ZERO FRICTION

Auditing Smart Contracts

What IT Auditors need to know

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

Founder

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

www.zerofriction.io

1

Learning Objectives

1. What are smart contracts, their benefits and use cases.
2. Understand the legality of smart contracts. Are smart controls legal means for use in establishing agreements? What legal frameworks support/not support the use of smart contracts? What governance consideration should organization have in place?
3. Familiar with the key concepts of smart contracts, and how the smart contracts may differ between permissioned and permissionless blockchains.
4. What specific audit elements to review and examine during a course of an IT audit?
5. Recognize the cybersecurity risks of smart contracts, and what controls can be implemented to minimize the risks from the use of smart contracts.

2

Agenda

- What is a Smart Contract?
- Use Cases
- Regulatory Drivers
- Legality
- Characteristics and Programming
- Smart Contract Audit Considerations
- Best Practices
- Final Words

3

What is a Smart Contract?

4

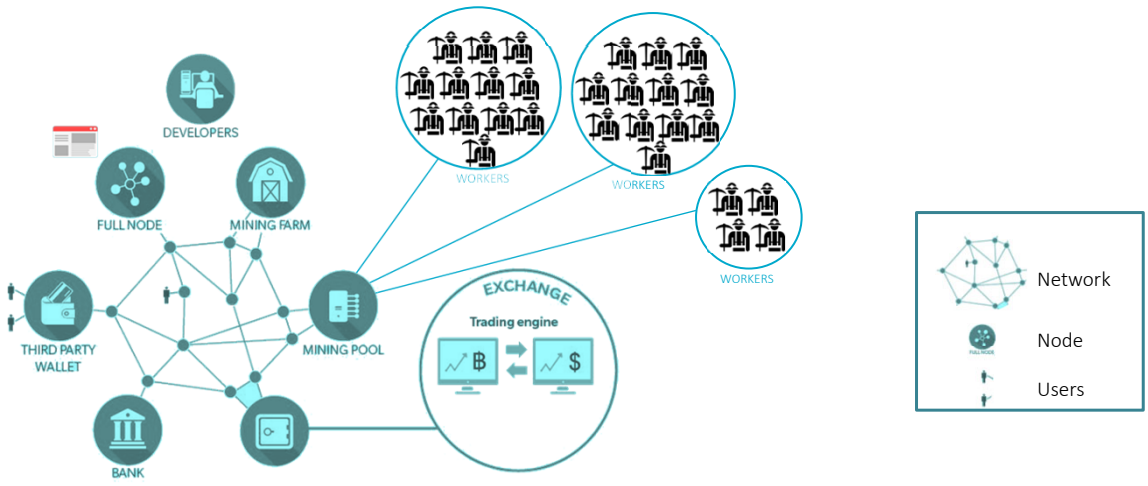
What is a Smart Contract

- Is a computer program that prescribes its conditions and outcomes.
- Is stored and processed on 2nd generation blockchain.
- 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 {}
}
```


5

Typical Blockchain Network



6

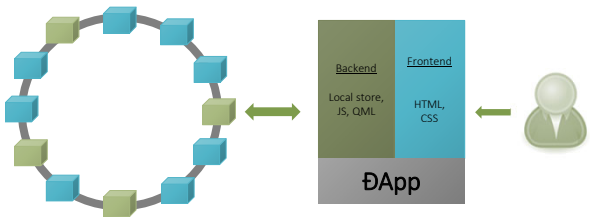
Why Smart Contracts?

	Traditional Contracts	Smart Contracts
Speed	1 to 3 days	Within minutes
Need for Escrow	Required	May not be required
Cost	Expensive	Fraction of the cost of traditional contract
Location	Physically presence required	Virtually from anywhere
Trust & Autonomy	Dependence on other parties	Automated and unalterable
Transparency	Opaque – Need lawyers	Visible – Lawyers may not be required.

7

Đapps and Smart Contracts

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



8

Use Cases for Smart Contracts

- Games
- Defi/Exchanges
- Gambling
- Property
- Social
- Security
- Identity
- Marketplaces
- Health

The DeFi ecosystem generated 99% of Ethereum activity.

9

Key Regulatory Drivers

United States

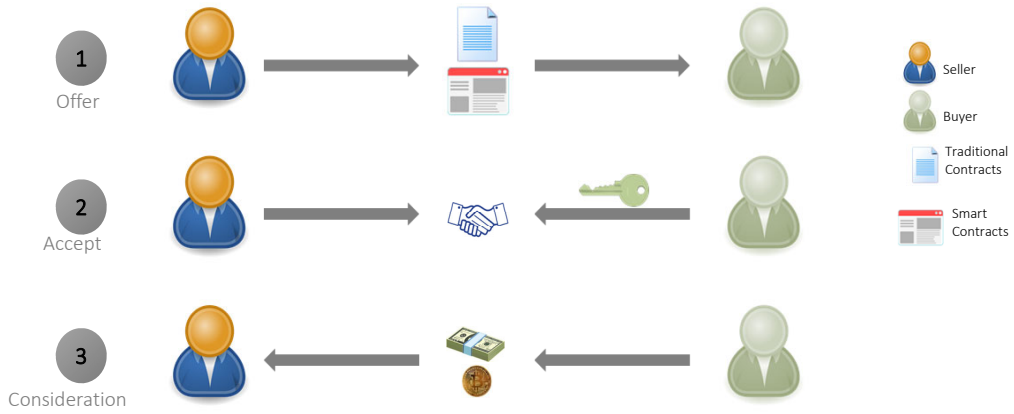
- Electronic Signatures in Global and National Commerce (ESIGN) Act
- Uniform Electronic Transactions Act (UETA)
- FDA's 21 CFR Part 11

European Union

- Electronic Identification and Trust Services Regulation (910/2014/EC)
- Electronic Signature Directive (1999/93EC) [obsoleted]

10

Are Smart Contracts Legally Binding?



11

Key Properties of a Smart Contract

12

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 {}
}
```


13

Key Characteristics of Smart Contract

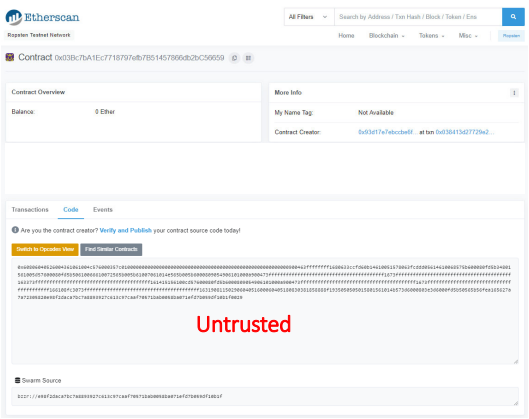
Immutability

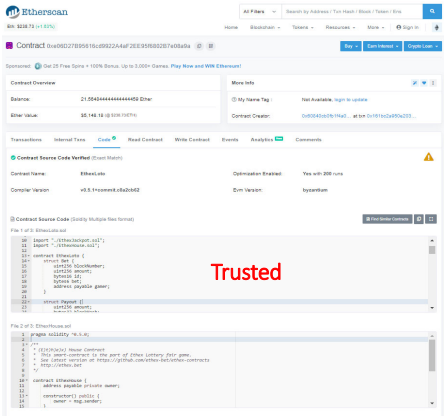
- Cannot be changed.
- Cannot be disabled.
- Cannot be removed.
- May be self-destruct if preprogrammed.

14

Key Characteristics of Smart Contract

Visibility

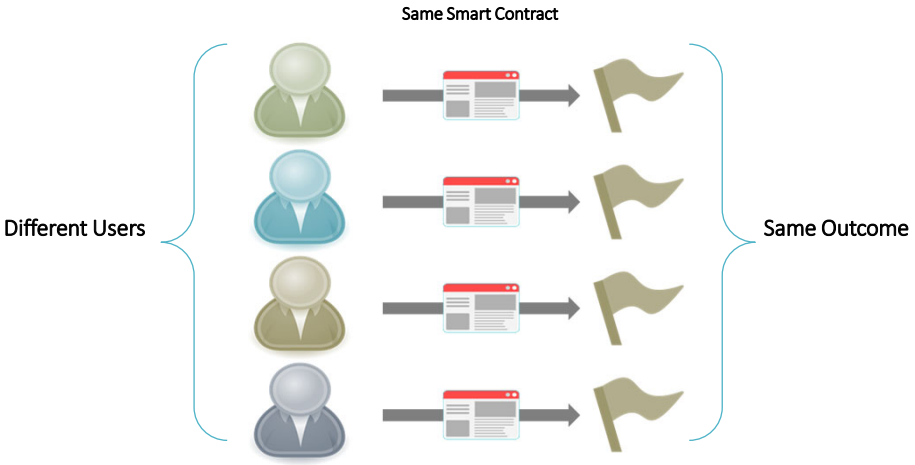




15

Key Characteristics of Smart Contract

Deterministic



16

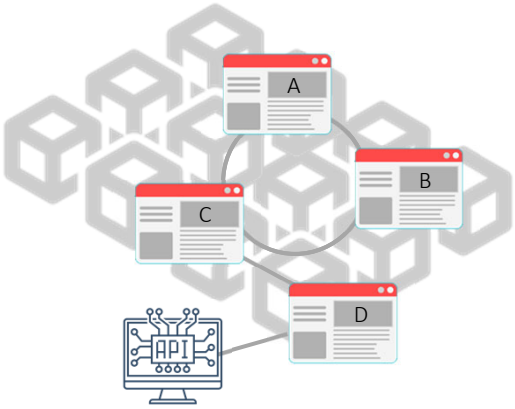
Key Characteristics of Smart Contract

Atomic

17

Key Characteristics of Smart Contract

Interaction with Other Interfaces



18

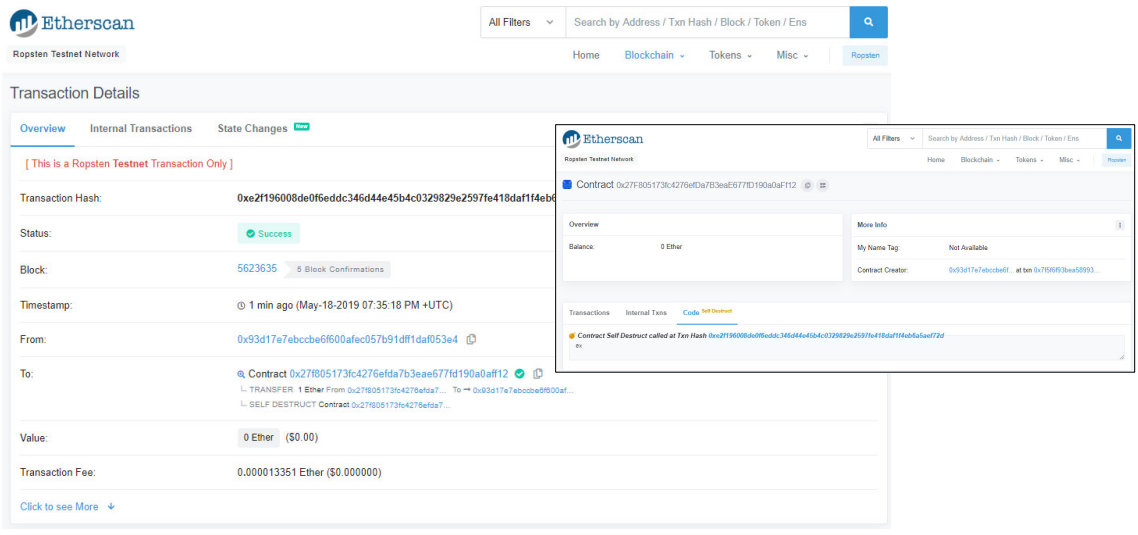
Key Characteristics of Smart Contract

Self-Destruct

```
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);
        }
}
```


19

Self-Destructed Smart Contracts



20

Audit Considerations

21

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.

22

Administrative Audit Considerations

23

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

24

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*

25

Administrative: Audit Considerations for the Smart Contract

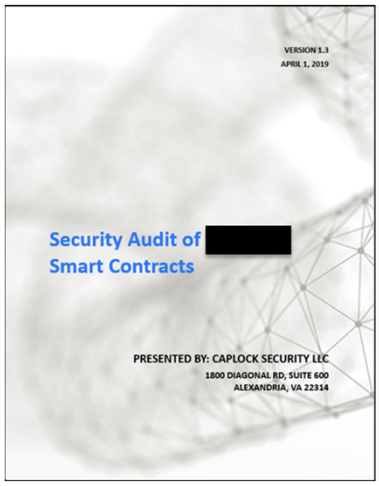
- Accurately represents the promises of the smart contract
- Clear agreement addressing non-operational issues
- Escrow or not?
- Security audit performed?

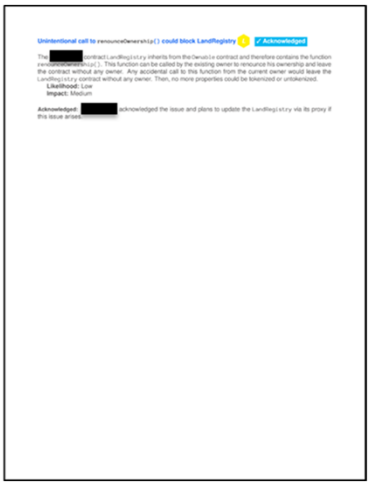
26

Operational Audit Considerations

27

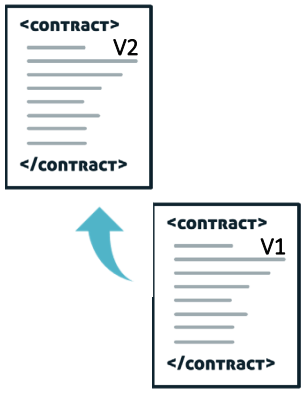
Operational: Availability of Third-Party Security Audit Report





28

Operational: Upgrade of Smart Contract

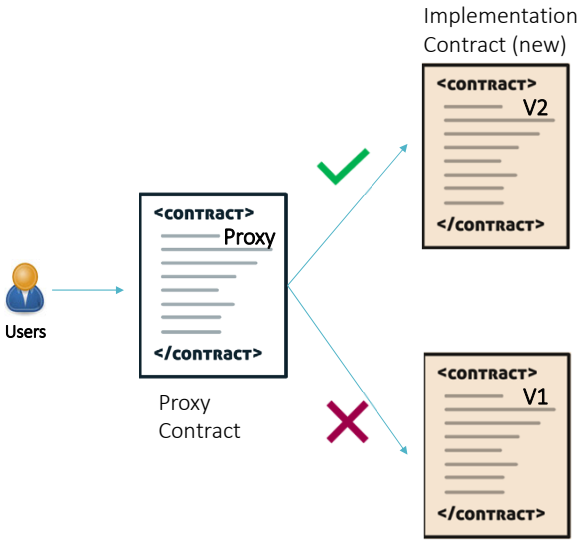


The Old Way

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.

29

Operational: Usage of Proxy Contract

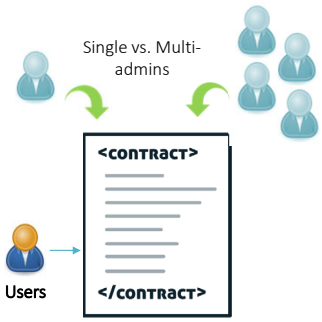


A Better Way

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.

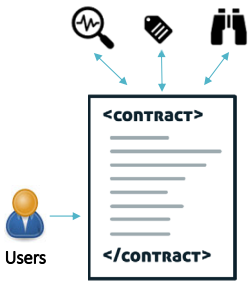
30

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.*

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.*

Technical Audit Considerations

33

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

34

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 }
```


35

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 }
```


36

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.

```
pragma solidity ^0.4.21;

contract HashForEther {

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

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


37

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);
    }
}
```


38

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;
}
```


39

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()().

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

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    require(msg.sender.call.value(amountToWithdraw)());
    // The user's balance is already 0, so future
    invocations won't withdraw anything
}
```


40

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 }
```


41

Security Considerations for Security Audit

Integer Overflow & Underflow (mitigation)

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

```
1  pragma solidity ^0.4.15;
2
3  library SafeMath {
4      function add(uint256 a, uint256 b) internal constant returns
       (uint256) {
5          uint256 c = a + b;
6          assert(c >= a);
7          return c;
8      }
9  }
10
11 contract Overflow {
12     uint private sellerBalance=0;
13
14     function safe_add(uint value) returns (bool) {
15         require(value + sellerBalance >= sellerBalance);
16         sellerBalance += value;
17     }
18 }
```


42

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.

```
1  pragma solidity ^0.4.21;
2
3  contract UncheckedSendValue {
4      uint weiLeft;
5      uint balance;
6      mapping(address => uint256) public balances;
7
8      function deposit () public payable {
9          balances[msg.sender] += msg.value;
10     }
11
12     function withdraw (uint _amount) public {
13         require(balances[msg.sender] >= _amount);
14         weiLeft -= _amount;
15         msg.sender.send(_amount);
16         }
17
18     function GetBalance() public constant returns(uint){
19         return this.balance;
20     }
21 }
```


43

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.

```
1  pragma solidity ^0.4.21;
2
3  contract UncheckedSendValue {
4      uint weiLeft;
5      uint balance;
6      mapping(address => uint256) public balances;
7
8      function deposit () public payable {
9          balances[msg.sender] += msg.value;
10     }
11
12     function withdraw (uint _amount) public {
13         require(balances[msg.sender] >= _amount);
14         if (msg.sender.send(_amount))
15             weiLeft -= _amount;
16         else throw;
17         }
18
19     function GetBalance() public constant returns(uint){
20         return this.balance;
21     }
22 }
```


44

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);
    }
}
```


45

Security Considerations for Security Audit

Timestamp Manipulation (mitigation)

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

```
const contract = web3.eth.contract(contractAbi);
const contractInstance = contract.at(contractAddress);

contractInstance.timer('time_counterjs');
// send current time value
    time_counterjs +=1;
});
```

```
pragma solidity ^0.4.21;

contract TimestampManipulation {
    address public owner;
    uint time_counter;
    uint max_counter = 1521763200;

    function TimestampManipulation() public {
        owner = msg.sender
    }

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

    // Using an external initiator such as a JS
    // function to trigger at some intervals
    function timer(currenttime_count) public {
        require(msg.sender == owner);
        time_counter = currenttime_count;
    }

}
```


46

Security Considerations for Security Audit

Bad Randomness (anti-pattern)

Poor implementation of pseudo-random number generator

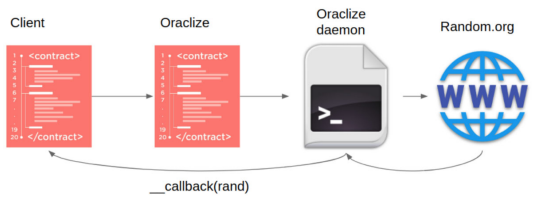
- Private variables are set via a transaction at some point in time and are visible on the blockchain.
- Block variables such as block.timestamp, block.coinbase, block.number can be manipulated by miners.

```
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 }
```


47

Security Considerations for Security Audit

Bad Randomness (mitigation)



48

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);
    }
}
```


49

Security Considerations for Security Audit

Front Running (mitigation)

- Usage of commit-reveal approach (RandDAO)

50

How do I get Hands-on Experience?

51

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

52

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!

53

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

58

Final Words

- The effectiveness of the audit 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.

59

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

60