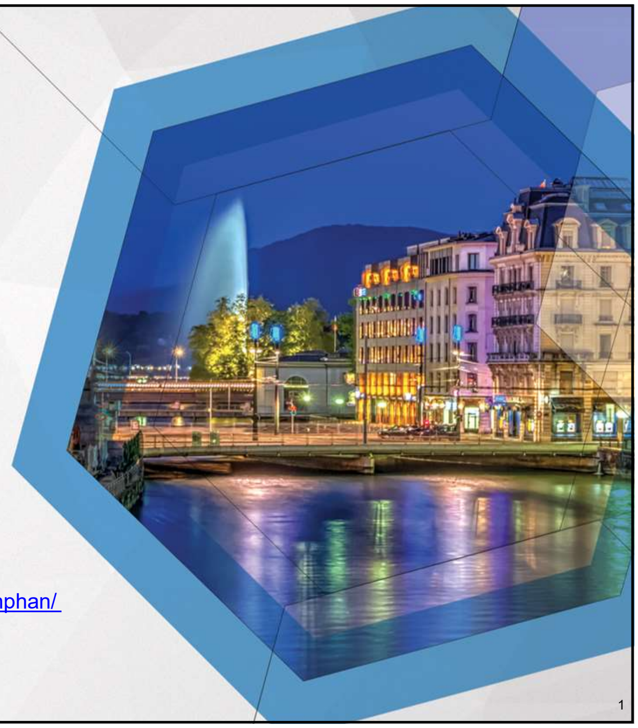
An ISACA® 2019 European Conference



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.

#EuroCACSCSX

An ISACA® 2019 European Conference

2

Agenda

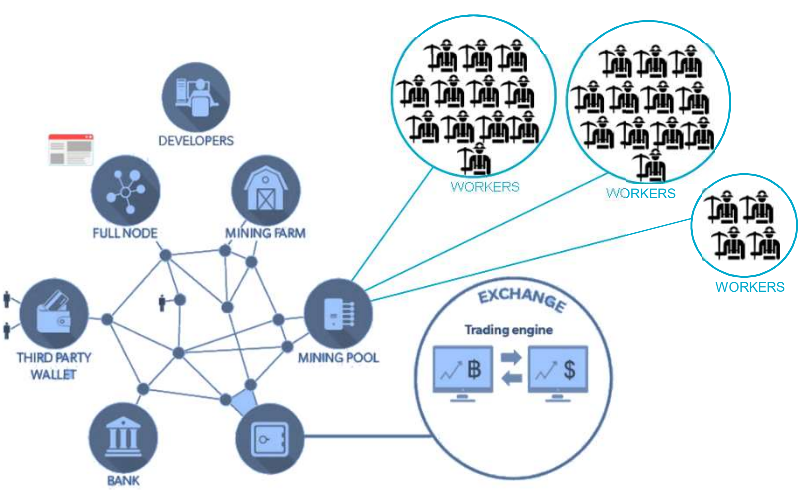
- What is a Smart Contract?
- Use Cases
- Regulatory Drivers
- Legality
- Characteristics and Programming
- Smart Contract Audit Considerations
- Tools
- Best Practices
- Final Words

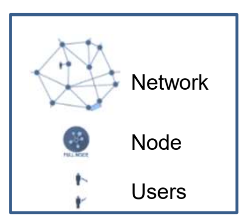
What is a Smart Contract?

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.

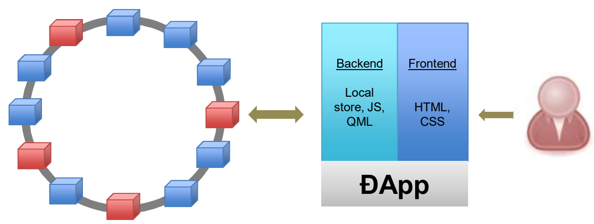
Typical Blockchain Network



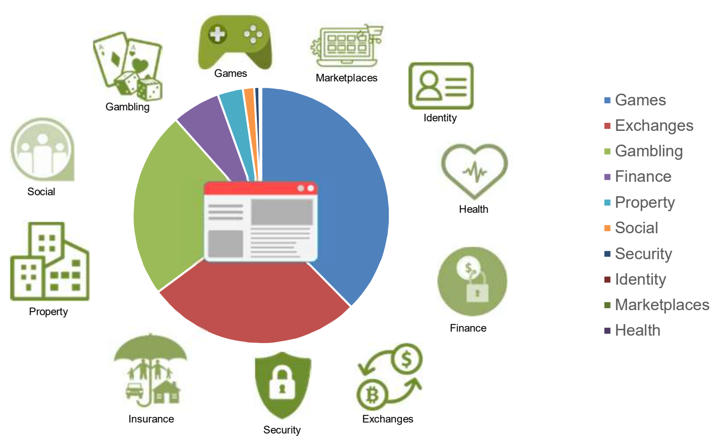


Đapps and Smart Contracts

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



Use Cases for Smart Contracts

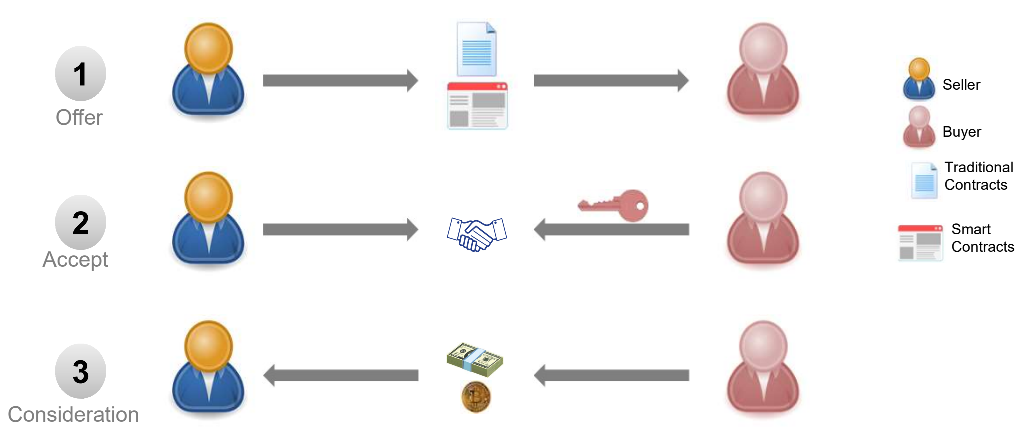


Key Regulatory Drivers

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?



Smart Contract vs. Traditional Contract

#EuroCACSCSX

CACS | CSX CYBERSECURITY NEXUS

An ISACA® 2019 European Conference

Traditional (natural language) Contracts vs. Smart Contracts

- Consumer-centric
- Completeness
- Governance, amendments and disputes

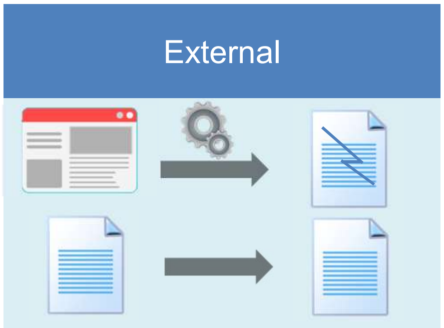
#EuroCACSCSX

CACS | CSX CYBERSECURITY NEXUS

An ISACA® 2019 European Conference

12

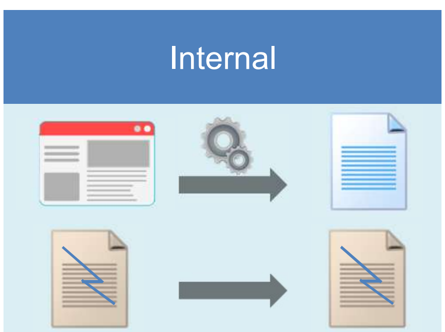
External Model



- The traditional contract is the contract for terms and conditions (T&C) 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.

#EuroCACSCSX

Internal Model



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

#EuroCACSCSX

Key Properties of a Smart Contract

#EuroCACSCSX

An ISACA® 2019 European Conference

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 {}
}
```


#EuroCACSCSX

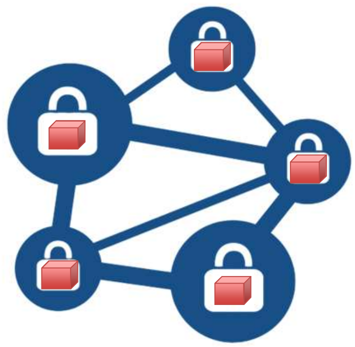
An ISACA® 2019 European Conference

16

Key Characteristics of Smart Contract

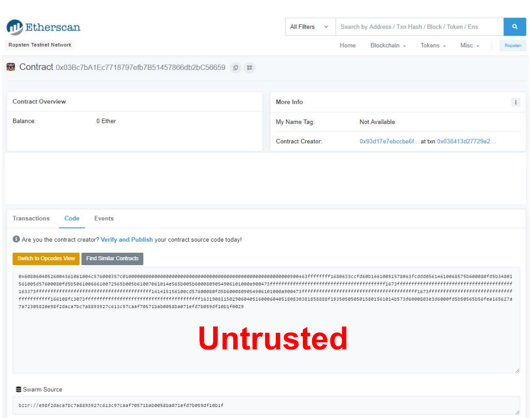
Immutability

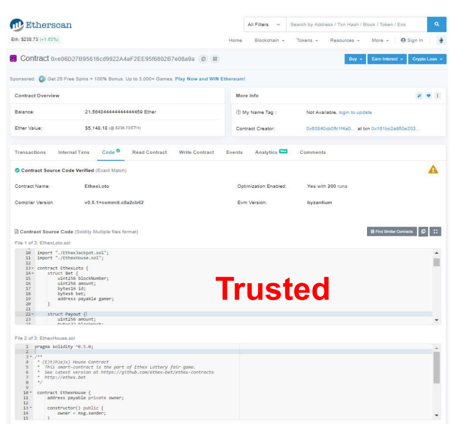
- Cannot be changed.
- Cannot be disabled.
- Cannot be removed.
- May be self-destruct if preprogrammed.



Key Characteristics of Smart Contract

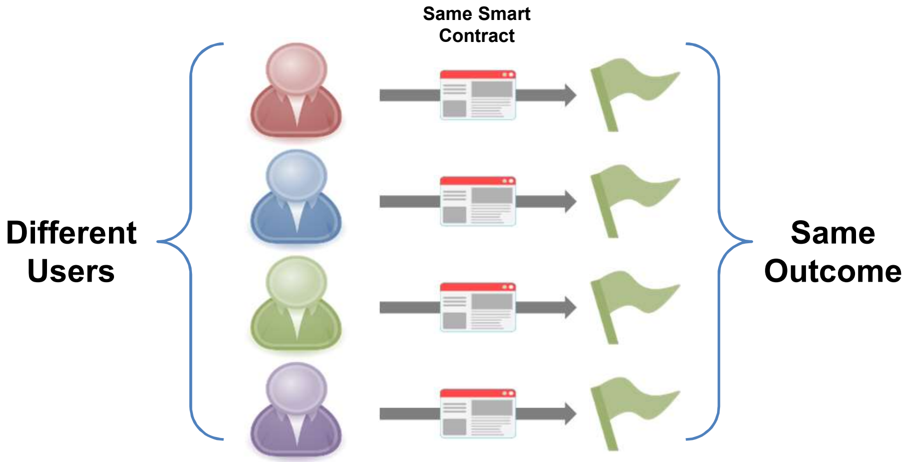
Visibility





Key Characteristics of Smart Contract

Deterministic

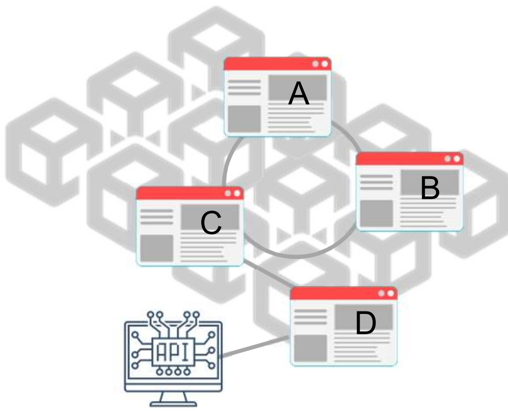


Key Characteristics of Smart Contract

Atomic

Key Characteristics of Smart Contract

Interaction with Other Interfaces



#EuroCACSCSX

An ISACA® 2019 European Conference

21

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);
        }
}
```

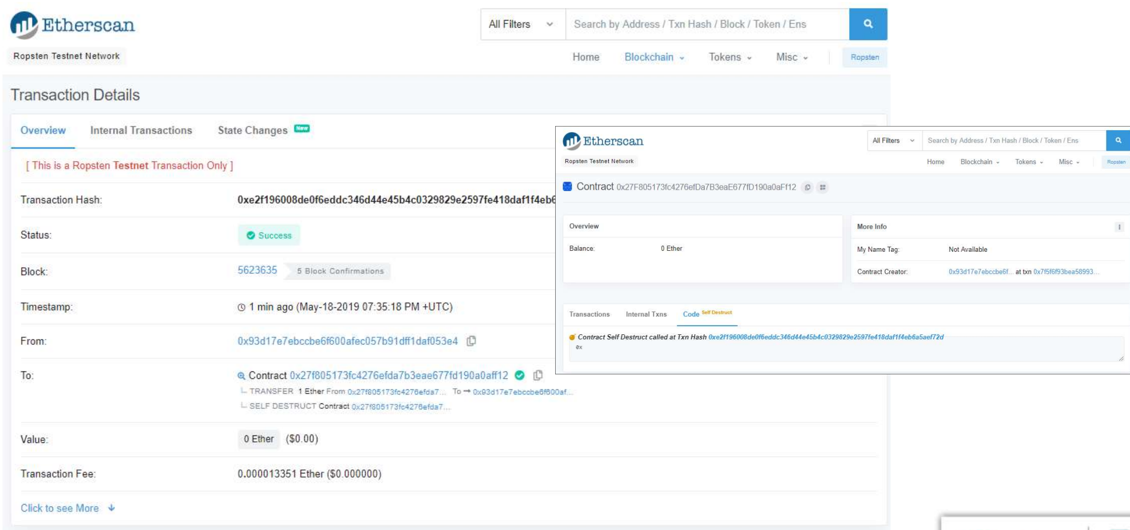
- Preprogrammed
- One-time event
- Does not remove transaction history.
- Return any values in the contract back to the contract owner when called.
- Any value sent to self-destructed contract is lost forever.

#EuroCACSCSX

An ISACA® 2019 European Conference

22

Self-Destructed Smart Contracts



#EuroCACSCSX

23

How do I get Hands-on Experience?

#EuroCACSCSX

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

#EuroCACSCSX

25

Audit Considerations

#EuroCACSCSX

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

#EuroCACSCSX

An ISACA® 2019 European Conference

Audit Considerations from External Factors

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

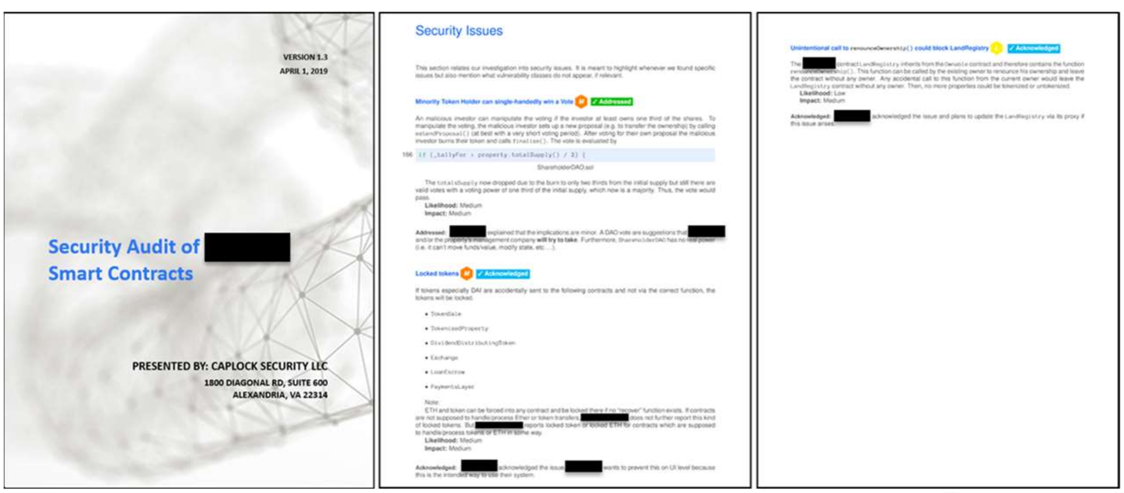
#EuroCACSCSX

An ISACA® 2019 European Conference

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)

```
pragma solidity ^0.4.21;

contract OwnerWallet {
    address public owner;

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

    // Fallback. Collect ether.
    function () payable {}

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

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)

```
pragma solidity ^0.4.21;

contract OwnerWallet {
    address public owner;

    // constructor to initialize ownership
    function OwnerWallet() public {
        owner = msg.sender;
    }

    // Fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

- 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)

```
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);
    }
}
```

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)

```
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
}

```

- 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)

```
pragma solidity ^0.4.15;

contract Overflow {
    uint private sellerBalance=0;

    function add(uint value) returns (bool){
        sellerBalance += value; // possible overflow

        // possible auditor assert
        // assert(sellerBalance >= value);
    }
}
```

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)

```
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;
    }
}
```

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

#EuroCACSCSX

An ISACA® 2019 European Conference

40

Security Considerations for Security Audit

Unchecked Return Values (anti-pattern)

```
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;
    }
}
```

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.

#EuroCACSCSX

An ISACA® 2019 European Conference

41

Security Considerations for Security Audit

Unchecked Return Values (mitigation)

```
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;
    }
}
```

- 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)

```
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);
    }
}
```

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)

```
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;
    }

}
```

- 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;
});
```


Security Considerations for Security Audit

Bad Randomness (anti-pattern)

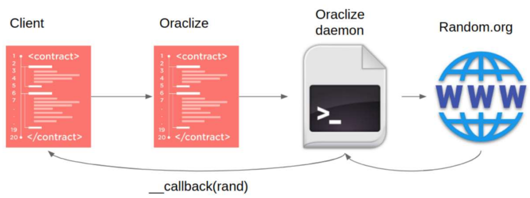
```
uint256 constant private salt =  block.timestamp;

function random(uint Max) constant private returns (uint256 result){
    //get the best seed for randomness
    uint256 x = salt * 100/Max;
    uint256 y = salt * block.number/(salt % 5) ;
    uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
    uint256 h = uint256(block.blockhash(seed));

    return uint256((h / x)) % Max + 1; //random number between 1 and
    Max
}
```

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)
- Use of RandDAO

#EuroCACSCSX

Security Considerations for Security Audit

Front Running (anti-pattern)

```
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);
    }
}
```

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.

#EuroCACSCSX

Security Considerations for Security Audit

Front Running (mitigation)

- Usage of commit-reveal approach (RandDAO)

#EuroCACSCSX

An ISACA® 2019 European Conference

48

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!

#EuroCACSCSX

An ISACA® 2019 European Conference

53

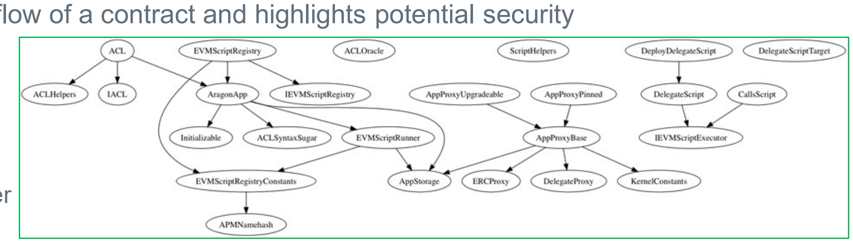
Help is Available

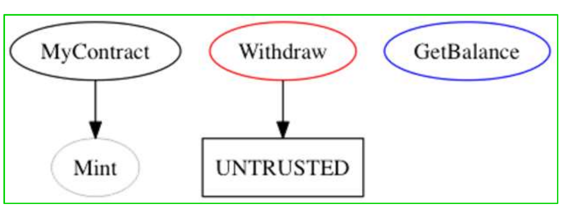
#EuroCACSCSX

An ISACA® 2019 European Conference

Static and Dynamic Analysis Tools

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





#EuroCACSCSX

An ISACA® 2019 European Conference

55

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 testing evaluates all of the code under test.
- Improve code quality by enforcing rules for style and composition, making code easier to read and review.
  - Solidity Coverage (test coverage)
  - 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.

#EuroCACSCSX

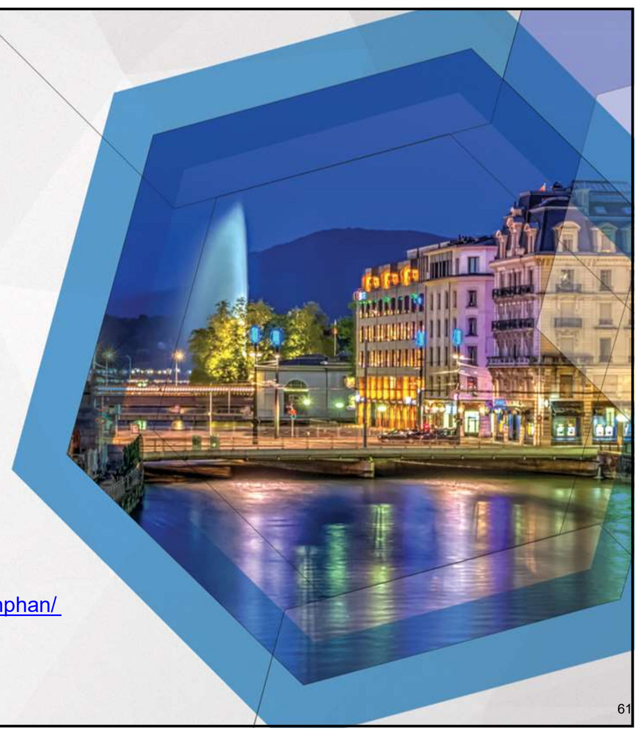
An ISACA® 2019 European Conference

60

An ISACA® 2019 European Conference

THANK YOU FOR YOUR TIME.

PLEASE COMPLETE THE SURVEY!



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

61