Blockchains & Distributed Ledgers

Lecture 04

Dimitris Karakostas

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

Denial-of-Service

DoS: 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: 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.

Error handling

- If a send/transfer **call fails**, the contract might get **stuck**.
- It is **possible to force** a call to fail (e.g., by forcing a contract to send to another contract).
- **Errors** need to be **handled**, instead of simply reverting
- *transfer* is **preferable** to *send*, as it returns an **error object** that can be **examined** and act upon accordingly

Pull over push: example

```
// BAD DESIGN (PUSH)

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

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

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

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

Pull over push: example

```
// BAD DESIGN (PUSH)

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

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

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

```
// PULL DESIGN

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

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**.

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.

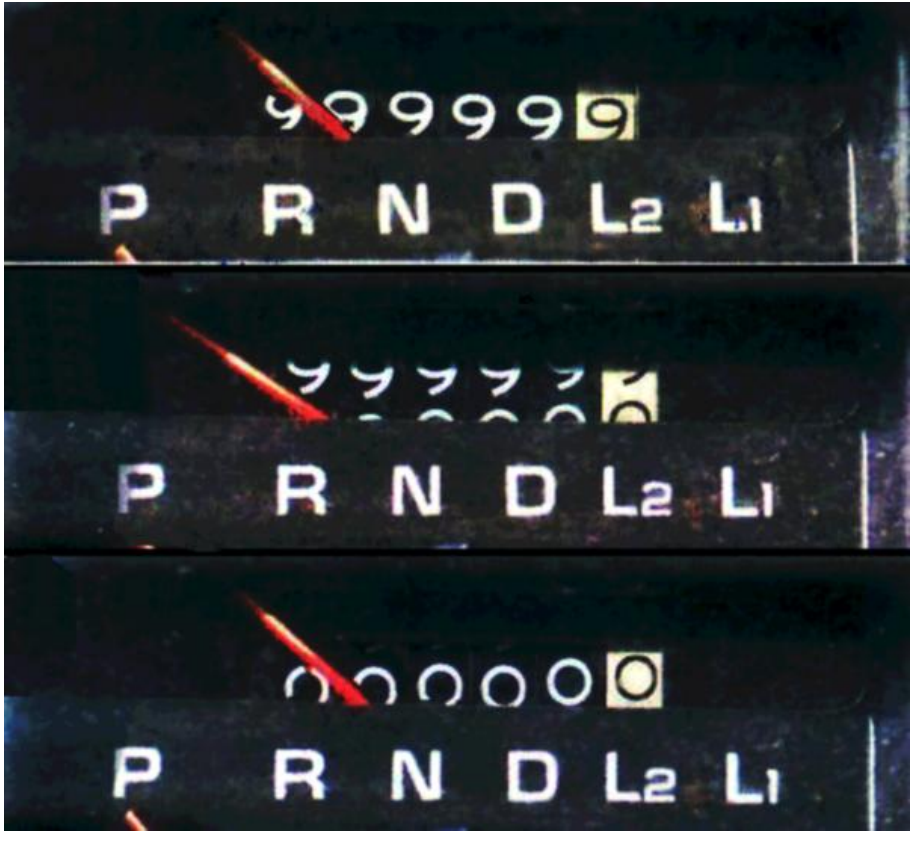
DoS: Unbounded operation

```
// 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) { … }
}
```

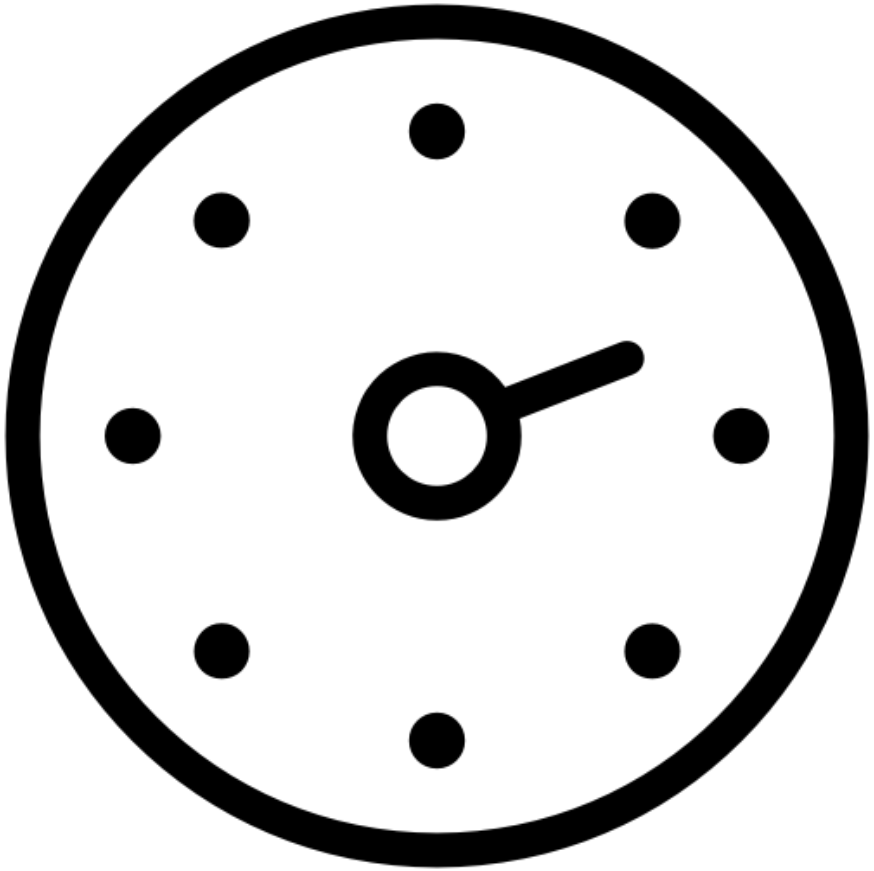
Source: MadMax: surviving out-of-gas conditions in Ethereum smart contracts. Grech N., Kong M., Jurisevic A., Brent L., Scholz B., and Smaragdakis Y. OOPSLA ‘18.

Overflow/Underflow

Integer Overflow and Underflow



$2^{256} - 1$ 0



2^{254}

2^{255}

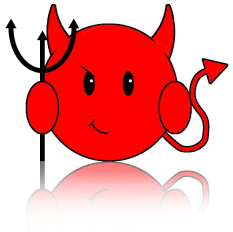
Integer Overflow and Underflow

```
// 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 donate(uint256 _value) public payable {
      require(msg.value == value);
      balanceOf[msg.sender] += _value;
}
```

```
function attack() {
      performAttack = true;
      victim.donate(1);
      victim.withdraw(1);
}
function() {
      if (performAttack) {
            performAttack = false;
            victim.withdraw(1);
      }
}
```

Integer Overflow and Underflow: solutions

Use OpenZeppelin's SafeMath library

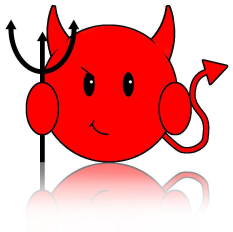
```
// OpenZeppelin: SafeMath.sol

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;
}
```

Reentrancy

Reentrancy

Fallback function

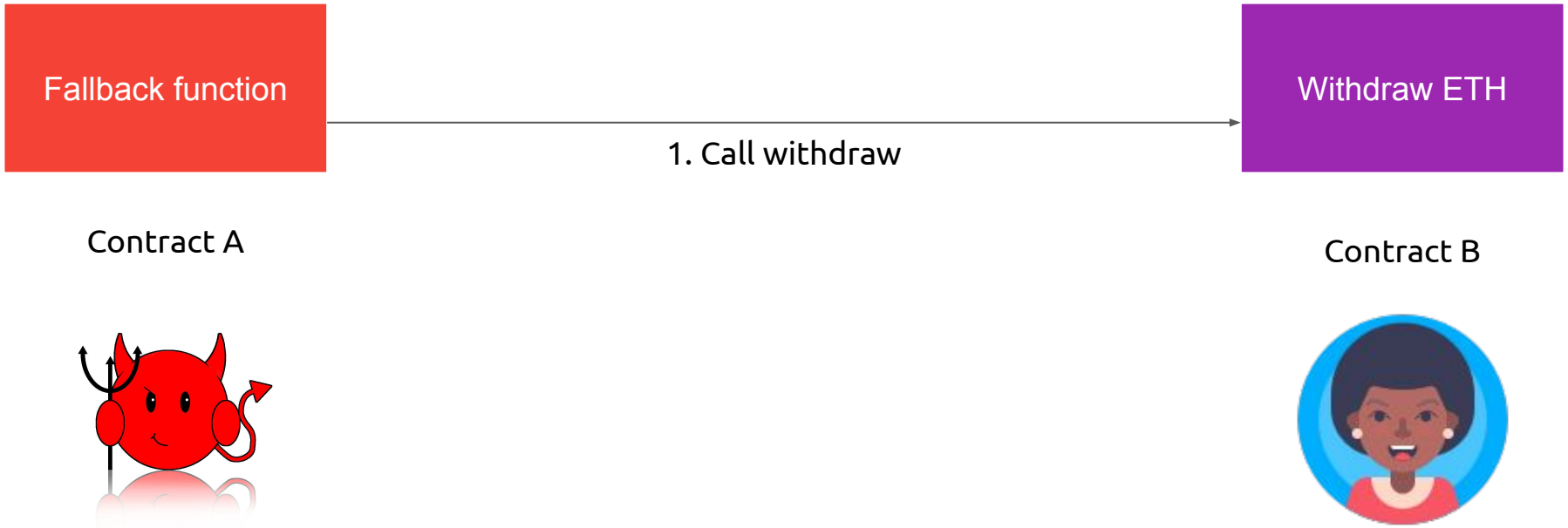
Contract A



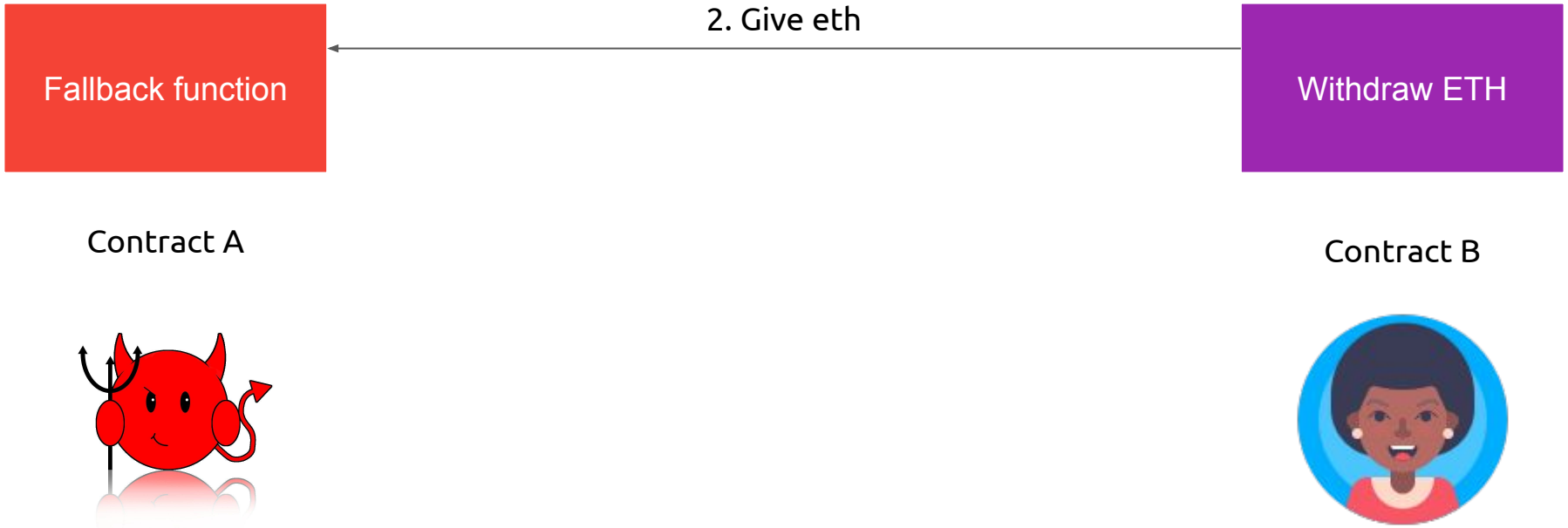
Withdraw ETH

Contract B

Reentrancy


Fallback function
1. Call withdraw
Withdraw ETH
Contract A
Contract B

Reentrancy


Fallback function
2. Give eth
Withdraw ETH
Contract A
Contract B

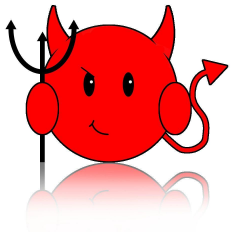
Reentrancy

Fallback function

3. Call withdraw again

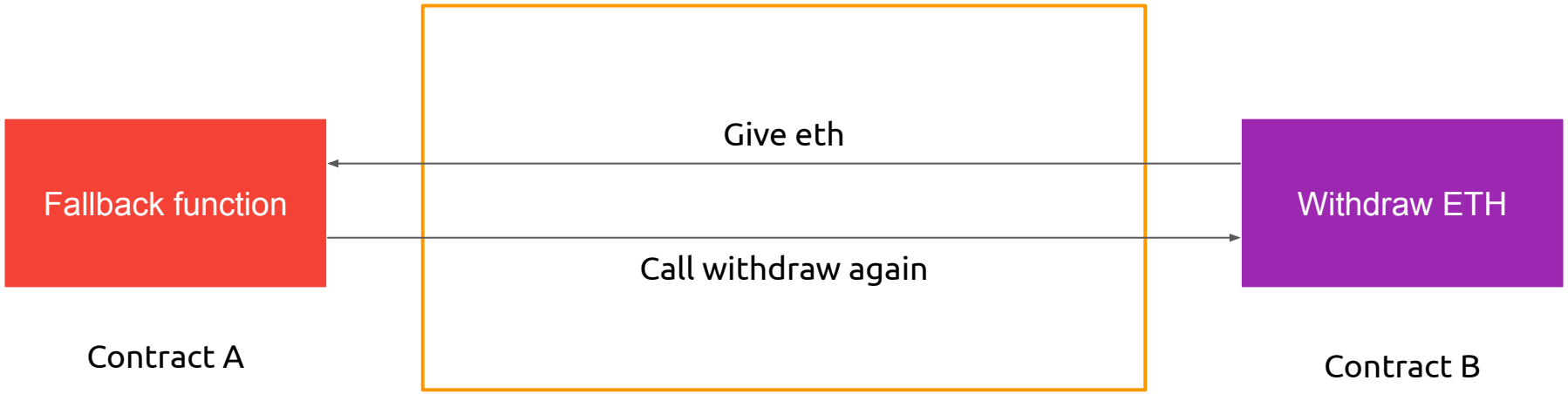
Withdraw ETH

Contract A



Contract B

Reentrancy


Fallback function
Give eth
Withdraw ETH
Call withdraw again
Contract A
Contract B
Loop of function calls

Reentrancy example

```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

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

      userBalances[msg.sender] = 0;

}
```

Contract B

Reentrancy example

```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

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

      userBalances[msg.sender] = 0;

}
```

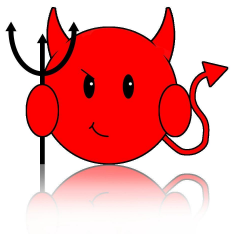
Contract B

Reentrancy example

Begin attack by sending msg.value eth

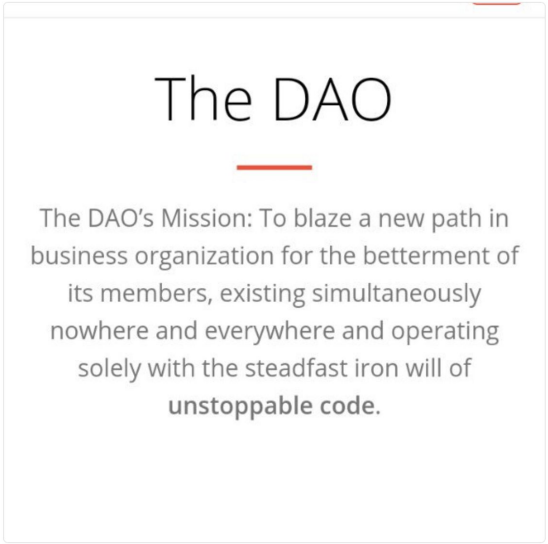
```
// INSECURE
mapping (address => uint) private userBalances;
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)());
    userBalances[msg.sender] = 0;
}
```

```
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 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

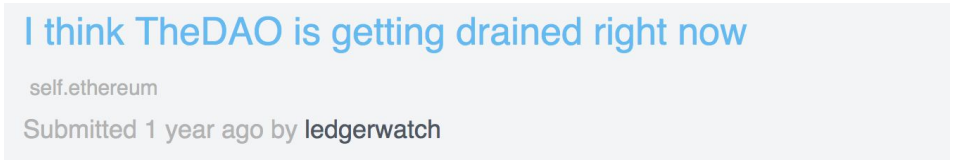
15 hours
NEXT PRICE PHASE

11 days
LEFT
ENDS 28 MAY 09:00 GMT

~150 million USD in ~ 1 month

The DAO Attack (2016)

- 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



Reentrancy: solutions

```
// SECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

      userBalances[msg.sender] = 0;

      msg.transfer(amountToWithdraw);

}
```

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

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)

Solidity/Ethereum hazards

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance`
- How can you **send ether** to a contract **without** firing contact's **fallback** function ?

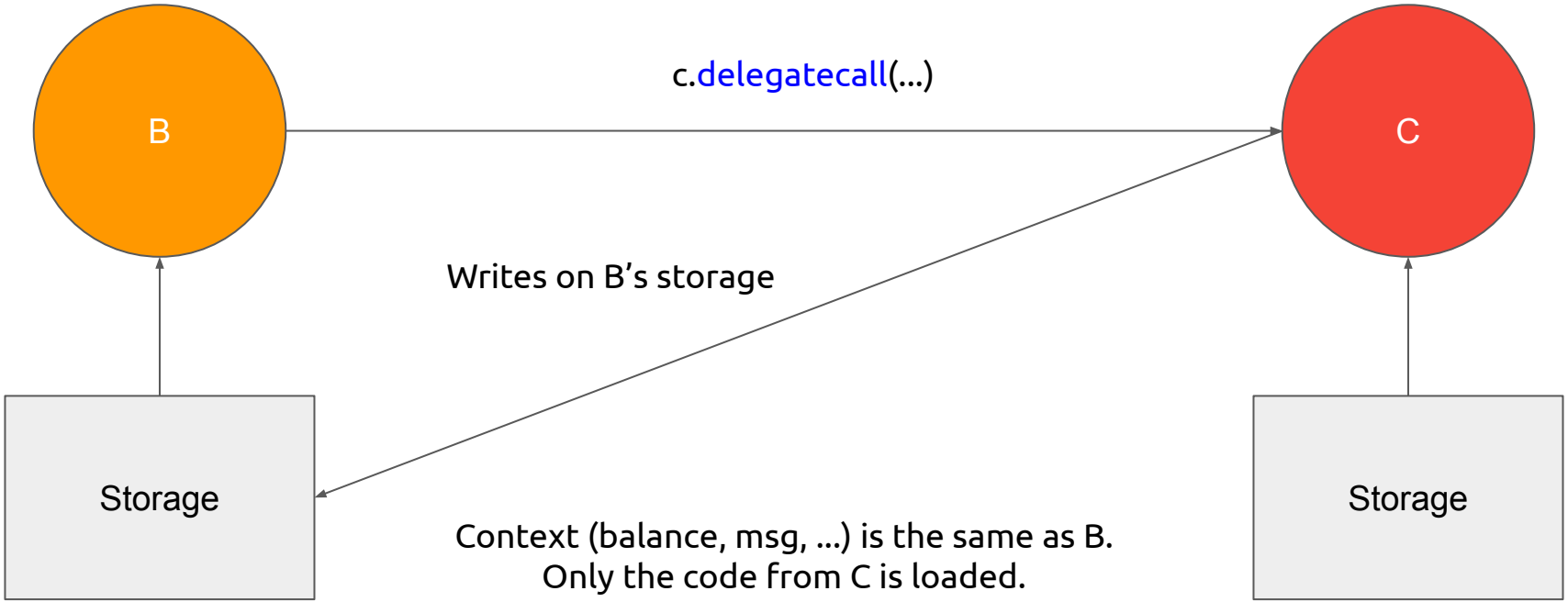
Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance`
- How can you **send ether** to a contract **without** firing contact's **fallback** function ?
  - 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.

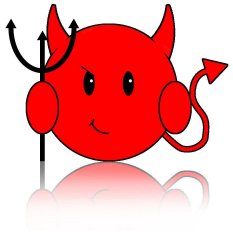
Delegate call

Delegate call

Delegate call



Delegate call

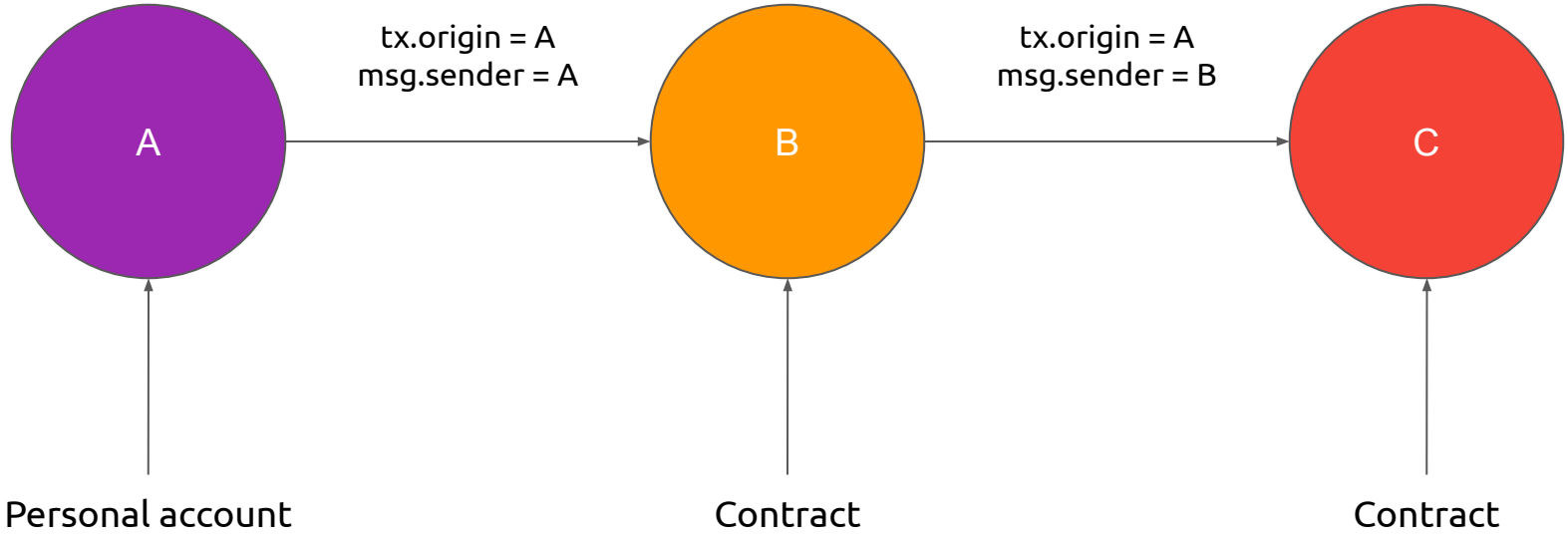


```
// INSECURE
address public owner;

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



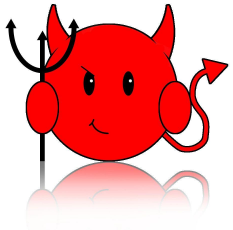
Use of tx.origin

```
// 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);

}
```

Keep fallback function simple

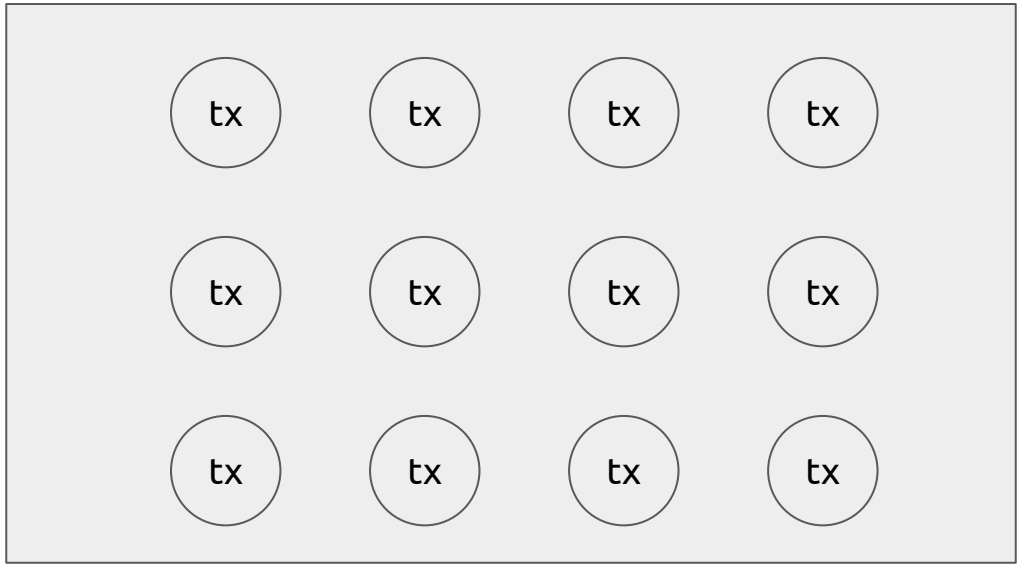
```
// BAD

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

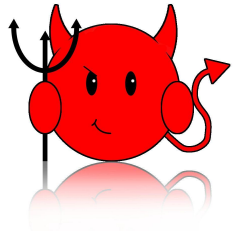
Front-running

Front-Running



Miner: sortByGasPrice(txs, ‘desc’)

Front-Running: user



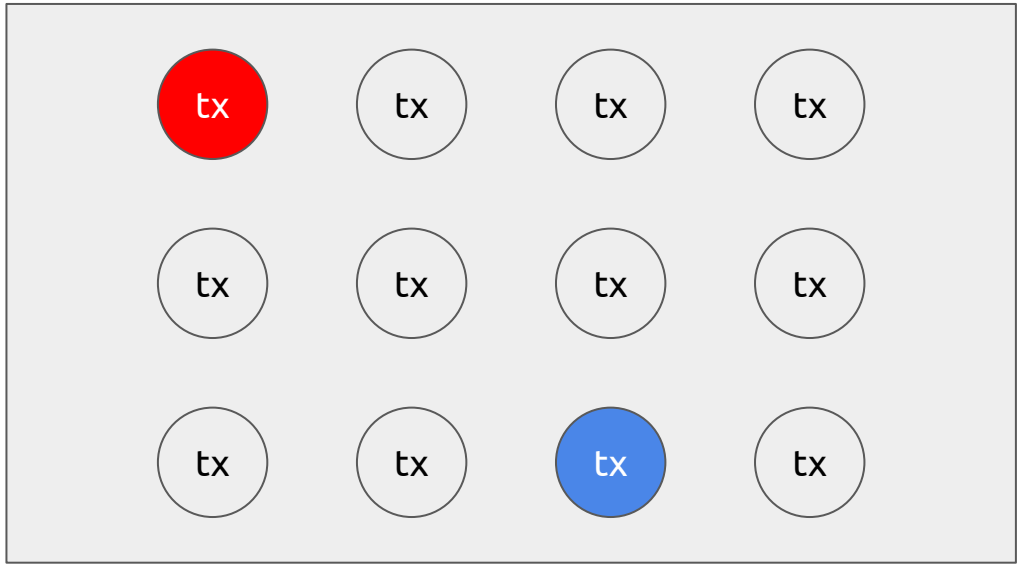
50 GWei

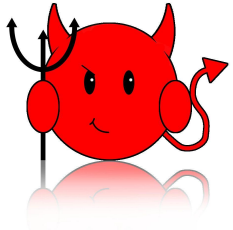
tx

2 GWei

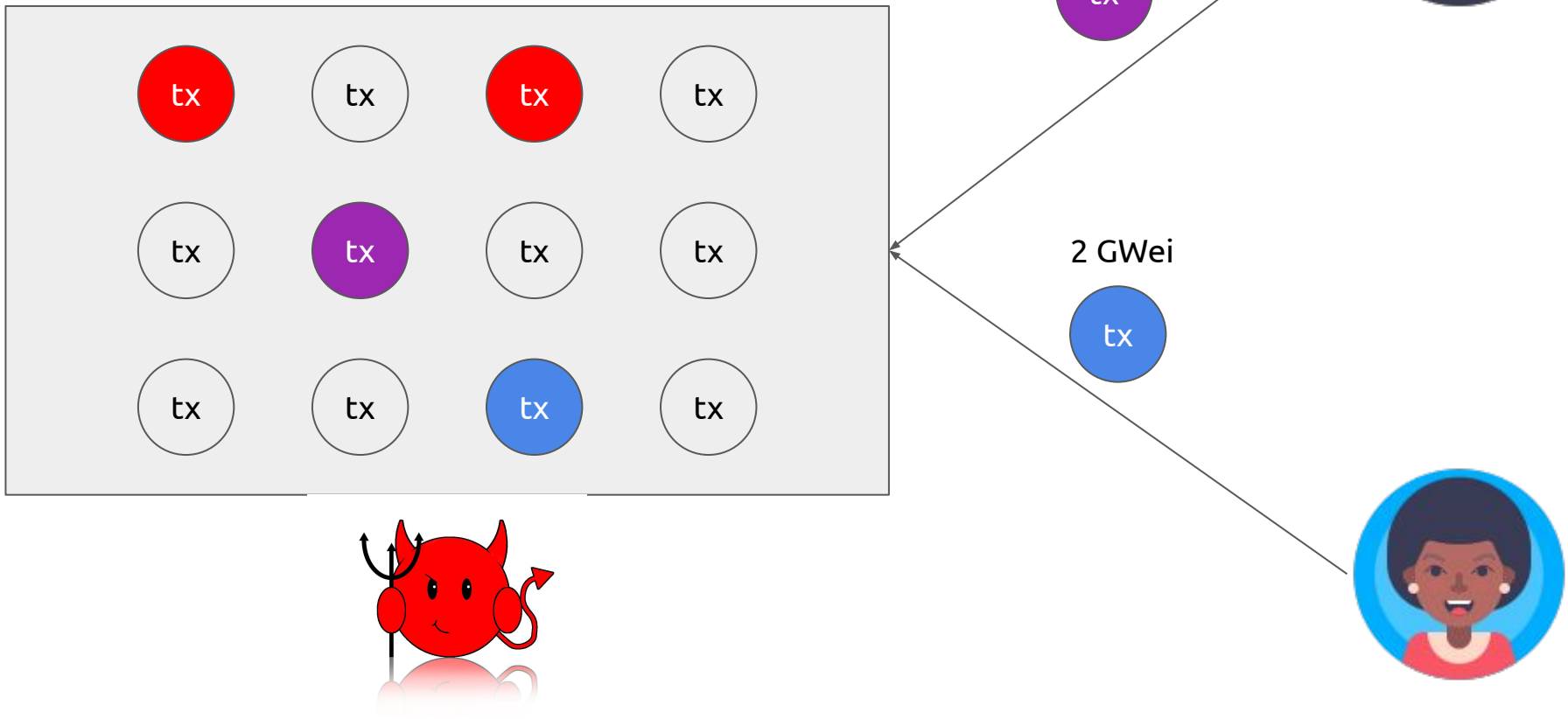
tx

Front-Running: user





Front-Running: miner



Front-Running: example

```
// INSECURE

function registerName(bytes32 name) public {

     names[name] = msg.sender;

}
```

Front-Running: example

```
// INSECURE

function registerName(bytes32 name) public {

     names[name] = msg.sender;

}
```

registerName becomes a non-interactive cryptographic commitment function.
Properties:

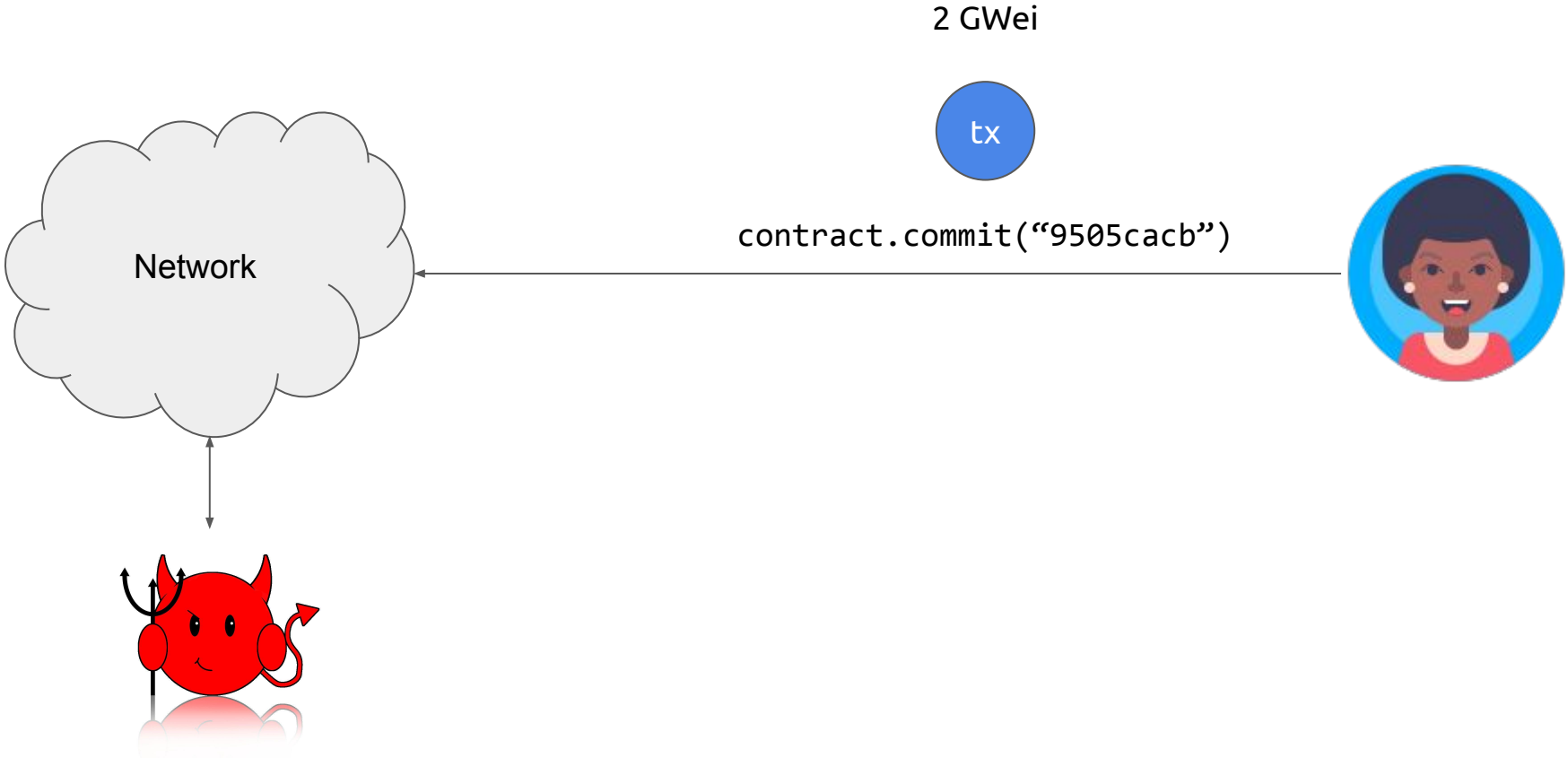
- Binding. A commitment can be only opened to its committed value.
- Hiding. The commitment reveals no information about its committed value.

A way to implement is via a cryptographic hash function, e.g., keccak256

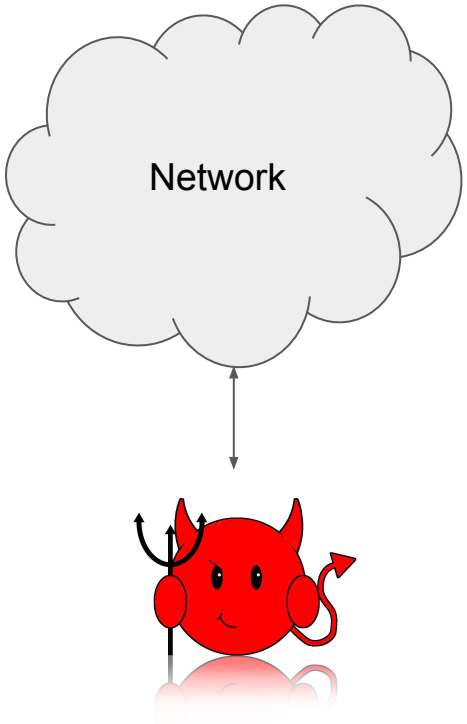
Note: nonce space should be large!

```
function registerName(bytes32 name, bytes32 nonce) public {

     require(commitments[makeCommitment(name, nonce)] == msg.sender, “Not found!”);

     names[name] = msg.sender;

}
```

Front-Running: example



Front-Running: example

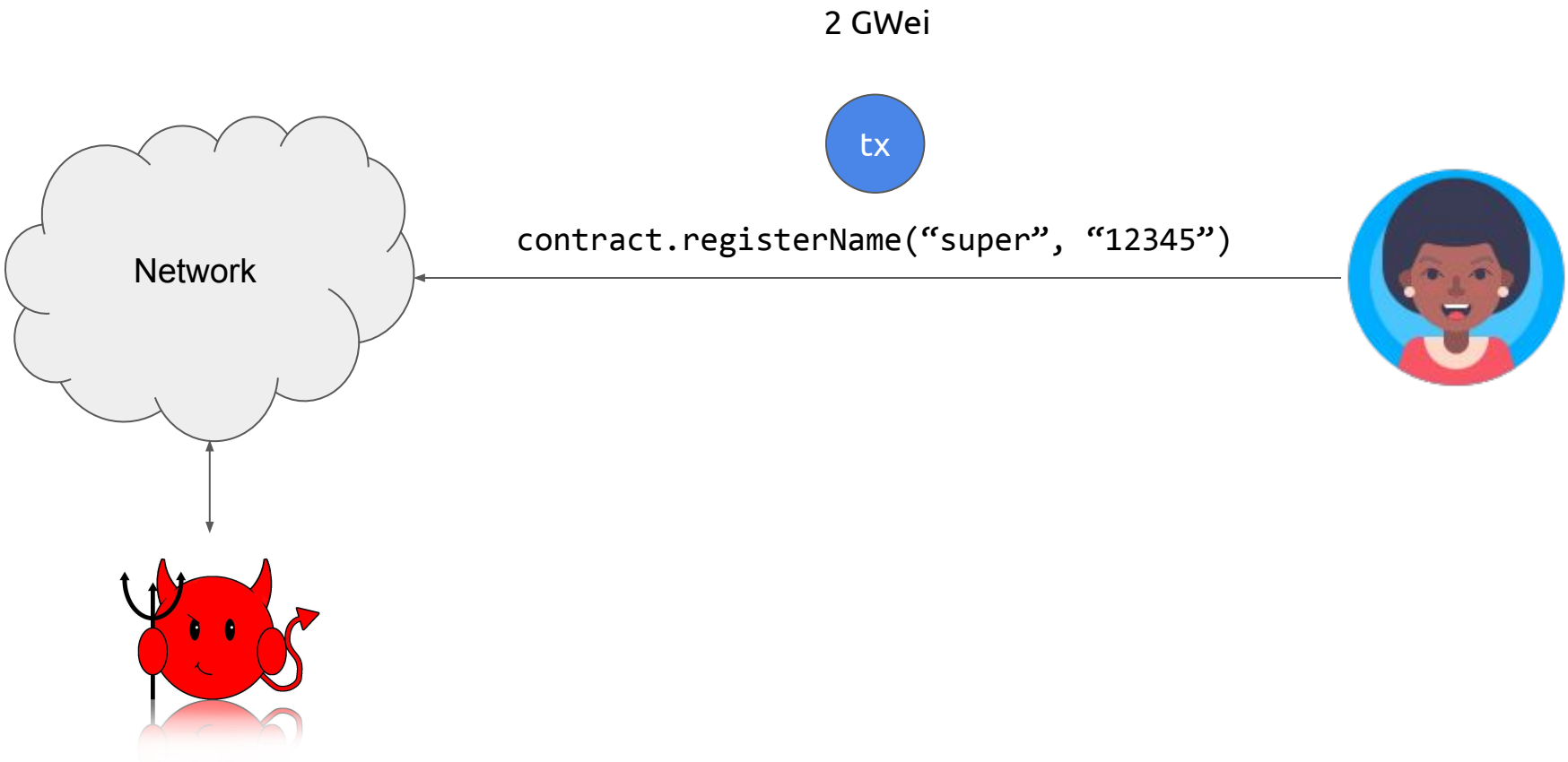


2 GWei

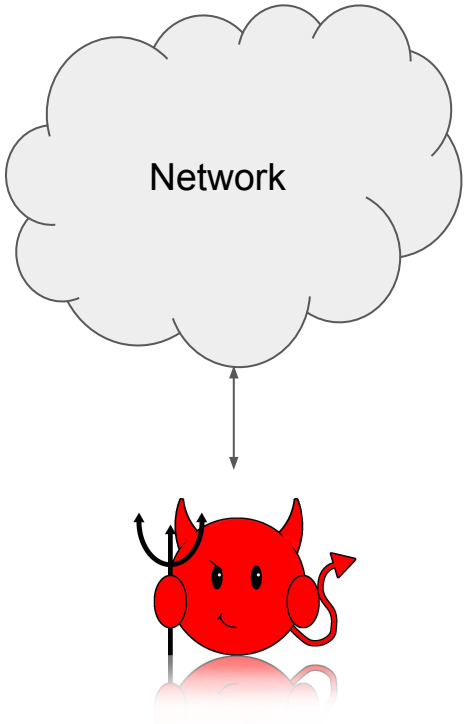
tx

```
contract.commit(“9505cacb”)
```

Front-Running: example



Front-Running: example



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

Randomness: sources (?)

- block.number
- block.time
- block.has
- block.diff

- lock.coinbase
- k.gasLimit
- n
- ender

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

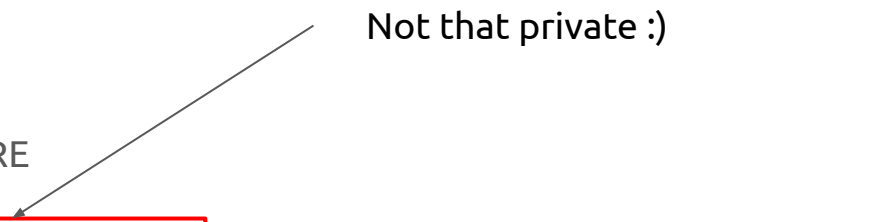
Randomness

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

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

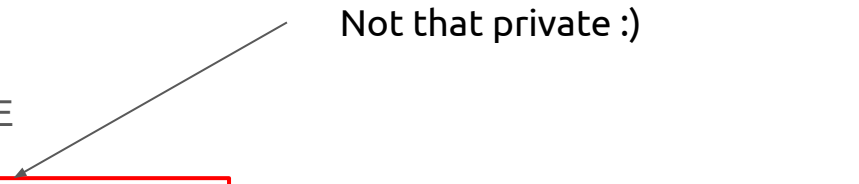
```
// INSECURE
address seed1 = contestants[uint(block.coinbase) % totalTickets].addr;
address seed2 = contestants[uint(msg.sender) % totalTickets].addr;
uint seed3 = block.difficulty;
bytes32 randHash = keccak256(seed1, seed2, seed3);
uint winningNumber = uint(randHash) % totalTickets;
address winningAddress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

Randomness: blockhash




```
// 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




```
// INSECURE

uint256 constant private FACTOR =
115792089237316195423570985008687907853269984665640564039457584007913129639 9;

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: intra-transaction information leak

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

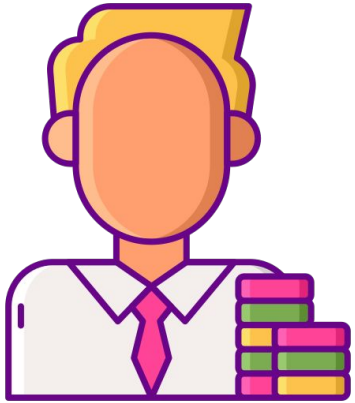
Source: Bad Randomness Is Even Dicier than You Think. Yannis Smaragdakis

Sources of randomness

- **Block information** can be **manipulated by miner**
- Block information **shared** by all users in the same block
- In Ethereum, **all data** posted on the chain are **visible**
- “private” vars are only private w.r.t. object-oriented programming **visibility**
- If same-block txs share randomness source, attacker can **check** whether conditions are favorable **before** acting

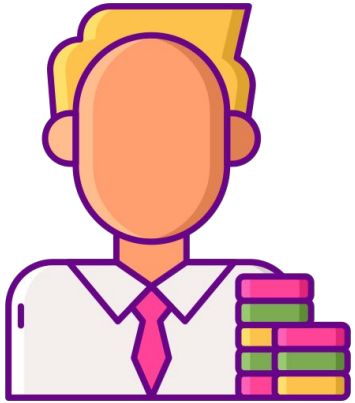
What about future blocks ?

Casino



Player

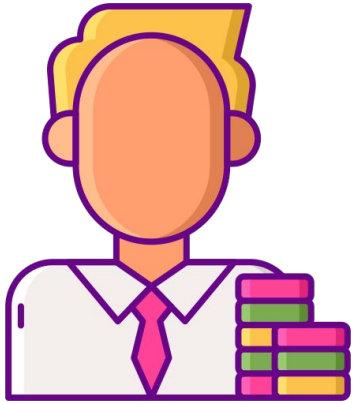
1. Player makes a bet and the casino stores the block.number of the transaction



Casino

Player

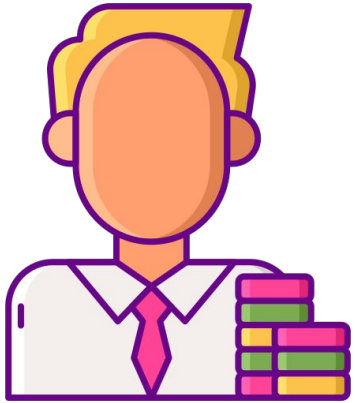
2. A few blocks later, player requests from the casino to announce the winning number



Casino

Player

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

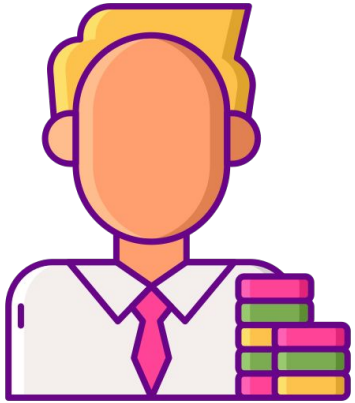


Casino

Player

Validate block.number age!

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



Casino

Player

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

- A contract can access the hashes of only the **last 256** blocks; blockhash older than that defaults to 0
- Always **validate** block's age
- A **miner** can keep newly-mined **blocks hidden**, until they mine a favorable one

Randomness: towards safer PRNG

- Commit - reveal schemes
- Example:
  - Casino and player each commit to a random value.
  - Casino and player reveal their values.
  - Casino XORs the random values to a seed; the seed can be combined with the hash of a future block.
  - If either casino or player honest, then seed is random.

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

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

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

Funding paid by last contributor

R pays for funding

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

VS.

Crowdfunding Contract #3

R sets a threshold

Contract collects contributions

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

Funding paid by last contributor

R pays for funding

R and contributors pay for funding

A (horribly insecure) ✊🖐✌ contract

```
3  pragma solidity >=0.7.0 <0.9.0;
4
5  contract RockPaperScissors { // Winner gets 1 ETH
6      struct round {
7          address payable player;
8          bytes32 commitment;
9          uint256 hand;
10     }
11     round[] private rounds;
12
13     function commit(uint256 hand) payable public {
14         require((hand == 1 || hand == 2 || hand == 3) && (rounds.length < 2));
15         rounds.push(round(payable(msg.sender), sha256(abi.encode(hand)), 0));
16     }
17
18     function open(uint256 hand) public {
19         require(rounds.length == 2);
20         for (uint256 i = 0; i < 2; i++) {
21             if (rounds[i].commitment == sha256(abi.encode(hand))) {
22                 rounds[i].hand = hand;
23             }
24             if (rounds[(i + 1) % 2].hand == 0) {
25                 return;
26             }
27         }
28         if ((rounds[0].hand == 1 && rounds[1].hand == 2) ||
29             (rounds[0].hand == 2 && rounds[1].hand == 3) ||
30             (rounds[0].hand == 3 && rounds[1].hand == 1)) {
31             rounds[0].player.transfer(1 ether);
32         }
33         else if (rounds[0].hand != rounds[1].hand) {
34             rounds[1].player.transfer(1 ether);
35         }
36         selfdestruct(payable(msg.sender));
37     }
38 }
```