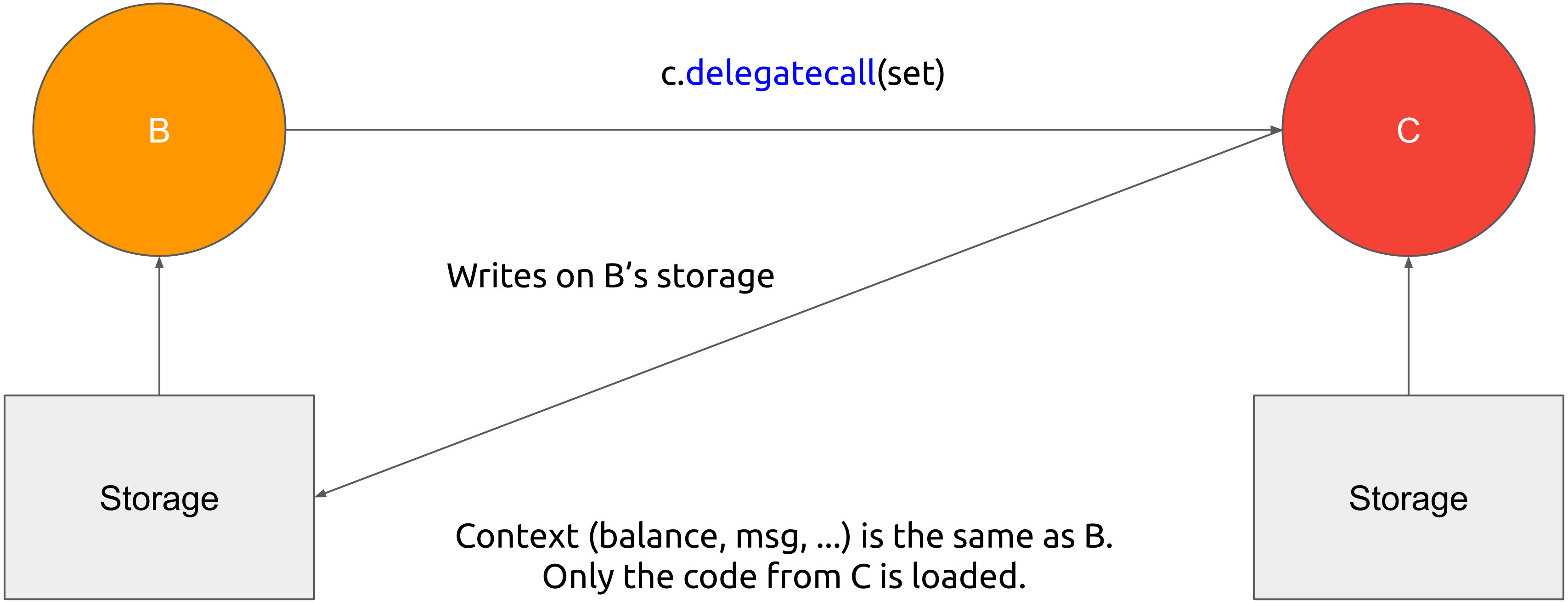
Delegate call

B
C
Storage
Storage

Delegate call

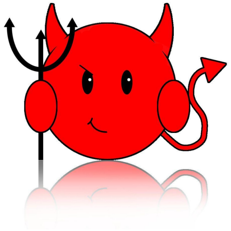


Delegate call



Context (balance, msg, ...) is the same as B.
Only the code from C is loaded.

Delegate call

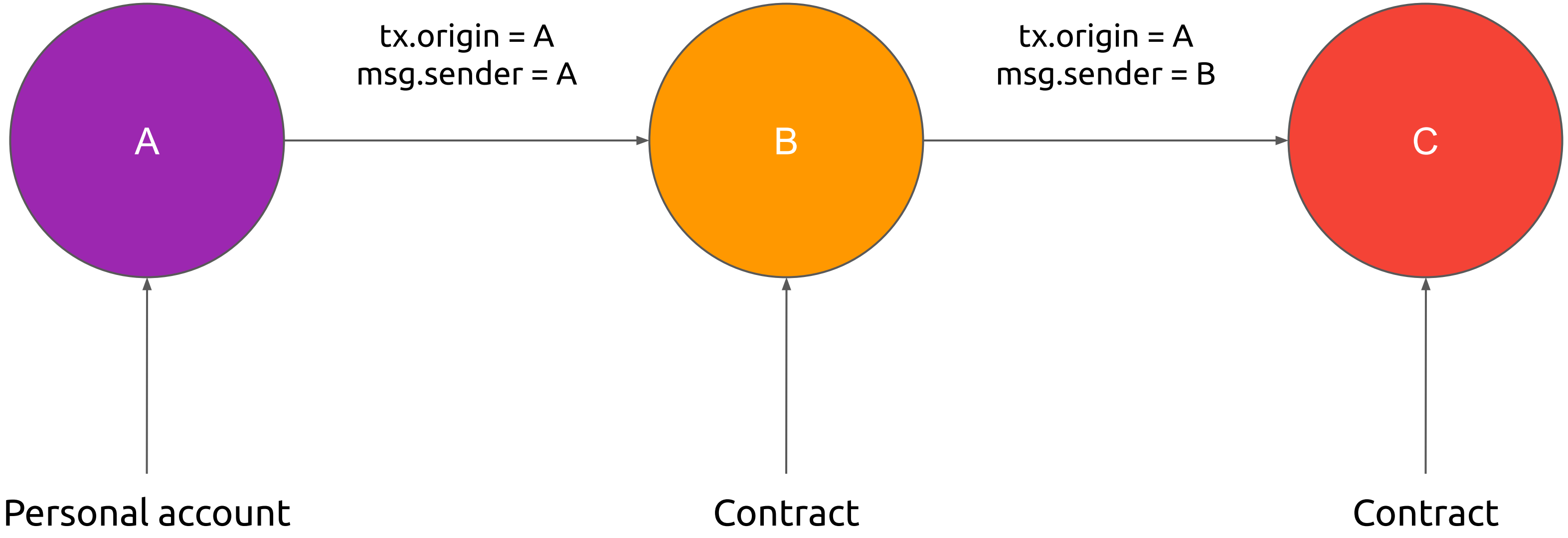


```
// INSECURE
address public owner;

Library library =


function() public {
      require(library.delegatecall(msg.data));
}
```

```
  address public owner;

  constructor (address _owner) public {
    owner = _owner;
  }

  function pwn() public {
    owner = msg.sender;
  }
}
```

Source: https://ethernaut.zeppelin.solutions

Use of tx.origin



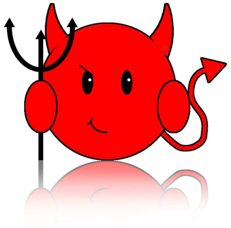
Use of tx.origin

```
// INSECURE
contract Bank {

    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.call.value(amount)();
    }

}
```

Use of tx.origin

```
// INSECURE
contract Bank {

    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.call.value(amount)();
    }

}
```

Use of tx.origin



```
// INSECURE
contract Bank {

    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function sendTo(address payable receiver, uint amount)
public {
        require(tx.origin == owner);
        receiver.call.value(amount)();
    }

}
```

```
function() external payable {

   victim.sendTo(attacker,msg.sender.balance);

}
```

Pull over push

- **Do not transfer ether** to **users** (push) but **let** the **users withdraw** (pull) their funds.
- **Isolates** each **external call** into its own transaction.
- **Avoids** multiple `send()` calls in a single transaction.
- **Reduces** problems with **gas limits**.
- **Trade-off** between **security** and **user experience**.

Pull over push: example

```
// INSECURE

function bid() payable {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
      }

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

```
// SECURE

function bid() payable external {
      require(msg.value >= highestBid);

      if (highestBidder != address(0)) {
            refunds[highestBidder] += highestBid;
      }

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

function withdrawRefund() external {
      uint refund = refunds[msg.sender];
      refunds[msg.sender] = 0;
      msg.sender.transfer(refund);
}
```

Source: https://consensys.github.io/smart-contract-best-practices

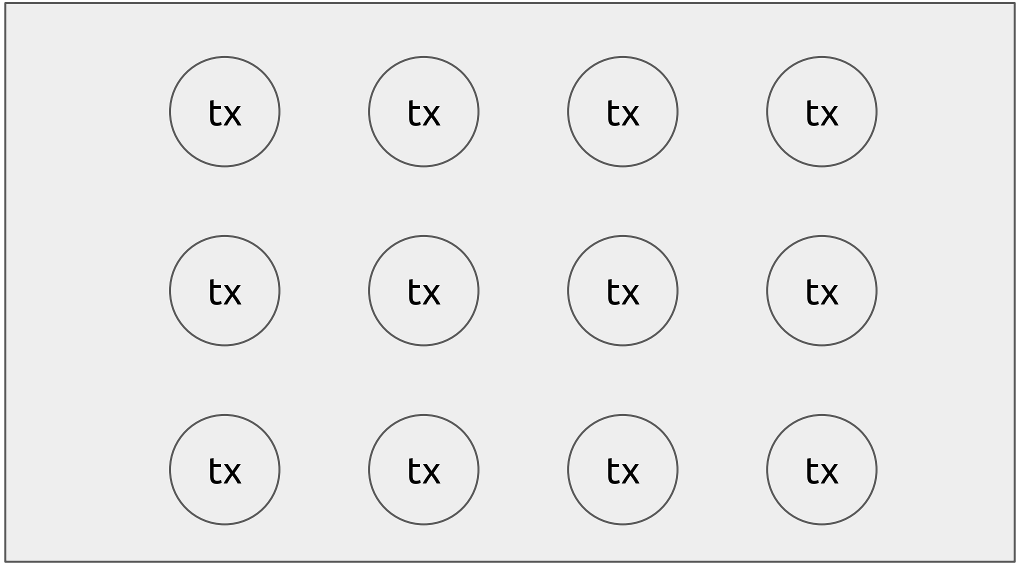
Keep fallback function simple

```
// BAD

function() payable {
      balances[msg.sender] += msg.value;
}
```

```
// GOOD

function deposit() payable external {
      balances[msg.sender] += msg.value;
}

function() payable {
      require(msg.data.length == 0);
      emit LogDepositReceived(msg.sender);
}
```

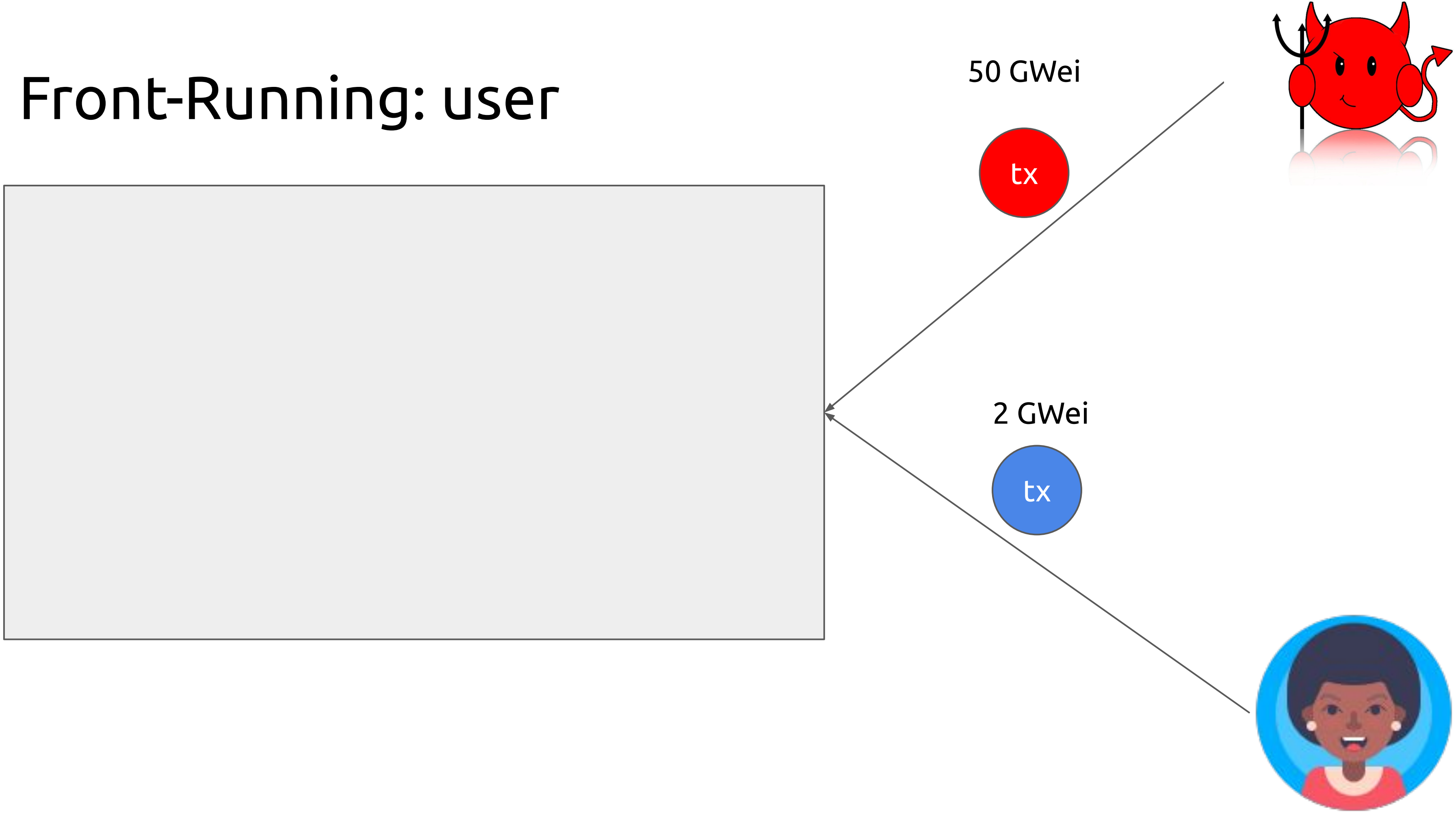
Source: https://consensys.github.io/smart-contract-best-practices

Front-Running



```
sortByGasPrice(txs, ‘desc’)
```

Front-Running: user



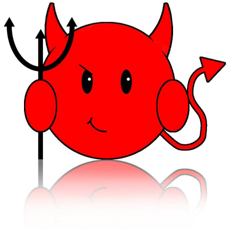
50 GWei

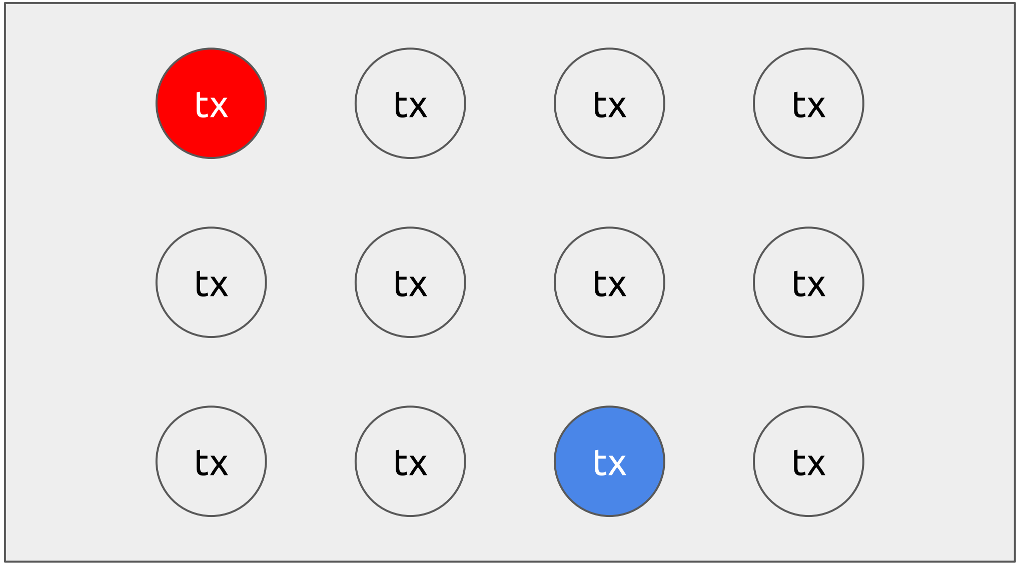
tx

2 GWei

tx

Front-Running: user



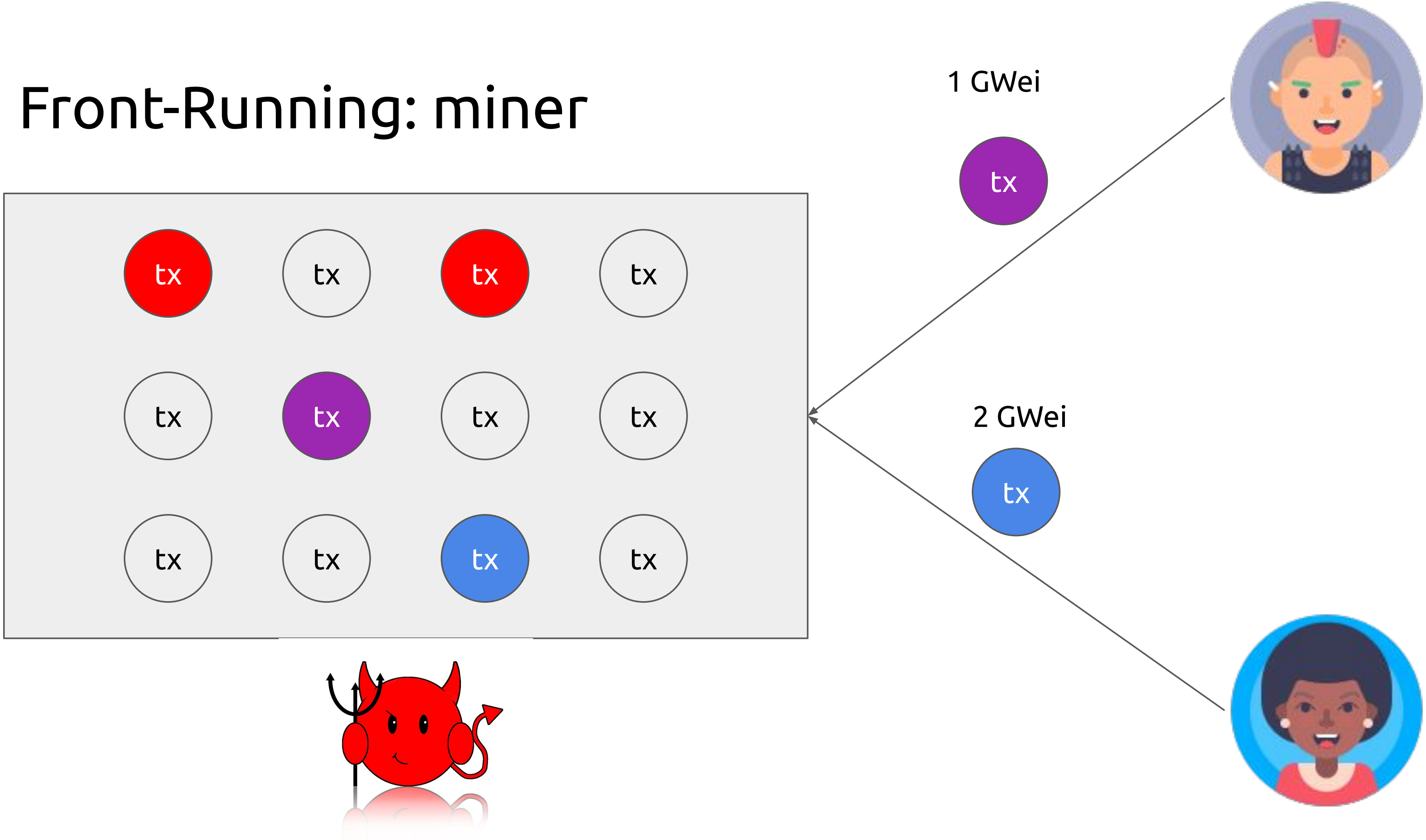


tx tx tx tx

tx tx tx tx

tx tx tx tx

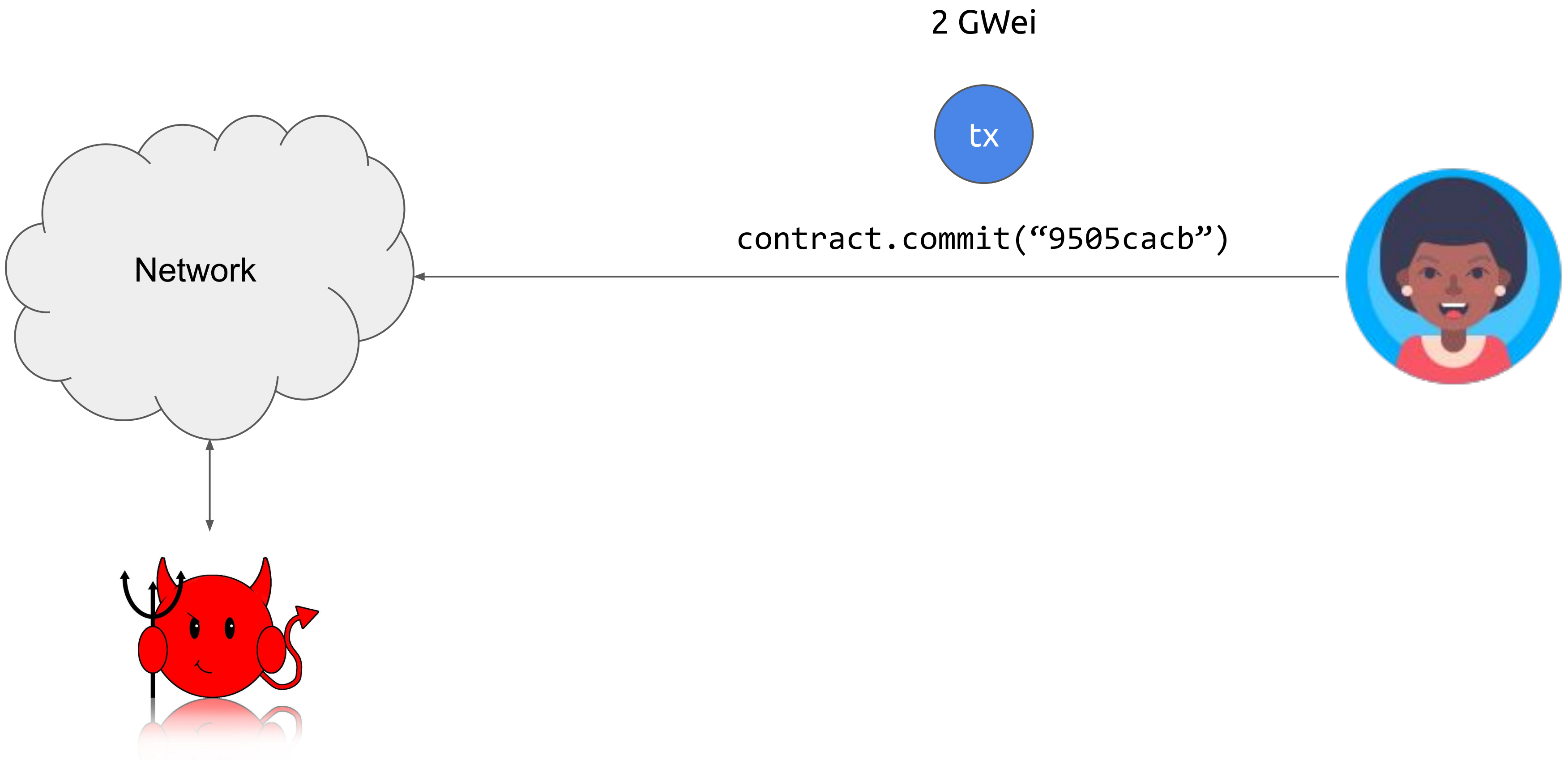
Front-Running: miner



Front-Running: example

```
// INSECURE
function commit(bytes32 commitment) public {
    commitments[commitment] = msg.sender;
}

function registerName(bytes32 name, bytes32 nonce) public {
    require(commitments[makeCommitment(name, nonce)] == msg.sender, “Not found!”);
    names[name] = msg.sender;
}
```

Front-Running: example



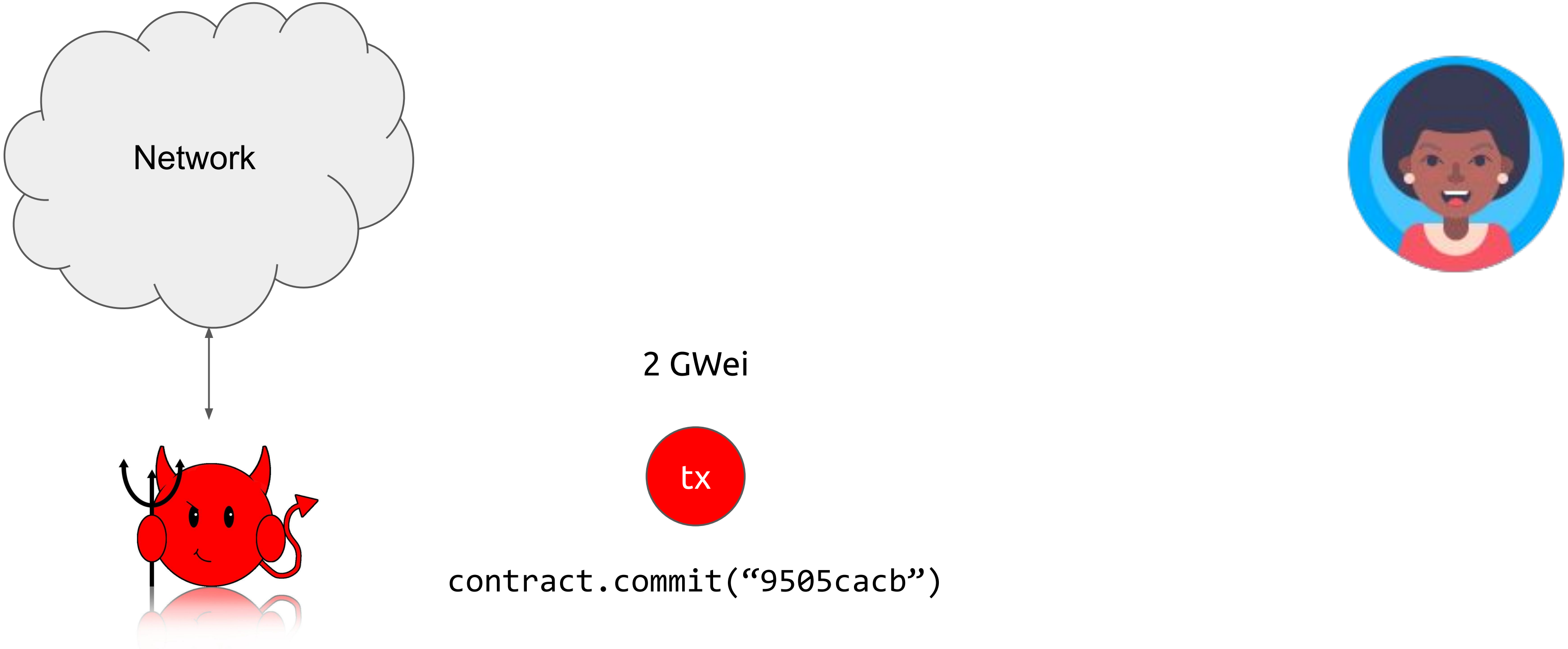
2 GWei

tx

```
contract.commit(“9505cacb”)
```

Network

Front-Running: example



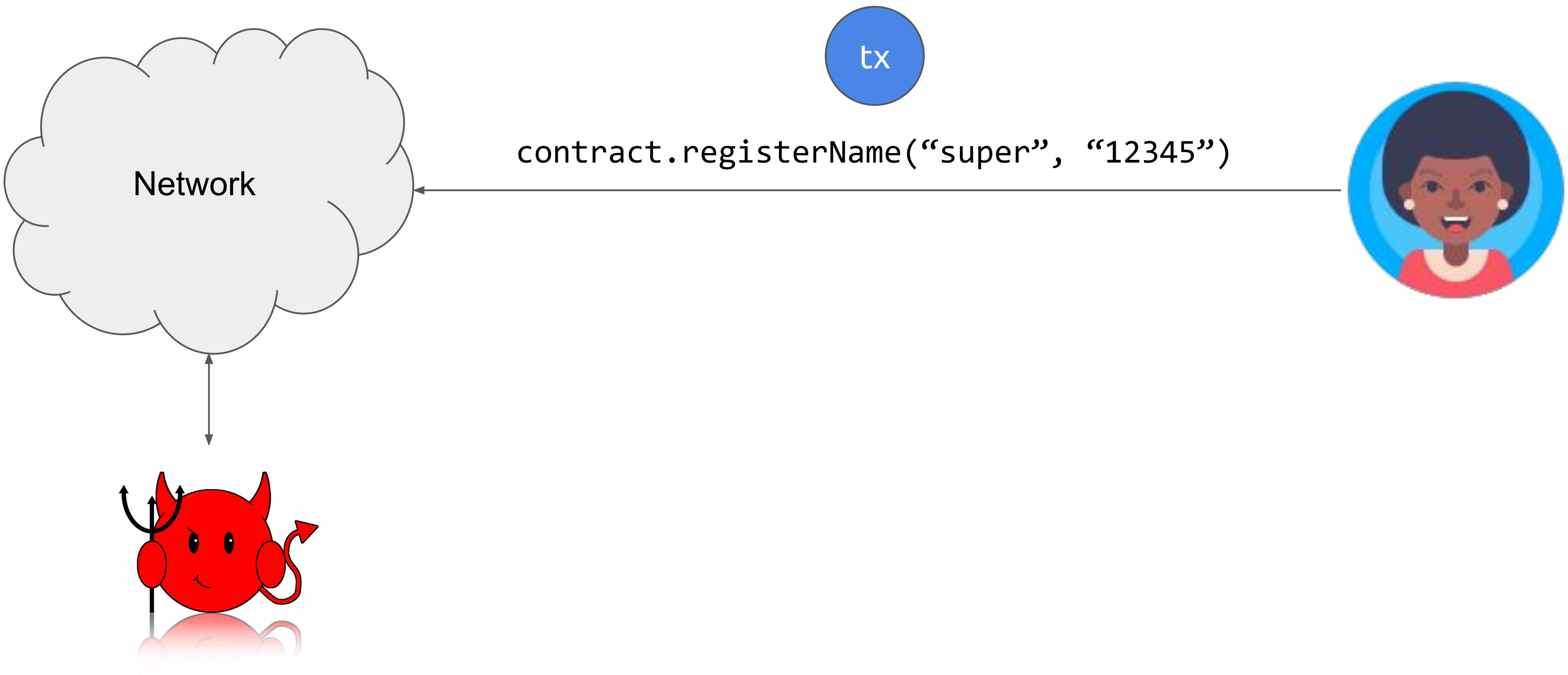
Network

2 GWei

tx

```
contract.commit(“9505cacb”)
```

Front-Running: example



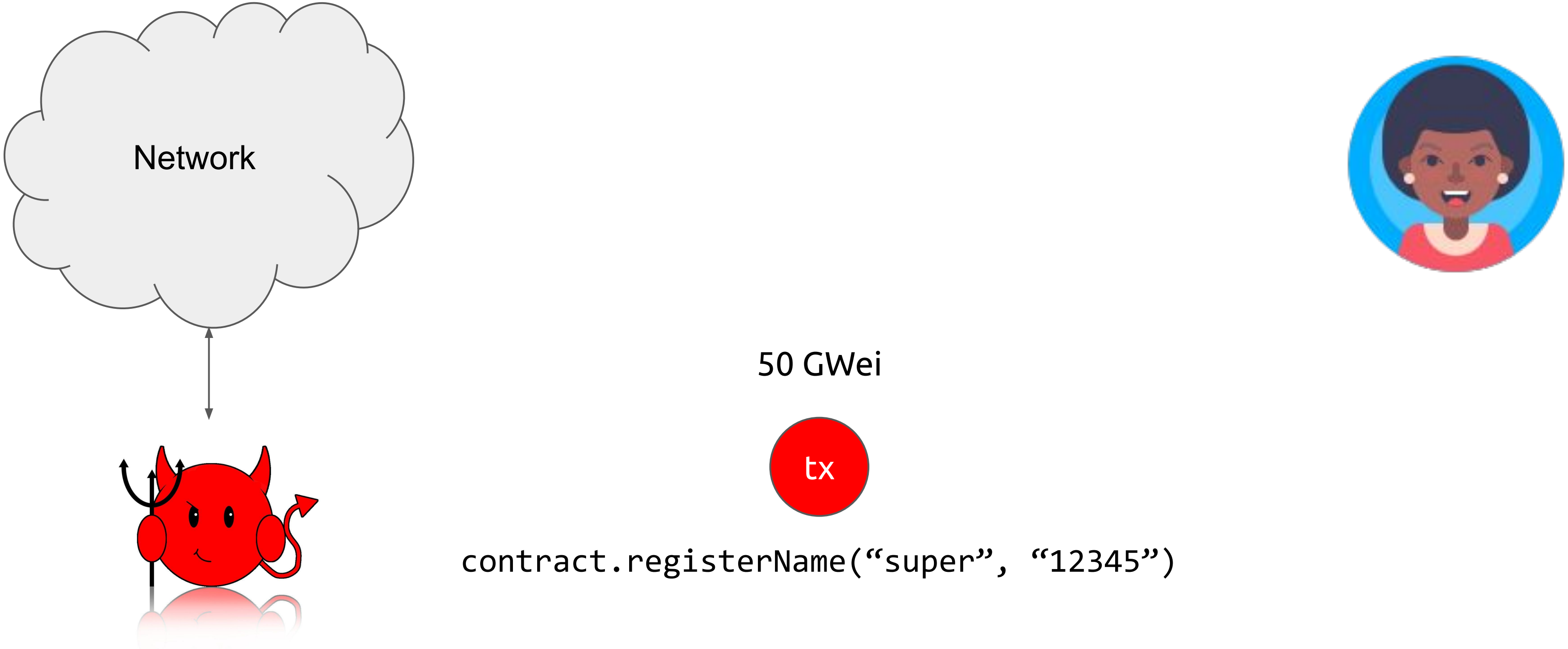
2 GWei

tx

```
contract.registerName(“super”, “12345”)
```

Network

Front-Running: example



Network

50 GWei

tx

```
contract.registerName(“super”, “12345”)
```

Randomness

Randomness: sources (?)

- block.number
- block.timestamp
- block.hash
- block.difficulty
- block.coinbase
- block.gasLimit
- now
- msg.sender

uint(keccak256(

timestamp	msg.sender	hash	...

)) % n

Randomness: sources (?)

- block.number
- block.timestamp
- block.hash
- block.difficulty
- block.coinbase
- block.gasLimit
- now
- msg.sender

They can be manipulated by a malicious miner.
They are shared within the same block to all users.

Randomness

```
// INSECURE
bool won = (block.number % 2) == 0;
```

```
// INSECURE
uint random = uint(keccak256(block.timestamp)) % 2;
```

```
// INSECURE
address seed1 = contestants[uint(block.coinbase) % totalTickets].addr;
address seed2 = contestants[uint(msg.sender) % totalTickets].addr;
uint seed3 = block.difficulty;
bytes32 randHash = keccak256(seed1, seed2, seed3);
uint winningNumber = uint(randHash) % totalTickets;
address winningAddress = contestants[winningNumber].addr;
```

Randomness: blockhash

Not that private :)

```
// INSECURE

uint256 private _seed;

function random(uint64 upper) public returns (uint64 randomNumber) {
        _seed = uint64(keccack256(keccack256(block.blockhash(block.number), _seed), now));
        return _seed % upper;
}
```

Randomness: blockhash

Not that private :)

```
// INSECURE

uint256 constant private FACTOR =
1157920892373161954235709850086879078532699846656405640394575840079131296399;

function rand(uint max) constant private returns (uint256 result) {
        uint256 factor = FACTOR * 100 / max;
        uint256 lastBlockNumber = block.number - 1;
        uint256 hashVal = uint256(block.blockhash(lastBlockNumber));
        return uint256((uint256(hashVal) / factor)) % max;
}
```

Randomness: attack pattern

```
if (replicatedVictimConditionOutcome() == favorable)
    victim.tryMyLuck();
```

Source: Bad Randomness Is Even Dicier than You Think. Yannis Smaragdakis

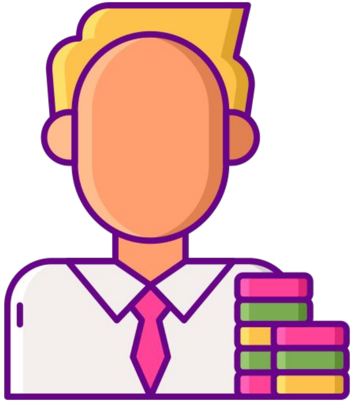
Randomness: intra-transaction information leak

```
victim.tryMyLuck();
require(victim.conditionOutcome() == favorable);
```

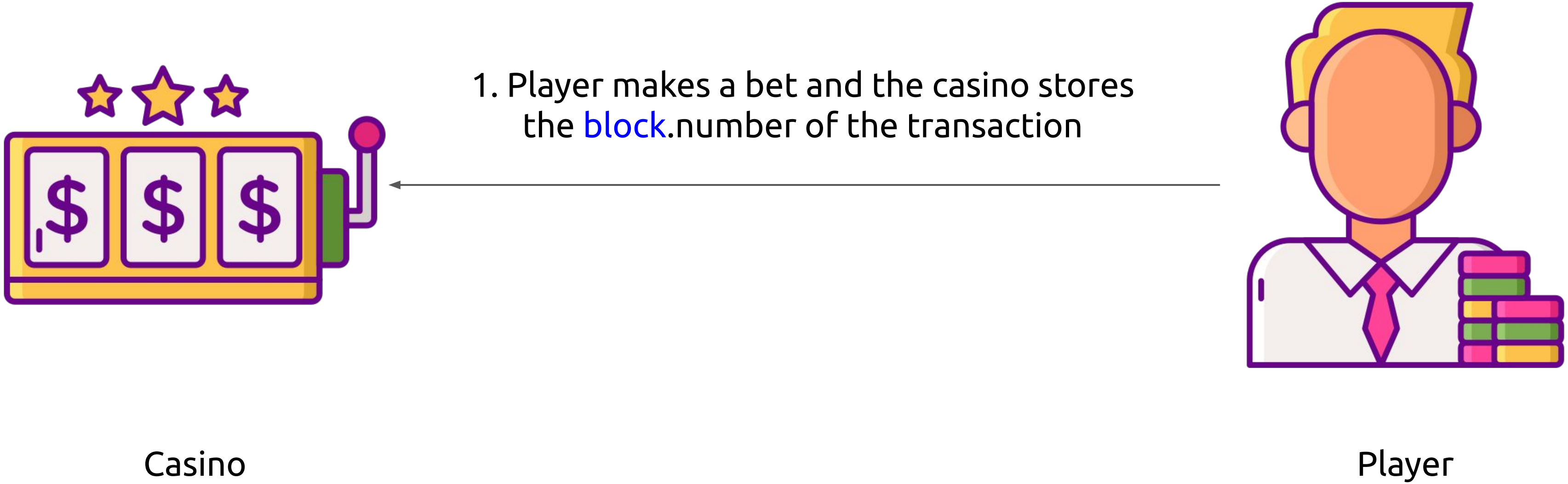
Source: Bad Randomness Is Even Dicier than You Think. Yannis Smaragdakis

What about future blocks ?

Casino



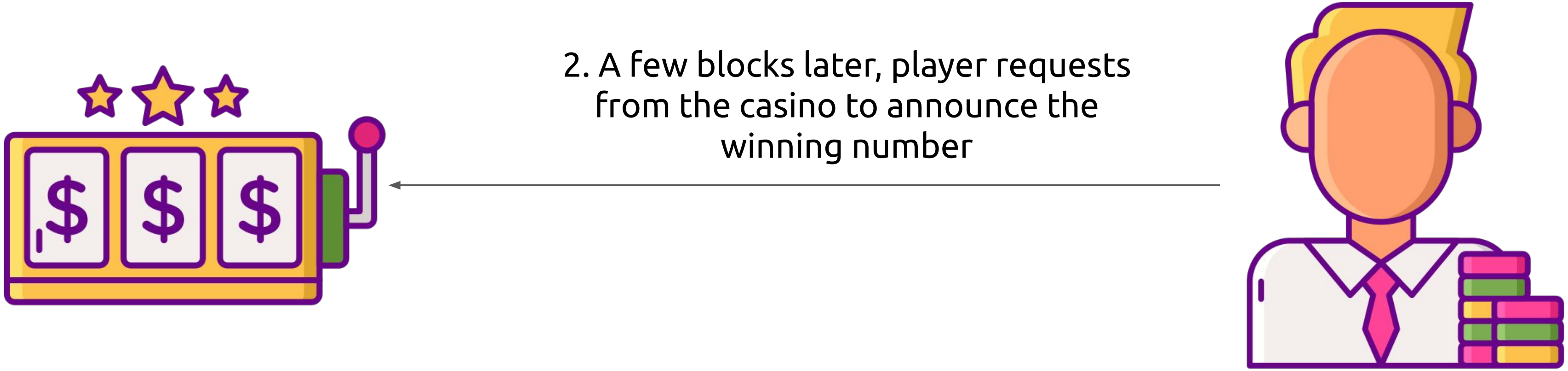
Player



1. Player makes a bet and the casino stores the block.number of the transaction

Casino

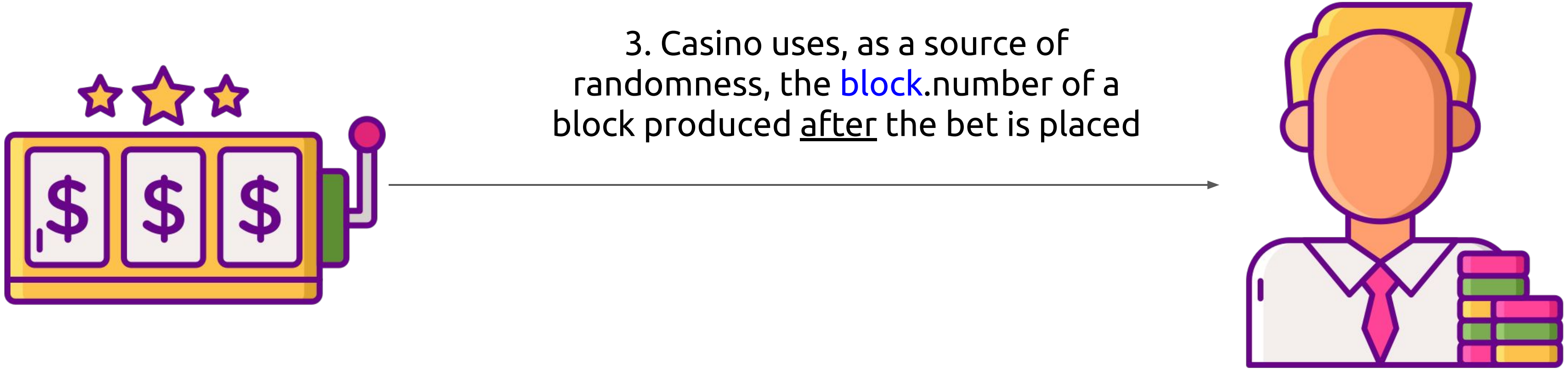
Player



2. A few blocks later, player requests from the casino to announce the winning number

Casino

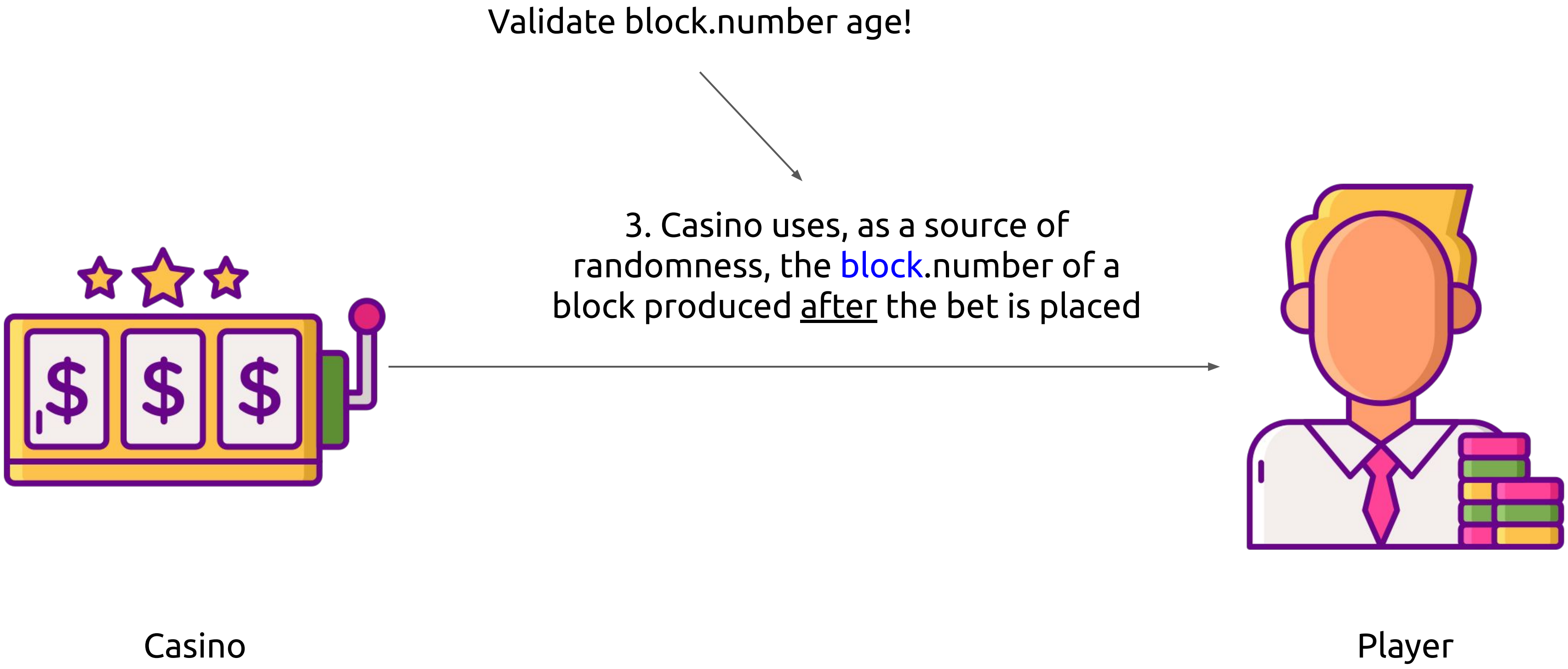
Player



3. Casino uses, as a source of randomness, the block.number of a block produced after the bet is placed

Casino

Player



Validate block.number age!

3. Casino uses, as a source of randomness, the block.number of a block produced after the bet is placed

Casino

Player

Is the hash of a block in the future a good source of randomness against a malicious miner ?

Randomness: towards safer PRNG

- Commit - reveal schemes
- Example:
  - Casino and player commit each to a random value.
  - Casino and player reveal their random values.
  - Casino XORs the random values to a seed. The seed can be combined with the hash of a future block.
- RANDAO (decentralized)

On-chain data is public

- Applications (games, auctions, etc) required data to be private up until some point in time.
- Best strategy: commitment schemes
    - Commit phase: Submit the hash of the value.
    - Reveal phase: Submit the value.
- Be aware of front-running!

Gas Fairness

Crowdfunding Contract #1

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it sends funds to R and returns any surplus to contributors.

vs.

Crowdfunding Contract #2

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it allows R to withdraw the threshold and return any surplus to contributors

Gas Fairness

Crowdfunding Contract #1

R sets a threshold

Contract collects contributions

When balance exceeds threshold, it sends funds to R and returns any surplus to contributors.

Funding paid by last contributor

vs.

Crowdfunding Contract #2

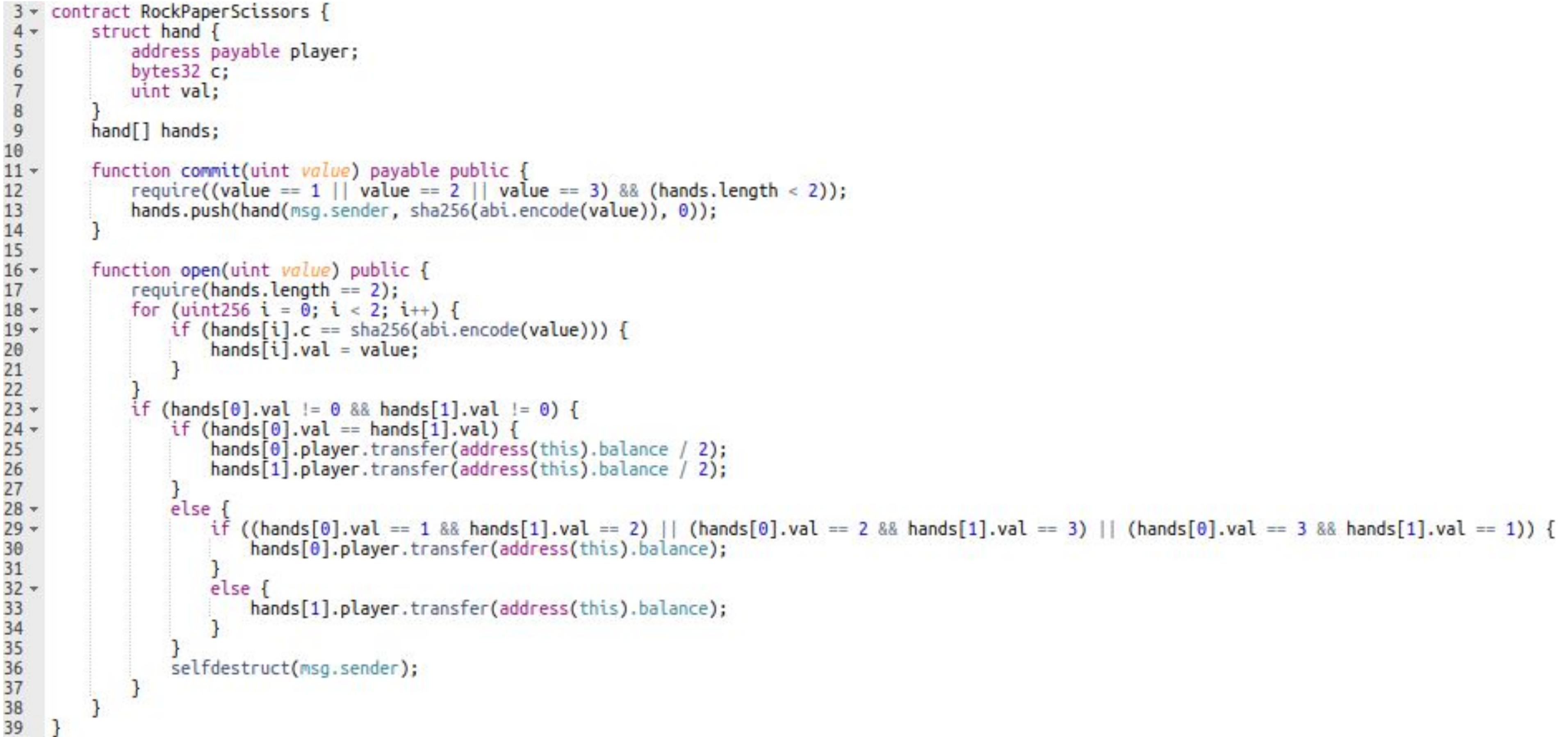
R sets a threshold

Contract collects contributions

When balance exceeds threshold, it allows R to withdraw the threshold and return any surplus to contributors

Each contributor pays for withdrawal

A horrible ✊✋✌️ contract



```solidity
contract RockPaperScissors {
    struct hand {
        address payable player;
        bytes32 c;
        uint val;
    }
    hand[] hands;

    function commit(uint value) payable public {
        require((value == 1 || value == 2 || value == 3) && (hands.length < 2));
        hands.push(hand(msg.sender, sha256(abi.encode(value)), 0));
    }

    function open(uint value) public {
        require(hands.length == 2);
        for (uint256 i = 0; i < 2; i++) {
            if (hands[i].c == sha256(abi.encode(value))) {
                hands[i].val = value;
            }
        }
        if (hands[0].val != 0 && hands[1].val != 0) {
            if (hands[0].val == hands[1].val) {
                hands[0].player.transfer(address(this).balance / 2);
                hands[1].player.transfer(address(this).balance / 2);
            }
            else {
                if ((hands[0].val == 1 && hands[1].val == 2) || (hands[0].val == 2 && hands[1].val == 3) || (hands[0].val == 3 && hands[1].val == 1)) {
                    hands[0].player.transfer(address(this).balance);
                }
                else {
                    hands[1].player.transfer(address(this).balance);
                }
            }
            selfdestruct(msg.sender);
        }
    }
}
```

Thank you!