ZERO FRICTION

Methods and Techniques of Smart Contract Security

Tuan Phan, CISSP, PMP, CBSP, CTCE, Security+, SSBB
Founder

@ChainOpSec
LinkedIn.com/in/tuanphan/
github.com/tuanp703

www.zerofriction.io

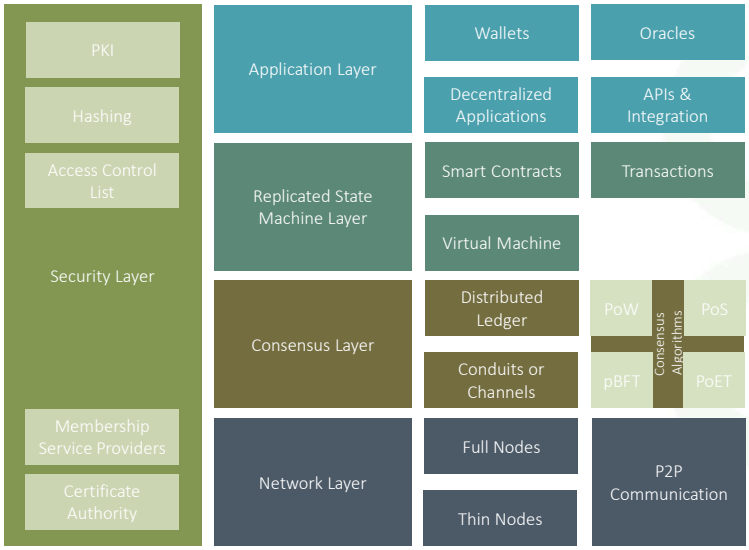
1

Learning Objectives

1. Brief Technical Review of Blockchain Mechanics
2. Types of attacks on blockchain network
3. How such attacks can be exploited.
4. Familiar with the key concepts of smart contract.
5. What specific audit elements to review and examine during a course of an IT audit?
6. Recognize the cybersecurity risks of smart contracts, and what controls can be implemented to minimize the risks from the use of smart contracts.

2

Generic Blockchain Reference Architecture



- From ISACA Blockchain Framework and Guide: (https://www.isaca.org/bookstore/bookstore-misc-digital/wbfg)
- Provide a common language to frame discussion around different blockchain designs and implementation.

3

What is a Smart Contract

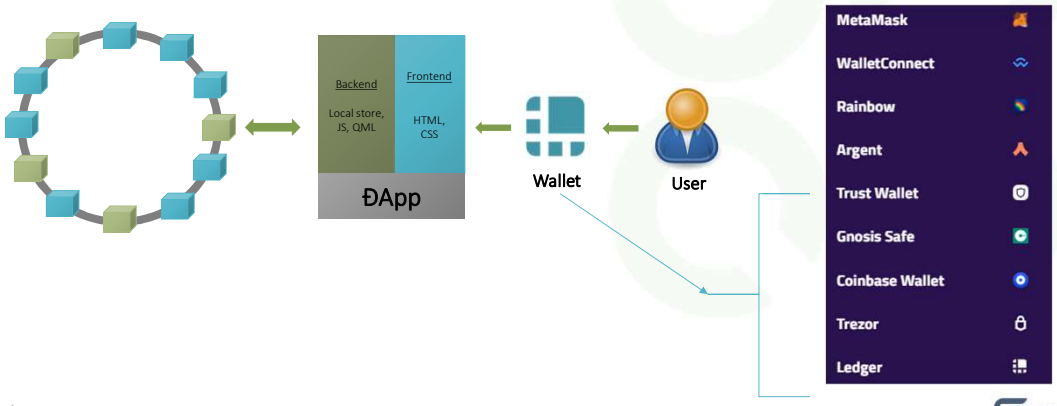
- Is a computer program that prescribes its conditions and outcomes.
- Associated with 2nd generation blockchain and later.
- Stays dormant until called by a transaction.
- Transactions performed are written onto the distributed *ledger.*


```
pragma solidity ^0.4.24;

contract Messenger {
    address owner;
    string[] messages;
    uint256 balance;

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

    function add(string newMessage) public {
        require(msg.sender == owner);
        messages.push(newMessage);
    }

    function count() view public returns(uint){
        return messages.length;
    }

    function getMessages(uint index) view public returns(string){
        return messages[index];
    }

    function GetBalance() public constant returns(uint256){
        return this.balance;
    }

    function deposit() payable {}
}
```


4

Decentralized Applications (Đapps) and User Interaction (Ethereum)

- ĐApps are blockchain-enabled applications/websites
- Rely on smart contracts for logic processing.

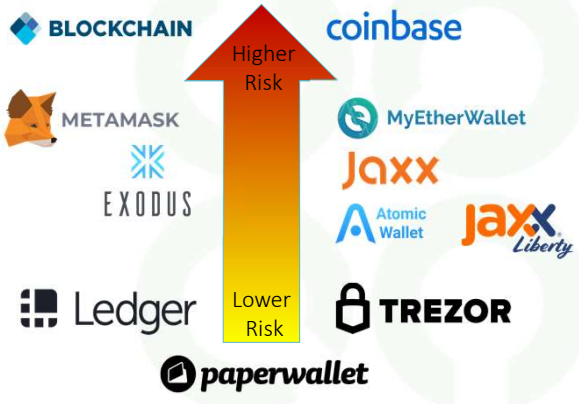


5

Types of Wallet

Hot Wallets
- Web wallets
- Browser wallets
- Desktop/Mobile wallets

Cold Wallets
- Hardware wallets
- Paper wallets



6

Types of Attacks

7

Blockchain Attacks

Network
- 51% Attack (PoW) or Casper (PoS)
- Eclipse Attack
- Denial of Service Attack
- Sybil Attack
- Routing Attack

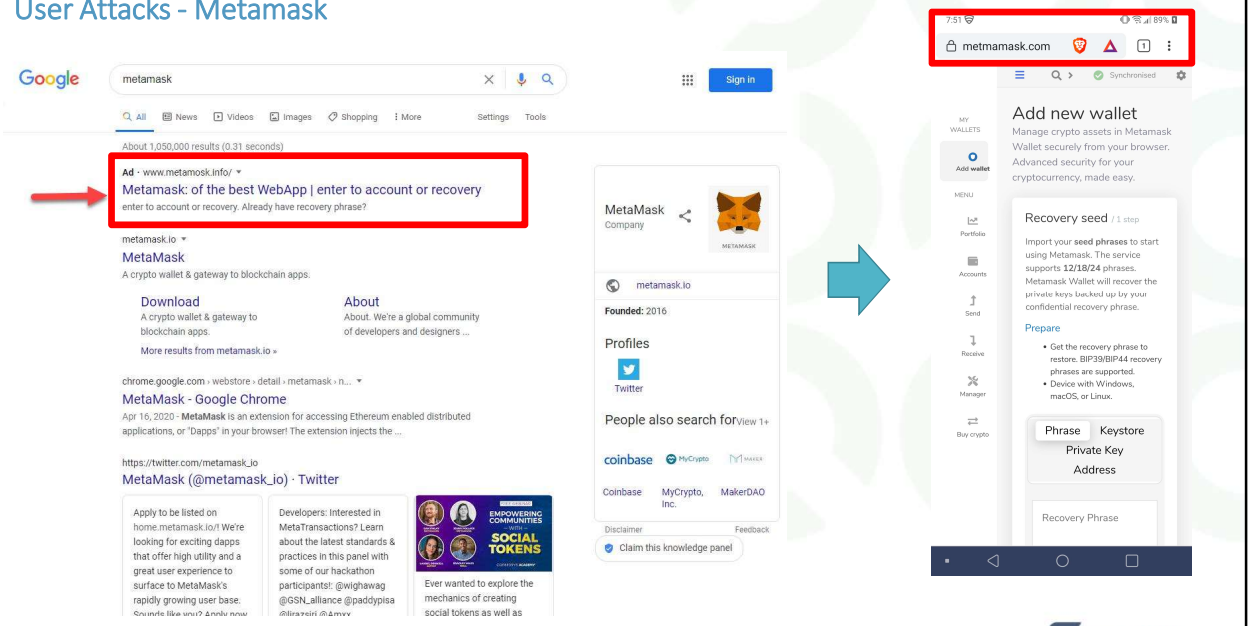
Node
- Cryptojacking Attack
- Remote Manager Exploit (miner exploit)
- Theft/Seizure

User
- Scams
- Phishing
- Web Search
- SIM Swap

Smart Contract
- Access control
- Default Visibility
- Reentrancy
- Integer Over/Underflow
- Unchecked Return
- Timestamp Manipulation
- Bad Randomness
- Front Running
- Denial of Services
- Oracle manipulation

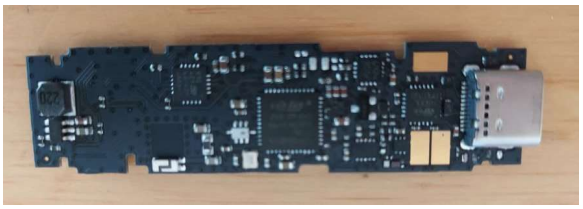
8

User Attacks - Metamask

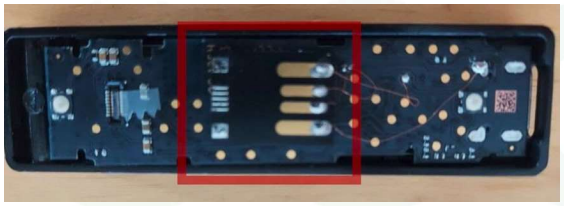


9

User Attacks - Ledger



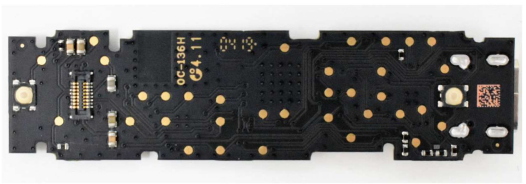
FAKE Front



FAKE Back

REAL Front

REAL Back



From Reddit

10

Key Properties of a Smart Contract

11

Key Characteristics of Smart Contract

Turing Completeness

```
pragma solidity ^0.4.8;

contract Victim {
  uint256 balance;

  // return the victim contract's balance
  function GetBalance() public constant returns(uint256){
        return this.balance;
  }

  // this function sends 0.05 ether when it is call.
  function withdraw() {
        uint transferAmt = 0.05 ether;
        if (!msg.sender.call.value(transferAmt)()) throw;
  }

  function deposit() payable {}
}
```


12

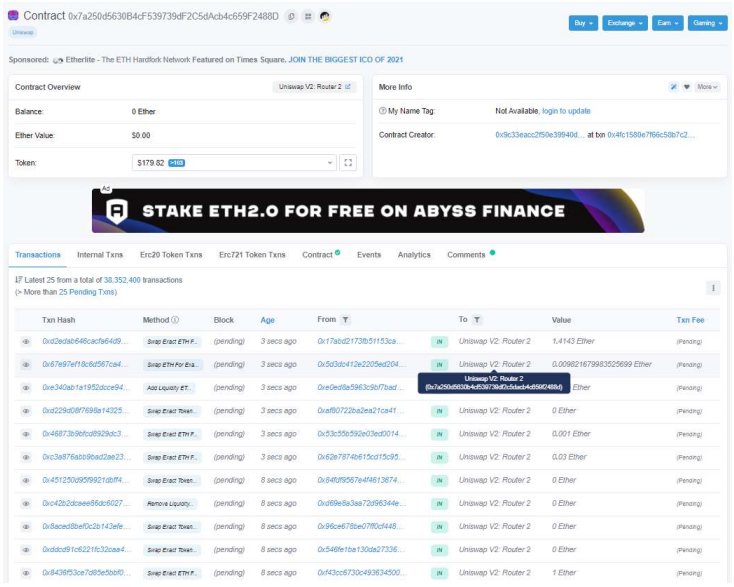
Key Characteristics of Smart Contract

Immutability

- Cannot be changed.
- Cannot be disabled.
- Cannot be removed.
- May be self-destruct if preprogrammed.

Key Characteristics of Smart Contract

Immutability - Example



Key Characteristics of Smart Contract (unique to Ethereum)

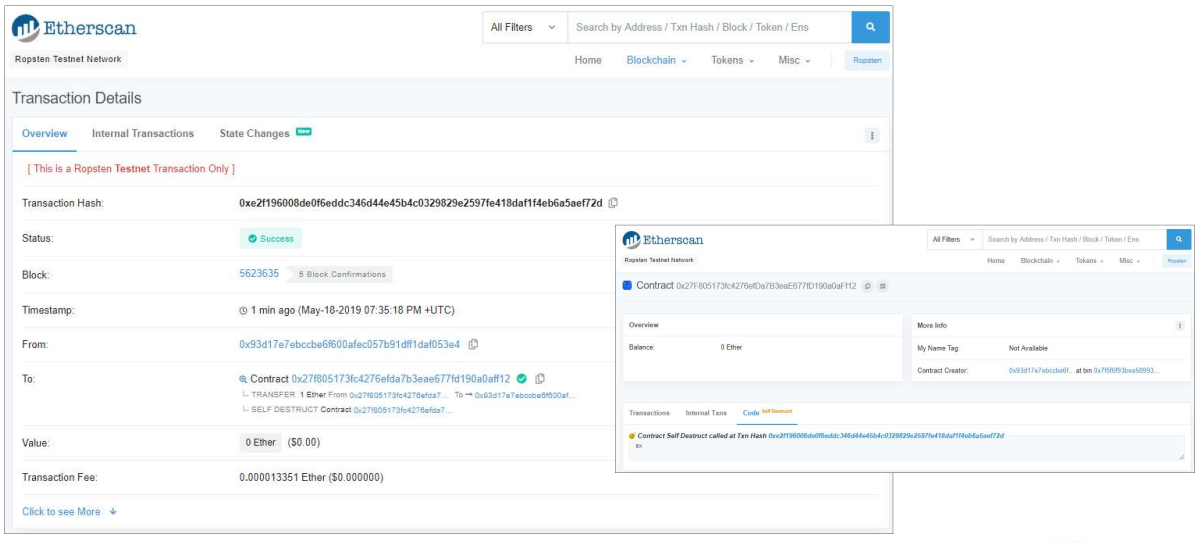
Self-Destruct – Special Case

```
pragma solidity ^0.5.0;

contract Destruct_demo{
        address owner;

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

        function deposit() public payable {
          require(msg.value > 0.1 ether);
        }

        function kill_it() public {
                require(msg.sender == owner);
                selfdestruct(msg.sender);
        }
}
```

- Preprogrammed
- One-time event
- Does not remove transaction history.
- Return any values in the contract back to the contract owner when called.
- Any value sent to self-destructed contract is lost forever.

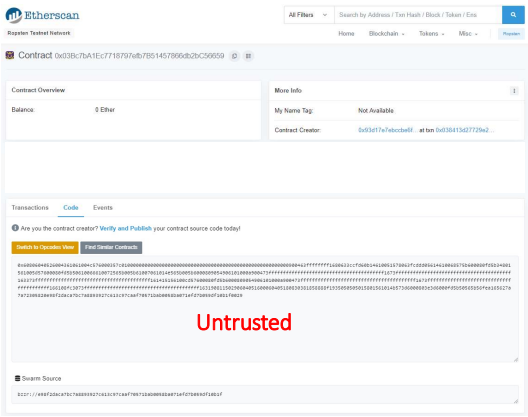
Key Characteristics of Smart Contract (unique to Ethereum)

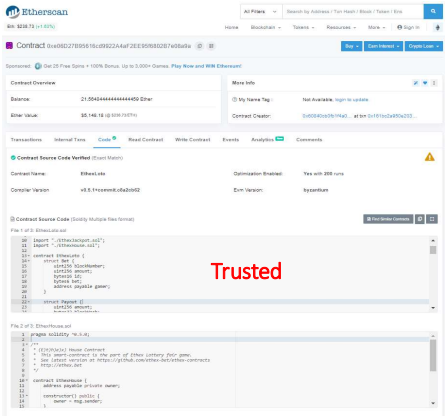
Self-Destruct – Example Details



Key Characteristics of Smart Contract

Visibility

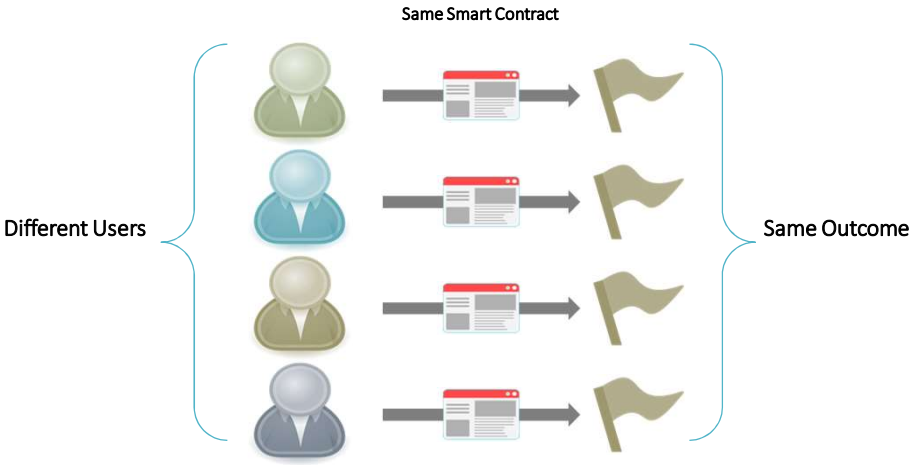




17

Key Characteristics of Smart Contract

Deterministic



18

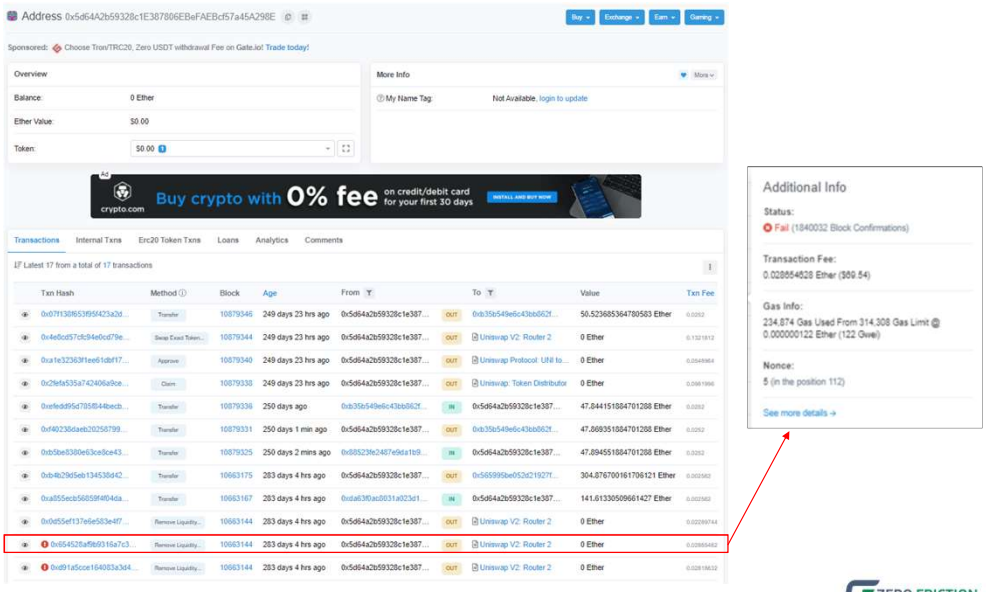
Key Characteristics of Smart Contract

Atomic

19

Key Characteristics of Smart Contract

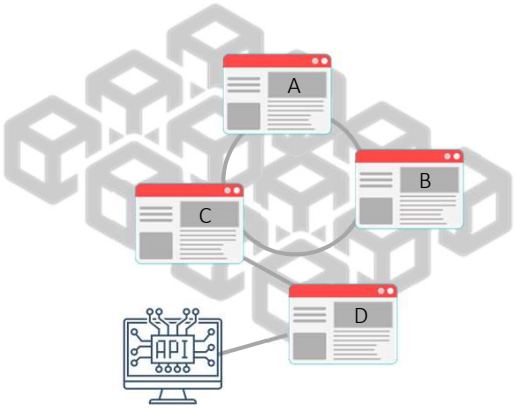
Atomic - Example



20

Key Characteristics of Smart Contract

Interaction with Other Interfaces




```
// This contract setup the owner.
// This contract provides inheritance to the main contract.
contract FromParent {
    address payable public owner;
    address payable public beneficiary = address(0);    // Set beneficiary to null

    constructor() public { // Set at start of deployment
        owner = msg.sender;             // Set the owner of contract
    }
}

// This is the Donations contract
// The contract does the following:
// 1. Initilizes the contract with a donation limit to be collected and a beneficiary. Similar to gofund me.
// 2. Users donate to the contract. Donation must be at least 1 ether.
// 3. Track donations until the donation limit has been reached, or terminate if at least 10% has been collected
// 4. Send 90% of the donations to the beneficiary
// 5. Owner gets 10% of the contract dopnation

// This contract inherits from the FromParent contract.
contract Donations is FromParent {
    using SafeMath for uint;

    mapping (address => uint) private balances;     // Track the donate amount from the user

    bool private hasBeneficiary = false;        // Set no beneficiary selected
        bool private isAtLimit = false;         // Set default for reaching donation limit flag
    bool private stopped = false;           // Set default circuit breaker/ emergency stop flag
        bool private lock = false;          // Safeguard against reentrancy

    string public emergencyTxt = 'Emergency circuit breaker is NOT ACTIVE';     // Track emergency circuit breaker status
    string public lockTxt = 'Lock is NOT ACTIVE';   // Track lock status

    uint public targetAmt;          // Track donation limit to be sought
        uint pubiic donationAmt;        // Track current donation so far.
    uint public oldDonationAmt;     // Track donation amount before withdraw payout
    uint public oldBalance;         // Track the balance before withdraw payout
    uint public newBalance;         // Track the balance after withdraw payout

    uint private round_expired;     // Track time to expire round
        uint private escrowAmt;         // Track the escrow amount from the beneficiary
        uint private beneficiaryAmt;    // Track the beneficiary amount from the total donations
```


21

Smart Contract Audit Considerations

22

Smart Contract Audit Considerations

1. Understand the technology.
2. Identify risk and appropriate controls to mitigate risk to an acceptable level.
   - *Administrative*
   - *Operational*
   - *Technical*
3. Achieve and monitor ongoing compliance effectively.

Administrative Audit Considerations

Administrative: Audit Considerations for Buyer and Seller

- Financially stable/viable, experienced, and knowledgeable
- Collusions, misconduct and manipulations
- Number of parties
- Conflicts of interest
- Able to deliver on the promises

25

Administrative: Audit Considerations from External Factors

- Regulators
- Herstatt (settlement) risk
- Privacy
- Platform dependencies:
  - *Development/Ongoing Support*
  - *Security issues*
  - *Speed of transactions*
  - *Cost of transactions*
  - *Scalability*

26

Administrative: Audit Considerations for the Smart Contract

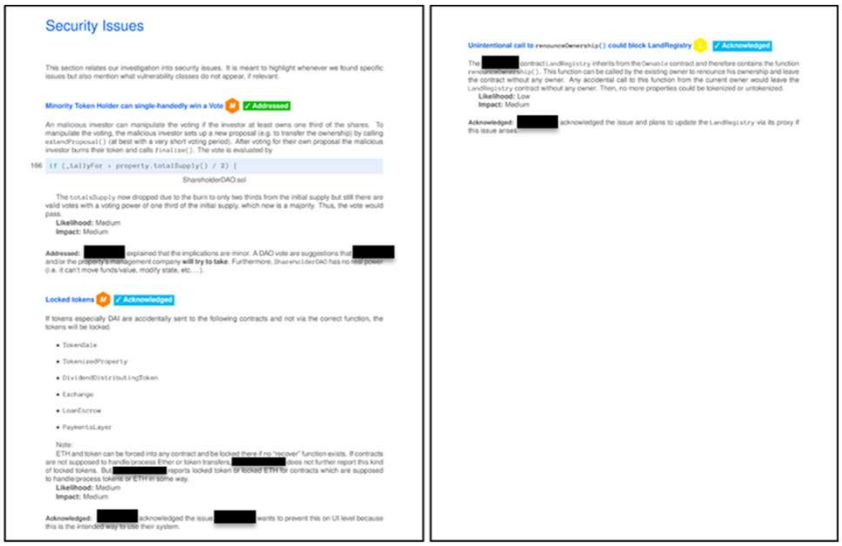
- Accurately represents the promises of the smart contract
- Clear agreement addressing non-operational issues
- Escrow or not?
- Security audit performed?

27

Operational Audit Considerations

28

Operational: Availability of Third-Party Security Audit Report



Security Issues

This section relates our investigation into security issues. It is meant to highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Minority Token Holder can single-handedly win a Vote ✓ Addressed

An malicious investor can manipulate the voting if the investor at least owns one third of the shares. To manipulate the voting, the malicious investor sets up a new proposal (e.g. to transfer the ownership) by calling `extendProposal()` (at best with a very short voting period). After voting for their own proposal the malicious investor burns their token and calls `finalize()`. The vote is evaluated by

```
166  if (_tallyFor > property.totalSupply() / 2) {
```

ShareholderDAO.sol

The `totalSupply` now dropped due to the burn to only two thirds from the initial supply but still there are valid votes with a voting power of one third of the initial supply, which now is a majority. Thus, the vote would pass.

Likelihood: Medium
Impact: Medium

Addressed: [redacted] explained that the implications are minor. A DAO vote are suggestions that [redacted] and/or the property's management company **will try to take**. Furthermore, `ShareholderDAO` has no real power (i.e. it can't move funds/value, modify state, etc...).

Locked tokens ✓ Acknowledged

If tokens especially DAI are accidentally sent to the following contracts and not via the correct function, the tokens will be locked.

- `TokenSale`
- `TokenizedProperty`
- `DividendDistributingToken`
- `Exchange`
- `LoanEscrow`
- `PaymentsLayer`

Note:
ETH and token can be forced into any contract and be locked there if no "recover" function exists. If contracts are not supposed to handle/process Ether or token transfers, [redacted] does not further report this kind of locked tokens. But [redacted] reports locked token or locked ETH for contracts which are supposed to handle/process tokens or ETH in some way.

Likelihood: Medium
Impact: Medium

Acknowledged: [redacted] acknowledged the issue [redacted] wants to prevent this on UI level because this is the intended way to use their system.

Unintentional call to `renounceOwnership()` could block LandRegistry ✓ Acknowledged

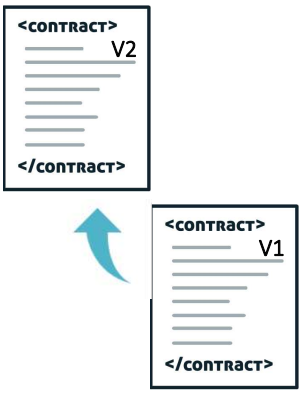
The [redacted] contract `LandRegistry` inherits from the `Ownable` contract and therefore contains the function `renounceOwnership()`. This function can be called by the existing owner to renounce his ownership and leave the contract without any owner. Any accidental call to this function from the current owner would leave the `LandRegistry` contract without any owner. Then, no more properties could be tokenized or untokenized.

Likelihood: Low
Impact: Medium

Acknowledged: [redacted] acknowledged the issue and plans to update the `LandRegistry` via its proxy if this issue arises.

29

Operational: Upgrade of Smart Contract

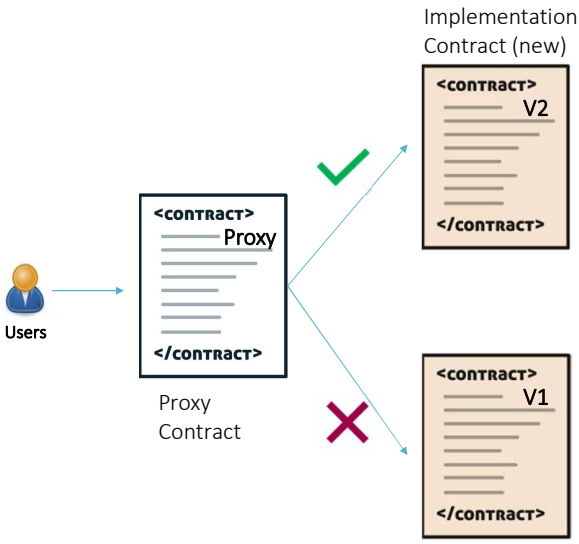


The Challenge

1. Deploy a new version of the contract at a new contract address.
2. Manually migrate all states from the old one contract to the new one.
3. Update all contracts that interacted with the old contract to use the address of the new one.
4. Reach out to all your users and convince them to start using the new deployment
5. Handle both contracts being used simultaneously, as users are slow to migrate.

30

Operational: Usage of Proxy Contract

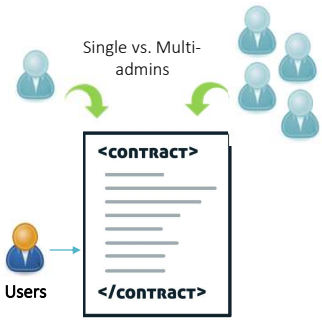


A Better Way

1. Users always interact with the Proxy contract.
2. Deploy a new version of the contract to the Implementation contract.
3. Use delegate call allowing the code to be executed in the context of the caller (proxy), not of the callee (implementation).
4. Allowing for prior states to be maintained vs. migrated.
5. Eliminate the dual-maintenance cycle needed until all users migrated.

31

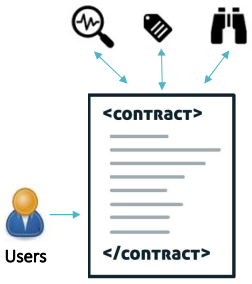
Operational: Contract Management – Who Has Control?



- The contract owner must maintain access to the smart contract and critical core functions such as fund transfer, pause deposits, and other contract's emergency circuit breaker mechanisms.
- Implement through either single admin or multi-admin design.
  - *Govern the management of the smart contracts*
  - *Offer redundancy and protection against lost/obsolete keys*
  - *Safeguard against actions from rogue admin.*

32

Operational: Usage of Oracles



- Software oracles extract online information from various sources and transmit the data to the blockchain.
- Hardware oracles obtain data from hardware devices such as barcode scanners, temperature and humidity sensors and relay such data to the blockchain.
- Minimize the Oracle Problem:
  - *Use multiple oracles to ensure accuracy of data supplied.*
  - *Implement the use of correctness check in the computed data.*

33

Manual Analysis

34

Common Vulnerabilities

1. Check the known vulnerabilities.
2. Argument values should be checked.
3. Solidity operates using integers. For better precision, perform multiplication operation first before division.
4. Verify the ordering of calls when there are multiple contracts with layered inheritance.
5. For crowdsale, verify if hardcap is reachable.
6. Inline assembly should be used very carefully if used at all.
7. Pay attention to modularity. Sometimes it is better to split contract's logic into several contracts. For example, it is not a good idea to have token and crowdsale logic in one contract.
8. It is a bad practice to modify commonly used libraries, such as OpenZeppelin. If one is used in the project, you should check, whether it is added via npm. If it is not, then check, what version is used and whether it is modified.

Standard-specific Issues

ERC20

1. If the token does not have a single way to counteract the ERC20 approve issue, we recommend the developer to warn users about that vulnerability and to add functions increaseApproval() and decreaseApproval().
2. Sometimes the developer decides to reject zero token transfers. This is an issue since ERC20 token standard states that "Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event".
3. When token contract creates new tokens, it SHOULD fire Transfer event with the _from address equal to 0x0.

ERC223

1. The auditor should verify that all token receiving contracts have implemented tokenFallback function.
2. Also one should ensure that all the token sending contracts are resistant to reentrancy vulnerability.

Code Logic

1. Check the known vulnerabilities.
2. Argument values should be checked.
3. Solidity operates using integers. For better precision, perform multiplication operation first before division.
4. Verify the ordering of calls when there are multiple contracts with layered inheritance.
5. For crowdsale, verify if hardcap is reachable.
6. Inline assembly should be used very carefully if used at all.
7. Pay attention to modularity. Sometimes it is better to split contract's logic into several contracts. For example, it is not a good idea to have token and crowdsale logic in one contract.
8. It is a bad practice to modify commonly used libraries, such as OpenZeppelin. If one is used in the project, you should check, whether it is added via npm. If it is not, then check, what version is used and whether it is modified.

Technical Audit Considerations

Security Considerations for Security Audit

What – Why – How to Mitigate

- Access Control
- Default Visibility
- Reentrancy
- Integer Over/Underflow
- Unchecked Return
- Timestamp Manipulation
- Bad Randomness
- Front Running
- Denial of Services
- Short Address

Security Considerations for Security Audit

Access Control (anti-pattern)

Is an attack that seizes ownership of a contract from its rightful owner.

- Incorrect usage or lack of constructor to initialize ownership.
- Failure to check for ownership prior to execute key functions.

```
1  pragma solidity ^0.4.21;
2
3  contract OwnerWallet {
4      address public owner;
5
6      function initWallet() public {
7          owner = msg.sender;
8      }
9
10     // Fallback. Collect ether.
11     function () payable {}
12
13     function withdraw() public {
14         msg.sender.transfer(this.balance);
15     }
16 }
```

Security Considerations for Security Audit

Access Control (mitigation)

- Properly initialized to maintain contract ownership.
- Require contract owner check before any allowing any execution intended for the contract owner.

```
1  pragma solidity ^0.4.21;
2
3  contract OwnerWallet {
4      address public owner;
5
6      // constructor to initialize ownership
7      function OwnerWallet() public {
8          owner = msg.sender;
9      }
10
11     // Fallback. Collect ether.
12     function () payable {}
13
14     function withdraw() public {
15         require(msg.sender == owner);
16         msg.sender.transfer(this.balance);
17     }
18 }
```


Security Considerations for Security Audit

Default Visibility (anti-pattern)

Misuse of visibility modifiers expose certain functions for manipulation by other contracts.

- No visibility identifier stated.

```
1  pragma solidity ^0.4.21;
2
3  contract HashForEther {
4
5      function withdrawWinnings() {
6          // Winner if the last 8 hex characters of the address are 0
7          require(uint32(msg.sender) == 0);
8          _sendWinnings();
9      }
10
11     function _sendWinnings() {
12         msg.sender.transfer(this.balance);
13     }
14 }
```

Security Considerations for Security Audit

Default Visibility (mitigation)

- Explicitly state the visibility identifier.
- Use the correct visibility identifiers:
  - *Public (visible to everyone; is the default if not specified)*
  - *Private (visible for only the current contract)*
  - *Internal (can be called inside the current contract)*
  - *External (can be called from other contracts and transactions)*

```
pragma solidity ^0.4.21;

contract HashForEther {

    function withdrawWinnings() public {
        // Winner if the last 8 hex characters of the address are 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
    }

    function _sendWinnings() private internal {
        msg.sender.transfer(this.balance);
    }
}
```


Security Considerations for Security Audit

Reentrancy (anti-pattern)

Is a classic attack that takes over control flow of a contract and manipulate the data to prevent the correct updating of state.

- Making external calls

```
// INSECURE
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)());
    // At this point, the caller's code is executed, and
    can call withdrawBalance again
    userBalances[msg.sender] = 0;
}
```

```
// INSECURE
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
        userBalances[to] += amount;
        userBalances[msg.sender] -= amount;
    }
}

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    // At this point, the caller's code is executed,
    // and can call transfer()
    require(msg.sender.call.value(amountToWithdraw)());
    userBalances[msg.sender] = 0;
}
```

Security Considerations for Security Audit

Reentrancy (mitigation)

- Finish all internal work (e.g., state changes) first and only then calling the external function.
- Use send() instead of call.value()().

```
1  mapping (address => uint) private userBalances;
2
3  function withdrawBalance() public {
4      uint amountToWithdraw = userBalances[msg.sender];
5      userBalances[msg.sender] = 0;
6      require(msg.sender.call.value(amountToWithdraw)());
       // The user's balance is already 0, so future
       invocations won't withdraw anything
7  }
8
```


Security Considerations for Security Audit

Integer Overflow & Underflow (anti-pattern)

Occurs when an operation is performed that requires a fixed-size variable to store a number (or piece of data) that is outside the range of the variable's data type.

- An unsigned integer gets incremented above its maximum value (overflow)
- An unsigned integer gets decremented below zero (underflow)

```
1  pragma solidity ^0.4.15;
2
3  contract Overflow {
4      uint private sellerBalance=0;
5
6      function add(uint value) returns (bool){
7          sellerBalance += value; // possible overflow
8
9          // possible auditor assert
10         // assert(sellerBalance >= value);
11     }
12 }
```

Security Considerations for Security Audit

Integer Overflow & Underflow (mitigation)

- Use SafeMath library
- Check both storage and calculated variables for valid condition.

```
pragma solidity ^0.4.15;

library SafeMath {
    function add(uint256 a, uint256 b) internal constant returns
    (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}

contract Overflow {
    uint private sellerBalance=0;

    function safe_add(uint value) returns (bool) {
        require(value + sellerBalance >= sellerBalance);
        sellerBalance += value;
    }
}
```


Security Considerations for Security Audit

Unchecked Return Values (anti-pattern)

Failure to verify low-level function state after call may result in incorrect variable states.

- Low-level functions are call(), callcode(), delegatecall() and send().
- Level-level calls return boolean false when fail instead of a roll-back.

```
pragma solidity ^0.4.21;

contract UncheckedSendValue {
    uint weiLeft;
    uint balance;
    mapping(address => uint256) public balances;

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

    function withdraw (uint _amount) public {
        require(balances[msg.sender] >= _amount);
        weiLeft -= _amount;
        msg.sender.send(_amount);
        }

    function GetBalance() public constant returns(uint){
        return this.balance;
    }
}
```

Security Considerations for Security Audit

Unchecked Return Values (mitigation)

- Check the return value of send() to see if it completes successfully.
- If it doesn't, then throw an exception so all the state is rolled back.

```
pragma solidity ^0.4.21;

contract UncheckedSendValue {
    uint weiLeft;
    uint balance;
    mapping(address => uint256) public balances;

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

    function withdraw (uint _amount) public {
        require(balances[msg.sender] >= _amount);
        if (msg.sender.send(_amount))
            weiLeft -= _amount;
        else throw;
        }

    function GetBalance() public constant returns(uint){
        return this.balance;
    }
}
```


Security Considerations for Security Audit

Timestamp Manipulation (anti-pattern)

Misuse of block.timestamp function by miners.

- Miners can set their time to any period in the future.
- If mined time is within 15 minutes, the block will be accepted on the network.

```
pragma solidity ^0.4.21;

contract TimestampManipulation {
    uint time_counter;
    uint max_counter = 1521763200;

    function play() public {
        require(now > 1521763200 && neverPlayed == true);
        neverPlayed = false;
        msg.sender.transfer(1500 ether);
    }
}
```

Security Considerations for Security Audit

Timestamp Manipulation (mitigation)

- Do not relying on the time as advertised.
- Use external initiator to track time.

```
1  const contract = web3.eth.contract(contractAbi);
2  const contractInstance = contract.at(contractAddress);
3
4  contractInstance.timer('time_counterjs');
5  // send current time value
6      time_counterjs +=1;
7  });
```

```
1  pragma solidity ^0.4.21;
2
3  contract TimestampManipulation {
4      address public owner;
5      uint time_counter;
6      uint max_counter = 1521763200;
7
8      function TimestampManipulation() public {
9          owner = msg.sender
10     }
11
12     function play() public {
13         require(time_counter > max_counter && neverPlayed == true);
14         neverPlayed = false;
15         msg.sender.transfer(1500 ether);
16     }
17
18     // Using an external initiator such as a JS
19     // function to trigger at some intervals
20     function timer(currenttime_count) public {
21         require(msg.sender == owner);
22         time_counter = currenttime_count;
23     }
24
25 }
```


Security Considerations for Security Audit

Bad Randomness (anti-pattern)

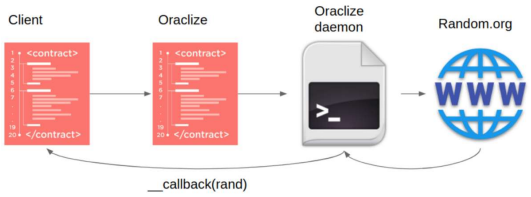
Poor implementation of pseudo-random number generator

- Private variables are set via a transaction at some point in time and are visible on the blockchain.
- Block variables such as block.timestamp, block.coinbase, block.number can be manipulated by miners.

```
1  uint256 constant private salt =  block.timestamp;
2
3  function random(uint Max) constant private returns (uint256 result){
4      //get the best seed for randomness
5      uint256 x = salt * 100/Max;
6      uint256 y = salt * block.number/(salt % 5) ;
7      uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
8      uint256 h = uint256(block.blockhash(seed));
9
10     return uint256((h / x)) % Max + 1; //random number between 1 and
       Max
11 }
```

Security Considerations for Security Audit

Bad Randomness (mitigation)



- Use random seed from a random oracle such as Oraclize
- Use a random seed from a future block hash

Security Considerations for Security Audit

Front Running (anti-pattern)

Pay higher gas fees to have copied transactions mined more quickly to preempt the original solution.

- Attacker watches the pool of pending transactions for the winning transaction.
- Attacker submits his bet with higher gas price to beat out the winning transaction.

```
pragma solidity ^0.4.21;

contract FindThisHash {
    bytes32 constant public hash =
      0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee
      0a;

    function FindingThisHash(address _owner) public payable {}
    // constructor() public payable {}  // load with ether

    function solve(string solution) public {
        // If you can find the pre-image of the hash, receive 1000
        ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Security Considerations for Security Audit

Front Running (mitigation)

- Usage of commit-reveal approach (RandDAO)

55

Security Considerations for Security Audit

There will always be new Unknowns

- Smart contracts → emerging
- Coding language ≠ stable
- There will be new classes of vulnerabilities
- Developers and auditors need to stay on their feet!

56

Best Practices for Smart Contracts

- Maintain control
- Be aware of smart contract properties
- Prepare for failure (circuit breaker)
- Rollout carefully (rate limiting, max usage, correctness checks)
- Keep contracts simple
- Stay up to date (refactoring, latest compiler)

Follow Occam's razor

Final Words

- The effectiveness of the security of the blockchain is highly dependent on the auditor's understanding of the mechanisms for both the blockchain platform and the underlying smart contracts.
- It is important to consider the complete design of the blockchain application and all interconnected parts.
- Smart contracts have limitations, therefore, third-party audit and security reviews are paramount to bring perspective.
- Transparency, expert reviews, user testing and use of automated security tools are mechanisms to minimize vulnerabilities.

ZERO FRICTION

THANK YOU FOR YOUR TIME.

Tuan Phan, CISSP, PMP, CBSP, Security+, SSBB
Founder

@ChainOpSec
LinkedIn.com/in/tuanphan/
github.com/tuanp703

www.zerofriction.io

59

Brief Blockchain Primer

60

Blockchain Platforms

APPLICATIONS

OPEN PLATFORMS

ENTERPRISE PLATFORMS

HYPERLEDGER SAWTOOTH

IROHA

r3.

HYPERLEDGER FABRIC

HYPERLEDGER INDY

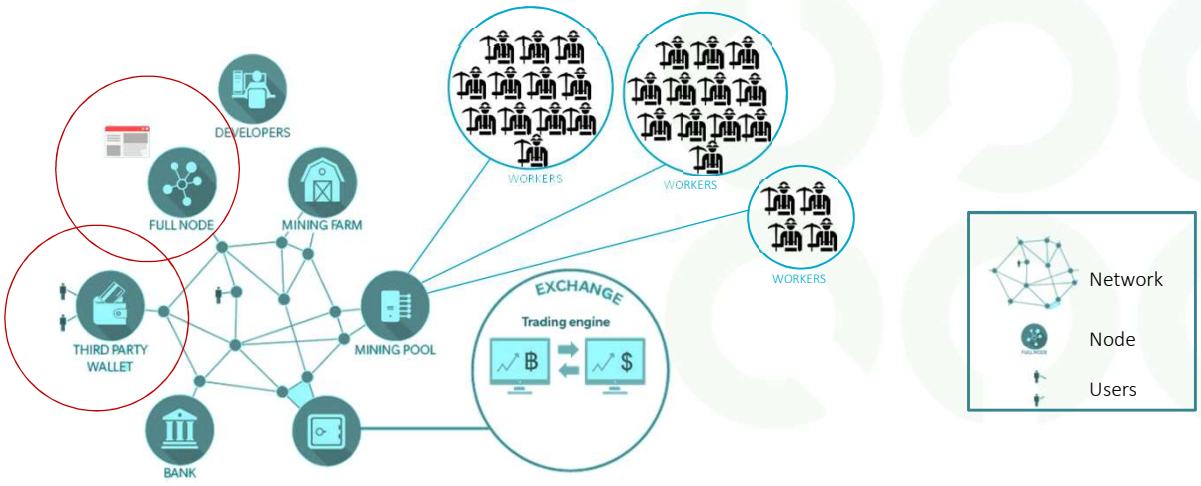
HYPERLEDGER BURROW

www.zerofriction.io

ZERO FRICTION 61

61

Typical Blockchain Network

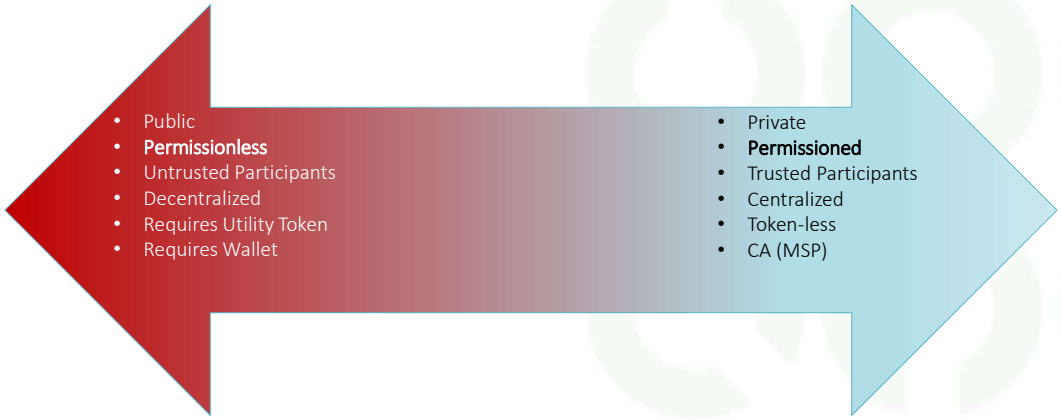


www.zerofriction.io

ZERO FRICTION 62

62

Deployment Models of Blockchain



63

Four Core Characteristics of Blockchain

Shared Ledger

- History of all transactions
- Append-only with immutable past
- Distributed and replicated

Cryptography

- Integrity of ledger
- Authenticity of transactions (user wallet)
- Privacy of transactions
- Identity of participants (user address)

64

User Wallets

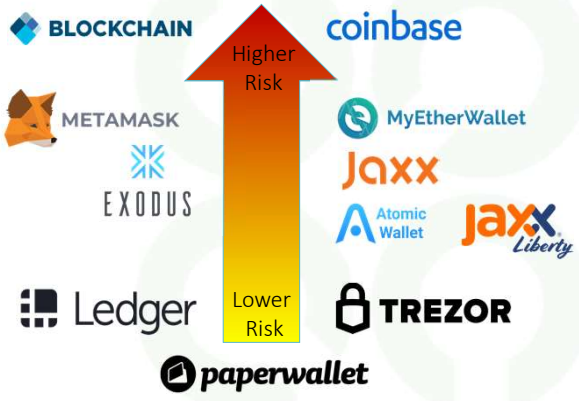
- Serves as the primary interface for the user to the blockchain
  - Ethereum wallet address → 0x47faAD405C3338112A02CE7fDfBF13d1d229F1B3
  - Bitcoin wallet address → bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh
- Is the container for private keys and as the system for managing these keys.
- Controls access to a user's money, manages keys and addresses, tracks user balance, creates and signs transactions, and interacts with contracts.
- Modern wallets use seed words to generate the private keys based on a defined standard (BIP39).

65

Types of Wallet

Hot Wallets
- Web wallets
- Browser wallets
- Desktop/Mobile wallets

Cold Wallets
- Hardware wallets
- Paper wallets



66

Four Core Characteristics of Blockchain (2nd Gen and later)

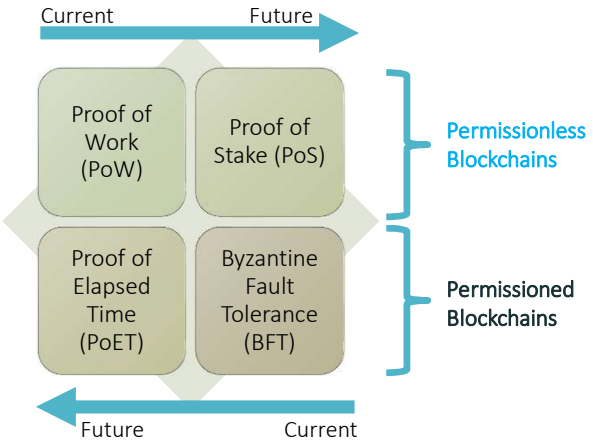
Shared Ledger
- History of all transactions
- Append-only with immutable past
- Distributed and replicated

Cryptography
- Integrity of ledger
- Authenticity of transactions
- Privacy of transactions
- Identity of participants

Trust Model
- Consensus protocol
- Transaction validation
- Tolerate disruption

67

Consensus Protocols and Outlook



- Consensus participants seek the chance to create and append the block to the digital ledger.

Use specialty hardware (ASIC miners), GPU and electricity.

Use locked investments to win a chance proportion to the investment.

Use hardened chip to determine the block author from designated decision nodes.

Use a majority voting scheme from designated decision nodes.

68

Four Core Characteristics of Blockchain (2nd Gen and later)

Shared Ledger
- History of all transactions
- Append-only with immutable past
- Distributed and replicated

Cryptography
- Integrity of ledger
- Authenticity of transactions
- Privacy of transactions
- Identity of participants

Trust Model
- Consensus protocol
- Transaction validation
- Tolerate disruption

Smart Contract
- Logic embedded in the ledger
- Executed together with transactions

Test Your Knowledge (True or False)

1. Smart contracts exist in 1st and 2nd generation blockchain.
2. In a permissioned blockchain the participants are known to the blockchain operator.
3. Decentralized applications depend on smart contracts for the business logic.
4. The shared ledger is distributed across nodes.
5. Cold wallets are more risky than hot wallets.
6. Private blockchains are generally more centralized.
7. Transaction outcomes can be different (e.g., random likelihood) for every users interacting with the same smart contract.
8. All conditions for a given transaction in a smart contract must be met in order to complete and record the transaction.
9. Access control is critical to maintain ownership oof a smart contract.
10. Reentrancy exists because contract states are not properly set prior to making external calls.

Answer to Test Your Knowledge

1. False: Smart contracts exist in 1st and 2nd generation blockchain.
2. True: In a permissioned blockchain the participants are known to the blockchain operator.
3. True: Decentralized applications depend on smart contracts for the business logic.
4. True: The shared ledger is distributed across nodes.
5. False: Cold wallets are more risky than hot wallets.
6. True: Private blockchains are generally more centralized.
7. False: Transaction outcomes can be different (e.g., random likelihood) for every users interacting with the same smart contract.
8. True: All conditions for a given transaction in a smart contract must be met in order to complete and record the transaction.
9. True: Access control is critical to maintain ownership oof a smart contract.
10. True: Reentrancy exists because contract states are not properly set prior to making external calls.

How do I get Hands-on Experience?

Programming Smart Contracts

Ethereum

- Solidity via an IDE (Remix IDE, EthFiddle)
- Wallet with some test currencies
- Local development environment or web-based at Remix (https://remix.ethereum.org/)
- Connection to the actual blockchain network, local or testnet

Hyperledger Fabric

- Go/Javascript (popular for permissioned blockchains) via an IDE (HLFV Composer, VSCode or similar editors)
- Local development environment or IBM Bluemix Console (https://cloud.ibm.com/login)
- Connection to the actual blockchain network, local or testnet