LEARN SMART CONTRACT PROGRAMMING IN 1 HOUR

Sichao Yang

Co-Founder and CEO of Nakamoto & Turing Lab

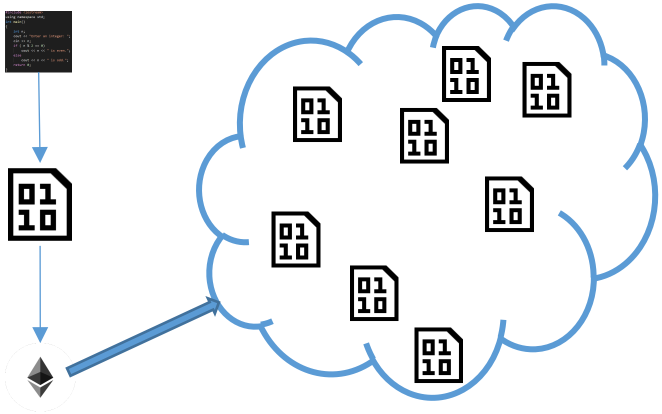
June 17, 2020

What is a smart contract?

What is a smart contract?

Executable file

Smart contract



Fundamentally, a “smart contract” is a set of coded computer functions.

• May incorporate the elements of a binding contract (*e.g.*, offer, acceptance, and consideration), or may simply execute certain terms of a contract.

•Allows self-executing computer code to take actions at specified times and/or based on reference to the occurrence or non-occurrence of an action or event (*e.g.*, delivery of an asset, weather conditions, or change in a reference rate).

Key Attributes of a Smart Contract

NAKAMOTO &TURING LABS

Can authenticate counter party identities, ownership of assets and claims of right

Can access or refer to outside information or data to trigger action(s)

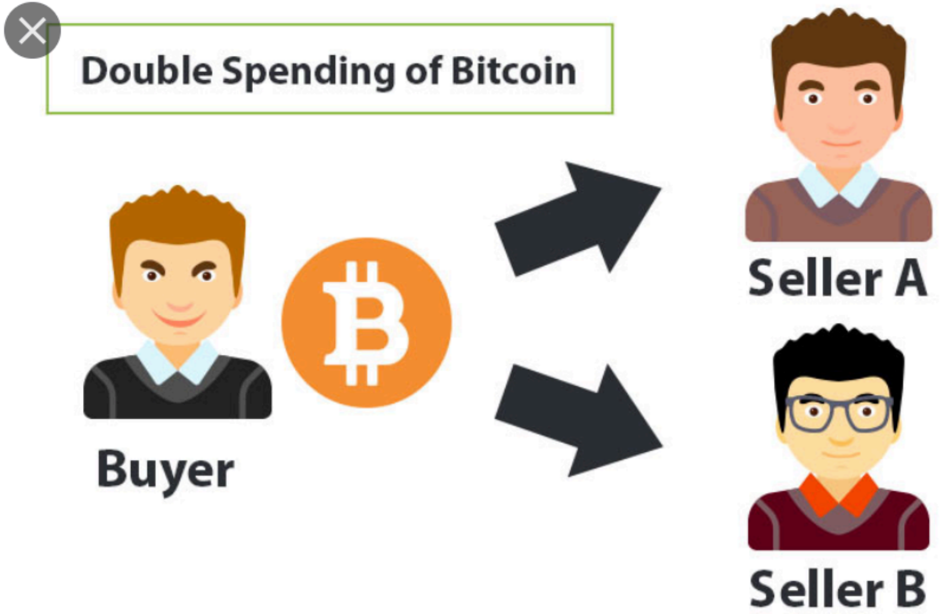
Can automate execution processes

Smart contracts use ***digital signatures*** – private cryptographic keys held by each party to verify participation and assent to agreed terms.

Smart contracts use ***oracles*** – a mutually agreed upon, network- authenticated reference data provider (potentially a third-party); this is a source of information to determine actions and/or contractual outcomes, for example, commodity prices, weather data, interest rates, or an event occurrence.

Self-execution: A smart contract will take actions, *e.g.*, disperse payments, without further action by the counterparties.

Why blockchain?



The blockchain technology (more specifically, the consensus algorithm) determines the order of transaction.

Smart contract development cycle

Steps to be taken to develop a secure smart contract

NAKAMOTO &TURING LABS

1. Understand the use-case of smart contract.
2. Create a basic architecture of smart contracts interaction or flowchart how functions will interact with each other.
3. Start development using any IDE or development tools like Truffle, remix with proper documentation of each and every function.
4. Once the development is completed start testing smart contracts on test-net or private blockchain.(this is called a manual testing).
5. Record all the transaction while testing on test-net, analyze results of all transactions with actual use case or business logic of smart contract.
6. Unit testing will be the next step in smart contract development life cycle, there are multiple frameworks for unit and integration testing that can be use to test smart contract. Example : Truffle framework.
7. Once unit testing is done using truffle framework on ganache, smart contract author should go for 3rd party Audit of smart contract.
8. Last but not the least, bug bounty programs are also very efficient to secure smart contracts. Communities like 0x protocol is offering $100,000 in bounty programs.

Understand Use case

NAKAMOTO &TURING LABS

Use case of smart contracts should be clear before development is started, developer should gather all the information of smart contracts like business logic, also all the 3rd party libraries that developer will use while developing a smart contract.

Use cases

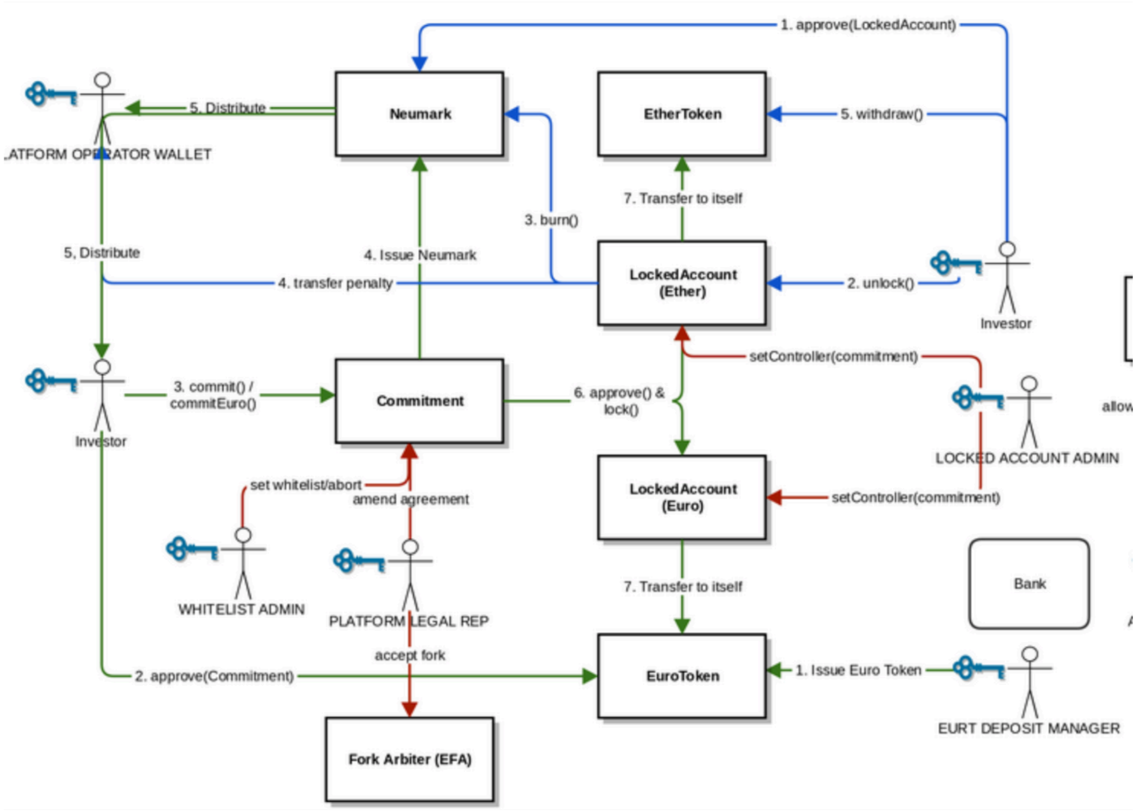
- Trading
- Records
- Voting
- Game
- etc.

3rd party libraries

- Consensus Lab
- Openzeppelin

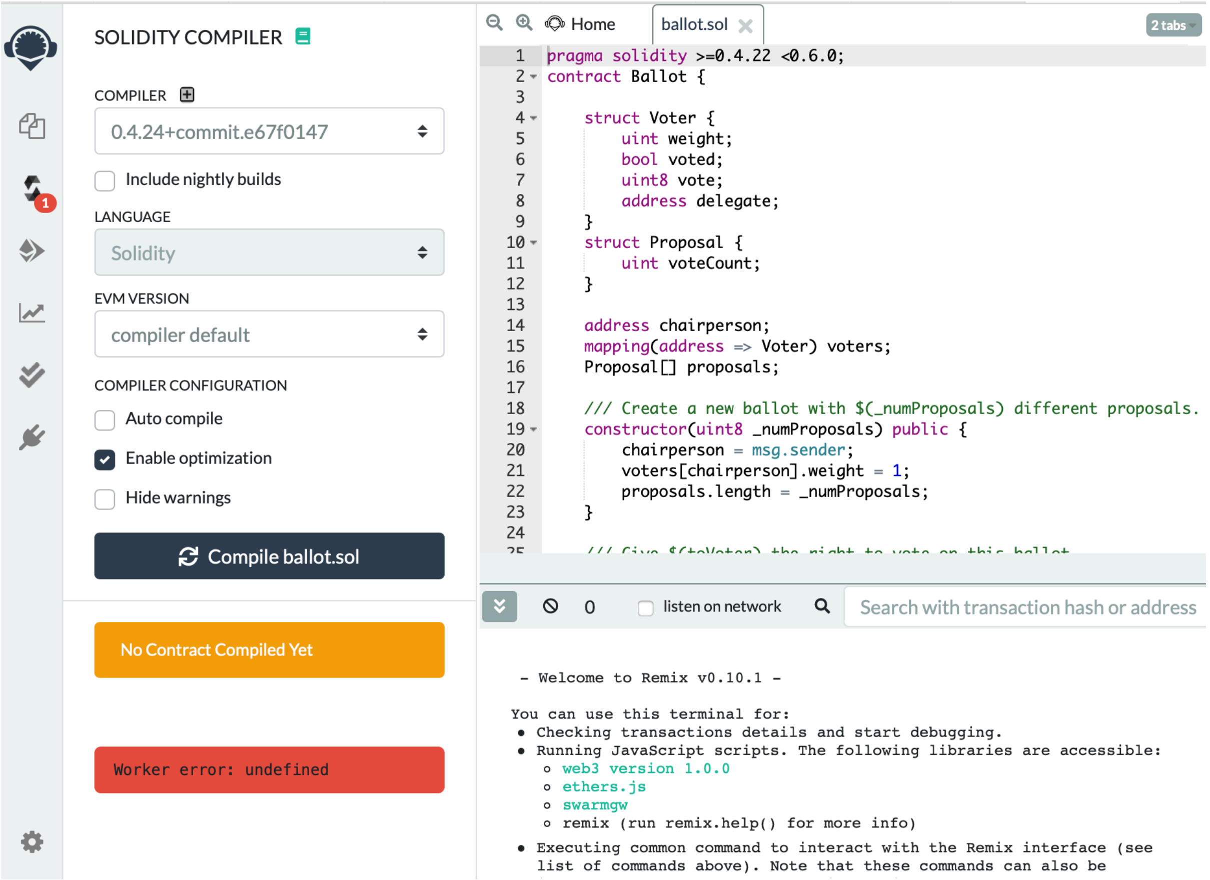
Architecture design

A basic architecture depicts the business logic of smart contract. Architecture design in the initial phase help developers to follow the exact path during development phase.



Development phase

In this phase actual development is started, developer can use any Code editor or IDE to develop a smart contract.



Manual Testing

In this phase smart contract should be tested well on test-net (Rinkeby/Ropsten), all the transaction and state changes should be recorded to verify that smart contract's behavior is same that intend to be.

Transaction hashes

Network : rinkeby

Contract creation : 0x8f5502fb08f74cef7f2ecbd37afa36317bc3e39865fe229bf5a2f37d361208ec

Transfer tokens : 0x6ab77473cf529db7f209a46c53f81974f75192064cf004265fbdef61dd2a7716

Approve : 0xf5f5729791b87e48667da71d0c7f0b1f8d90ba23afc07376e50d5c471e026b73

Send ether to token contract (Should fail):
0xd439077b3707154b39a64bdd62f7a0973e5590d9ec83712f2bf696d6631500e0

transferFrom :0x1914e66e3ced288ecb7c9bed412b6339d344f55881abf7abb66529a30ec43059

transferOwnership : *0xd374fc752aeaa0c0fd4c39891cbfbe597d7c5e1c89f608a4738f2787bba2a863*

acceptOwnership: 0x4ec5bdbe76d4280ca4c66be5a0562e789d2d0bcb6668bfbff2245a06bc5bc430

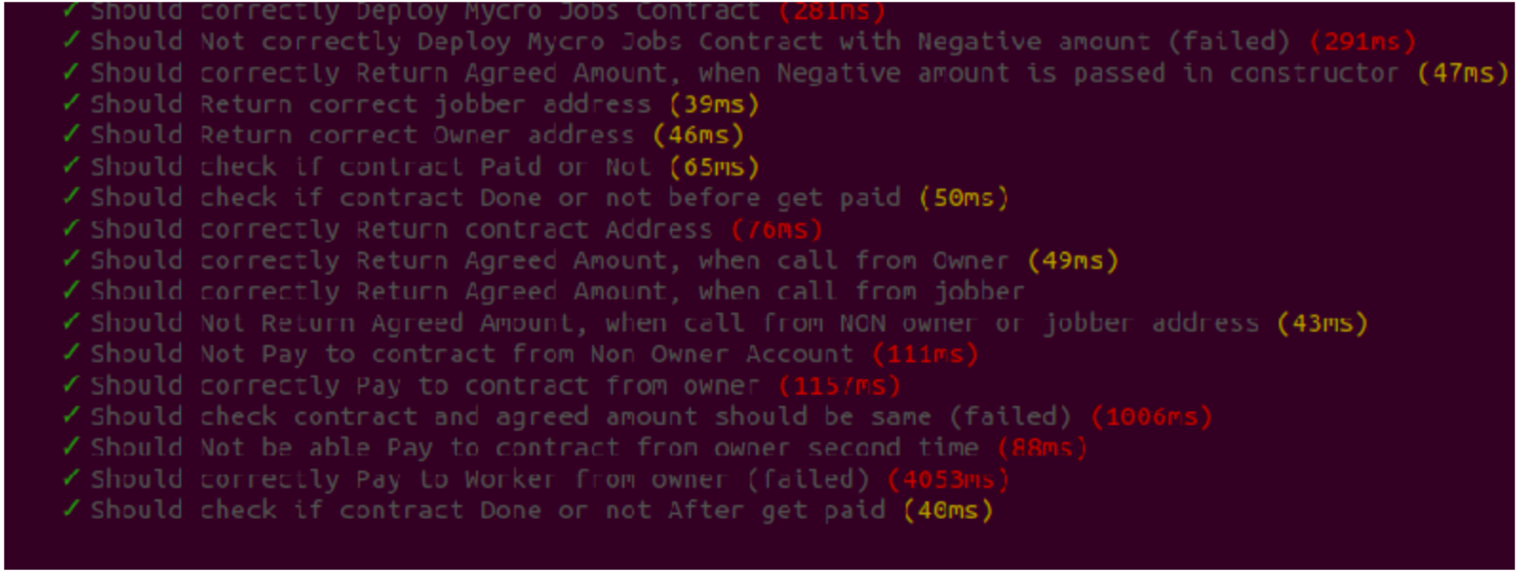
Burn tokens : 0x99375a0f74d872ef83c282b98b7294aa414174acef96deb0586c3b7d19fcb092

Tokens(Sample tokens) transfer to MM token contract
0x7d69373b29dd6281ff987286dcdca172063ab915f2d402fa5d9b615a1afa3e6d

transferAnyERC20Tokens
0xc54934d8ef31a8f56868ed82439df6c4c512c3365c8623620afc636df239a873

Unit Testing

Unit testing can be done using [truffle]() framework, developer should write test cases for all the functions of smart contract, test cases should reflect correct the business logic of smart contract.



3rd party security audit

All the Smart contracts ready for production should be audited before deploying on main net, because even though business logic of smart contracts is tested on test-net several times, smart contract cannot be declared as secured or bug free contract, smart contract may contain some logical errors that can be identified by audit.

The cost could be **> $100,000.**

QuillAudits

CERTIK

Quantstamp

Bug Bounty

NAKAMOTO
&TURING
LABS

Bug bounty programs are very useful in identifying bugs in smart contract, as your smart contract will come under the eye of multiple experienced auditors or developers to find the loopholes in smart contract, even after two successful 3rd party audits, 0x protocol project have also conduct a bounty program in order to find the potential bug in smart contract.

Other tools

NAKAMOTO &TURING LABS

Code coverage

- code coverage is a special tool that evaluate how efficient your test cases

File		Statements		Branches		Functions		Lines	
ERC20.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
ERC20Basic.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
NoOwner.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
ACAToken.sol		98.65%	73/74	59.62%	31/52	95.45%	21/22	97.01%	65/67
SafeMath.sol		91.67%	11/12	50%	4/8	100%	4/4	91.67%	11/12
ACATokenSale.sol		91.36%	222/243	53.16%	101/190	93.85%	61/65	90.43%	208/230

Up-gradable Smart Contracts

- Tradeoff: decentralization v.s. convenience

Solidity programming

Basic Features

Similar to Java, Javascript, Python

- Comments
- Primitive Types
- Strings
- Arrays
- Statements
- Boolean, Conditional, and Arithmetic Expressions
- Loops
- Variables
- Literals
- Methods

Gas costs

NAKAMOTO &TURING LABS

- Every theorem block has a gas limit
- The sender of a transaction has to pay the gas cost
- The best practice is to write an optimized code that uses a minimum amount of gas.
- The amount of gas you will use during a transaction depends on

  1. The data location of variables
  2. The algorithm complexity

Data Locations – Storage

NAKAMOTO &TURING LABS

Storage

The storage location is permanent data, which means that this data can be accessed into all functions within the contract. To make it more simple, you can think of it as the hard disk data of your computer where all the data gets stored permanently. Similarly, the storage variable is stored in the state of a smart contract and is persistent between function calls. Keep in mind that storage data location is expensive compared to other data locations.

Data Locations – Memory

NAKAMOTO &TURING LABS

Memory

The memory location is temporary data and cheaper than the storage location. It can only be accessible within the function. Usually, Memory data is used to save temporary variables for calculation during function execution. Once the function gets executed, its contents are discarded. You can think of it as a RAM of each individual function.

Data Locations – Calldata

Calldata

Calldata is non-modifiable and non-persistent data location where all the passing values to the function are stored. Also, Calldata is the default location of parameters (not return parameters) of external functions.

Data Locations – Stack

Stack

Stack is a non-persistent data maintained by EVM (Ethereum Virtual Machine). EVM uses stack data location to load the variables during execution. Stack location has the limitation up to 1024 levels.

Data Locations – Default Rules #1

State variables are always in storage

```
pragma solidity ^0.5.0;

contract DataLocation {

    //storage
    uint stateVariable;
    uint[] stateArray;
}
```

You can not explicitly override the location

```
pragma solidity ^0.5.0;

contract DataLocation {

    uint storage stateVariable; // error
    uint[] memory stateArray; // error
}
```

Data Locations – Default Rules #2

Function parameters including return parameters are stored in memory

```
pragma solidity ^0.5.0;

contract Location {

    uint stateVariable;
    uint[] stateArray;

    function calculation(uint num1, uint num2) public pure returns (uint result){
        return num1 + num2;
    }
}
```

Data Locations – Default Rules #3

Local variables with a value type are stored in the memory.

However, for a reference type, you need to specify the data location explicitly.

```
pragma solidity ^0.5.0;

contract Locations {

  /* these all are state variables  */

  //stored in the storage
  bool flag;
  uint number;
  address account;

  function doSomething() public  {

    /* these all are local variables  */

    //value types
    //so they are stored in the memory
    bool flag2;
    uint number2;
    address account2;

    //reference type
    uint[] memory localArray;
  }
}
```

Data Locations – Default Rules #4

Function parameters (not including returns parameters) of external function are stored in the Calldata.

Data Locations – Default Copy Behavior #5

Assignment of one state variable to another state variable creates a new copy

```
pragma solidity ^0.5.0;

contract Locations {

  uint public stateVar1 = 10;
  uint stateVar2 = 20;

  function doSomething() public returns (uint) {

    stateVar1 = stateVar2;
    stateVar2 = 30;

    return stateVar1; //returns 20
  }
}
```

Data Locations – Default Rules #6

Assignment to storage state variable from a memory variable always creates a new copy.

```
pragma solidity ^ 0.5.0;

contract Locations {

    uint stateVar = 10; //storage

    function doSomething() public returns(uint) {

        uint localVar = 20; //memory
        stateVar = localVar;
        localVar = 40;

        return stateVar; //returns 20
    }
}
```

Data Locations – Default Rules #7

Assignment to a memory variable from state storage variable will create a copy.

```
pragma solidity ^ 0.5.0;

contract Locations {

    uint stateVar = 10; //storage

    function doSomething() public returns(uint) {

        uint localVar = 20; //memory

        localVar = stateVar;
        stateVar = 40;

        return localVar; //returns 10
    }
}
```

Data Locations – Default Rules #8

Assignment from one memory variable to another memory variable will not create a copy. This is applicable to reference type variables only. Local variable still creates a new copy.

```
pragma solidity ^ 0.5.0;

contract Locations {

    function doSomething()
        public pure returns(uint[] memory, uint[] memory) {

        uint[] memory localMemoryArray1 = new uint[](3);
        localMemoryArray1[0] = 4;
        localMemoryArray1[1] = 5;
        localMemoryArray1[2] = 6;

        uint[] memory localMemoryArray2 = localMemoryArray1;
        localMemoryArray1[0] = 10;

        return (localMemoryArray1, localMemoryArray2);
       //returns 10,4,6 | 10,4,6
    }
}
```

Global variables

NAKAMOTO
&TURING
LABS

msg.sender

- the address of the sender in the current call

msg.value

- the amount of wei sent with the message

Now

- the current unix timestamp in seconds

Visibility for functions and state variables

NAKAMOTO &TURING LABS

Public

- can be called either internally or from messages, default for functions

Private

- can only be called from the contract that it is defined in and not from derived contracts

Internal

- can be called from the contract it is defined in or in derived contracts default for state variables

External

- can only be called from other contracts and via transactions
- cannot be called internally

Inheritance

For contracts inheriting from multiple other contracts, only a single contract is created on the blockchain

The code from the base contracts is copied into the final contract

Use “is” to inherit from another contract

```
pragma solidity ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}


contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

```

Multiple inheritance is possible

```
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


contract Base1 is mortal {
    function kill() { /* do cleanup 1 */ mortal.kill(); }
}


contract Base2 is mortal {
    function kill() { /* do cleanup 2 */ mortal.kill(); }
}


contract Final is Base1, Base2 {
}
```

- Use super to call the function in immediate parents in the inheritance hierarchy

Address Type

- The address type comes in two flavours, which are largely identical:
  - address: Holds a 20 byte value (size of an Ethereum address).
  - address payable: Same as address, but with the additional members transfer and send.

- Operators: <=, <, ==, !=, >= and >

- Member of Addresses
  - balance
  - transfer
  - send
  - call, delegatecall and staticcall

Smart contract security

Smart Contracts: Challenges and Risks

NAKAMOTO &TURING LABS

Although Smart Contracts could:

Enhance market activity and efficiency

Verify customer and counterparty identity

Facilitate trade execution and contract fulfillment

Ensure accurate books and recordkeeping

Complete prompt regulatory reporting

Smart Contracts could also:

Unlawfully circumvent rules and protections.

Diminish transparency and accountability.

Impair market integrity.

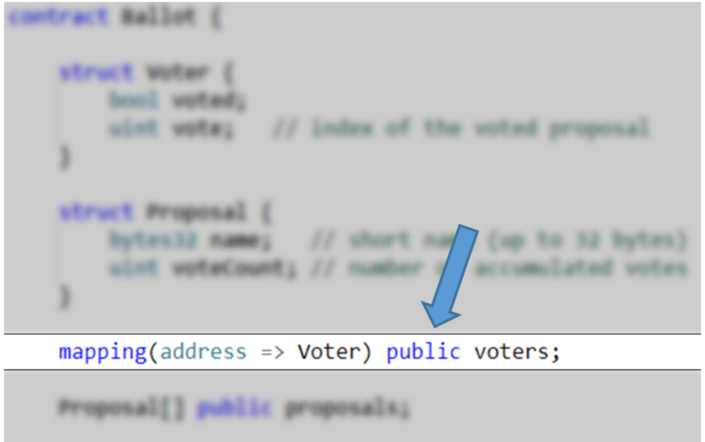
Introduce risk, including operational, technical and cybersecurity.

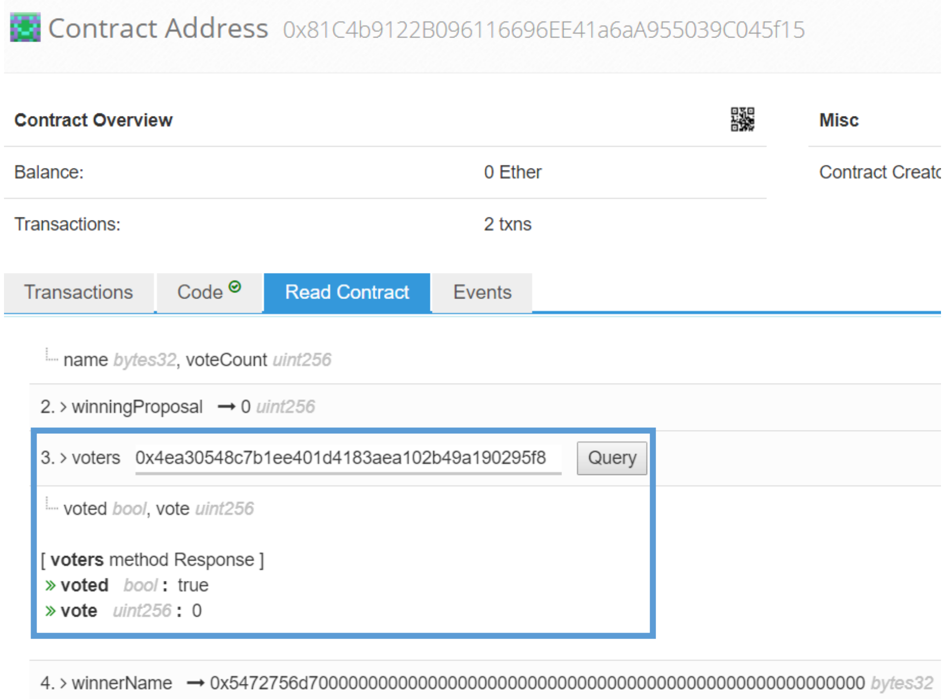
Be subject to fraud and manipulation

NAKAMOTO &TURING LABS

Vulnerability scenario #1 – all your data is public

Variables

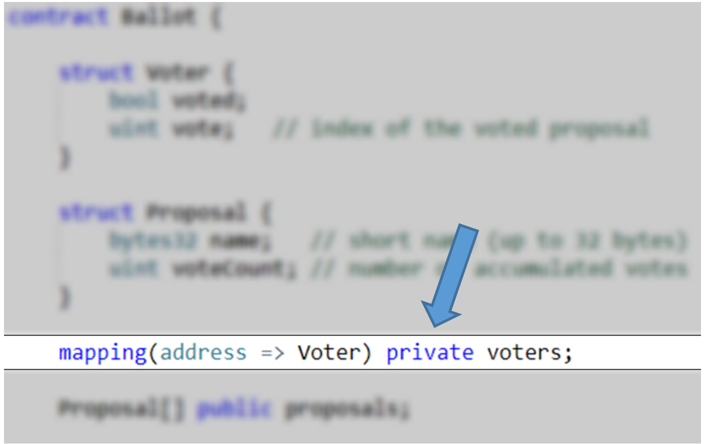


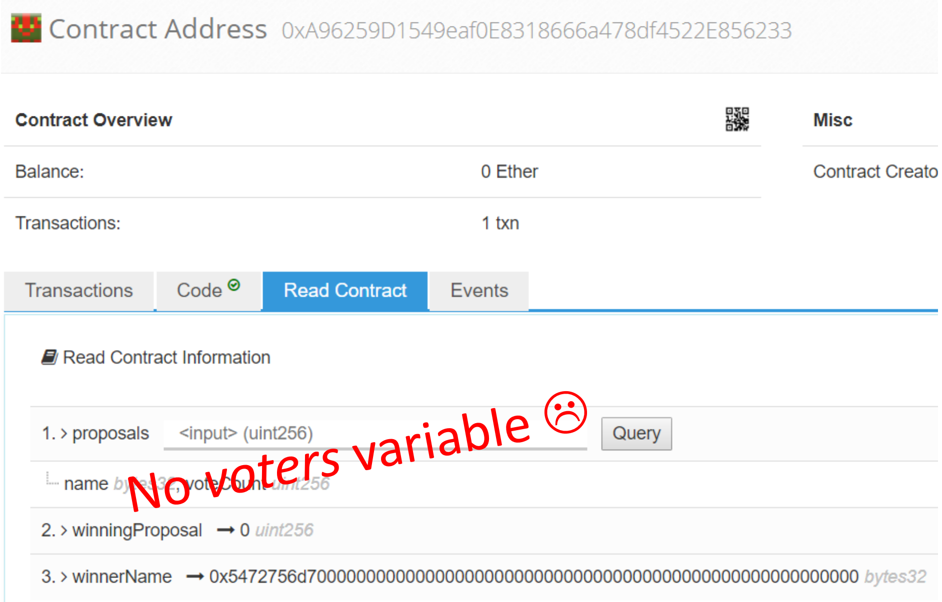


Vulnerability scenario #1 – all your data is public

NAKAMOTO &TURING LABS

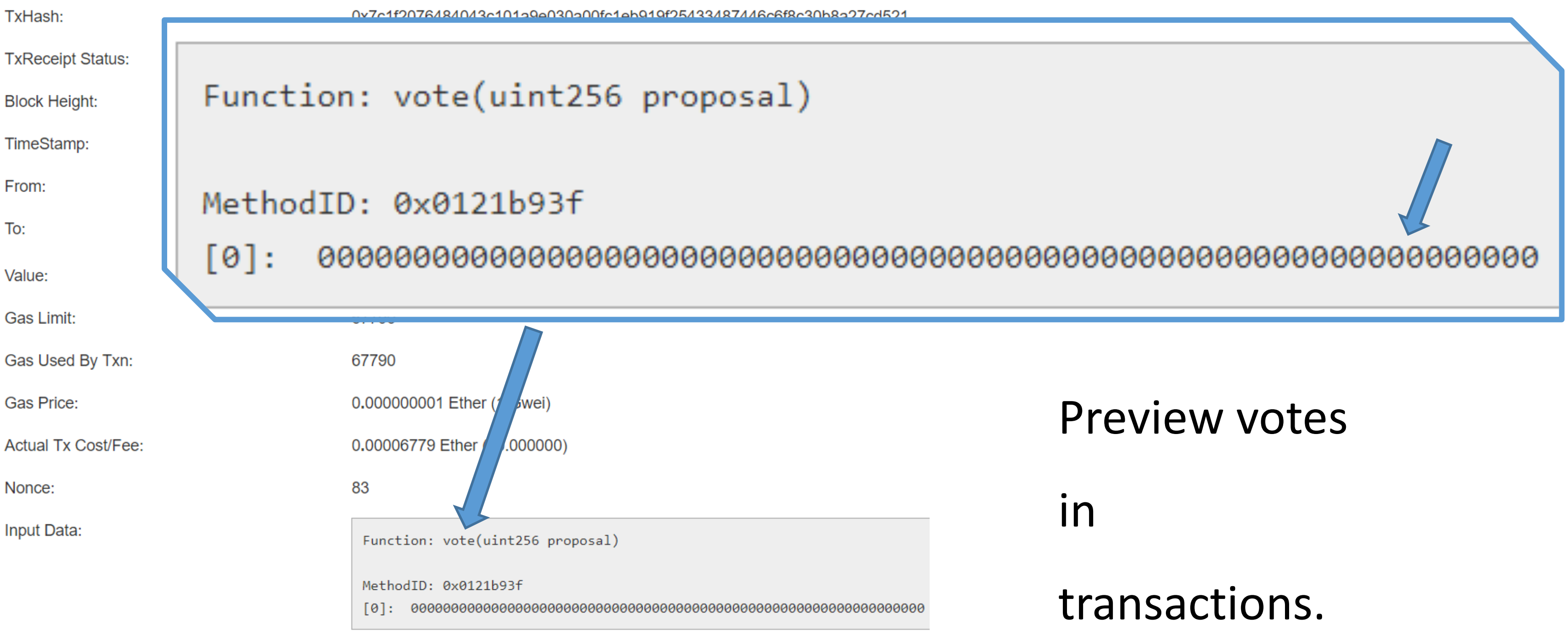
Variables





Vulnerability scenario #1 – all your data is public

NAKAMOTO &TURING LABS



Preview votes

in

transactions.

Vulnerability scenario #1 – public functions

NAKAMOTO &TURING LABS

Functions

```
pragma solidity ^0.4.18;

contract PrivateFunctions {

  function maliciousFunction1() private {
    ...
  }

  function maliciousFunction2() {
    ...
  }

}
```

- Public functions can be executed by anyone.
- Can anyone execute maliciousFunction2() ?

Functions are **public by default!**

Vulnerability scenario #1 – public functions

Parity Hack worth 30 mln $

Public function which changes the owner.


$30 Million: Ether Reported Sto
to Parity Wallet Breach
https://www.coindesk.com/30-million-ether-reported-stolen-p

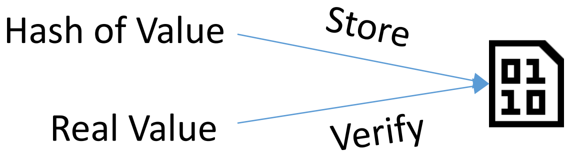
```
function initWallet(address[] _owners, uint _required, uint _daylimit) {
```

The race!

30 mln $	80 mln $
worth	today
90 mln $	240 mln $

Lessons learned

- Set visibility type to **all** functions.
- Do not keep secret data as plaintext in smart contract.
- Examples:
  - Rock Paper Scissors
  - Blind Auctions
- Use blind commitments.



Vulnerability scenario #2 – smart contract is a program

Integer Overflow

- Ethereum Tokens – your own cryptocurrency on Ethereum.
- The attack: **empty victim's wallet.**

```
contract Token {
  ...
  mapping (address => uint256) public balanceOf;

  function transfer(address _to, uint256 _value) {
    require(balanceOf[msg.sender] >= _value);
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
  }

  ...
}
```

Integer Overflow

- Balances:
  - Victim -> (MAXUINT-9) tokens (e.g. founder of contract).
  - Attacker -> 10 tokens.
- Attacker transfers 10 tokens to victim.
- Both have **zero tokens.**

```
contract Token {
  ...
  mapping (address => uint256) public balanceOf;

  function transfer(address _to, uint256 _value) {
    require(balanceOf[msg.sender] >= _value);
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
  }
  ...
}
```

NAKAMOTO
&TURING
LABS

Vulnerability scenario #2 – smart contract is a program

Insecure libraries

```
...
// 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);
}

// kills the contract sending everything to `_to`.
function kill(address _to) onlymanyowners(sha3(msg.data)) external {
  suicide(_to);
}
...
```

ETHEREUM NEWS

Ethereum's Parity Hacked, Half a Million ETH Frozen

November 7, 2017 1:58 pm

Lessons learned

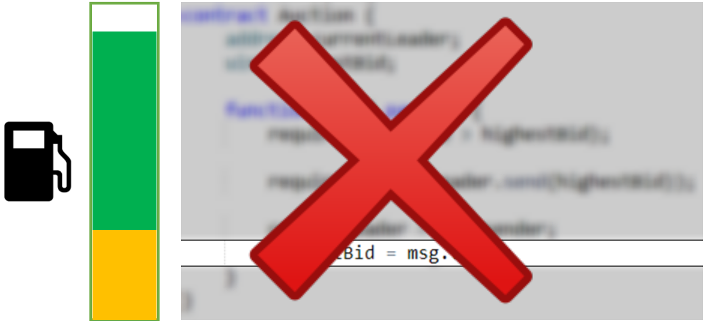
- Use open source libraries to handle typical errors (e.g. SafeMath for overflows).
- Write tests for boundary conditions.
- Verify the correctness and test libraries that you plan to use.

Vulnerability scenario #3 – smart contract has limitations

NAKAMOTO &TURING LABS

Gas Limit

- All transactions are given some gas.
- All operations cost some gas.
- Transaction is rejected if gas limit is exceeded.



- The idea: **to prevent infinite loops**.
- The attack: **DoS the contract**.

Gas Limit – DoS on auction contract

Further bids are blocked.

Lessons learned

- Learn the limitations of Ethereum (gas, randomness, etc.).
- Learn the way of handling these limitations.
- Write tests for handling limitations.

Get access to your first smart contract

A number guess games

- N players
- Each player bet 1 ETH and guess one number between 1 and 50
- The smart contract randomly generates a number
- The player whose guess is closest to the number wins all the money
- If there are more than 1 player whose guesses are closest to the number, they divide the money equally

https://rinkeby.etherscan.io/address/0x1c32e7eeab3948fab5a7f0cc540c6118424ef571#code

NAKAMOTO
&TURING
LABS