Blockchains & Distributed Ledgers

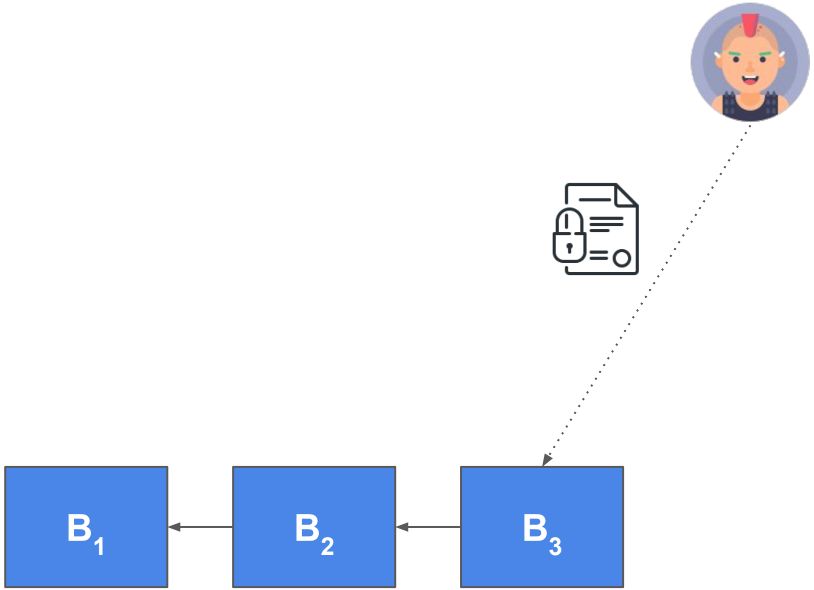
Lecture 04

Dimitris Karakostas

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

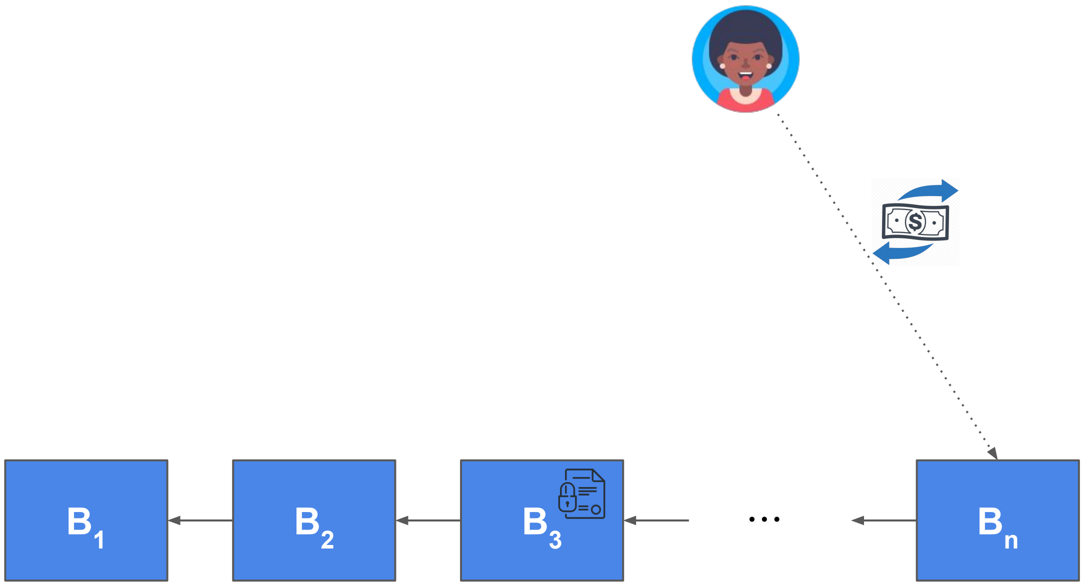
Smart Contracts

- `The developer writes and deploys the contract`



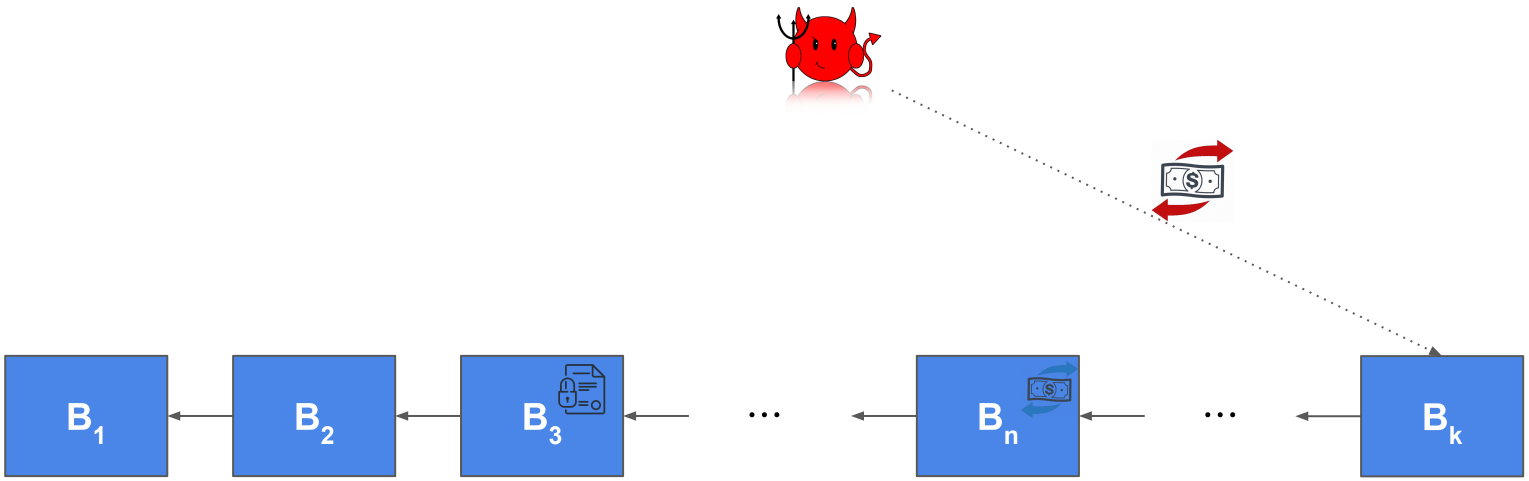
Smart Contracts

- The developer writes and deploys the contract
- A user interacts with the contract via a transaction



Smart Contracts

- The developer writes and deploys the contract
- A user interacts with the contract
- An adversary exploits a hazard in the contract, by sending a transaction that somehow breaks its functionality



Smart Contracts

- The developer writes and deploys the contract
- A user interacts with the contract
- An adversary exploits a hazard in the contract, by sending a transaction that somehow breaks its functionality

In this lecture, you will learn:

- How to identify hazards in contracts written by others
- How to protect users (of your contracts) from known attacks

Denial-of-Service

DoS: Unbounded operation

```
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.transfer(investors[i].dividendAmount);
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Unbounded operation

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.transfer(investors[i].dividendAmount);
}
```

- Operation requires more gas as array becomes larger
- After some point, it might be impossible (beyond gas limits) to execute it

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Griefing

```
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.transfer(investors[i].dividendAmount));
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Griefing

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.transfer(investors[i].dividendAmount));
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

DoS: Griefing

```
// INSECURE
for (uint i = 0; i < investors.length; i++) {
    investors[i].addr.transfer(investors[i].dividendAmount);
}

// ALSO INSECURE
for (uint i = 0; i < investors.length; i++) {
    if (!(investors[i].addr.send(investors[i].dividendAmount)) {
        revert();
    }
}
```

https://cs.pomona.edu/~michael/courses/csci190s21/papers/madmax.pdf

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 getting the victim contract to send to another contract that fails)
- **Errors** need to be **handled**, instead of simply reverting
- *transfer* is **preferable** to *send*, as it returns an **error object** that can be **examined** to act accordingly

Pull over push: example

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

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

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

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;
}
```

```
// GOOD DESIGN (PULL)

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);
}
```

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

Pull over push

- **Do not transfer** ETH to users (push); let them **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**.
- Possibly increases **gas fairness** (each user pays the gas for receiving their own funds).
- **Tradeoff** between **security** and **user experience**.

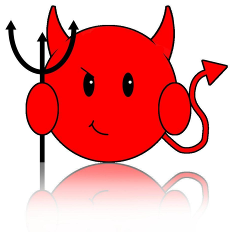
Reentrancy

Reentrancy

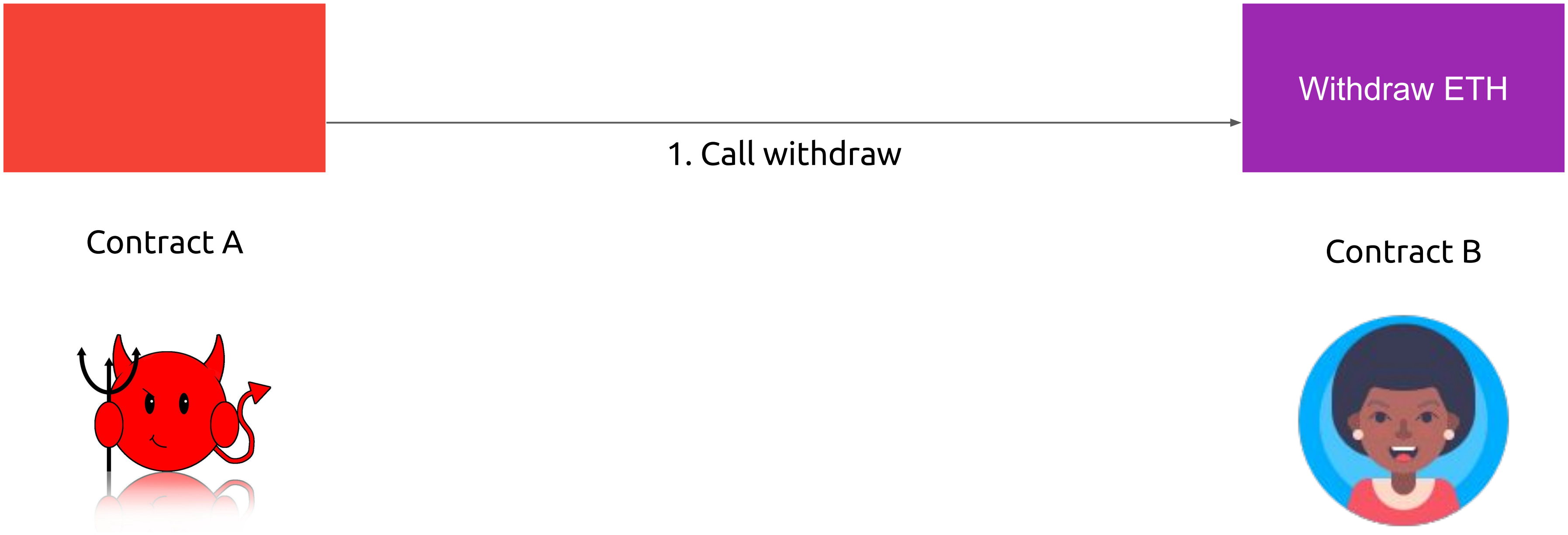
Withdraw ETH

Contract A

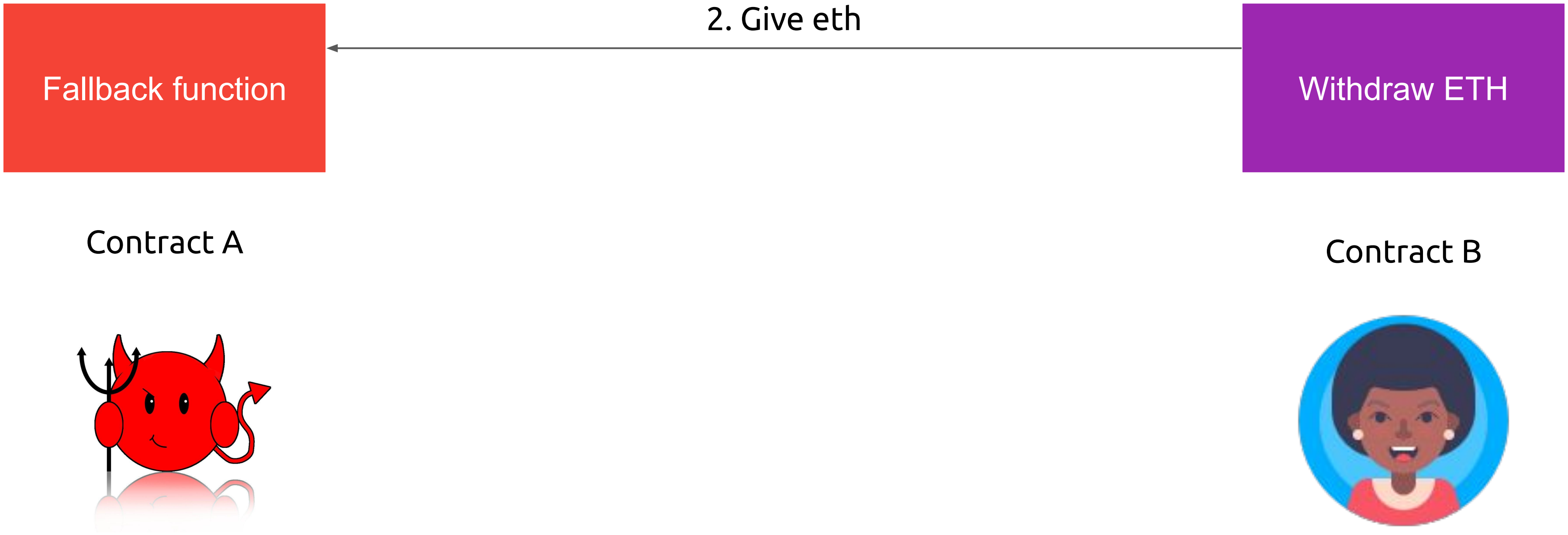
Contract B



Reentrancy



Reentrancy


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

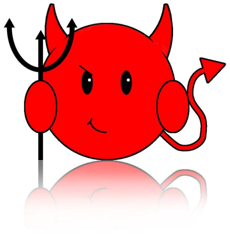
Reentrancy

Fallback function

3. Call withdraw again

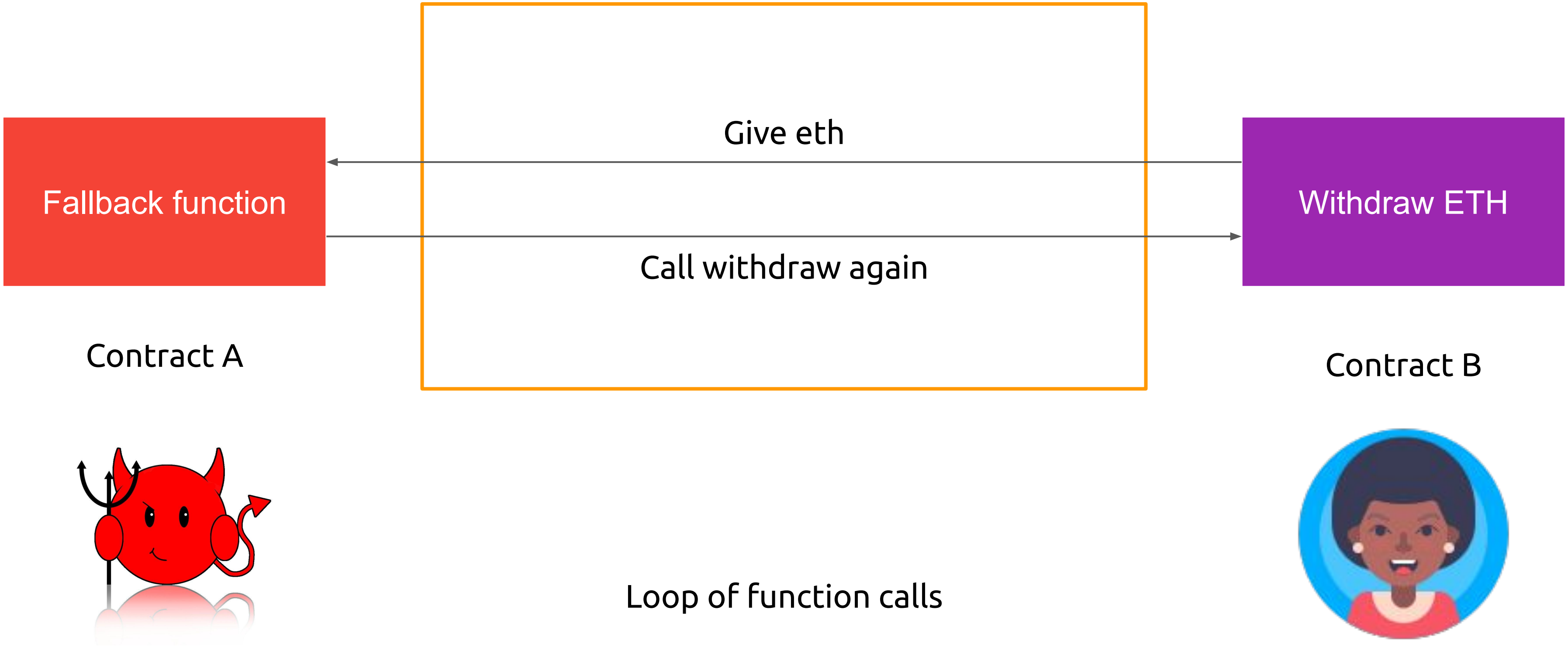
Withdraw ETH

Contract A



Contract B

Reentrancy


Give eth
Fallback function
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;

}
```

Reentrancy example

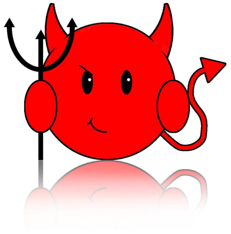
```
// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

      uint amountToWithdraw = userBalances[msg.sender];

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

      userBalances[msg.sender] = 0;

}
```

Reentrancy example

Begin attack by sending 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 receive() payable {

  if (victimContract.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.**

*According to the SEC, neither "distributed" nor "autonomous": https://www.sec.gov/news/press-release/2017-131

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)

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

}
```

- Finish all internal work (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., on inputs, sender, value, arguments etc
2. Enforce **effects** and update the **state** accordingly
3. **Interact** with other accounts via external calls or send/transfer

Solidity/Ethereum hazards

Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance (when contract relies on it)`

```
contract Vulnerable {
    function receive() external {
        revert();
    }

    function fallback() external {
        revert();
    }

    function somethingBad() {
        require(this.balance > 0);
        // Do something bad
    }
}
```

https://github.com/demining/Solidity-Forcibly-Send-Ether-Vulnerability

Forcibly Sending Ether to a Contract

- Possible exploit
    - **misuse** of `this.balance (when contract relies on it)`
- 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 (when contract relies on it)`
- 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** and send ether to it
    - **selfdestruct**(victimContractAddress) does **not** trigger fallback
    - Set contract's address as **recipient of block rewards**

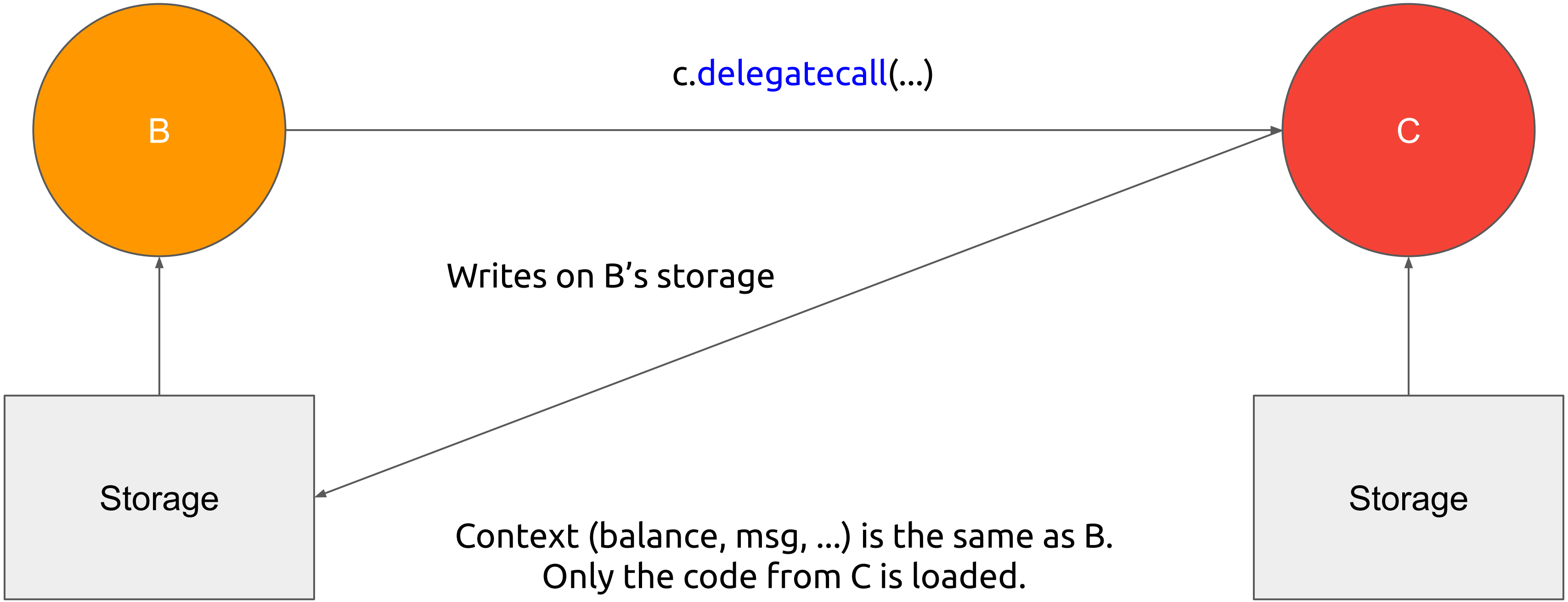
Forcibly Sending Ether to a Contract

- Possible exploit
  - **misuse** of `this.balance (when contract relies on it)`
- 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** and send ether to it
  - **selfdestruct**(victimContractAddress) does **not** trigger fallback
  - Set contract's address as **recipient of block rewards**
- Lesson: **Avoid** strict equality checks with the contract's balance

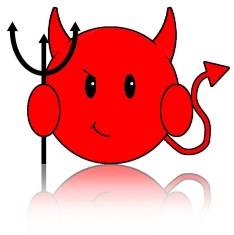
Delegate call

Delegate call

Delegate call



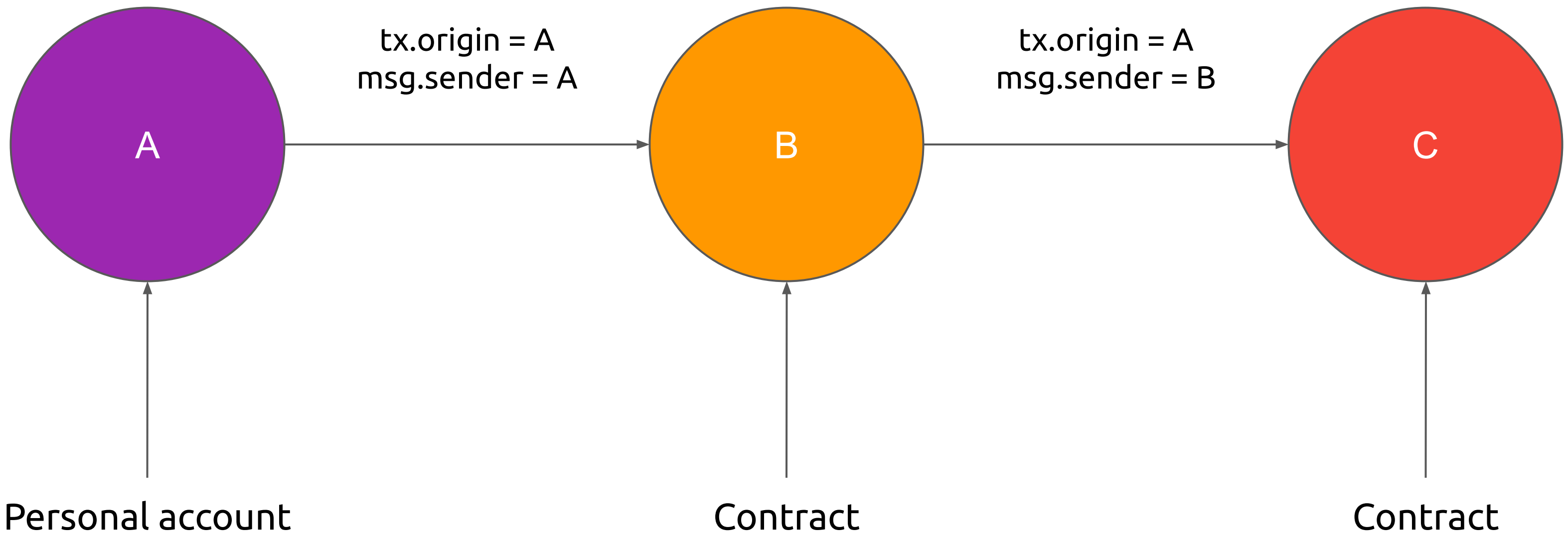
Delegate call



```
// 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;
  }
}
```

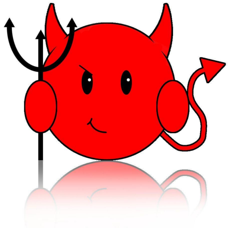
Use of tx.origin



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)();
    }

}
```

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 receive() external payable {

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

}
```

Keep fallback function simple

```
// BAD

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

```
// GOOD

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

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

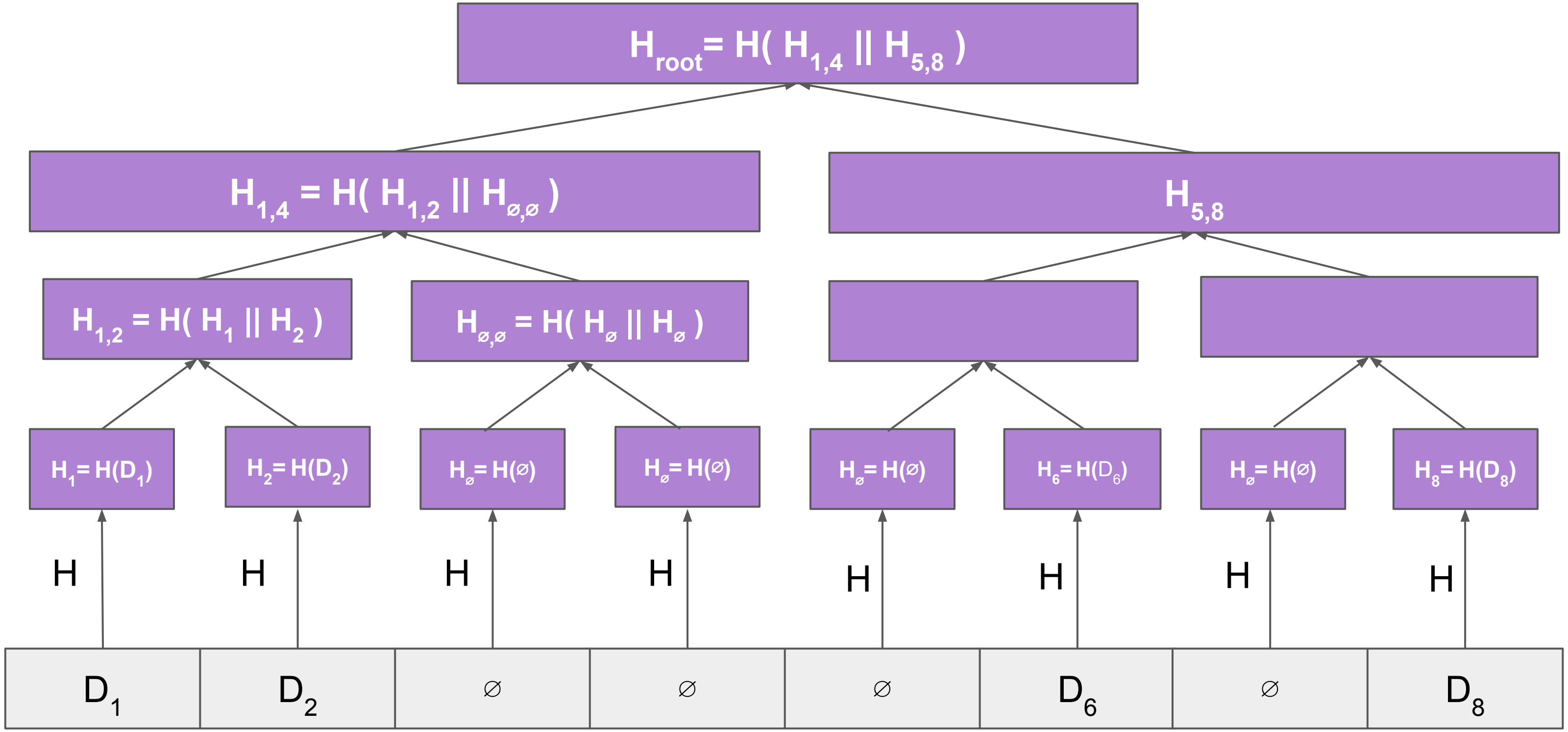
https://consensys.github.io/smart-contract-best-practices

Default values
And
Merkle Trees

Sparse Merkle Trees

- Perfect Binary Merkle Tree
- Unfilled leaves take default values

Sparse Merkle Trees



Sparse Merkle Trees: key-value stores

- Assume that keys are 256 bits (e.g., a SHA256 hash)
- Construct a Sparse Merkle Tree with 2^{256} leaves
- Insert a (key, value) element in the store
  - Insert the value in the **leaf** that corresponds to the **key**
  - Construct the root of the new Merkle Tree
- Proof of inclusion: as usual
- Proof of non-inclusion: prove **empty value** in leaf for **corresponding key**
- Constructing such tree for 2^{256} leaves from scratch is extremely **consuming**
  - Optimizations?

Solidity’s default values

- Solidity does not support None/null types
- Every variable is initialized to a (respective) **zero value**
    - uint256: 0
    - bytes32: bytes32(0)
    - …
- Verifying whether a string is not initialized:
    - bytes(myVariable).length != 0
    - sha3(myVariable) != sha3("")

The Nomad Bridge Hack

- Nomad contract kept:
  - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
    - Intended use: Timestamp after which an MTR can be used for message validation

```
function acceptableRoot(bytes32 _root) public view returns (bool) {
    // ...
    uint256 _time = confirmAt[_root];
    if (_time == 0) {
       return false;
    }
    return block.timestamp >= _time;
}
```

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

- Nomad contract kept:
    - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
        - Intended use: Timestamp after which an MTR can be used for message validation
    - mapping of message hashes to MTRs: **mapping(bytes32 => bytes32) messages**
        - Intended use: if a message is validated, the mapping keeps the message's hash and the MTR used to validate it

```
function process(bytes memory _message) public returns (bool _success) {
   // ...
   require(acceptableRoot(messages[_messageHash]), "!proven");
   // ...
}
```

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

- Nomad contract kept:
  - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
    - Intended use: Timestamp after which an MTR can be used for message validation
  - mapping of message hashes to MTRs: **mapping(bytes32 => bytes32) messages**
    - Intended use: if a message is validated, the mapping keeps the message's hash and the MTR used to validate it
- On 21 June 2022, a new version of the contract was created
  - During initialization, Nomad set: **confirmAt[bytes32(0)] = 1**
  - Attack!

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

- Nomad contract kept:
  - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
  - mapping of message hashes to MTRs: **mapping(bytes32 => bytes32) messages**
- On 21 June 2022, a new version of the contract was created
  - During initialization, Nomad set: **confirmAt[bytes32(0)] = 1**
  - Attack!
    - Every non-validated message is initialized to the zero MTR in the *messages* mapping

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

- Nomad contract kept:
  - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
  - mapping of message hashes to MTRs: **mapping(bytes32 => bytes32) messages**
- On 21 June 2022, a new version of the contract was created
  - During initialization, Nomad set: **confirmAt[bytes32(0)] = 1**
  - Attack!
    - Every non-validated message is initialized to the zero MTR in the *messages* mapping
    - By setting *confirmAt[bytes32(0)] = 1*, the zero MTR gets “confirmed” at timestamp 1

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

- Nomad contract kept:
    - mapping of MTRs to timestamps: **mapping(bytes32 => uint256) confirmAt**
    - mapping of message hashes to MTRs: **mapping(bytes32 => bytes32) messages**
- On 21 June 2022, a new version of the contract was created
    - During initialization, Nomad set: **confirmAt[bytes32(0)] = 1**
    - Attack!
        - Every non-validated message is initialized to the zero MTR in the *messages* mapping
        - By setting *confirmAt[bytes32(0)] = 1,* the zero MTR gets “confirmed” at timestamp 1
        - So, every previously non-validated message now becomes valid

https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd

The Nomad Bridge Hack

Another crypto bridge attack: Nomad loses $190 million in ‘chaotic’ hack

By Jennifer Korn

Published 12:39 PM EDT, Wed August 3, 2022

How a crypto bridge bug led to a $200m 'decentralized crowd looting'

Flash mob exploits Nomad's validation code blunder

Hackers Return $9M to Nomad Bridge After $190M Exploit

The popular Ethereum to Moonbeam bridge is working with law enforcement and data analytics firms.

By Oliver Knight   Aug 3, 2022 at 10:52 a.m. GMT   Updated Aug 3, 2022 at 2:53 p.m. GMT

samczsun @samczsun · Aug 2

Replying to @samczsun

11/ This is why the hack was so chaotic - you didn't need to know about Solidity or Merkle Trees or anything like that. All you had to do was find a transaction that worked, find/replace the other person's address with yours, and then re-broadcast it

24   129   1,062

The Nomad Bridge Hack

QSP-19 Proving With An Empty Leaf

Severity: Low Risk

Status: Acknowledged

File(s) affected: `Replica.sol`

Description: The function `Replica.sol:prove` accepts the input `_leaf` and checks if it is part of the Merkle tree. Nomad architecture uses a sparse Merkle tree, in which all the non-used leaves default with empty `bytes32`. This nature of the sparse Merkle tree makes it possible for one to pass an empty `bytes32` as the `_leaf` and some artificial Merkle proof with a specified index to pass the inclusion check. The "empty leaf" message status can later be flagged as `PROVEN`, resulting in the `messages` mapping in an undesired state.

Recommendation: Validate that the `_leaf` input of the function `Replica.sol:prove` is not empty.

Update: The Nomad team responded that "We consider it to be effectively impossible to find the preimage of the empty leaf". We believe the Nomad team has misunderstood the issue. It is not related to finding the pre-image of the empty bytes. Instead, it is about being able to prove that empty bytes are included in the tree (empty bytes are the default nodes of a sparse Merkle tree). Therefore, anyone can call the `prove` function with an empty leaf and update the status to be proven.

https://github.com/nomad-xyz/docs/blob/1ff0c55dba2a842c811468c57793ff9a6542ef0f/docs/public/Nomad-Audit.pdf

The Nomad Bridge Hack - Lessons

- Always **check user input** thoroughly
  - Especially for empty values
- **Every object** has a value
  - Even if never accessed before, it has a **zero** value
- When an auditor flags a bug, **fix it**

Binance Bridge Hack

- Binance Bridge used a sophisticated implementation of AVL Merkle Trees
  - AVL trees: self-balancing binary search trees
  - In this implementation, verification contains special *operations* that need to succeed
  - Root hash is computed in a pretty complex manner (source code)
- Attacker
  - Changed a leaf's value, inserting the malicious payload
  - Added an inner node in a way that verification for original MTR passed

https://twitter.com/samczsun/status/1578167198203289600

Binance Bridge Hack

Binance hit by $100 million blockchain bridge hack

Carly Page @carlypage_ / 2:36 PM GMT+1 • October 7, 2022

Binance Hit By $570 Million Blockchain Bridge Hack

By RAHUL NAMBIAMPURATH Published October 07, 2022

Key takeaways

- The world's largest crypto exchange, Binance, had to suspend deposits and withdrawals due to a hack.
- BNB is the fifth largest crypto by market cap, and the hack was for 2 million BNB tokens, which resulted in $570 million.

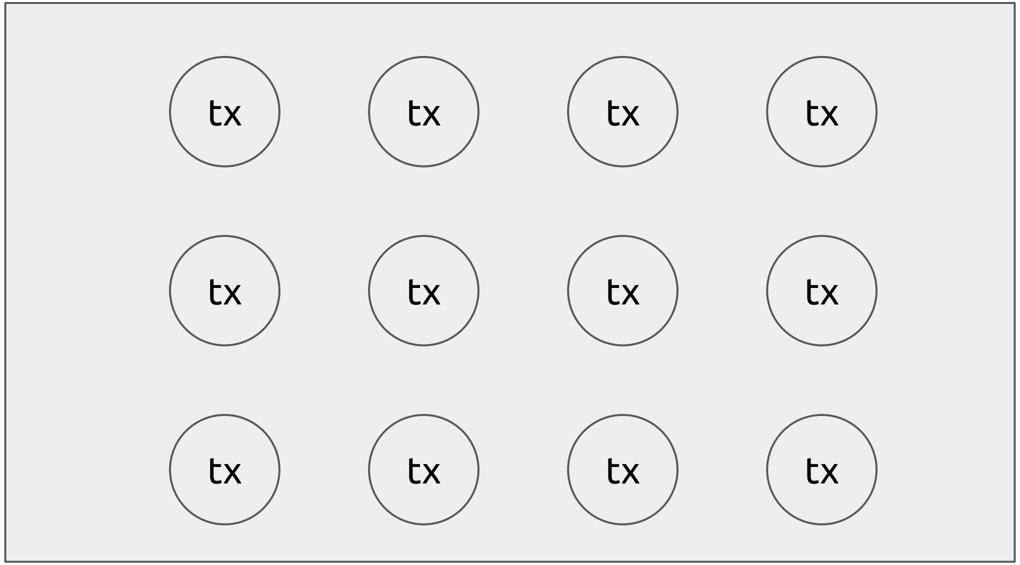
Binance Bridge Hack

- Binance Bridge used a sophisticated implementation of AVL Merkle Trees
  - AVL trees: self-balancing binary search trees
  - In this implementation, verification contains special *operations* that need to succeed
  - Root hash is computed in a pretty complex manner (source code)
- Attacker
  - Changed a leaf's value, inserting the malicious payload
  - Added an inner node in a way that verification for original MTR passed
- Lessons:
  - **Keep it simple**
  - **Don't roll your own crypto**

https://twitter.com/samczsun/status/1578167198203289600

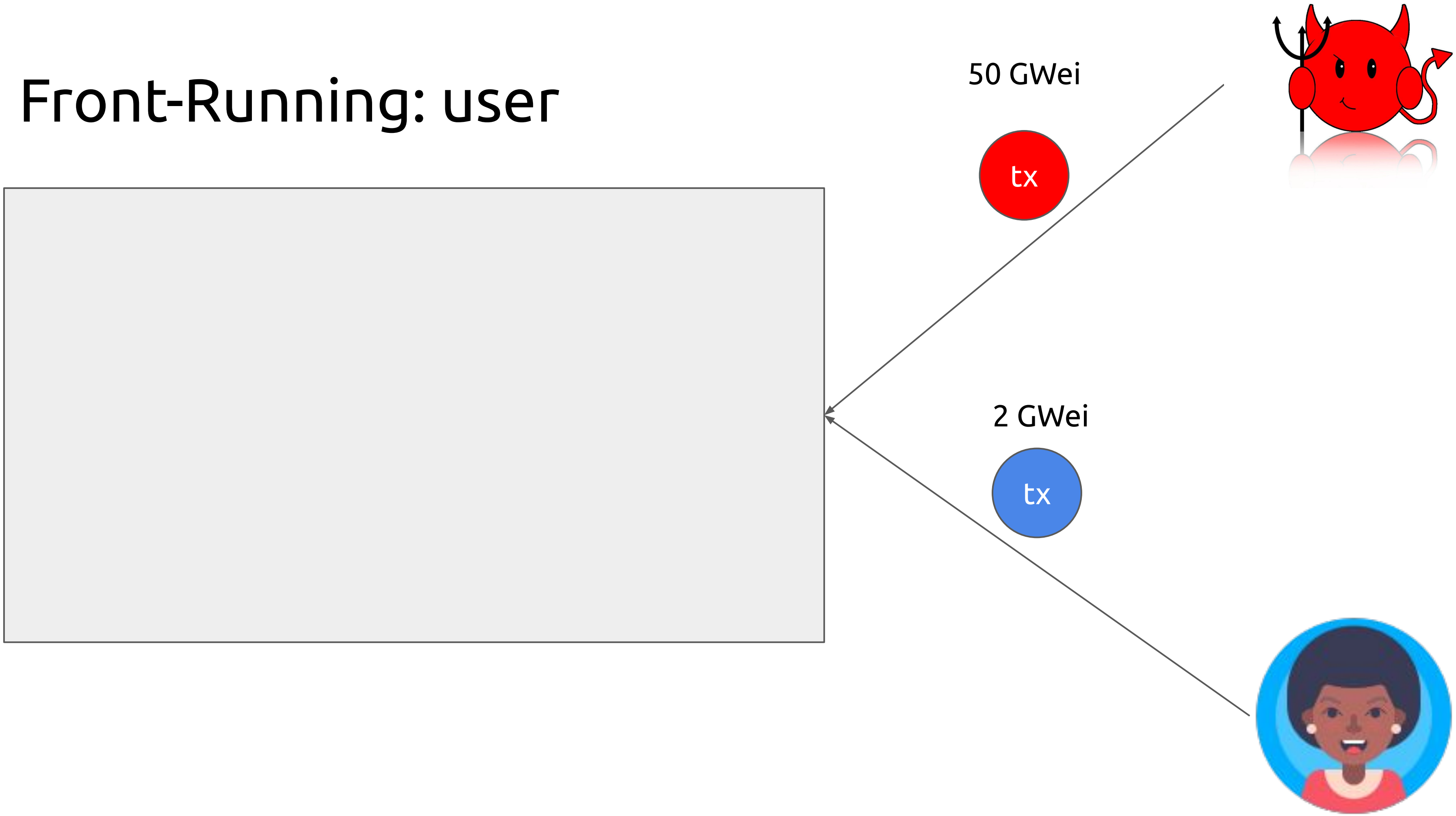
Front-running

Front-Running

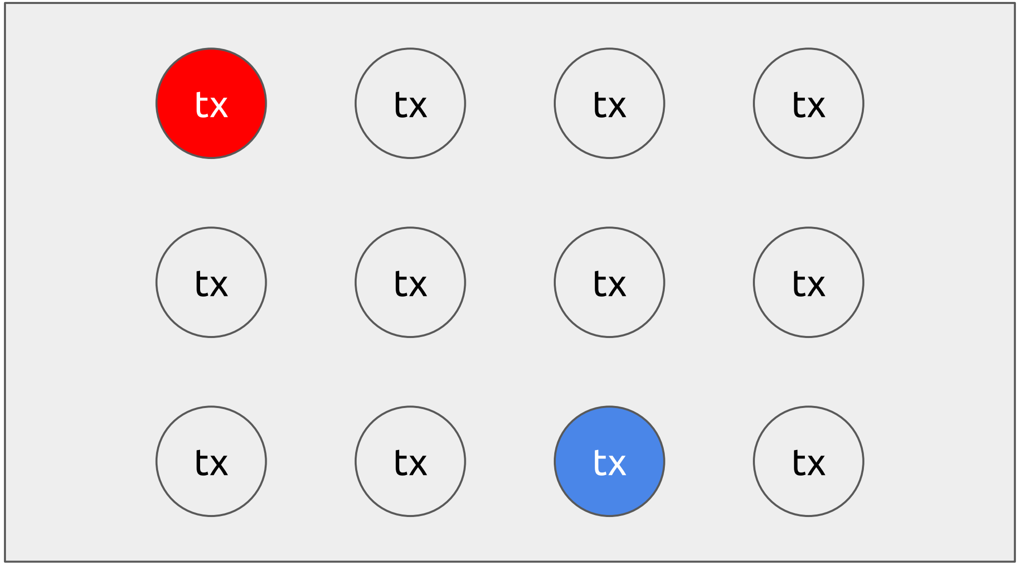


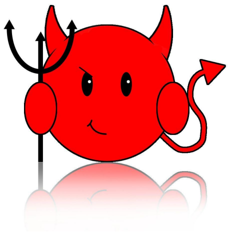
Miner: `sortByGasPrice(txs, ‘desc’)`

Front-Running: user

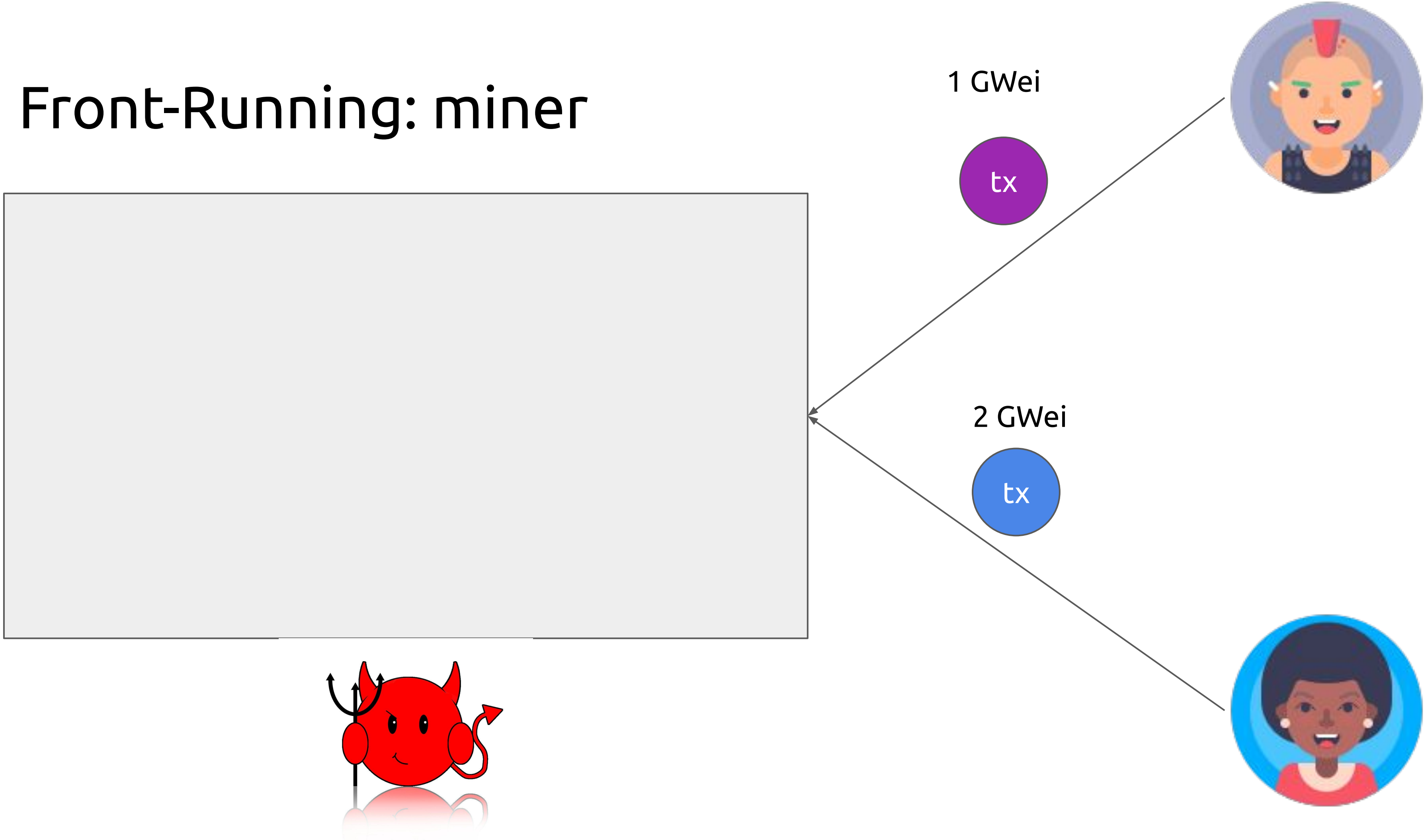


Front-Running: user

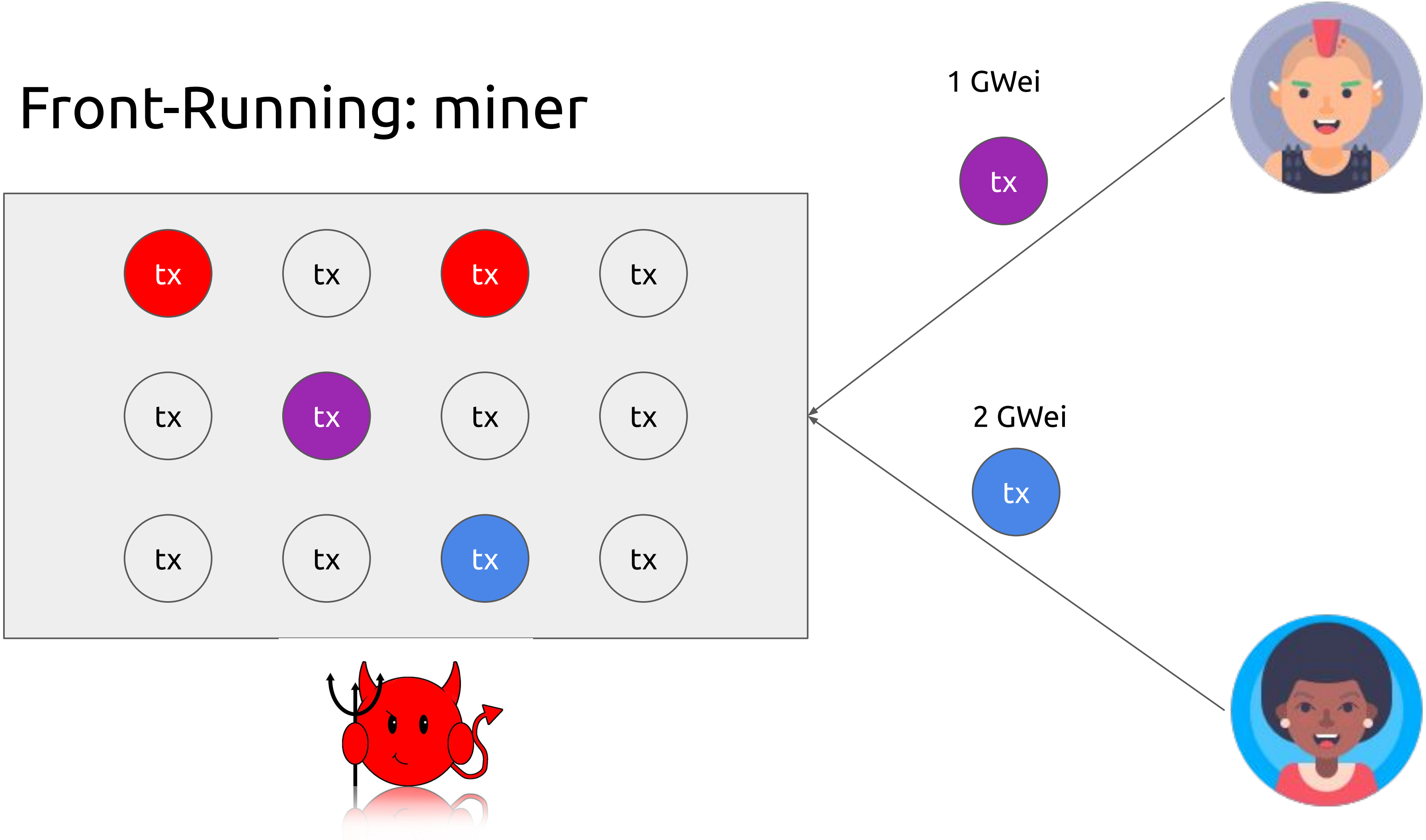




Front-Running: miner



Front-Running: miner



Front-Running: example

```
// INSECURE

function registerName(bytes32 name) public {

    names[name] = msg.sender;

}
```

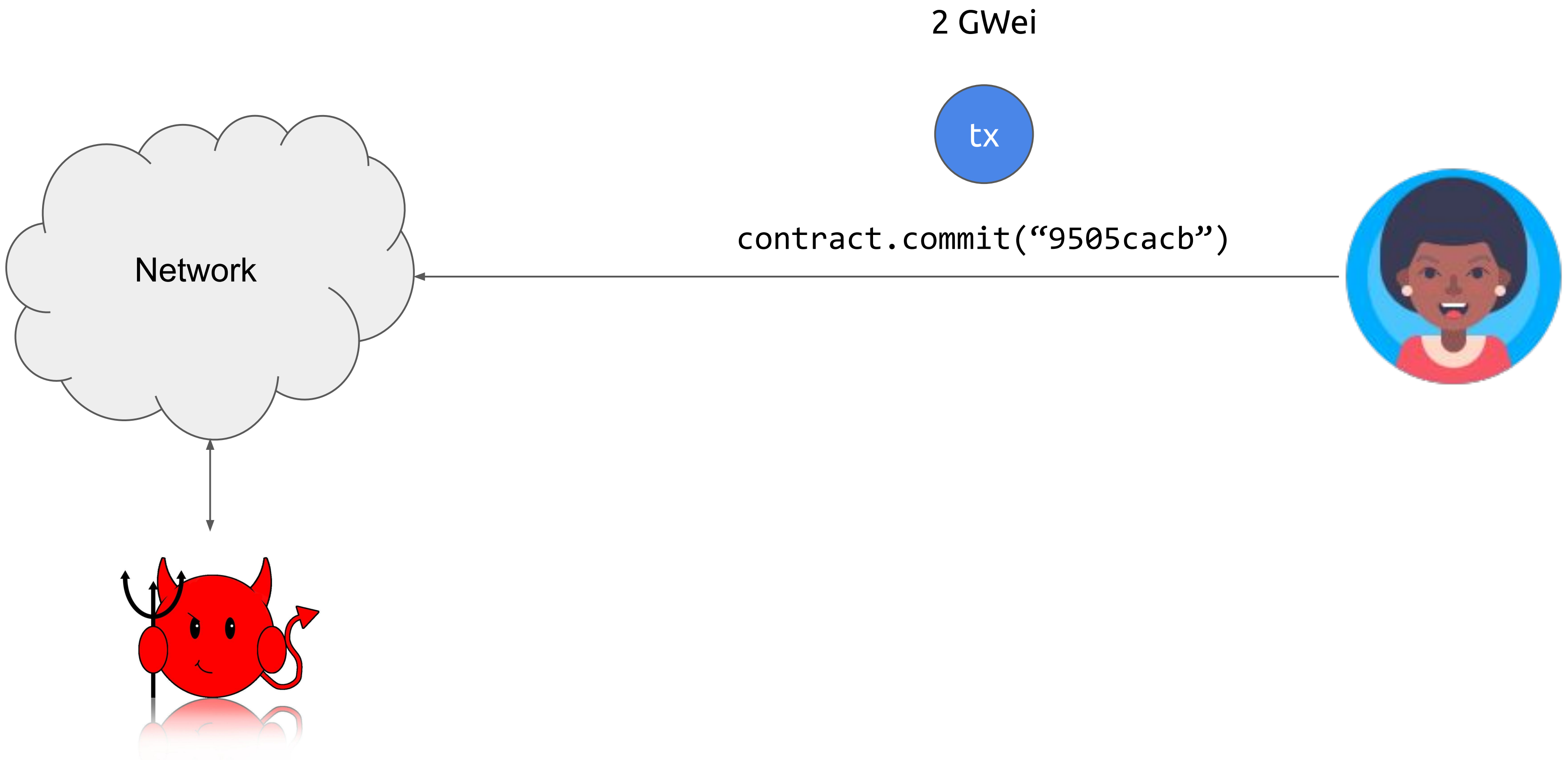
Front-Running: solution

- Employ a cryptographic **commitment scheme**
- Implementation
    - commit: c = hash(<value, nonce>) *(Note: nonce space should be large!)*
    - reveal: v = <value’, nonce’>
    - verify: c == hash(v)
- Properties
    - **Binding**: a commitment can be opened only to its committed value
    - **Hiding**: a commitment reveals no information about its committed value

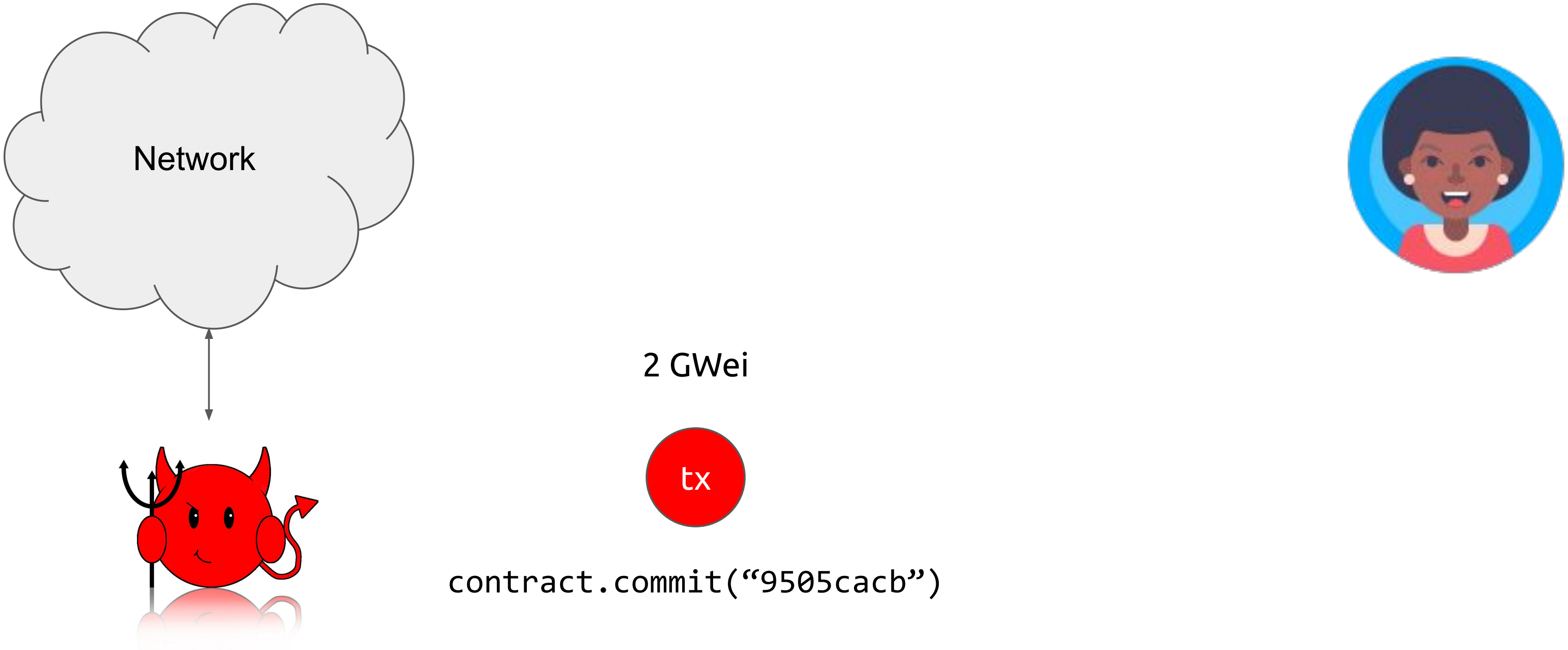
Front-Running: solution example

```
// INSECURE

function registerName(bytes32 name) public {

     names[name] = msg.sender;

}


// MORE SECURE, BUT…

function registerName(bytes32 name, bytes32 nonce) public {

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

     names[name] = msg.sender;

}
```

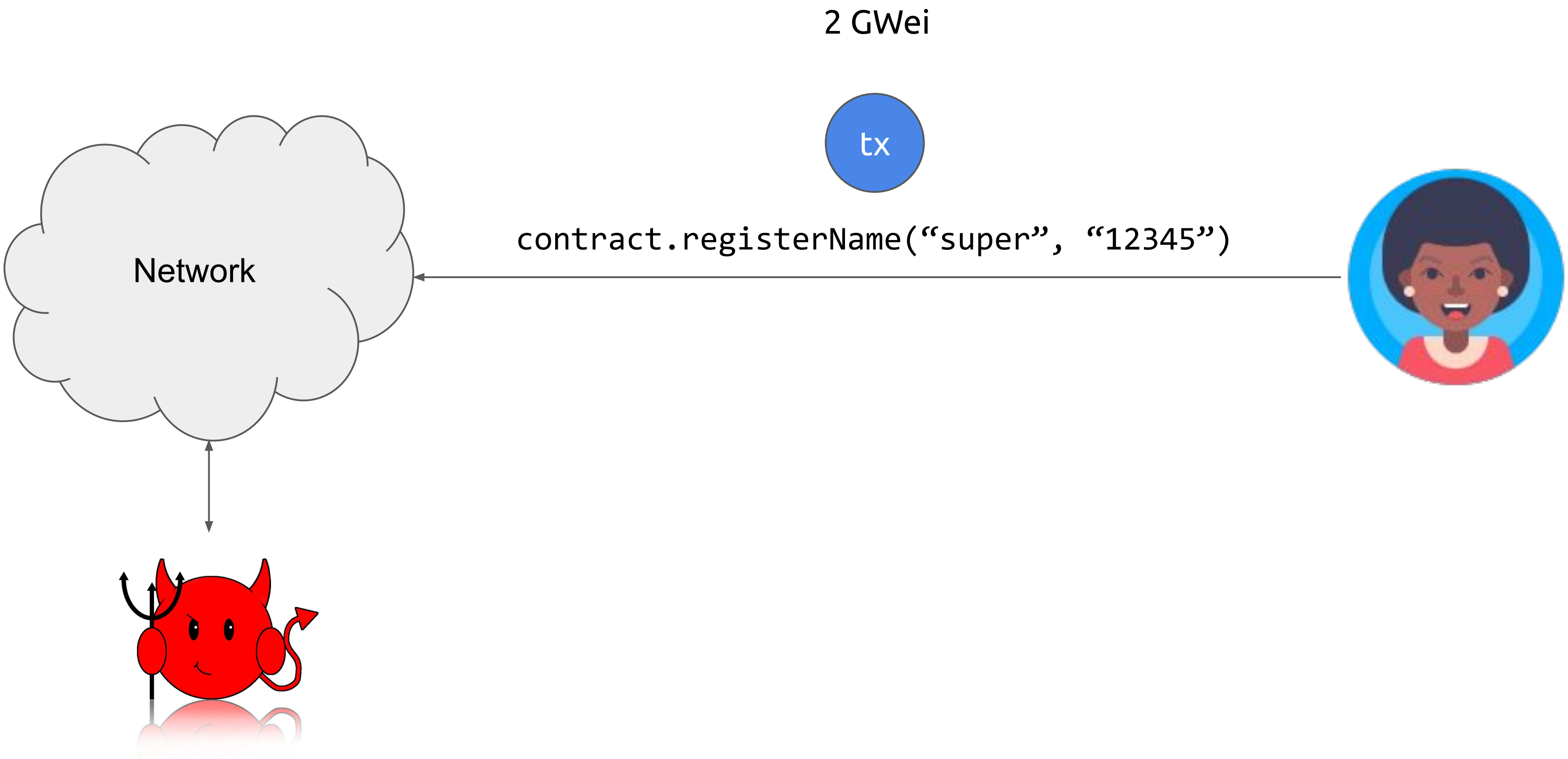
Front-Running: example



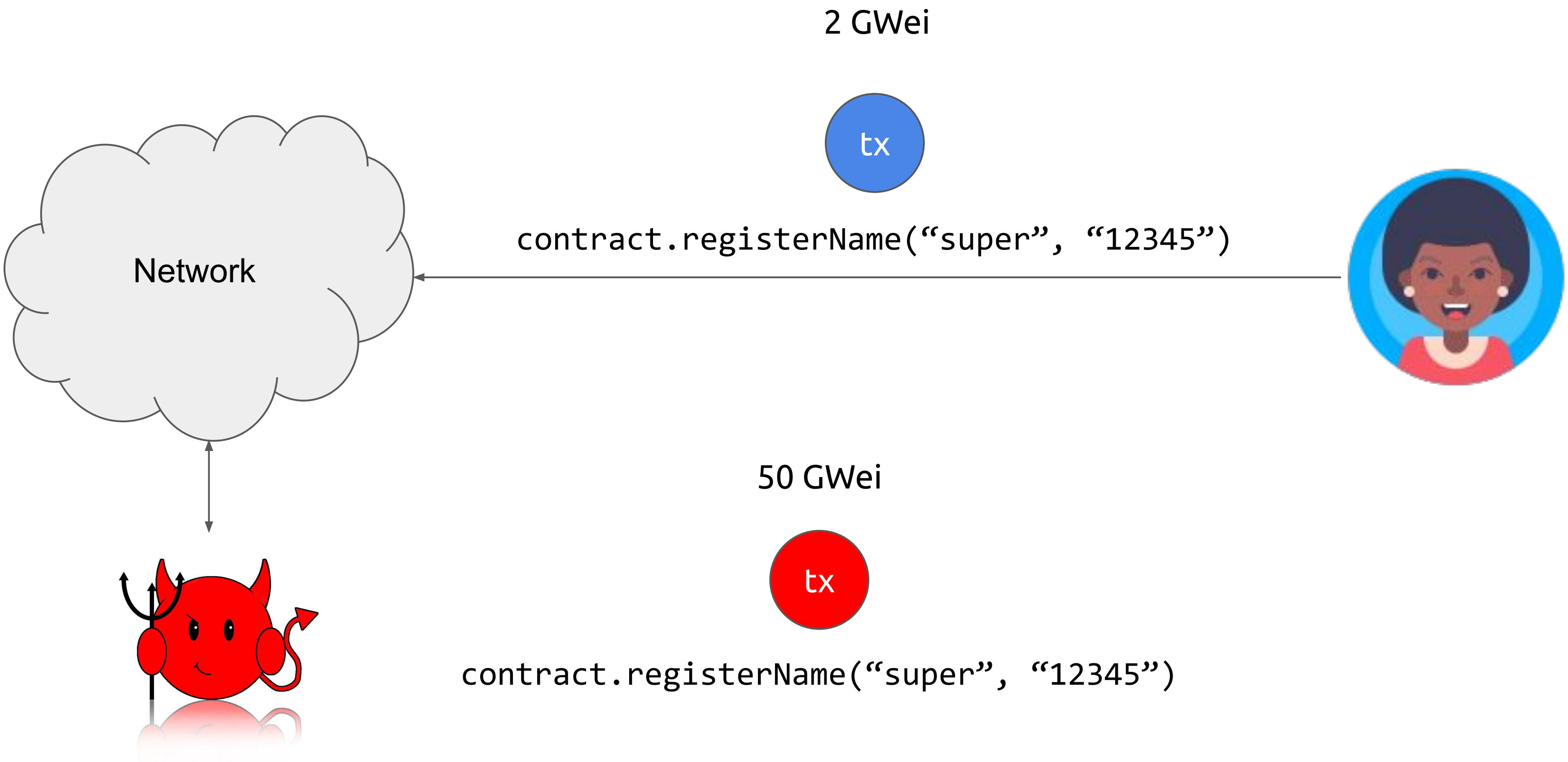
Front-Running: example



Front-Running: example



Front-Running: example



Front-Running: another solution

- Employ a cryptographic **commitment scheme**
- **Keep track** of committed values
  - Prevent a user from posting a commitment already posted by another user
- Possible **DoS** and **forced gas cost**
  - Attacker can front-run a user's commit operation and post the commitment as their own
  - User is forced to spend extra gas for new tx that posts new commitment
  - Attacker can continue front-running until they run out of money (to pay gas)

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.time p
- block.has
- block.diff y
- lock.coinbase
- k.gasLimit
- m
- m 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

Not really private

Also not 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 really 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: intra-transaction information leak

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

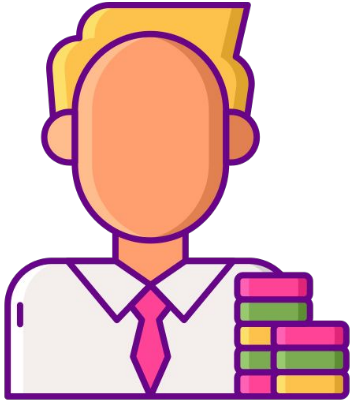
https://media.dedaub.com/bad-randomness-is-even-dicier-than-you-think-7fa2c6e0c2cd

Sources of randomness

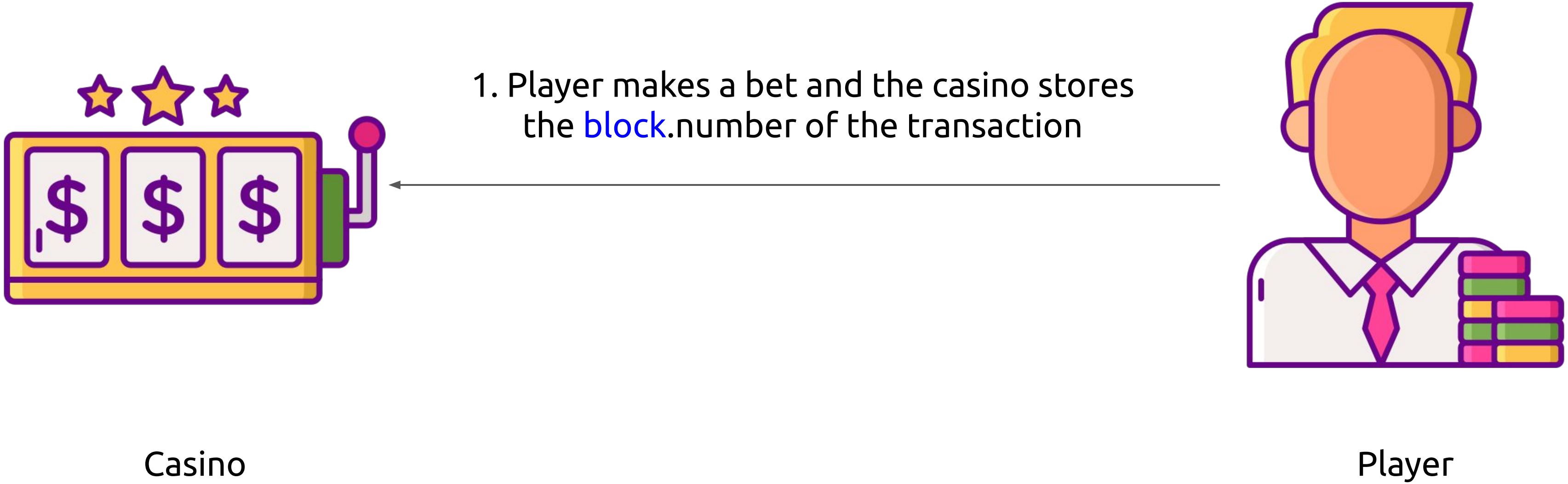
- **Block information** can be **manipulated by miner**
- Block information **shared** by all users in the same block
- In Ethereum, **all data** posted on the chain are **visible**
- “private” vars are only private w.r.t. object-oriented programming **visibility**
- If same-block txs share randomness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

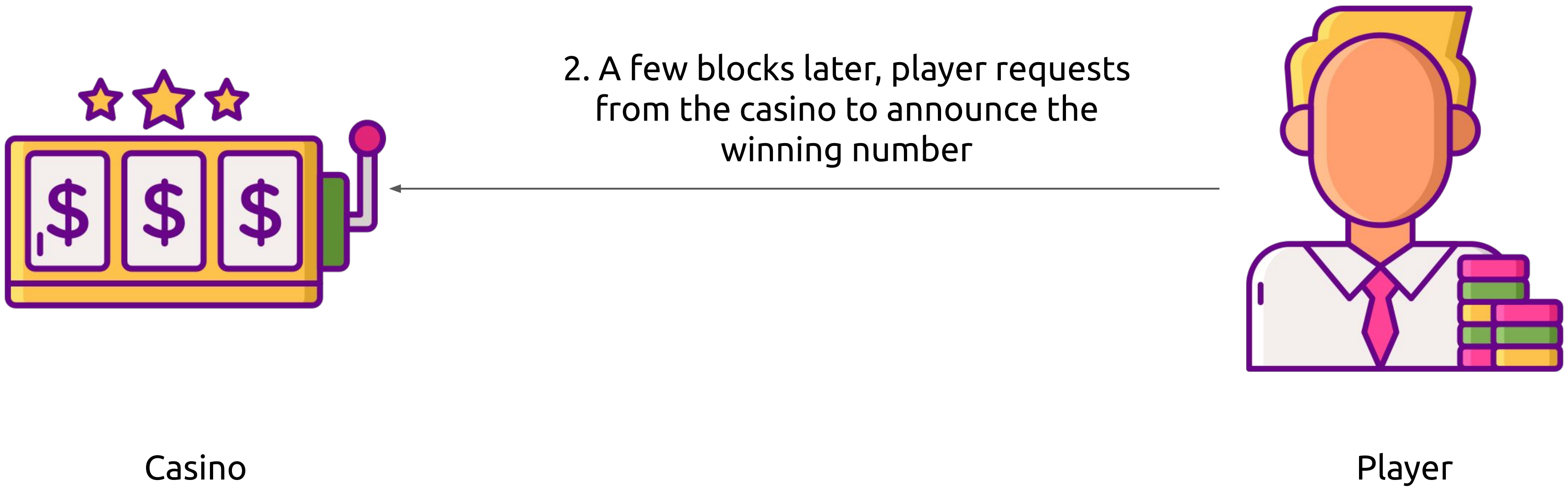
What about future blocks ?

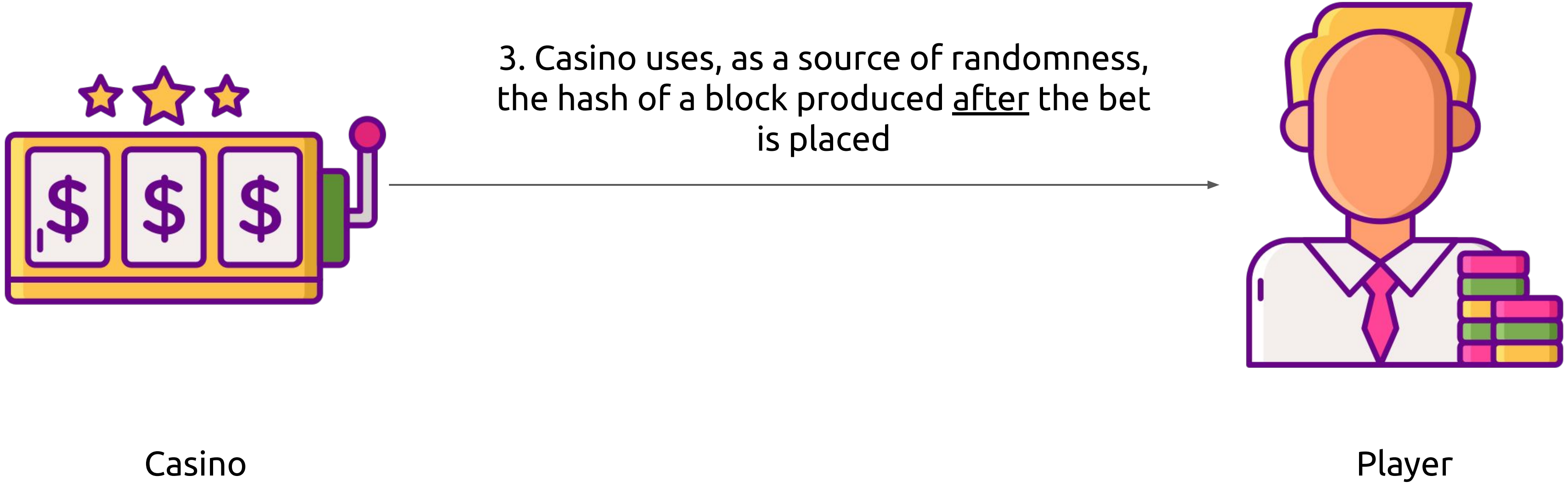
Casino



Player


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

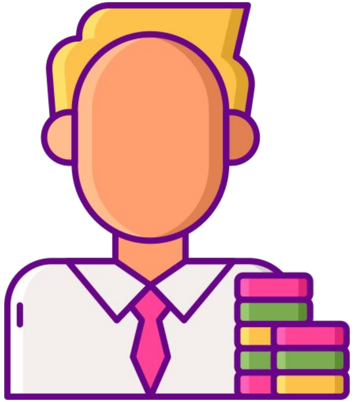

2. A few blocks later, player requests from the casino to announce the winning number
Casino
Player


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

Validate block.number age!

3. Casino uses, as a source of randomness, the hash of a block 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
- With some probability (how high?), a malicious **miner** will **create the specific future block**
- In PoS, the **proposer** of a future block might be **known beforehand**
- A **miner** can keep newly-mined **blocks hidden**, until they mine a favorable one

Randomness: towards safer PRNG

- Commitment schemes
  - Prover commits to a message *m* by publishing *h = H(m)* (H is a hash function)
  - After some time, prover reveals message *m*
  - Verifier wants to be sure that the originally committed message is the revealed one
    - Verifier checks that: *h == H(m)*
  - Binding property:
    - Collision resistance: it should be infeasible to find m’ s.t. H(m) == H(m’)
  - Hiding property:
    - Honest prover wants no information about *m* to be retrievable from *H(m)*
    - H needs to behave as a random oracle
    - *m* should be unpredictable; if domain is small, use salt

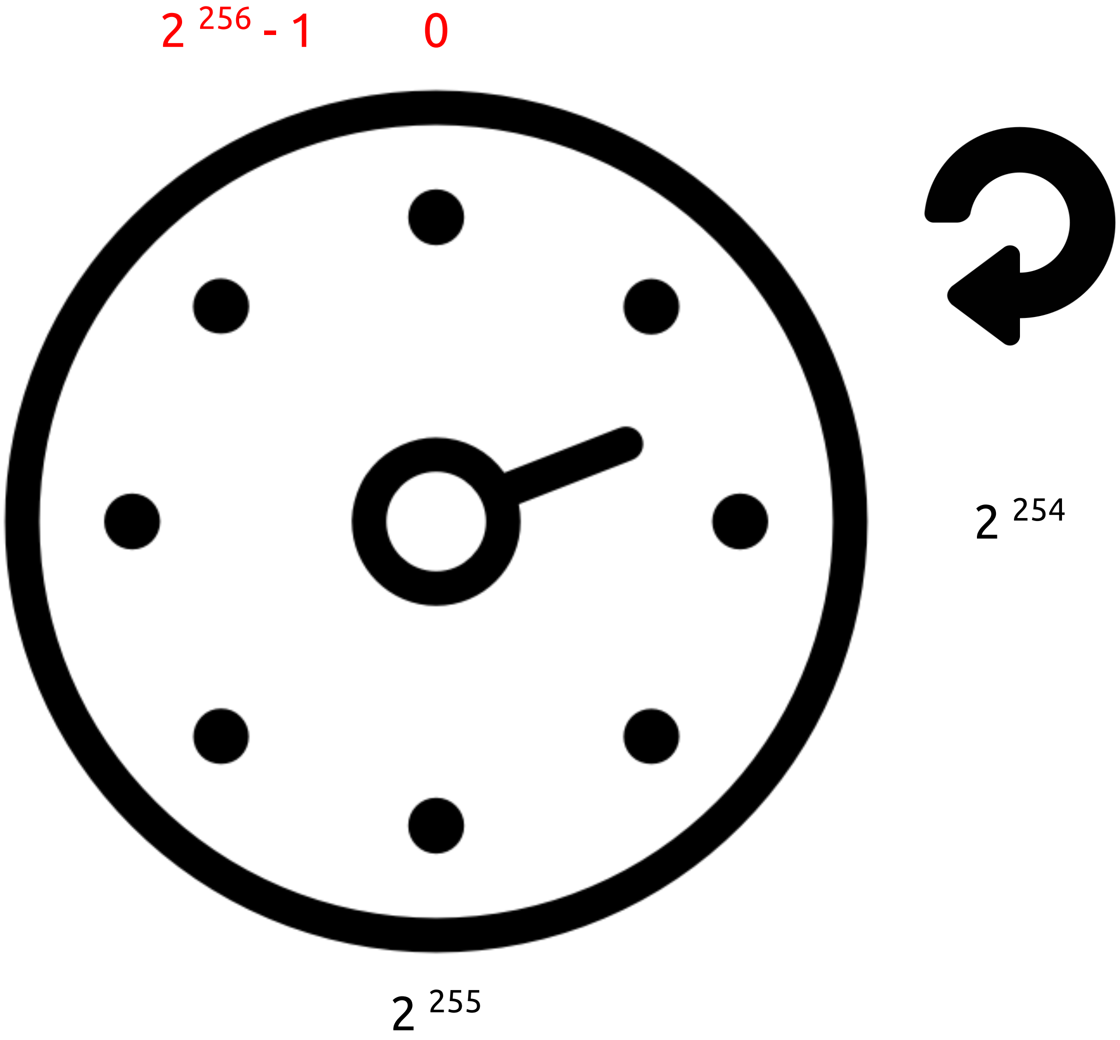
Randomness: towards safer PRNG

- Commitment schemes
- Example:
  - Casino and player each commit to a random value
  - Casino and player reveal their values
  - Casino XORs the random values to produce a seed
    - the seed can also be combined with the hash of a future block
  - If *either* casino *or* player honest, then the seed is random (why?)

On-chain data is public

- Applications (games, auctions, etc) required **data** to be **private** up until some point in time
- Every data that is published on-chain is **visible** by everyone
- Best strategy: **commitment schemes**
- Watch out for front-running!

Overflow/Underflow


2^256 - 1
0
2^254
2^255

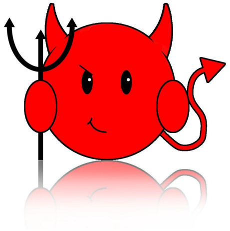
Integer Overflow and Underflow

```
// 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

Solidity 0.8+ protects natively against over/underflows.

For older versions, 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;
}
```

(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

```
pragma solidity >=0.7.0 <0.9.0;

contract RockPaperScissors { // Winner gets 1 ETH
    struct round {
        address payable player;
        bytes32 commitment;
        uint256 hand;
    }
    round[] private rounds;

    function commit(uint256 hand) payable public {
        require((hand == 1 || hand == 2 || hand == 3) && (rounds.length < 2));
        rounds.push(round(payable(msg.sender), sha256(abi.encode(hand)), 0));
    }

    function open(uint256 hand) public {
        require(rounds.length == 2);
        for (uint256 i = 0; i < 2; i++) {
            if (rounds[i].commitment == sha256(abi.encode(hand))) {
                rounds[i].hand = hand;
            }
            if (rounds[(i + 1) % 2].hand == 0) {
                return;
            }
        }
        if ((rounds[0].hand == 1 && rounds[1].hand == 2) ||
            (rounds[0].hand == 2 && rounds[1].hand == 3) ||
            (rounds[0].hand == 3 && rounds[1].hand == 1)) {
            rounds[0].player.transfer(1 ether);
        }
        else if (rounds[0].hand != rounds[1].hand) {
            rounds[1].player.transfer(1 ether);
        }
        selfdestruct(payable(msg.sender));
    }
}
```