2019 AFRICA CACS AN ISACA EVENT

# Auditing Smart Contracts

Tuan Phan, CISSP, PMP, CBSP, Security+, SSBB
Partner, Caplock Security LLC
tphan@caplocksecurity.com @ChainOpSec LinkedIn.com/in/tuanphan/



1

## Learning Objectives

1. Identify possible use cases of smart contracts.
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. Recognize the technical, operational, and cybersecurity risks of smart contracts, and what controls can be implemented to minimize the risks from the use of smart contracts.

## Agenda

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

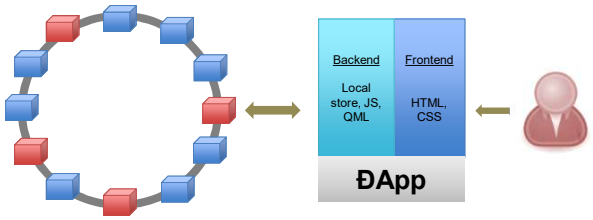



## What is a Smart Contract

- Is a computer program that prescribes its conditions and outcomes.
- Is stored and processed on a distributed ledger.
- Stays dormant until called by a transaction.
- Transactions performed are written onto the distributed ledger.

## Đapps and Smart Contracts

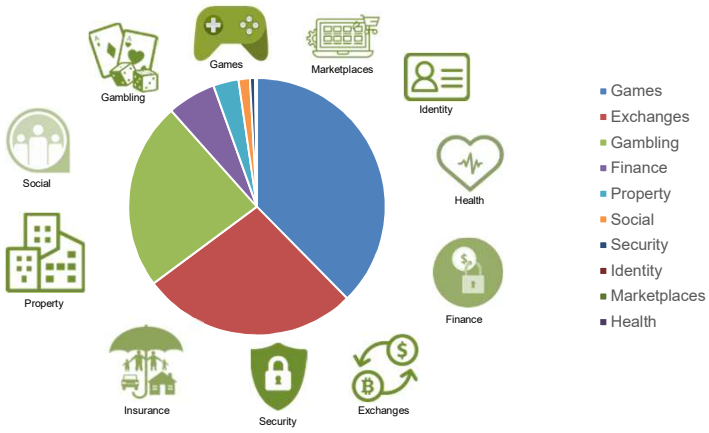
- ĐApps are blockchain-enabled applications/websites
- Rely on smart contracts for logic processing.







## Use Cases for Smart Contracts



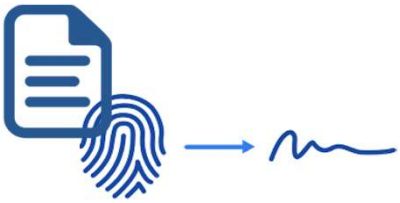
## Key Regulatory Drivers

United States

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

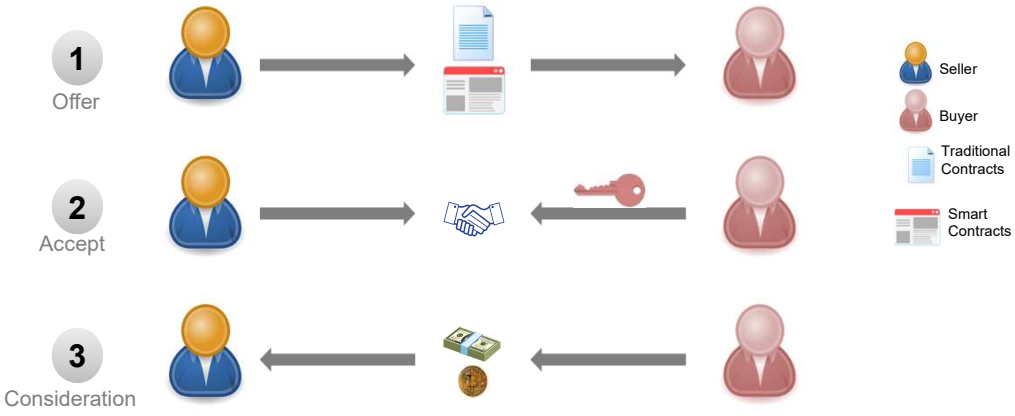
European Union

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






## Are Smart Contracts Legally Binding?

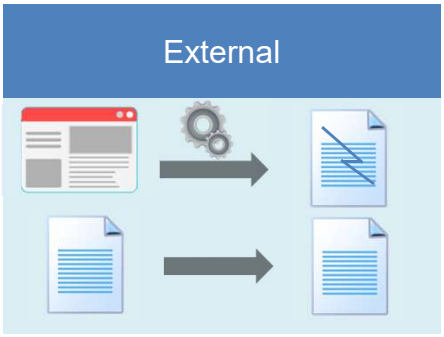


## Implementation Models of Smart Contracts

- Smart contracts vs. traditional (natural language) contracts
  - Consumer-centric
  - Completeness
  - Governance, amendments and disputes

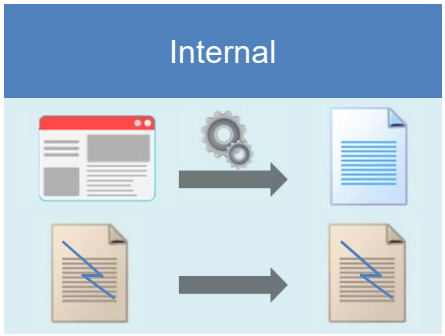



## External Model




- The traditional contract is the contract for terms and conditions between the parties.
- Limited automation are handled by the smart contract portion.
- The specific automation is explicitly defined in the T&C of the contract.

## Internal Model




- Recognizes the code as law and the smart contract is the contract.
- Only non-operational attributes are handled thru. the traditional contract.




## Key Characteristics of Smart Contract

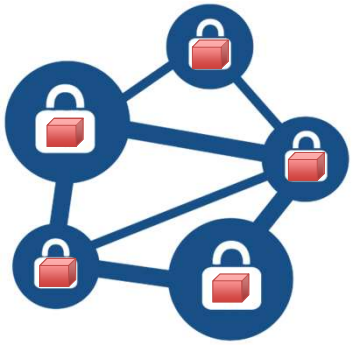
### Turing Completeness

```
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 {}
}
```

## Key Characteristics of Smart Contract

### Immutability

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

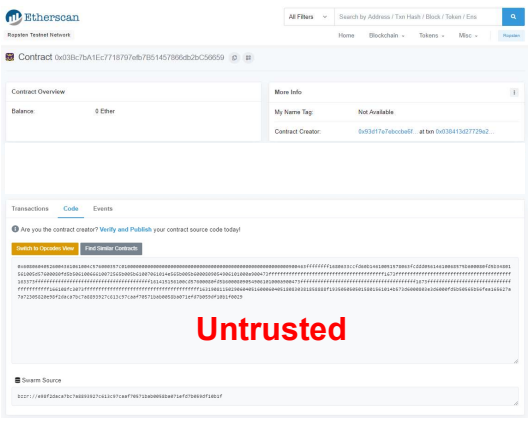


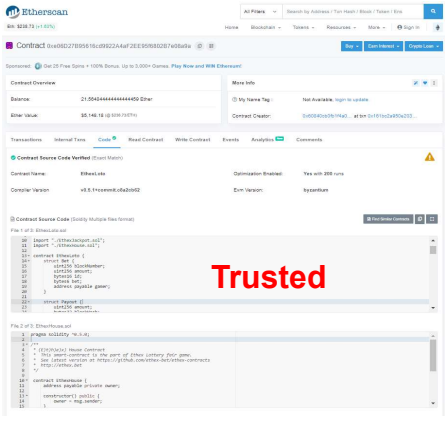




## Key Characteristics of Smart Contract

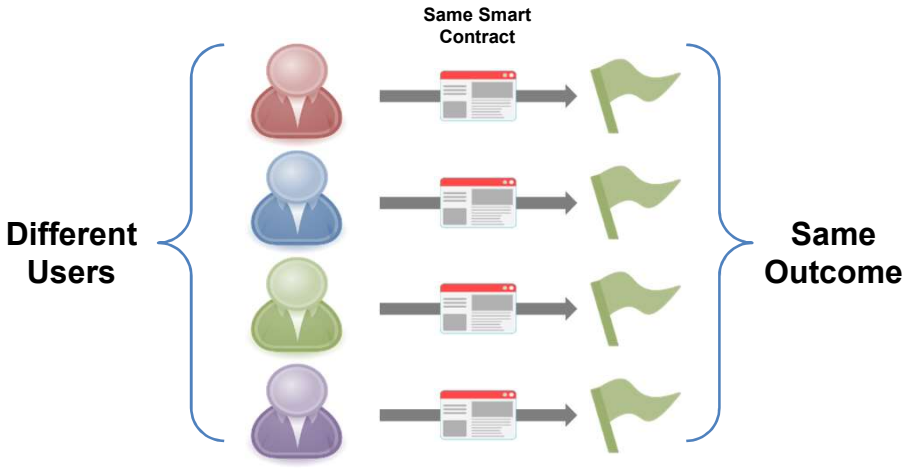
### Visibility



## Key Characteristics of Smart Contract

### Deterministic

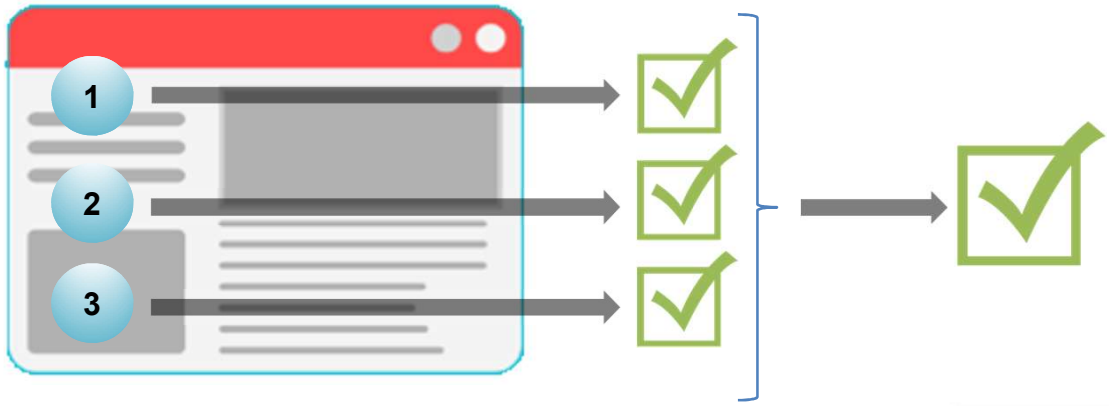






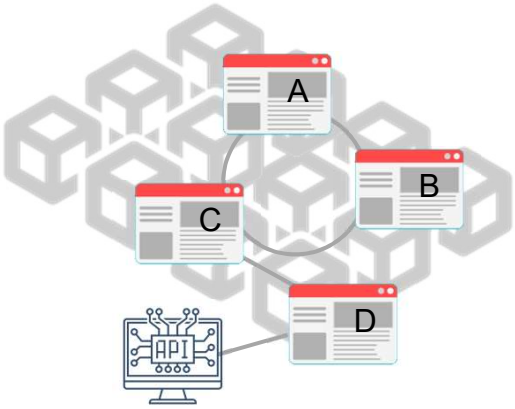
## Key Characteristics of Smart Contract

### Atomic



## Key Characteristics of Smart Contract

### Interaction with Other Interfaces







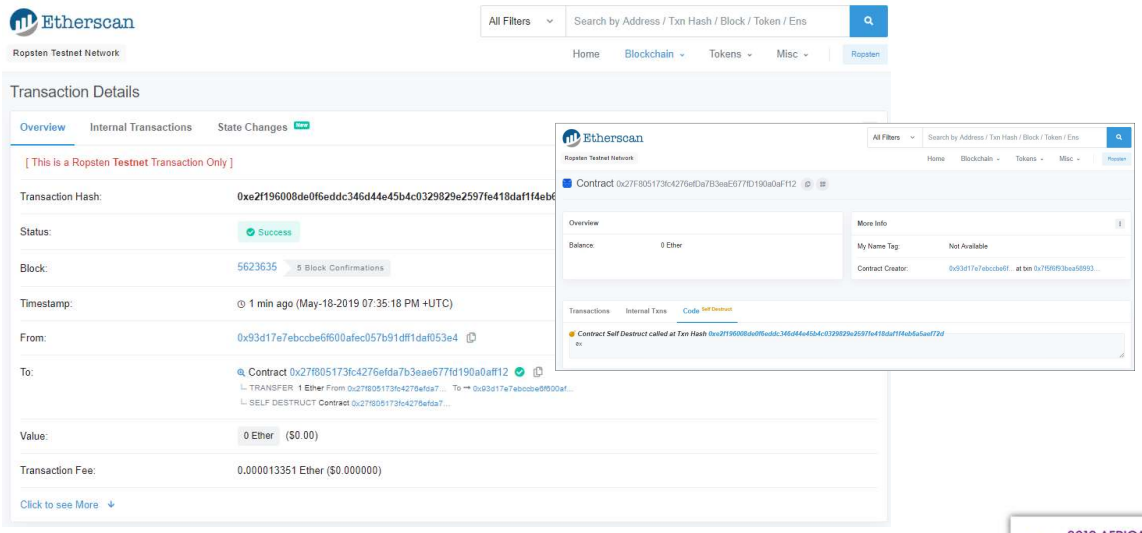
## Key Characteristics of Smart Contract

### 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);
        }
}
```

- 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.

## Self-Destructed Smart Contracts








## 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

## 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




## Audit Considerations from External Factors

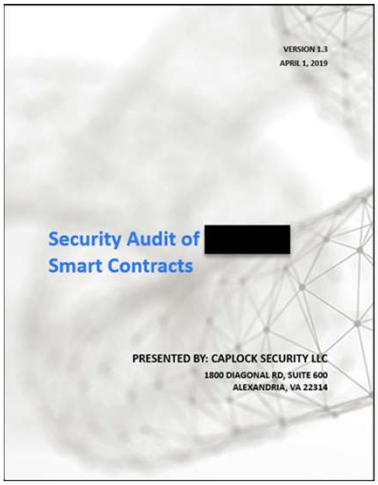

- Regulators
- Herstatt (settlement) risk
- Privacy
- Platform dependencies:
  - Development/Ongoing Support
  - Security issues
  - Speed of transactions
  - Cost of transactions
  - Scalability

## Audit Considerations for the Contract


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




## Sample of Security Audit Report





## 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)**

```
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 }
```

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.

## Security Considerations for Security Audit

### Access Control (mitigation)

```
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 }
```

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




## Security Considerations for Security Audit

### Default Visibility (anti-pattern)

```
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 }
```

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

---

- No visibility identifier stated.

## Security Considerations for Security Audit

### Default Visibility (mitigation)

```
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);
    }
}
```

- 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)





## Security Considerations for Security Audit

### Reentrancy (anti-pattern)

```
// 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;
}
```

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

## Security Considerations for Security Audit

### Reentrancy (mitigation)

```
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
```

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




## Security Considerations for Security Audit

### Integer Overflow & Underflow (anti-pattern)

```
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 }
```

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)

## Security Considerations for Security Audit

### Integer Overflow & Underflow (mitigation)

```
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 }
```

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




## Security Considerations for Security Audit

### Unchecked Return Values (anti-pattern)

```
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 }
```

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.

## Security Considerations for Security Audit

### Unchecked Return Values (mitigation)

```
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 }
```

- 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.





## Security Considerations for Security Audit

### Timestamp Manipulation (anti-pattern)

```
1  pragma solidity ^0.4.21;
2
3  contract TimestampManipulation {
4      uint time_counter;
5      uint max_counter = 1521763200;
6
7      function play() public {
8          require(now > 1521763200 && neverPlayed == true);
9          neverPlayed = false;
10         msg.sender.transfer(1500 ether);
11     }
12 }
```

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.

## Security Considerations for Security Audit

### Timestamp Manipulation (mitigation)

```
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(now > 1521763200 && 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 }
```

- 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  });
```





## Security Considerations for Security Audit

### Bad Randomness (anti-pattern)

```
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 }
```

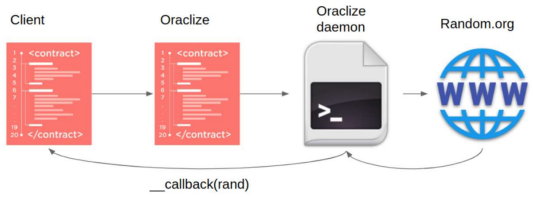
Poor implementation of pseudo-random number generator

---

- 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.

## Security Considerations for Security Audit

### Bad Randomness (mitigation)




- Use blockhash of some future block.
- Use external oracles (Oraclize)




## Security Considerations for Security Audit

### Front Running (anti-pattern)

```
pragma solidity ^0.4.21;

contract FindThisHash {
    bytes32 constant public hash =

      0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    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);
    }
}
```

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.

## Security Considerations for Security Audit

### Front Running (mitigation)


- Usage of commit-reveal approach (RandDAO)




## Security Considerations for Security Audit

### Denials of Service (anti-pattern)

```
function becomePresident() payable {
    // must pay the price to become president
    require(msg.value >= price);

    // we pay the previous president
    president.transfer(price);

    // we crown the new president
    president = msg.sender;

    // we double the price to become president
    price = price * 2;
}
```

```
function selectNextWinners(uint256 _largestWinner) {
    for(uint256 i = 0; i < largestWinner, i++) {
        // some code
    }
    largestWinner = _largestWinner;
}
```

The attacker manipulates one or more conditions causing the contract to fail and essentially disables the smart contract.

---

- Blocking access to owner account
- Locked functions
- Artificially inflating arrays
- Exceeding gas limit

## Security Considerations for Security Audit

### Denials of Service (mitigation)

```
1  function becomePresident() payable {
2      // must pay the price to become president
3      require(msg.value >= price);
4
5      // we pay the previous president
6      president.transfer(price);
7
8      // we crown the new president
9      president = msg.sender;
10
11     // we double the price to become president
12     price = price * 2;
13 }
```

```
1  function selectNextWinners(uint256 _largestWinner) {
2      for(uint256 i = 0; i < largestWinner, i++) {
3          // some code
4      }
5      largestWinner = _largestWinner;
6  }
```

- Coding and user testing in testnet
- Expert code review
- Safeguarding private keys




## Security Considerations for Security Audit

### Short Address (anti-pattern)

Actual data

Msg.data = [function hash, address, amount]

function = transfer (address, amount)

Address = 0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f00

Amount = uint256

Manipulation

transfer(0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f0, uint256,0)

0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f00

Amount increases by a factor of 2

Is when a contract receives less data than it was expecting, and Solidity fills the missing bytes with zeros.

---

- Higher amount for the transaction vs. the intended transaction amount

## Security Considerations for Security Audit

**Short Address (mitigation)**

```
1  contract NonPayloadAttackableToken {
2    modifier onlyPayloadSize(uint size) {
3      assert(msg.data.length == size + 4);
4      _;
5    }
6
7    function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
8      // do stuff
9    }
10 }
```

- Verify if the amount of data is the same amount expected by the contract.
- Perform coding and user testing in testnet.
- Conduct expert code review.



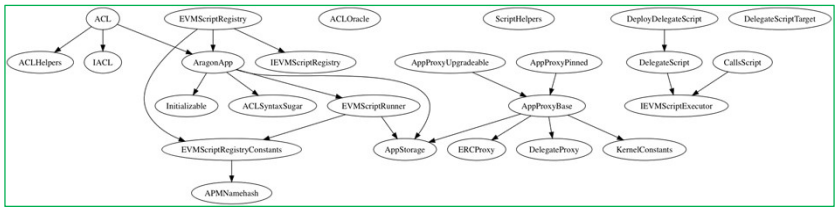

## Security Considerations for Security Audit

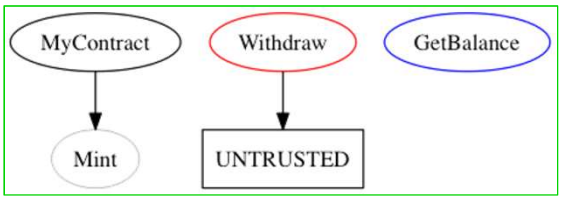
**Unknown Unknowns**

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

## Visualization Tools

- Visualize function control flow of a contract and highlights potential security vulnerabilities:
  - Surya
  - Solgraph
  - EVM Lab
  - Ethereum graph debugger







## Static and Dynamic Analysis Tools

- Use symbolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities.
  - Mythril
  - Slither
  - Echidna
  - Oyente
  - Securify
  - SmartCheck
  - Octopus
  - Chaincode Scanner*

## Test Coverage & Linters

- Ensure that test evaluate all of the code under test.
  - Solidity Coverage (test coverage)
- Improve code quality by enforcing rules for style and composition, making code easier to read and review.
  - Solcheck
  - Solint
  - Solium
  - Solhint




## Best Practices for Smart Contracts

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

**Follow Occam's razor**

## What have we learned from the Learning Objectives

1. Highlighted the many use cases of smart contracts.
2. Defined the basis for the legality of smart contracts.
3. Discussed the seven key concepts of smart contracts, and how the smart contracts may differ between permissioned and permissionless blockchains.
4. Addressed the technical, operational, and cybersecurity risks of smart contracts, and what controls can be implemented to minimize the risks from the use of smart contracts.




## Final Words

- Smart contracts are highly experimental and with limitations.
- The effectiveness of the audit is highly dependent on the auditor's understanding of the underlying mechanisms of the smart contract and the blockchain platform.
- Security audit is paramount for smart contracts.
- Transparency, expert reviews, user testing and use of automated security tools are mechanisms to minimize vulnerabilities.

2019 AFRICA CACS
AN ISACA EVENT

# THANK YOU FOR YOUR TIME

Tuan Phan, CISSP, PMP, CBSP, Security+, SSBB
Partner, Caplock Security LLC
tphan@caplocksecurity.com @ChainOpSec LinkedIn.com/in/tuanphan/

ISACA



54