

시큐어 코딩 가이드

스마트 컨트랙트를 위한 솔리디티 시큐어 코딩 가이드

Institute: 고려대학교 블록체인 연구센터

Date: October. 13, 2022

Version: 1.3



Contents

1 시큐어코딩 가이드	1
1.1 입력데이터 검증 및 표현	1
1. 함수 기본 가시성(Function Default Visibility)	1
2. 정수 오버플로우 및 언더플로우 (Integer Overflow and Underflow)	3
3. 오래된 컴파일러 버전(Outdated Compiler Version)	5
1.2 보안기능	6
1. 플로팅 프래그마(FloatingPragma)	6
2. 확인되지 않은 반환 값(Unchecked Call Return Value)	8
3. 보호되지 않은 이더 인출(Unprotected Ether Withdrawal)	10
4. 보호되지 않은 SELFDESTRUCT 명령어(Unprotected SELFDESTRUCT instruction)	13
5. 재진입(Reentrancy)	15
6. 상태 변수 기본 가시성(State Variable Default Visibility)	19
7. 초기화되지 않은 스토리지 포인터(Uninitialized Storage Pointer)	21
8. 어서션 위반(Assert Violation)	24
1.3 시간 및 상태	26
1. 더 이상 사용되지 않는 Solidity 함수 사용(Use of Deprecated Solidity Functions)	26
2. 신뢰할 수 없는 Callee에게 Delegatecall(Delegatecall to Untrusted Callee)	29
3. 호출실패로 인한 DoS(DoS with Failed Call)	31
4. 트랜잭션 주문 종속성(Transaction Order Dependence)	33
5. tx.origin을 통한 인증(Authorization through tx.origin)	35
6. 시간의 대체값으로 블록값 사용(Block values as a proxy for time)	37
7. 서명 가단성(Signature Malleability)	39
8. 잘못된 생성자 이름(Incorrect Constructor Name)	44
1.4 에러 처리	46
1. 새도잉 상태 변수(Shadowing State Variables)	46
2. 부적절한 난수값(Weak Sources of Randomness from Chain Attributes)	48
3. 서명 재생 공격에 대한 보호 부재(Missing Protection against Signature Replay Attacks)	51
4. 적절한 서명 검증 부족(Lack of Proper Signature Verification)	52
5. 요구사항 위반(Requirement Violation)	53
6. 임의의 저장 위치에 쓰기(Write to Arbitrary Storage Location)	55
7. 잘못된 상속 순서(Incorrect Inheritance Order)	58
8. 불충분한 가스 그리핑(Insufficient Gas Griefing)	61
9. 함수 유형 변수를 사용한 임의 점프(Arbitrary Jump with Function Type Variable)	64
1.5 코드 오류	66
1. 블록 가스 한도를 사용한 DoS(DoS With Block Gas Limit)	66
2. 오타(Typographical Error)	68
3. RLO 유니코드기법 (Right-To-Left-Override control character (U+202E))	69
4. 사용되지 않는 변수 존재(Presence of unused variables)	71
5. 예상치 못한 이더 잔액(Unexpected Ether balance)	73

6.	여러 가변 길이 인수가 있는 해시 충돌(Hash Collisions With Multiple Variable Length Arguments)	74
7.	하드 코딩된 가스량 (Message call with hardcoded gas amount)	77
8.	효과가 없는 코드(Code With No Effects)	79
9.	암호화 되지 않은 개인 데이터 온체인(Unencrypted Private Data On-Chain)	81

제 1장 시큐어코딩 가이드

1.1 입력데이터 검증 및 표현

1. 함수 기본 가시성(Function Default Visibility)

가. 정의 (Description)

함수 가시성이 지정되지 않은 함수는 기본적으로 `public`의 타입으로 설정됩니다. 이는 개발자가 가시성을 설정하는 것을 빠트리고 악의적인 사용자가 무단 또는 의도하지 않은 상태 변경을 할 수 있는 경우 취약점으로 이어질 수 있습니다.

나. 관련 보안약점

- CWE-710: Improper Adherence to Coding Standards

다. 시큐어 코딩기법

- 함수의 유형은 `external`, `public`, `internal` 또는 `private`으로 지정할 수 있습니다. 함수에 적합한 가시성 유형을 명시적으로 선언하는 것이 좋습니다. 이것을 통해 스마트 컨트랙트 시스템의 공격 표면을 크게 줄일 수 있습니다.

라. 예제

```
1 *******/
2 /*                      Insecure code                      */
3 *******/
4
5 pragma solidity ^0.4.24;
6
7 contract HashForEther {
8     function withdrawWinnings() {
9         // Winner if the last 8 hex characters of the address are 0.
10        require(uint32(msg.sender) == 0);
11        _sendWinnings();
12    }
13    function _sendWinnings() {
14        msg.sender.transfer(this.balance);
15    }
16 }
17 *******/
18 /*                      Secure code                      */
19 *******/
20
21
22 pragma solidity ^0.4.24;
23
24 contract HashForEther {
25     function withdrawWinnings() public {
```

```

26     // Winner if the last 8 hex characters of the address are 0.
27     require(uint32(msg.sender) == 0);
28     _sendWinnings();
29 }
30 function _sendWinnings() internal{
31     msg.sender.transfer(this.balance);
32 }
33 }
```

마. 참고문헌

- [1] Ethereum Smart Contract Best Practices - Explicitly mark visibility in functions and state variables
- [2] SigmaPrime - Visibility

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
			V	V

2. 정수 오버플로우 및 언더플로우 (Integer Overflow and Underflow)

카테고리: V5: Arithmetic

가. 정의 (Description)

정수 오버플로우 및 언더플로우는 연산 시 정수형 변수가 취할 수 있는 값의 범위를 초과할 경우에 발생한다. 이로 인해 스마트 컨트랙트에서 비정상적인 거래가 진행될 수 있는 문제가 있다.

나. 관련 보안약점

- CWE-682 : Incorrect Calculation
- CWE-190 : Integer Overflow or Wraparound

다. 시큐어 코딩기법

- 일반적인 연산 라이브러리를 사용하는 것 대신에 SafeMath 라이브러리를 사용한다.

라. 예제

```

1  /************************************************************************/
2  /*                               Insecure code                         */
3  /************************************************************************/
4
5 pragma solidity 0.4.24;
6
7 contract Overflow_Add {
8     uint public balance = 1;
9
10    function add(uint256 deposit) public {
11        balance += deposit;
12    }
13}
14
15 /************************************************************************/
16 /*                               Secure code                          */
17 /************************************************************************/
18
19 pragma solidity ^0.4.24;
20
21 contract Overflow_Add {
22     uint public balance = 1;
23
24     function add(uint256 deposit) public {
25         balance = add(balance, deposit);
26
27     } //from SafeMath
28
29     function add(uint256 a, uint256 b) internal pure returns (uint256) {
30         uint256 c = a + b;
31         require(c >= a);

```

```
32     return c;
33 }
34 }
```

마. 참고문헌

- [1] CWE-682: Incorrection Calculation, <https://cwe.mitre.org/data/definitions/682.html>
- [2] CWE-190: Integer Overflow or Wraparound, <https://cwe.mitre.org/data/definitions/190.html>
- [3] CWE-191: Integer Underflow(Wrap or Wraparound), <https://cwe.mitre.org/data/definitions/191.html>
- [4] SWC-101: Integer Overflow and Underflow, <https://swcregistry.io/docs/SWC-101>
- [5] Ethereum Smart Contract Best Practices – Integer Overflow and Underflow

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
			V	V

3. 오래된 컴파일러 버전(Outdated Compiler Version)

카테고리: V1: ARCHITECTURE, DESIGN AND THREAT MODELLING

가. 정의 (Description)

오래된 컴파일러를 사용하는 것은 특히 현재 컴파일러 버전에도 영향을 미치는 버그가 있을 경우 문제가 될 수 있다.

나. 관련 보안약점

- CWE-937 : Using Components with Known Vulnerabilities

다. 시큐어 코딩기법

- 최신 버전의 솔리디티 컴파일러를 사용한다.

라. 예제

```

1  /*#####
2  /* Insecure code
3  #####*/
4
5 pragma solidity 0.4.13;
6
7 contract OutdatedCompilerVersion {
8     uint public x = 1;
9 }
```

마. 참고문헌

- [1] Solidity Release Notes
- [2] Etherscan Solidity Bug Info

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V				V

1.2 보안기능

1. 플로팅 프래그마(FloatingPragma)

가. 정의 (Description)

컨트랙트는 철저하게 테스트된 것과 동일한 버전의 컴파일러 및 플래그로 배포되어야 한다. Pragma를 고정하면 이전 버전의 컴파일러를 사용하여 버그가 존재할 수 있는 컨트랙트가 실수로 배포되지 않도록 하는 데 도움이 된다.

나. 관련 보안약점

- CWE-664 : Improper Control of a Resource Through its Lifetime

다. 시큐어 코딩기법

- Pragma 버전을 고정하고 선택한 컴파일러 버전에 대해 알려진 버그를 고려한다. Pragma는 라이브러리 또는 EthPM 패키지의 컨트랙트에서와 같이 다른 개발자들이 사용하도록 의도된 컨트랙트일 때 유동화되도록 허용할 수 있다. 그렇지 않으면, 개발자가 로컬에서 컴파일하기 위해 수동으로 pragma를 업데이트 해야 한다.

라. 예제

```

1 *******/
2 /*           Insecure code           */
3 *******/
4
5 pragma solidity ^0.4.0;
6
7 contract PragmaNotLocked {
8     uint public x = 1;
9 }
10
11 *******/
12 /*           Secure code           */
13 *******/
14
15 pragma solidity 0.4.25;
16
17 contract PragmaFixed {
18     uint public x = 1;
19 }
```

마. 참고문헌

- [1] Ethereum Smart Contract Best Practices - Lock pragmas to specific compiler version

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V	V	V

2. 확인되지 않은 반환 값(Unchecked Call Return Value)

가. 정의 (Description)

메시지 호출의 반환 값을 확인되지 않는다. 호출된 컨트랙트가 예외를 둬도 실행이 재개된다. 호출이 실패하거나 공격자가 강제로 호출을 실패하게 하면 이후 프로그램 논리에서 예기치 않은 동작이 발생할 수 있다.

나. 관련 보안약점

- CWE-252 : Unchecked Return Value

다. 시큐어 코딩기법

- 저수준 호출 방법을 사용할 경우 반환 값을 확인하여 호출이 실패할 가능성을 처리해야 한다.

라. 예제

```

1  *****
2  /*                               Insecure code                         */
3  *****
4
5  pragma solidity 0.4.25;
6
7  contract ReturnValue {
8      function callnotchecked(address callee) public {
9          callee.call();
10     }
11 }
12
13 *****
14 /*                               Secure code                         */
15 *****
16
17 pragma solidity 0.4.25;
18
19 contract ReturnValue {
20     function callchecked(address callee) public {
21         require(callee.call());
22     }
23 }
```

마. 참고문현

- [1] Ethereum Smart Contract Best Practices - Handle errors in external calls

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
		V	V	V

3. 보호되지 않은 이더 인출(Unprotected Ether Withdrawal)

가. 정의 (Description)

액세스 제어가 없거나 불충분한 경우에, 악의적인 당사자(parties)는 컨트랙트 계좌에서 이더의 일부 또는 전체를 인출할 수 있다. 이 버그는 의도치 않게 초기화 함수를 노출하여 발생할 수 있다. 생성자로 의도된 함수의 이름을 잘못 지정함으로써 생성자 코드는 런타임 바이트 코드로 종료되고 누구든지 컨트랙트를 초기화하기 위해 호출 될 수 있다.

나. 관련 보안약점

- CWE-284 : Improper Access Control

다. 시큐어 코딩기법

- 권한이 있는 당사자 또는 스마트 컨트랙트 사양에 따라서만 인출이 될 수 있도록 제어를 구현한다.

라. 예제

```

1 *******/
2 /*           Insecure code          */
3 ******/
4
5 pragma solidity ^0.4.23;
6
7 /**
8  * @title MultiOwnable
9  */
10 contract MultiOwnable {
11     address public root;
12     mapping (address => address) public owners; // owner => parent of owner
13
14 /**
15  * @dev The Ownable constructor sets the original 'owner' of the contract to the sender
16  * account.
17  */
18 constructor() public {
19     root = msg.sender;
20     owners[root] = root;
21 }
22
23 /**
24  * @dev Throws if called by any account other than the owner.
25  */
26 modifier onlyOwner() {
27     require(owners[msg.sender] != 0);
28     -
29 }
30
31 /**
32  * @dev Adding new owners

```

```

33 * Note that the "onlyOwner" modifier is missing here.
34 */
35 function newOwner(address _owner) external returns (bool) {
36   require(_owner != 0);
37   owners[_owner] = msg.sender;
38   return true;
39 }
40
41 /**
42  * @dev Deleting owners
43 */
44 function deleteOwner(address _owner) onlyOwner external returns (bool) {
45   require(owners[_owner] == msg.sender || (owners[_owner] != 0 && msg.sender == root));
46   owners[_owner] = 0;
47   return true;
48 }
49 }
50
51 contract TestContract is MultiOwnable {
52
53   function withdrawAll() onlyOwner {
54     msg.sender.transfer(this.balance);
55   }
56
57   function() payable {
58   }
59 }
60
61
62 /*#####*/
63 /*           Secure code           */
64 /*#####*/
65
66 pragma solidity ^0.4.23;
67
68 /**
69  * @title MultiOwnable
70 */
71 contract MultiOwnable {
72   address public root;
73   mapping (address => address) public owners; // owner => parent of owner
74
75 /**
76  * @dev The Ownable constructor sets the original 'owner' of the contract to the sender
77  * account.
78 */
79 constructor() public {
80   root = msg.sender;
81   owners[root] = root;
82 }
83
84 /**
85  * @dev Throws if called by any account other than the owner.

```

```

86  /*
87   modifier onlyOwner() {
88     require(owners[msg.sender] != 0);
89     _;
90   }
91
92 /**
93 * @dev Adding new owners
94 * Note that the "onlyOwner" modifier is missing here.
95 */
96 function newOwner(address _owner) onlyOwner external returns (bool) {
97   require(_owner != 0);
98   owners[_owner] = msg.sender;
99   return true;
100 }
101
102 /**
103 * @dev Deleting owners
104 */
105 function deleteOwner(address _owner) onlyOwner external returns (bool) {
106   require(owners[_owner] == msg.sender || (owners[_owner] != 0 && msg.sender == root));
107   owners[_owner] = 0;
108   return true;
109 }
110 }
111
112 contract TestContract is MultiOwnable {
113
114   function withdrawAll() onlyOwner {
115     msg.sender.transfer(this.balance);
116   }
117
118   function() payable {
119   }
120 }

```

마. 참고문헌

[1] Rubixi smart contract

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V		V

4. 보호되지 않은 SELFDESTRUCT 명령어(Unprotected SELFDESTRUCT instruction)

가. 정의 (Description)

접근 제어가 없거나 불충분한 경우에 악의적인 당사자가 컨트랙트를 자체 파기할 수 있다.

나. 관련 보안약점

- CWE-284 : improper Access Control

다. 시큐어 코딩기법

- 절대적으로 필요한 경우가 아니면 자체 파기 기능을 제거하는 것이 좋다. 해당 기능이 유효한 케이스가 있는 경우 여러 당사자가 승인해야 사용할 수 있도록 multisig scheme을 구현하는 것이 좋다.

라. 예제

```

1  /************************************************************************/
2  /*          Insecure code           */
3  /************************************************************************/
4
5 pragma solidity ^0.4.23;
6
7 contract SuicideMultiTxFeasible {
8     uint256 private initialized = 0;
9     uint256 public count = 1;
10
11    function init() public {
12        initialized = 1;
13    }
14
15    function run(uint256 input) {
16        if (initialized == 0) {
17            return;
18        }
19
20        selfdestruct(msg.sender);
21    }
22}
23
24  /************************************************************************/
25/*          Secure code           */
26/************************************************************************/
27
28pragma solidity ^0.4.23;
29
30contract SuicideMultiTxFeasible {
31    uint256 private initialized = 0;
32    uint256 public count = 1;
33
34    function init() public {
35        initialized = 1;

```

```

36 }
37
38 function run(uint256 input) {
39     if (initialized != 2) {
40         return;
41     }
42
43     selfdestruct(msg.sender);
44 }
45 }
```

마. 참고문헌

- [1] Parity “I accidentally killed it” bug

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V			V

5. 재진입(Reentrancy)

가. 정의 (Description)

외부 컨트랙트를 호출할 때 가장 큰 위험 중 하나는 제어 흐름(control flow)를 이어받을 수 있다는 점이다. 재진입 공격에서 악의적인 컨트랙트는 함수의 첫 번째 호출이 완료되기 전에 컨트랙트 호출로 다시 호출된다. 이것은 함수의 다른 호출들이 바람직하지 않은 방식으로 상호작용하게 할 수 있다.

나. 관련 보안약점

- CWE-841 : Improper Enforcement of Behavioral Workflow

다. 시큐어 코딩기법

- 호출이 실행되기 전에 내부 상태 변경이 모두 수행되었는지 확인한다. 이것을 Check-Effects-Interactions 패턴이라고 한다. 또는 OpenZeppelin의 ReentrancyGuard 같은 재진입 잠금을 사용한다.

라. 예제

```

1  /************************************************************************/
2  /* Insecure code */
3  /************************************************************************/
4
5 pragma solidity ^0.5.0;
6
7 contract ModifierEntrancy {
8     mapping (address => uint) public tokenBalance;
9     string constant name = "Nu Token";
10    Bank bank;
11
12    constructor() public{
13        bank = new Bank();
14    }
15
16    //If a contract has a zero balance and supports the token give them some token
17    function airDrop() hasNoBalance supportsToken public{
18        tokenBalance[msg.sender] += 20;
19    }
20
21    //Checks that the contract responds the way we want
22    modifier supportsToken() {
23        require(keccak256(abi.encodePacked("Nu Token")) == bank.supportsToken());
24        -
25    }
26
27    //Checks that the caller has a zero balance
28    modifier hasNoBalance {
29        require(tokenBalance[msg.sender] == 0);
30        -
31    }
32 }
```

```

33
34 contract Bank{
35     function supportsToken() external returns(bytes32) {
36         return keccak256(abi.encodePacked("Nu Token"));
37     }
38 }
39
40 /*#####
41 /*          Secure code
42 /*#####
43
44 pragma solidity ^0.5.0;
45
46 contract ModifierEntrancy {
47     mapping (address => uint) public tokenBalance;
48     string constant name = "Nu Token";
49     Bank bank;
50     constructor() public{
51         bank = new Bank();
52     }
53
54     //If a contract has a zero balance and supports the token give them some token
55     function airDrop() supportsToken hasNoBalance public{
56         // In the fixed version supportsToken comes before hasNoBalance
57         tokenBalance[msg.sender] += 20;
58     }
59
60     //Checks that the contract responds the way we want
61     modifier supportsToken() {
62         require(keccak256(abi.encodePacked("Nu Token")) == bank.supportsToken());
63        _;
64     }
65
66     //Checks that the caller has a zero balance
67     modifier hasNoBalance {
68         require(tokenBalance[msg.sender] == 0);
69        _;
70     }
71 }
72
73 contract Bank{
74     function supportsToken() external returns(bytes32){
75         return keccak256(abi.encodePacked("Nu Token")));
76     }
77 }
```

```

1 /*#####
2 /*          Insecure code
3 /*#####
4
5 pragma solidity 0.4.24;
6
7 contract SimpleDAO {
```

```

8   mapping (address => uint) public credit;
9
10  function donate(address to) payable public{
11      credit[to] += msg.value;
12  }
13
14  function withdraw(uint amount) public{
15      if (credit[msg.sender]>= amount) {
16          require(msg.sender.call.value(amount)());
17          credit[msg.sender]-=amount;
18      }
19  }
20
21  function queryCredit(address to) view public returns(uint){
22      return credit[to];
23  }
24}
25
26 /*#####
27 /*          Secure code
28 /*#####
29
30 pragma solidity 0.4.24;
31
32 contract SimpleDAO {
33     mapping (address => uint) public credit;
34
35     function donate(address to) payable public{
36         credit[to] += msg.value;
37     }
38
39     function withdraw(uint amount) public {
40         if (credit[msg.sender]>= amount) {
41             credit[msg.sender]-=amount;
42             require(msg.sender.call.value(amount)());
43         }
44     }
45
46     function queryCredit(address to) view public returns (uint){
47         return credit[to];
48     }
49 }
```

마. 참고문헌

- [1] Ethereum Smart Contract Best Practices - Reentrancy

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V	V	V	V

6. 상태 변수 기본 가시성(State Variable Default Visibility)

가. 정의 (Description)

가시성(Visibility)에 명시적으로 레이블을 지정하면 누가 변수에 접근할 수 있는지 쉽게 구별할 수 있다.

나. 관련 보안약점

- CWE-710 : Improper Adherence to Coding Standards

다. 시큐어 코딩기법

- 변수는 public, internal, private로 지정할 수 있다. 모든 상태 변수에 대한 가시성을 명시적으로 정의한다.

라. 예제

```

1  /*************************************************************************/
2  /*          Insecure code                                         */
3  /*************************************************************************/
4
5 pragma solidity 0.4.24;
6
7 contract TestStorage {
8
9     uint storeduint1 = 15;
10    uint constant constuint = 16;
11    uint32 investmentsDeadlineTimeStamp = uint32(now);
12
13    bytes16 string1 = "test1";
14    bytes32 private string2 = "test1236";
15    string public string3 = "lets string something";
16
17    mapping (address => uint) public uints1;
18    mapping (address => DeviceData) structs1;
19
20    uint[] uintarray;
21    DeviceData[] devicedataArray;
22
23    struct DeviceData {
24        string deviceBrand;
25        string deviceYear;
26        string batteryWearLevel;
27    }
28
29    function testStorage() public {
30        address address1 = 0xbccc714d56bc0da0fd33d96d2a87b680dd6d0df6;
31        address address2 = 0xaee905fdd3ed851e48d22059575b9f4245a82b04;
32
33        uints1[address1] = 88;
34        uints1[address2] = 99;
35

```

```

36     DeviceData memory dev1 = DeviceData("deviceBrand", "deviceYear", "wearLevel");
37
38     structs1[address1] = dev1;
39
40     uintarray.push(8000);
41     uintarray.push(9000);
42
43     devicedataArray.push(dev1);
44 }
45 }
```

마. 참고문헌

- [1] Ethereum Smart Contract Best Practices - Explicitly mark visibility in functions and state variables

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
		V		V

7. 초기화되지 않은 스토리지 포인터(Uninitialized Storage Pointer)

가. 정의 (Description)

초기화되지 않은 로컬 스토리지 변수는 컨트랙트에서 예상치 못한 스토리지 위치를 가리킬 수 있으며, 이로 인해 의도적이거나 의도하지 않은 취약점이 발생할 수 있다.

나. 관련 보안약점

- CWE-824 : Access of Uninitialized Pointer

다. 시큐어 코딩기법

- 실제로 그렇지 않은 경우가 많지만 컨트랙트에 스토리지 객체가 필요한지 확인한다. If a local variable is sufficient, mark the storage location of the variable explicitly with the memory attribute. 스토리지 변수가 필요한 경우 선언시 초기화 하고 스토리지 위치 저장을 추가적으로 명시한다. 컴파일러 버전 0.5.0 이상부터는 초기화되지 않은 스토리지 포인터가 있는 컨트랙트가 더 이상 컴파일되지 않으므로 이 문제가 해결되었다.

라. 예제

```

1  /*#####
2  *          Insecure code
3  #####*/
4
5 pragma solidity ^0.4.19;
6
7 contract CryptoRoulette {
8     uint256 private secretNumber;
9     uint256 public lastPlayed;
10    uint256 public betPrice = 0.1 ether;
11    address public ownerAddr;
12
13    struct Game {
14        address player;
15        uint256 number;
16    }
17    Game[] public gamesPlayed;
18
19    function CryptoRoulette() public {
20        ownerAddr = msg.sender;
21        shuffle();
22    }
23
24    function shuffle() internal {
25        // randomly set secretNumber with a value between 1 and 20
26        secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20 + 1;
27    }
28
29    function play(uint256 number) payable public {
30        require(msg.value >= betPrice && number <= 10);

```

```

31     Game game;
32     game.player = msg.sender;
33     game.number = number;
34     gamesPlayed.push(game);
35
36     if (number == secretNumber) {
37         // win!
38         msg.sender.transfer(this.balance);
39     }
40
41     shuffle();
42     lastPlayed = now;
43 }
44
45
46 function kill() public {
47     if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
48         suicide(msg.sender);
49     }
50 }
51
52 function() public payable {}
53 }
54
55 /*#####
56 /*          Secure code
57 /*#####
58
59 pragma solidity ^0.4.19;
60
61 contract CryptoRoulette {
62
63     uint256 private secretNumber;
64     uint256 public lastPlayed;
65     uint256 public betPrice = 0.1 ether;
66     address public ownerAddr;
67
68     struct Game {
69         address player;
70         uint256 number;
71     }
72     Game[] public gamesPlayed;
73
74     function CryptoRoulette() public {
75         ownerAddr = msg.sender;
76         shuffle();
77     }
78
79     function shuffle() internal {
80         // randomly set secretNumber with a value between 1 and 20
81         secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20 + 1;
82     }
83 }
```

```

84     function play(uint256 number) payable public {
85         require(msg.value >= betPrice && number <= 10);
86
87         Game memory game;
88         game.player = msg.sender;
89         game.number = number;
90         gamesPlayed.push(game);
91
92         if (number == secretNumber) {
93             // win!
94             msg.sender.transfer(this.balance);
95         }
96
97         shuffle();
98         lastPlayed = now;
99     }
100
101    function kill() public {
102        if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
103            suicide(msg.sender);
104        }
105    }
106
107    function() public payable { }
108 }
```

마. 참고문헌

[1] SigmaPrime - Uninitialized Storage Pointers

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V		V

8. 어서션 위반(Assert Violation)

가. 정의 (Description)

솔리디티의 assert() 함수는 불변성을 주장하기 위한 것이다. 올바르게 동작하는 코드는 결코 failing assert statement에 도달하면 안된다. 도달 가능한 assertion은 다음 두 가지 중 하나를 의미한다.

- 컨트랙트에 버그가 존재하여 유효하지 않은 상태로 들어갈 수 있다.
- assert statement가 잘못 사용된다. (e.g., 입력 유효성 검사)

나. 관련 보안약점

- CWE-670 : Always-Incorrect Control Flow Implementation

다. 시큐어 코딩기법

- assert()에서 확인된 조건이 실제로 변하지 않았는지 고려해야 한다. 그렇지 않은 경우 assert()함수를 require()함수로 대체한다. 예외가 예기치 않은 동작으로 인해 발생한 경우 어설션 위반을 허용하는 기본 버그를 수정해야 한다.

라. 예제

```

1  /*****  
2  /*                               Insecure code                         */  
3  *****/  
4  
5  pragma solidity ^0.4.21;  
6  
7  contract GasModel{  
8      uint x = 100;  
9      function check(){  
10         uint a = gasleft();  
11         x = x + 1;  
12         uint b = gasleft();  
13         assert(b > a);  
14     }  
15 }  
16  
17 /*****  
18 /*                               Secure code                          */  
19 *****/  
20  
21 pragma solidity ^0.4.21;  
22  
23 contract GasModelFixed{  
24     uint x = 100;  
25     function check(){  
26         uint a = gasleft();  
27         x = x + 1;  
28         uint b = gasleft();  
29         assert(b < a);  
30     }  
31 }
```

마. 참고문헌

[1] The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in EVM

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V			V

1.3 시간 및 상태

1. 더 이상 사용되지 않는 Solidity 함수 사용(Use of Deprecated Solidity Functions)

가. 정의 (Description)

솔리디티에서 더 이상 사용되지 않는 함수와 연산자가 있다. 이것들을 사용하면 코드의 품질이 저하되고 새로운 주요 컴파일러 버전에서는 더 이상 사용되지 않는 함수와 연산자로 인해 부작용과 컴파일 에러가 발생할 수 있다.

나. 관련 보안약점

- CWE-477 : Use of Obsolete Function

다. 시큐어 코딩기법

- 솔리디티는 사용되지 않는 명령어(constructs)에 대한 대안을 제공한다. 대부분은 별칭(alises)으로 오래된 명령어를 대체해도 현재 동작에 영향을 미치지 않는다.

Deprecated	Alternative
suicide(address)	selfdestruct(address)
block.blockhash(uint)	blockhash(uint)
sha3(...)	keccak256(...)
callcode(...)	delegatecall(...)
throw	revert()
msg.gas	gasleft
constant	view
var	corresponding type name

라. 예제

```

1 //#####
2 /*           Insecure code           */
3 //#####
4
5 pragma solidity ^0.4.24;
6
7 contract DeprecatedSimple {
8
9     // Do everything that's deprecated, then commit suicide.
10
11    function useDeprecated() public constant {
12
13        bytes32 blockhash = block.blockhash(0);
14        bytes32 hashofhash = sha3(blockhash);
15
16        uint gas = msg.gas;
17

```

```

18     if (gas == 0) {
19         throw;
20     }
21
22     address(this).callcode();
23
24     var a = [1,2,3];
25
26     var (x, y, z) = (false, "test", 0);
27
28     suicide(address(0));
29 }
30
31 function () public {}
32 }
33
34
35 /*#####
36 /*          Secure code
37 #####*/
38
39 pragma solidity ^0.4.24;
40
41 contract DeprecatedSimpleFixed {
42
43     function useDeprecatedFixed() public view {
44
45         bytes32 bhash = blockhash(0);
46         bytes32 hashofhash = keccak256(bhash);
47
48         uint gas = gasleft();
49
50         if (gas == 0) {
51             revert();
52         }
53
54         address(this).delegatecall();
55
56         uint8[3] memory a = [1,2,3];
57
58         (bool x, string memory y, uint8 z) = (false, "test", 0);
59
60         selfdestruct(address(0));
61     }
62
63     function () external {}
64 }

```

마. 참고문헌

- [1] List of global variables and functions, as of Solidity 0.4.25

- [2] Error handling: Assert, Require, Revert and Exceptions
- [3] View functions
- [4] Untyped declaration is deprecated as of Solidity 0.4.20
- [5] Solidity compiler changelog

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V				V

2. 신뢰할 수 없는 Callee에게 Delegatecall(Delegatecall to Untrusted Callee)

가. 정의 (Description)

delectcall이라는 이름의 메시지 호출의 특별한 변형이 존재하는데, 이는 대상 주소의 코드가 호출 컨트랙트의 맥락에서 실행되며 msg.sender와 msg.value가 그들의 값을 변경하지 않는다는 사실을 제외하고 메시지 호출과 동일하다. 이를 통해 스마트 컨트랙트는 런타임에 다른 주소에서 동적으로 코드를 로드할 수 있습니다. 보관소, 현주소, 잔액은 여전히 호출하는 컨트랙트를 참조한다.

신뢰할 수 없는 컨트랙트를 호출하는 것은 매우 위험하다. 대상 주소의 코드가 호출자의 저장 값을 변경할 수 있고 호출자의 균형을 완전히 제어할 수 있기 때문이다.

나. 관련 보안약점

- CWE-829 : Inclusion of Functionality from Untrusted Control Sphere

다. 시큐어 코딩기법

- delegatecall을 주의하여 사용하고 신뢰할 수 없는 계약에는 호출하지 않는다. 대상 주소가 사용자 입력에서 파생된 경우 신뢰할 수 있는 컨트랙트의 목록과 비교하여 확인해야 한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5  pragma solidity ^0.4.24;
6
7  contract Proxy {
8
9      address owner;
10
11     constructor() public {
12         owner = msg.sender;
13     }
14
15     function forward(address callee, bytes _data) public {
16         require(callee.delegatecall(_data));
17     }
18
19 }
20
21  /************************************************************************/
22  /*                         Secure code                           */
23  /************************************************************************/
24
25  pragma solidity ^0.4.24;
26
27  contract Proxy {
28
29     address callee;

```

```

30 address owner;
31
32 modifier onlyOwner {
33     require(msg.sender == owner);
34     _;
35 }
36
37 constructor() public {
38     callee = address(0x0);
39     owner = msg.sender;
40 }
41
42 function setCallee(address newCallee) public onlyOwner {
43     callee = newCallee;
44 }
45
46 function forward(bytes _data) public {
47     require(callee.delegatecall(_data));
48 }
49
50 }
```

마. 참고문헌

- [1] Solidity Documentation - Delegatecall / Callcode and Libraries
- [2] How to Secure Your Smart Contracts:6 Solidity Vulnerabilities and how to avoid them(Part 1) - Delegate Call
- [3] Solidity Security: Comprehensive list of known attack vectors and common anti-patterns - Delegatecall

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V				V

3. 호출실패로 인한 DoS(DoS with Failed Call)

가. 정의 (Description)

외부 호출은 의도적이거나 의도적이지 않게 실패할 수 있으며, 이로 인해 컨트랙트에 DoS 상황이 발생할 수 있다. 이런 장애로 인한 피해를 최소화하려면 각 외부 호출을 호출 수신자가 시작할 수 있도록 자체 트랜잭션으로 분리하는 것이 좋다. 이는 특히 결제와 관련이 있는데, 사용자가 자동으로 자금을 넣는 것보다 자금을 인출하도록 하는 것이 더 좋다 (이는 가스 한도에 문제가 생길 가능성은 감소시킨다).

나. 관련 보안약점

- CWE-703 : Improper Check or Handling of Exceptional Conditions

다. 시큐어 코딩기법

- 루프의 일부분으로 호출이 실행되는 경우, 단일 트랜잭션에서 여러 호출을 결합하는 것을 피한다.
- 항상 외부 호출이 실패할 수 있다고 가정한다.
- 실패한 호출을 처리하기 위한 컨트랙트 논리를 구현한다.

라. 예제

```

1  /*****  
2  /*                               Insecure code                         */  
3  *****/  
4  
5  contract auction {  
6      address highestBidder;  
7      uint highestBid;  
8  
9      function bid() payable {  
10         require(msg.value >= highestBid);  
11  
12         if (highestBidder != address(0)) {  
13             (bool success, ) = highestBidder.call.value(highestBid)("");  
14             require(success); // if this call consistently fails, no one else can bid  
15         }  
16  
17         highestBidder = msg.sender;  
18         highestBid = msg.value;  
19     }  
20 }  
21  
22  
23  /*****  
24  /*                               Secure code                          */  
25  *****/  
26  
27  contract auction {  
28      address highestBidder;  
29      uint highestBid;  
30      mapping(address => uint) refunds;

```

```

31
32     function bid() payable external {
33         require(msg.value >= highestBid);
34
35         if (highestBidder != address(0)) {
36             refunds[highestBidder] += highestBid; // record the refund that this user can claim
37         }
38
39         highestBidder = msg.sender;
40         highestBid = msg.value;
41     }
42
43     function withdrawRefund() external {
44         uint refund = refunds[msg.sender];
45         refunds[msg.sender] = 0;
46         (bool success, ) = msg.sender.call.value(refund)("");
47         require(success);
48     }
49 }
```

마. 참고문헌

- [1] ConsenSys Smart Contract Best Practices

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

4. 트랜잭션 주문 종속성(Transaction Order Dependence)

가. 정의 (Description)

이더리움 네트워크는 17초마다 새로운 블록이 확인되는 블록 단위로 트랜잭션을 처리한다. 채굴자들은 자신들이 받은 거래를 살펴보고 어떤 거래를 블록에 포함시킬지, 누가 충분히 높은 기름값을 지불했는지를 기준으로 선택한다. 또한 트랜잭션이 이더리움 네트워크로 전송되면 처리를 위해 각 노드로 전달된다. 따라서 이더리움 노드를 운영하는 사람은 어떤 트랜잭션이 발생할지 최종 확정되기 전에 미리 알 수 있다. 경합 조건 취약성은 코드가 제출된 트랜잭션 순서에 따라 달라질 때 발생한다.

나. 관련 보안약점

- CWE-362 : Concurrent Execution using Shared Resource with Improper Synchronization('Race Condition')

다. 시큐어 코딩기법

- 보상을 대가로 정보를 제출하는 과정에서 경합 조건을 해결할 수 있는 가능한 방법을 커밋 노출 해시 체계라고 한다. 답변을 제출하는 대신 계약 당사자는 해시(salt, address, answer)를 제출한다. 그리고 나서 보낸 사람이 보상을 요구하기 위해 salt과의 거래를 제출하고 답한다. 계약 해시(salt, msg.sender, answer)는 해시가 계약과 일치할 경우 생성된 해시를 저장된 해시에 대해 검사한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity ^0.4.16;
6
7 contract EthTxOrderDependenceMinimal {
8     address public owner;
9     bool public claimed;
10    uint public reward;
11
12    function EthTxOrderDependenceMinimal() public {
13        owner = msg.sender;
14    }
15
16    function setReward() public payable {
17        require (!claimed);
18
19        require(msg.sender == owner);
20        owner.transfer(reward);
21        reward = msg.value;
22    }
23
24    function claimReward(uint256 submission) {
25        require (!claimed);
26        require(submission < 10);

```

```
27     msg.sender.transfer(reward);
28     claimed = true;
29 }
30 }
31 }
```

마. 참고문헌

- [1] General Article on Race Conditions ERC20 Race Condition

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
		V	V	V

5. tx.origin을 통한 인증(Authorization through tx.origin)

가. 정의 (Description)

tx.origin은 트랜잭션을 보낸 계정의 주소를 반환하는 솔리디티의 전역 변수이다. 이미 인증된 계정이 인증을 위해 변수를 사용하여 악의적인 계약을 호출하는 경우 계약이 취약해질 수 있다. 이미 인증된 계정의 경우, tx.origin이 트랜잭션의 원래 발신자를 반환하기 때문에 인증 확인이 통과된 취약한 컨트랙트로부터 호출될 수 있다.

나. 관련 보안약점

- CWE-477: Use of Obsolete Function

다. 시큐어 코딩기법

- 인증을 위해 tx.origin을 사용하는 것 대신에 msg.sender를 사용한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity 0.4.24;
6
7 contract MyContract {
8     address owner;
9
10    function MyContract() public {
11        owner = msg.sender;
12    }
13
14    function sendTo(address receiver, uint amount) public {
15        require(tx.origin == owner);
16        receiver.transfer(amount);
17    }
18 }
19
20
21  /************************************************************************/
22 /*                         Secure code                           */
23  /************************************************************************/
24
25 pragma solidity 0.4.25;
26
27 contract MyContract {
28     address owner;
29
30     function MyContract() public {
31         owner = msg.sender;
32     }

```

```
33
34 function sendTo(address receiver, uint amount) public {
35     require(msg.sender == owner);
36     receiver.transfer(amount);
37 }
38 }
```

마. 참고문헌

- [1] Solidity Documentation - tx.origin
- [2] Ethereum Smart Contract Best Practices - Avoid using tx.origin
- [3] SigmaPrime - Visibility

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V	V	V	V

6. 시간의 대체값으로 블록값 사용(Block values as a proxy for time)

가. 정의 (Description)

컨트랙트는 종종 특정 유형의 기능을 수행하기 위해 시간 값에 접근해야 한다. `block.timestamp` 및 `block.number` 같은 값을 사용하면 현재 시간 또는 시간 델타를 알 수 있지만, 대부분의 목적으로 사용하기에는 안전하지 않다. 개발자는 시간에 의존적인 이벤트를 작동시키기 위해 `block.timestamp`를 사용하는 경우가 많다. 이더리움이 분산되면서 노드들은 어느 정도까지만 시간을 동기화 할 수 있다. 또한 악의적인 채굴자는 블록의 타임스탬프를 변경할 수 있으며, 특히 블록의 타임스탬프를 변경함으로써 이점을 얻을 수 있다. 그러나 채굴자는 타임스탬프를 이전 타임스탬프보다 작게 설정할 수 없으면, 향후 타임스탬프를 너무 면 미래에 설정할 수도 없다. 위의 모든 사항을 고려하여 개발자들은 제공된 타임스탬프의 정확성에 의존할 수 없다.

`block.number`의 경우 이더리움에서의 블록 타임이 일반적으로 14초 정도인 점을 고려하면 블록 간 시간 델타를 예측할 수 있다. 그러나 블록 타임은 일정하지 않으며 포크 재구성 및 난이도 문제 같은 이유로 변경될 수 있다. 블록 시간이 다양하기 때문에 정확한 시간계산을 위해 `block.number`에 의존하면 안된다.

나. 관련 보안약점

- CWE-829 : Inclusion of Functionality from Untrusted Control Sphere

다. 시큐어 코딩기법

- 개발자는 블록 값이 정확하지 않고 이를 사용하면 예기치 못한 문제가 발생할 수 있다는 개념으로 스마트 컨트랙트를 작성해야한다. 대신 오라클을 사용할 수 있다. Alternatively, they may make use oracle)

라. 예제

```

1  /************************************************************************/
2  /*                               Insecure code                         */
3  /************************************************************************/
4
5 pragma solidity ^0.5.0;
6
7 contract TimeLock {
8     struct User {
9         uint amount; // amount locked (in eth)
10        uint unlockBlock; // minimum block to unlock eth
11    }
12
13    mapping(address => User) private users;
14
15    // Tokens should be locked for exact time specified
16    function lockEth(uint _time, uint _amount) public payable {
17        require(msg.value == _amount, 'must send exact amount');
18        users[msg.sender].unlockBlock = block.number + (_time / 14);
19        users[msg.sender].amount = _amount;
20    }
21

```

```

22 // Withdraw tokens if lock period is over
23 function withdraw() public {
24     require(users[msg.sender].amount > 0, 'no amount locked');
25     require(block.number >= users[msg.sender].unlockBlock, 'lock period not over');
26
27     uint amount = users[msg.sender].amount;
28     users[msg.sender].amount = 0;
29     (bool success, ) = msg.sender.call.value(amount)("");
30     require(success, 'transfer failed');
31 }
32 }
```

마. 참고문헌

- [1] Safety: Timestamp dependence
- [2] Ethereum Smart Contract Best Practices - Timestamp Dependence
- [3] How do Ethereum mining nodes maintain a time consistent with the network?
- [4] Solidity: Timestamp dependency, is it possible to do safely?
- [5] Avoid using block.number as a timestamp

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V	V	V	V

7. 서명 가단성(Signature Malleability)

가. 정의 (Description)

이더리움 컨트랙트에서 암호화 서명 시스템의 구현은 서명이 고유하다고 가정하는 경우가 많지만 개인키를 소유하지 않아도 서명을 변경할 수 있다. EVM 규격은 사전에 컴파일 된 몇 개의 컨트랙트를 정의하는데, 그중 하나는 타원곡선 공개키 복구를 실행하는 `ecrecover`이다. 악의적인 사용자는 `v, r, s`의 값을 수정하여 다른 유효한 서명을 생성할 수 있다. 컨트랙트 레벨에서 서명 검증을 수행하는 시스템은 서명이 서명된 해시의 일부분일 경우 공격에 취약할 수 있다. 악의적인 사용자가 유효한 서명을 만들어 이전에 서명한 메시지를 재사용할 수 있다.

나. 관련 보안약점

- CWE-347 : Improper Verification of Cryptographic Signature

다. 시큐어 코딩기법

- 이전에 메시지가 컨트랙트에 의해 처리되었는지 확인하기 위해 서명된 메시지 해시를에 서명을 포함해서는 안된다.

라. 예제

```

1  /*****  
2  /* Insecure code */  
3  *****/  
4  
5  pragma solidity ^0.4.24;  
6  
7  contract transaction_malleability{  
8      mapping(address => uint256) balances;  
9      mapping(bytes32 => bool) signatureUsed;  
10  
11     constructor(address[] owners, uint[] init){  
12         require(owners.length == init.length);  
13         for(uint i=0; i < owners.length; i ++){  
14             balances[owners[i]] = init[i];  
15         }  
16     }  
17  
18     function transfer(  
19         bytes _signature,  
20         address _to,  
21         uint256 _value,  
22         uint256 _gasPrice,  
23         uint256 _nonce)  
24         public  
25     returns (bool)  
26     {  
27         bytes32 txid = keccak256(abi.encodePacked(getTransferHash(_to, _value, _gasPrice, _nonce),  
28             _signature));  
29         require(!signatureUsed[txid]);  
30     }  
31 }
```

```

29     address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);
30
31     require(balances[from] > _value);
32     balances[from] -= _value;
33     balances[_to] += _value;
34
35     signatureUsed[txid] = true;
36 }
37
38
39 function recoverTransferPreSigned(bytes _sig,
40     address _to,
41     uint256 _value,
42     uint256 _gasPrice,
43     uint256 _nonce)
44 public
45 view
46 returns (address recovered)
47 {
48     return ecrecoverFromSig(getSignHash(getTransferHash(_to, _value, _gasPrice, _nonce)), _sig);
49 }
50
51 function getTransferHash( address _to, uint256 _value, uint256 _gasPrice, uint256 _nonce)
52 public
53 view
54 returns (bytes32 txHash) {
55     return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasPrice, _nonce);
56 }
57
58 function getSignHash(bytes32 _hash)
59 public
60 pure
61 returns (bytes32 signHash)
62 {
63     return keccak256("\x19Ethereum Signed Message:\n32", _hash);
64 }
65
66 function ecrecoverFromSig(bytes32 hash, bytes sig)
67 public
68 pure
69 returns (address recoveredAddress)
70 {
71     bytes32 r;
72     bytes32 s;
73     uint8 v;
74
75     if (sig.length != 65) {
76         return address(0);
77     }
78
79     assembly {
80         r := mload(add(sig, 32))
81         s := mload(add(sig, 64))

```

```

82         v := byte(0, mload(add(sig, 96)))
83     }
84
85     if (v < 27) {
86         v += 27;
87     }
88
89     if (v != 27 && v != 28) {
90         return address(0);
91     }
92
93     return ecrecover(hash, v, r, s);
94 }
95
96
97
98 /*#####
99  *          Secure code
100 */
101
102 pragma solidity ^0.4.24;
103
104 contract transaction_malleability{
105     mapping(address => uint256) balances;
106     mapping(bytes32 => bool) signatureUsed;
107
108     constructor(address[] owners, uint[] init){
109         require(owners.length == init.length);
110
111         for(uint i=0; i < owners.length; i ++){
112             balances[owners[i]] = init[i];
113         }
114     }
115
116     function transfer(
117         bytes _signature,
118         address _to,
119         uint256 _value,
120         uint256 _gasPrice,
121         uint256 _nonce)
122     public
123     returns (bool)
124     {
125         bytes32 txid = getTransferHash(_to, _value, _gasPrice, _nonce);
126         require(!signatureUsed[txid]);
127
128         address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);
129
130         require(balances[from] > _value);
131         balances[from] -= _value;
132         balances[_to] += _value;
133         signatureUsed[txid] = true;
134     }

```

```

135
136     function recoverTransferPreSigned(
137         bytes _sig,
138         address _to,
139         uint256 _value,
140         uint256 _gasPrice,
141         uint256 _nonce)
142     public
143     view
144     returns (address recovered)
145     {
146         return ecrecoverFromSig(getSignHash(getTransferHash(_to, _value, _gasPrice, _nonce)), _sig);
147     }
148
149     function getTransferHash(
150         address _to,
151         uint256 _value,
152         uint256 _gasPrice,
153         uint256 _nonce)
154     public
155     view
156     returns (bytes32 txHash)
157     {
158         return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasPrice, _nonce);
159     }
160
161     function getSignHash(bytes32 _hash)
162     public
163     pure
164     returns (bytes32 signHash)
165     {
166         return keccak256("\x19Ethereum Signed Message:\n32", _hash);
167     }
168
169     function ecrecoverFromSig(bytes32 hash, bytes sig)
170     public
171     pure
172     returns (address recoveredAddress)
173     {
174         bytes32 r;
175         bytes32 s;
176         uint8 v;
177
178         if (sig.length != 65) {
179             return address(0);
180         }
181
182         assembly {
183             r := mload(add(sig, 32))
184             s := mload(add(sig, 64))
185             v := byte(0, mload(add(sig, 96)))
186         }
187

```

```

188     if (v < 27) {
189         v += 27;
190     }
191
192     if (v != 27 && v != 28) {
193         return address(0);
194     }
195
196     return ecrecover(hash, v, r, s);
197 }
198 }
```

마. 참고문헌

- [1] Bitcoin Transaction Malleability CTF - Challenge

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

8. 잘못된 생성자 이름(Incorrect Constructor Name)

가. 정의 (Description)

생성자는 컨트랙트 생성 중에 한번만 호출되는 함수이다. 생성자는 컨트랙트 소유자 설정과 같은 중요하고 권한 있는 작업을 수행한다. Solidity 버전 0.4.22 이전에는 생성자를 정의하는 유일한 방법으로 생성자를 포함하는 컨트랙트 클래스와 동일한 이름으로 함수를 생성하는 것이었다. 생성자가 되는 함수는 이름이 컨트랙트 이름과 정확히 일치하지 않는 경우 일반 호출이 가능한 함수가 된다. 이 동작은 특히 스마트 컨트랙트 코드가 다른 이름으로 재사용되지만 생성자 함수 이름이 그에 따라 변경되지 않는 경우 보안 문제로 이어질 수 있다.

나. 관련 보안약점

- CWE-665 : Improper Initialization

다. 시큐어 코딩기법

- Solidity 버전 0.4.22 constructor에는 생성자 정의를 더 명확하게 만드는 키워드가 도입되었다. 따라서 컨트랙트를 최신 버전의 Solidity 컴파일러로 업그레이드 하고 새로운 생성자 선언으로 변경하는 것이 좋다.

라. 예제

```

1  /*#####
2  *          Insecure code
3  #####*/
4
5 pragma solidity 0.4.24;
6
7 contract Missing{
8     address private owner;
9
10    modifier onlyowner {
11        require(msg.sender==owner);
12        -
13    }
14
15    function Constructor()
16        public
17    {
18        owner = msg.sender;
19    }
20
21    function () payable {}
22
23    function withdraw()
24        public
25        onlyowner
26    {
27        owner.transfer(this.balance);

```

```

28     }
29 }
30 }
31 }
32 }
33 /*#####
34 /*          Secure code          */
35 /*#####*/
36 }
37 pragma solidity ^0.4.24;
38 }
39 contract Missing{
40     address private owner;
41 }
42 modifier onlyowner {
43     require(msg.sender==owner);
44     _;
45 }
46 }
47 constructor()
48     public
49 {
50     owner = msg.sender;
51 }
52 }
53 function () payable {}
54 }
55 function withdraw()
56     public
57     onlyowner
58 {
59     owner.transfer(this.balance);
60 }
61 }
62 }
```

마. 참고문헌

[1] SigmaPrime - Constructors with Care

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V				V

1.4 에러 처리

1. 새도잉 상태 변수(Shadowing State Variables)

가. 정의 (Description)

Solidity는 상속이 사용될 때 상태 변수의 모호한 이름을 허용한다. 변수 x를 사용한 컨트랙트 A는 x라고 정의된 상태 변수를 가진 컨트랙트 B를 상속할 수 있다. 이렇게 두 개의 분리된 버전의 x를 만들게 되는데, 하나는 컨트랙트 A에서, 다른 하나는 컨트랙트 B에서 접근할 것이다. 복잡한 컨트랙트 시스템에서는 이러한 상태가 눈에 띄지 않고 보안 문제로 이어질 수 있다. 새도잉 상태 변수는 컨트랙트 및 함수 수준에 여러 정의가 있는 경우 단일 컨트랙트 내에서 발생할 수 있다.

나. 관련 보안약점

- CWE-710 : Improper Adherence to Coding Standards

다. 시큐어 코딩기법

- 계약 시스템의 스토리지 변수 레이아웃을 주의 깊게 검토하고 모호함을 제거한다. 컴파일러 경고는 단일 컨트랙트 내에서 문제를 플래그로 표시할 수 있으므로 항상 확인해야 한다.

라. 예제

```

1 *******/
2 /*           Insecure code           */
3 *******/
4
5 pragma solidity 0.4.24;
6
7 contract Tokensale {
8     uint hardcap = 10000 ether;
9
10    function Tokensale() {}
11
12    function fetchCap() public constant returns(uint) {
13        return hardcap;
14    }
15}
16
17 contract Presale is Tokensale {
18     uint hardcap = 1000 ether;
19
20     function Presale() Tokensale() {}
21 }
22
23
24 *******/
25 /*           Secure code           */
26 *******/
27

```

```

28 pragma solidity 0.4.25;
29
30 //We fix the problem by eliminating the declaration which overrides the prefered hardcap.
31
32 contract Tokensale {
33     uint public hardcap = 10000 ether;
34
35     function Tokensale() {}
36
37     function fetchCap() public constant returns(uint) {
38         return hardcap;
39     }
40 }
41
42 contract Presale is Tokensale {
43     //uint hardcap = 1000 ether;
44     //If the hardcap variables were both needed we would have to rename one to fix this.
45     function Presale() Tokensale() {
46         hardcap = 1000 ether; //We set the hardcap from the constructor for the Tokensale to be 1000
47         instead of 10000
48     }
}

```

마. 참고문헌

- [1] Issue on Solidity's Github - Shadowing of inherited state variables should be an error (override keyword)
- [2] Issue on Solidity's Github - Warn about shadowing state variables

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V		V

2. 부적절한 난수값(Weak Sources of Randomness from Chain Attributes)

가. 정의 (Description)

난수를 생성하는 기능은 모든 종류의 어플리케이션 분야에서 매우 유용하다. 한 가지 명백한 예는 도박용 DApps으로 pseudo-난수 생성기가 승자를 선택하는 데 사용된다. 하지만 이더리움에서 강력한 임의성 소스를 만드는 것은 매우 어렵다. 예를 들어, block.timestamp를 사용하는 것은 보안에 문제가 될 수 있다. 마이너는 몇 초 이내에 아무 타임스탬프를 제공하고 다른 사람이 자신의 블록을 승인하도록 선택할 수 있기 때문이다. blockhash나 block.difficulty 및 다른 필드 사용도 마이너에 의해 제어되기 때문에 보안에 문제가 될 수 있다.

나. 관련 보안약점

- CWE-330 : Use of Insufficiently Random Values

다. 시큐어 코딩기법

- Using commit scheme, e.g. RANDAO.
- 오라클을 통한 외부 임의성 소스를 사용한다. 이 방식은 오라클을 신뢰해야 하므로 여러 개의 오라클을 사용하는 것이 합리적일 수 있다.
- Using Bitcoin block hashes, as they are more expensive to mine.

라. 예제

```

1 *******/
2  *                               Insecure code
3  *****/
4
5 pragma solidity ^0.4.21;
6
7 contract GuessTheRandomNumberChallenge {
8     uint8 answer;
9
10    function GuessTheRandomNumberChallenge() public payable {
11        require(msg.value == 1 ether);
12        answer = uint8(keccak256(block.blockhash(block.number - 1), now));
13    }
14
15    function isComplete() public view returns (bool) {
16        return address(this).balance == 0;
17    }
18
19    function guess(uint8 n) public payable {
20        require(msg.value == 1 ether);
21
22        if (n == answer) {
23            msg.sender.transfer(2 ether);
24        }
25    }
26 }
```

```

27
28
29 /*#####*/
30 /*          Secure code          */
31 /*#####*/
32
33 pragma solidity ^0.4.25;
34
35 contract GuessTheRandomNumberChallenge {
36     uint8 answer;
37     uint8 committedGuess;
38     uint commitBlock;
39     address guesser;
40
41     function GuessTheRandomNumberChallenge() public payable {
42         require(msg.value == 1 ether);
43     }
44
45     function isComplete() public view returns (bool) {
46         return address(this).balance == 0;
47     }
48
49     //Guess the modulo of the blockhash 20 blocks from your guess
50     function guess(uint8 _guess) public payable {
51         require(msg.value == 1 ether);
52         committedGuess = _guess;
53         commitBlock = block.number;
54         guesser = msg.sender;
55     }
56     function recover() public {
57         //This must be called after the guessed block and before commitBlock+20's blockhash is
unrecoverable
58         require(block.number > commitBlock + 20 && commitBlock+20 > block.number - 256);
59         require(guesser == msg.sender);
60
61         if(uint(blockhash(commitBlock+20)) == committedGuess){
62             msg.sender.transfer(2 ether);
63         }
64     }
65 }

```

마. 참고문헌

- [1] How can I securely generate a random number in my smart contract?
- [2] When can BLOCKHASH be safely used for a random number? When would it be unsafe?
- [3] The Run smart contract

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

3. 서명 재생 공격에 대한 보호 부재(Missing Protection against Signature Replay Attacks)

가. 정의 (Description)

가용성 향상 또는 가스 비용 절감을 위해 스마트 컨트랙트에서 서명 검증을 수행해야 하는 경우도 있다. 보안에 대한 구현은 예를 들어 처리된 모든 메시지 해시를 추적하고 새 메시지 해시만 처리하도록 허용하여 서명 재생 공격으로부터 보호해야 한다. 악의적인 사용자는 이러한 제어 없이 컨트랙트를 공격하여 여러 번 처리된 다른 사용자가 보낸 메시지 해시를 얻을 수 있다.

나. 관련 보안약점

- CWE-347 : Improper Verification of Cryptographic Signature

다. 시큐어 코딩기법

- 서명 재생 공격으로부터 보호하기 위해 다음 권장 사항을 고려해야한다.
- 스마트 계약에서 처리한 모든 메시지 해시를 저장한다.
- 새 메시지가 수신되면 기존 메시지에 대해 확인하고 새 메시지 해시인 경우에만 비즈니스 논리로 진행한다.
- 메시지를 처리하는 컨트랙트 주소를 포함한다. 이렇게 하면 단일 컨트랙트에서만 메시지를 사용할 수 있다.
- 어떤 경우에도 서명을 포함한 메시지 해시를 생성하지 않는다. 이 ecrecover 기능은 signature malleability에 취약하다.(SWC-117 참조)

라. 예제

마. 참고문헌

[1] Medium - Replay Attack Vulnerability in Ethereum Smart Contracts Introduced by transferProxy()

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

4. 적절한 서명 검증 부족(Lack of Proper Signature Verification)

가. 정의 (Description)

스마트 컨트랙트 시스템이 제공하는 유연성과 증가된 전송 가능성 때문에 사용자가 온체인 트랜잭션을 수행하도록 직접 요청하는 대신 오프체인 메시지에 서명할 수 있도록 하는 것이 스마트 컨트랙트 시스템의 일반적인 패턴이다. 서명된 메시지를 처리하는 스마트 계약 시스템은 서명된 메시지를 추가로 처리하기 전에 서명된 메시지에서 신뢰성을 회복하기 위한 자신만의 논리를 구현해야 한다. 이러한 시스템의 한계는 스마트 컨트랙트 메시지에 서명할 수 없기 때문에 스마트 컨트랙트와 직접 상호작용할 수 없다는 것이다. 일부 서명 검증 구현에서는 이러한 제한이 없는 다른 방법을 기반으로 서명된 메시지의 유효성을 가정하여 이 문제를 해결하려고 한다. 이러한 방법의 예는 msg.sender에 의존하는 것이며, 만약 서명된 메시지가 발신인 주소에서 비롯되었다면 그것은 발신인 주소에 의해서도 생성되었다고 가정하는 것이다. 이는 특히 프록시를 사용하여 트랜잭션을 전달할 수 있는 시나리오에서 취약성을 유발할 수 있다.

나. 관련 보안약점

- CWE-345 : Insufficient Verification of Data Authenticity

다. 시큐어 코딩기법

- ecrecover()를 통해 적절한 서명 검증이 필요하지 않은 대체 검증 체제를 사용하지 않는 것이 좋다.

라. 예제

마. 참고문헌

[1] Consensys Diligence 0x Audit Report - Insecure signature validator undefined

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
		V		V

5. 요구사항 위반(Requirement Violation)

가. 정의 (Description)

솔리디티의 require() 구조는 함수의 외부 입력을 검증하기 위한 것이다. 대부분의 경우, 이러한 외부 입력은 발신자에 의해 제공되지만 호출자에 의해 반환될 수도 있다. 전자의 경우 전제조건 위반이라고 부른다. 요구사항 위반은 다음 두 가지 가능한 문제 중 하나를 나타낼 수 있다.

- 외부 입력을 제공한 컨트랙트에 버그가 있다.
- 요구사항을 표현하기 위해 사용되는 조건이 너무 강하다.

나. 관련 보안약점

- CWE-573 : Improper Following of Specification by Caller

다. 시큐어 코딩기법

- 필요한 논리 조건이 너무 강하면 유효한 외부 입력을 모두 허용하도록 약화해야 한다. 그렇지 않은 경우, 버그는 외부 입력을 제공한 컨트랙트에 있어야 하며 잘못된 입력이 제공되지 않는지 확인함으로써 코드를 수정하는 것을 고려한다.

라. 예제

```

1  /************************************************************************/
2  /*                               Insecure code                         */
3  /************************************************************************/
4
5  pragma solidity ^0.4.25;
6
7  contract Bar {
8      Foo private f = new Foo();
9      function doubleBaz() public view returns (int256) {
10         return 2 * f.baz(0);
11     }
12 }
13
14 contract Foo {
15     function baz(int256 x) public pure returns (int256) {
16         require(0 < x);
17         return 42;
18     }
19 }
20
21
22  /************************************************************************/
23 /*                               Secure code                          */
24  /************************************************************************/
25
26  pragma solidity ^0.4.25;
27
28  contract Bar {
29      Foo private f = new Foo();

```

```

30     function doubleBaz() public view returns (int256) {
31         return 2 * f.baz(1); //Changes the external contract to not hit the overly strong requirement.
32     }
33 }
34
35 contract Foo {
36     function baz(int256 x) public pure returns (int256) {
37         require(0 < x); //You can also fix the contract by changing the input to the uint type and
38             removing the require
39         return 42;
40     }
}

```

마. 참고문헌

- [1] The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

6. 임의의 저장 위치에 쓰기(Write to Arbitrary Storage Location)

가. 정의 (Description)

스마트 컨트랙트의 데이터는 EVM 수준의 일부 저장 위치에 지속적으로 저장된다. 컨트랙트는 인증된 사용자 또는 컨트랙트 계정만 중요한 저장 위치에 쓸 수 있도록 하는 역할을 한다. 만약 공격자가 컨트랙트의 임시 저장소 위치에 쓸 수 있는 경우 권한 검사를 쉽게 우회할 수 있다. 이를 통해 공격자는 컨트랙트 소유자의 주소를 저장하는 필드를 덮어써 저장소를 손상시킬 수 있다.

나. 관련 보안약점

- CWE-123 : Write-what-where Condition

다. 시큐어 코딩기법

- 모든 데이터 구조가 동일한 저장 공간을 공유한다는 점을 고려할 때, 한 데이터 구조에 쓰기가 실수로 다른 데이터 구조의 항목을 덮어쓰지 않도록 해야 한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity ^0.4.25;
6
7 contract Wallet {
8     uint[] private bonusCodes;
9     address private owner;
10
11    constructor() public {
12        bonusCodes = new uint[](0);
13        owner = msg.sender;
14    }
15
16    function () public payable {
17    }
18
19    function PushBonusCode(uint c) public {
20        bonusCodes.push(c);
21    }
22
23    function PopBonusCode() public {
24        require(0 <= bonusCodes.length);
25        bonusCodes.length--;
26    }
27
28    function UpdateBonusCodeAt(uint idx, uint c) public {
29        require(idx < bonusCodes.length);
30        bonusCodes[idx] = c;
31    }

```

```

32
33     function Destroy() public {
34         require(msg.sender == owner);
35         selfdestruct(msg.sender);
36     }
37 }
38
39
40 /*#####
41 /*          Secure code
42 /*#####
43
44 pragma solidity ^0.4.25;
45
46 contract Wallet {
47     uint[] private bonusCodes;
48     address private owner;
49
50     constructor() public {
51         bonusCodes = new uint[](0);
52         owner = msg.sender;
53     }
54
55     function () public payable {
56     }
57
58     function PushBonusCode(uint c) public {
59         bonusCodes.push(c);
60     }
61
62     function PopBonusCode() public {
63         require(0 < bonusCodes.length);
64         bonusCodes.length--;
65     }
66
67     function UpdateBonusCodeAt(uint idx, uint c) public {
68         require(idx < bonusCodes.length); //Since you now have to push very codes this is no longer an
69                         arbitrary write.
70         bonusCodes[idx] = c;
71     }
72
73     function Destroy() public {
74         require(msg.sender == owner);
75         selfdestruct(msg.sender);
76     }
}

```

마. 참고문헌

- [1] Entry to Underhanded Solidity Coding Contest 2017(honorable mention)

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

7. 잘못된 상속 순서(Incorrect Inheritance Order)

가. 정의 (Description)

솔리디티는 다중 상속을 지원하므로 하나의 컨트랙트가 여러 컨트랙트를 상속할 수 있다. 다중 상속은 다이아몬드 문제(둘 이상의 기본 계약이 동일한 함수를 정의한다면, 자식 컨트랙트에서 어떤 것을 호출해야 하는가?)라고 불리는 모호성을 야기한다. 솔리디티는 기본 컨트랙트 간의 우선 순위를 설정하는 역 C3 선형화를 사용하여 이러한 모호성을 처리한다. 이렇게 하면 기본 컨트랙트의 우선 순위가 다르기 때문에 상속 순서가 중요하다. 상속 순서를 무시하면 예기치 않은 동작이 발생할 수 있다.

나. 관련 보안약점

- CWE-696 : Incorrect Behavior Order

다. 시큐어 코딩기법

- 여러 컨트랙트를 상속할 때, 특히 동일한 기능을 가진 경우 개발자는 상속을 올바른 순서로 신중하게 지정해야 한다. The rule of thumb is to inherit contracts from more /general/ to more /specific/.

라. 예제

```

1  /*************************************************************************/
2  /*                         Insecure code                           */
3  /*************************************************************************/
4
5  contract Crowdsale {
6      using SafeMath for uint256;
7      ERC20Mintable public token;
8      uint256 public startBlock;
9      uint256 public endBlock;
10     address public wallet;
11     uint256 public rate;
12     uint256 public weiRaised;
13     event TokenPurchase(address indexed purchaser, address indexed beneficiary, uint256 value, uint256
14                           amount);
15
16     function Crowdsale(uint256 _startBlock, uint256 _endBlock, uint256 _rate, address _wallet) {
17         require(_startBlock >= block.number);
18         require(_endBlock >= _startBlock);
19         require(_rate > 0);
20         require(_wallet != 0x0);
21
22         token = createTokenContract();
23         startBlock = _startBlock;
24         endBlock = _endBlock;
25         rate = _rate;
26         wallet = _wallet;
27     }
28
29     function createTokenContract() internal returns (ERC20Mintable) {
30         return new ERC20Mintable();
31     }
32 }
```

```

30 }
31
32     function () payable {
33         buyTokens(msg.sender);
34     }
35
36     function buyTokens(address beneficiary) payable {
37         require(beneficiary != 0x0);
38         require(validPurchase());
39
40         uint256 weiAmount = msg.value;
41
42         uint256 tokens = weiAmount.mul(rate);
43
44         weiRaised = weiRaised.add(weiAmount);
45
46         token.mint(beneficiary, tokens);
47         TokenPurchase(msg.sender, beneficiary, weiAmount, tokens);
48
49         forwardFunds();
50     }
51
52     function forwardFunds() internal {
53         wallet.transfer(msg.value);
54     }
55
56     function validPurchase() internal constant returns (bool) {
57         uint256 current = block.number;
58         bool withinPeriod = current >= startBlock && current <= endBlock;
59         bool nonZeroPurchase = msg.value != 0;
60         return withinPeriod && nonZeroPurchase;
61     }
62
63     function hasEnded() public constant returns (bool) {
64         return block.number > endBlock;
65     }
66 }
67
68 contract CappedCrowdsale is Crowdsale {
69     using SafeMath for uint256;
70     uint256 public cap;
71
72     function CappedCrowdsale(uint256 _cap) {
73         require(_cap > 0);
74         cap = _cap;
75     }
76
77     function validPurchase() internal constant returns (bool) {
78         bool withinCap = weiRaised.add(msg.value) <= cap;
79         return super.validPurchase() && withinCap;
80     }
81
82     function hasEnded() public constant returns (bool) {

```

```

83     bool capReached = weiRaised >= cap;
84     return super.hasEnded() || capReached;
85   }
86 }
87
88 contract WhitelistedCrowdsale is Crowdsale {
89   using SafeMath for uint256;
90
91   mapping (address => bool) public whitelist;
92
93   function addToWhitelist(address addr) {
94     require(msg.sender != address(this));
95     whitelist[addr] = true;
96   }
97   function validPurchase() internal constant returns (bool) {
98     return super.validPurchase() || (whitelist[msg.sender] && !hasEnded());
99   }
100 }
101
102
103 contract MDTCrowdsale is CappedCrowdsale, WhitelistedCrowdsale {
104
105   function MDTCrowdsale()
106     CappedCrowdsale(50000000000000000000000000)
107     Crowdsale(block.number, block.number + 100000, 1, msg.sender) {
108       addToWhitelist(msg.sender);
109       addToWhitelist(0x0d5bda9db5dd36278c6a40683960ba58cac0149b);
110       addToWhitelist(0x1b6ddc637c24305b354d7c337f9126f68aad4886);
111     }
112 }
113 }
```

마. 참고문헌

- [1] Smart Contract Best Practices - Multiple Inheritance Caution
- [2] Solidity docs - Multiple Inheritance and Linearization
- [3] Solidity anti-patterns: Fun with inheritance DAG abuse

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V				V

8. 불충분한 가스 그리핑(Insufficient Gas Grieving)

가. 정의 (Description)

데이터를 수용하여 다른 컨트랙트의 하위 호출에 사용하는 컨트랙트에 대해 불충분한 가스 파괴 공격을 수행할 수 있다. 하위 호출에 실패하면 전체 트랜잭션을 되돌리거나 실행을 계속한다. 중계 컨트랙트의 경우, 트랜잭션을 실행하는 사용자인 ‘forwarder’는 트랜잭션을 실행하기에 충분한 가스만을 사용함으로써 효과적인 거래를 검열할 수 있지만 하위 호출이 성공하기에는 충분하지 않다.

나. 관련 보안약점

- CWE-691 : Insufficient Control Flow Management

다. 시큐어 코딩기법

- 불충분한 가스 그리핑을 방지하기 위한 두 가지 방법이 있다. 신뢰할 수 있는 사용자만 트랜잭션을 중계하도록 허용하거나 중계자가 충분한 가스를 제공하도록 요구한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity ^0.5.0;
6
7 contract Relayer {
8     uint transactionId;
9
10    struct Tx {
11        bytes data;
12        bool executed;
13    }
14
15    mapping (uint => Tx) transactions;
16
17    function relay(Target target, bytes memory _data) public returns(bool) {
18        // replay protection; do not call the same transaction twice
19        require(transactions[transactionId].executed == false, 'same transaction twice');
20        transactions[transactionId].data = _data;
21        transactions[transactionId].executed = true;
22        transactionId += 1;
23
24        (bool success, ) = address(target).call(abi.encodeWithSignature("execute(bytes)", _data));
25        return success;
26    }
27}
28
29// Contract called by Relayer
30contract Target {
31    function execute(bytes memory _data) public {

```

```

32         // Execute contract code
33     }
34 }
35
36
37 /*#####
38 *          Secure code
39 *#####
40 */
41 pragma solidity ^0.5.0;
42
43 contract Relayer {
44     uint transactionId;
45
46     struct Tx {
47         bytes data;
48         bool executed;
49     }
50
51     mapping (uint => Tx) transactions;
52
53     function relay(Target target, bytes memory _data, uint _gasLimit) public {
54         // replay protection; do not call the same transaction twice
55         require(transactions[transactionId].executed == false, 'same transaction twice');
56         transactions[transactionId].data = _data;
57         transactions[transactionId].executed = true;
58         transactionId += 1;
59
60         address(target).call(abi.encodeWithSignature("execute(bytes)", _data, _gasLimit));
61     }
62 }
63
64 // Contract called by Relayer
65 contract Target {
66     function execute(bytes memory _data, uint _gasLimit) public {
67         require(gasleft() >= _gasLimit, 'not enough gas');
68         // Execute contract code
69     }
70 }
```

마. 참고문헌

- [1] Consensys Smart Contract Best Practices
- [2] What does griefing mean?
- [3] Griefing Attacks: Are they profitable for the attacker?

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

9. 함수 유형 변수를 사용한 임의 점프(Arbitrary Jump with Function Type Variable)

가. 정의 (Description)

솔리디티는 함수 유형을 지원한다. 즉, 함수 유형의 변수를 일치하는 서명이 있는 함수에 대한 참조로 할당할 수 있다. 이러한 변수에 저장된 함수는 정규 함수처럼 호출할 수 있다. 이 문제는 사용자가 임의로 함수 유형 변수를 변경하여 임의 코드 명령을 실행할 수 있을 때 발생한다. 솔리디티는 포인터를 지원하지 않으므로 이러한 변수를 임의 값으로 변경할 수 없다. 하지만 개발자가 mstore 또는 대입 연산자와 같은 어셈블리 명령을 사용하면 최악의 경우 공격자가 필요한 유효성 검사 및 필요한 상태 변경을 위반하여 함수 유형 변수를 코드 명령으로 지정할 수 있다.

나. 관련 보안약점

- CWE-695 : Use of Low-Level Functionality

다. 시큐어 코딩기법

- 어셈블리 사용은 최소화해야 한다. 개발자는 사용자가 함수 유형 변수에 임의의 값을 할당하도록 허용하면 안된다.

라. 예제

```

1  /***** Insecure code *****/
2  /*
3   *          Insecure code
4   */
5
6  pragma solidity ^0.4.25;
7
8
9  contract FunctionTypes {
10
11     constructor() public payable { require(msg.value != 0); }
12
13     function withdraw() private {
14         require(msg.value == 0, 'dont send funds!');
15         address(msg.sender).transfer(address(this).balance);
16     }
17
18     function frwd() internal
19     { withdraw(); }
20
21     struct Func { function () internal f; }
22
23     function breakIt() public payable {
24         require(msg.value != 0, 'send funds!');
25         Func memory func;
26         func.f = frwd;
27         assembly { mstore(func, add(mload(func), callvalue)) }
28         func.f();
29     }
30 }
```

마. 참고문헌

- [1] Solidity CTF
- [2] Solidity docs - Solidity Assembly
- [3] Solidity docs - Function Types

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V			V

1.5 코드 오류

1. 블록 가스 한도를 사용한 DoS(DoS With Block Gas Limit)

가. 정의 (Description)

스마트 컨트랙트가 구축되거나 그 안의 기능이 호출될 때, 이러한 동작의 실행은 이를 완료하기 위해 얼마나 많은 계산이 필요한가에 기초하여 일정량의 가스를 필요로 한다. 이더리움 네트워크는 블록 가스 한도를 지정하며 블록에 포함된 모든 트랜잭션의 합계는 임계값을 초과할 수 없다. 중앙집중화된 어플리케이션에서 무해한 프로그래밍 패턴은 함수 실행 비용이 블록 가스 한도를 초과할 때 스마트 컨트랙트에서 서비스 거부 상태를 초래할 수 있다. 시간이 지남에 따라 크기가 증가하는 알 수 없는 크기의 배열을 수정하면 이러한 서비스 거부 상태가 발생할 수 있다.

나. 관련 보안약점

- CWE-400 : Uncontrolled Resource Consumption

다. 시큐어 코딩기법

- 시간이 지남에 따라 증가하는 큰 배열이 있을 경우 주의가 필요하다. 전체 데이터 구조를 반복하는 작업은 피해야 한다. 크기를 알 수 없는 배열을 무조건적으로 반복해야 하는 경우 여러 블록을 사용할 가능성이 있으므로 여러 트랜잭션이 필요하도록 계획해야 한다.

라. 예제

```

1 *******/
2 /*                               Insecure code                         */
3 *******/
4
5 pragma solidity ^0.4.25;
6
7 contract DosGas {
8
9     address[] creditorAddresses;
10    bool win = false;
11
12    function emptyCreditors() public {
13        if(creditorAddresses.length>1500) {
14            creditorAddresses = new address[](0);
15            win = true;
16        }
17    }
18
19    function addCreditors() public returns (bool) {
20        for(uint i=0;i<350;i++) {
21            creditorAddresses.push(msg.sender);
22        }
23        return true;
24    }
25

```

```

26     function iWin() public view returns (bool) {
27         return win;
28     }
29
30     function numberCreditors() public view returns (uint) {
31         return creditorAddresses.length;
32     }
33 }
```

마. 참고문헌

- [1] Ethereum Design Rationale
- [2] Ethereum Yellow Paper
- [3] Clear Large Array Without Blowing Gas Limit
- [4] GovernMental jackpot payout Dos Gas

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
	V			V

2. 오타(Typographical Error)

가. 정의 (Description)

예를 들어, 정의된 연산의 목적이 어떤 수를 변수에 합하는 것(+=)이지만 실수로 잘못된 방법(=+)으로 정의 하여 합을 계산하는 대신 변수를 다시 초기화하는 문제가 발생할 수 있습니다. 단항 + 연산자는 새 솔리디티 컴파일러 버전에서 더 이상 사용되지 않는다.

나. 관련 보안약점

- CWE-480 : Use of Incorrect Operator

다. 시큐어 코딩기법

- 이 취약점은 수학 연산에 대한 사전 조건 검사를 수행하거나 OpenZepplin이 개발한 SafeMath와 같은 산술 계산을 위한 라이브러리를 사용하면 피할 수 있다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity ^0.4.25;
6
7 contract TypoOneCommand {
8     uint numberOne = 1;
9
10    function alwaysOne() public {
11        numberOne += 1;
12    }
13}
14
15
16 /************************************************************************/
17 /*                         Secure code                          */
18 /*/************************************************************************/

```

마. 참고문헌

- [1] HackerGold Bug Analysis
- [2] SafeMath by OpenZeppelin
- [3] Disallow Unary plus

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V			V

3. RLO 유니코드기법 (Right-To-Left-Override control character (U+202E))

가. 정의 (Description)

악의적인 실행자는 Right-To-Left-Override 유니코드 문자를 사용하여 RTL 텍스트 랜더링을 강제하고 컨트랙트의 진짜 의도를 사용자에게 혼동시킬 수 있다.

나. 관련 보안약점

- CWE-451 : User Interface(UI) Misrepresentation of Critical Information

다. 시큐어 코딩기법

- U+202E 문자는 거의 사용되지 않는다. 스마트 컨트랙트의 소스 코드에 나타나지 않아야 한다.

라. 예제

```

1  /*#####
2   *          Insecure code
3  #####*/
4
5 pragma solidity ^0.5.0;
6
7 contract GuessTheNumber
8 {
9     uint _secretNumber;
10    address payable _owner;
11    event success(string);
12    event wrongNumber(string);
13
14    constructor(uint secretNumber) payable public
15    {
16        require(secretNumber <= 10);
17        _secretNumber = secretNumber;
18        _owner = msg.sender;
19    }
20
21    function getValue() view public returns (uint)
22    {
23        return address(this).balance;
24    }
25
26    function guess(uint n) payable public
27    {
28        require(msg.value == 1 ether);
29
30        uint p = address(this).balance;
31        checkAndTransferPrize(/*The prize/*rebmun desseug*/n , p/*
32                               /*The user who should benefit */ ,msg.sender);
33    }
34
35    function checkAndTransferPrize(uint p, uint n, address payable guesser) internal returns(bool)

```

```

36  {
37      if(n == _secretNumber)
38      {
39          guesser.transfer(p);
40          emit success("You guessed the correct number!");
41      }
42      else
43      {
44          emit wrongNumber("You've made an incorrect guess!");
45      }
46  }
47
48  function kill() public
49  {
50      require(msg.sender == _owner);
51      selfdestruct(_owner);
52  }
53 }
```

마. 참고문헌

[1] Outsmarting Smart Contracts

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V		V

4. 사용되지 않는 변수 존재(Presence of unused variables)

가. 정의 (Description)

솔리디티에서는 사용되지 않는 변수가 허용되며 직접적인 보안 문제를 일으키지 않는다. 하지만 다음과 같은 문제를 야기할 수 있으므로 가능한 피하는 것이 좋다.

- 계산량 증가 및 불필요한 가스 소비량 증가
- 버그 또는 잘못된 데이터 구조를 나타내며 일반적으로 코드 품질이 저하
- 코드 노이즈를 유발 및 코드의 가독성 감소

나. 관련 보안약점

- CWE-1164 : Irrelevant Code

다. 시큐어 코딩기법

- 코드 베이스에서 사용하지 않는 변수를 제거한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5  pragma solidity >=0.5.0;
6  pragma experimental ABIEncoderV2;
7
8  import "./base.sol";
9
10 contract DerivedA is Base {
11     // i is not used in the current contract
12     A i = A(1);
13
14     int internal j = 500;
15
16     function call(int a) public {
17         assign1(a);
18     }
19
20     function assign3(A memory x) public returns (uint) {
21         return g[1] + x.a + uint(j);
22     }
23
24     function ret() public returns (int){
25         return this.e();
26
27     }
28
29 }
```

```

32 /*#####*/
33 /*          Secure code          */
34 /*#####*/
35
36 pragma solidity >=0.5.0;
37 pragma experimental ABIEncoderV2;
38
39 import "./base_fixed.sol";
40
41 contract DerivedA is Base {
42
43     int internal j = 500;
44
45     function call(int a) public {
46         assign1(a);
47     }
48
49     function assign3(A memory x) public returns (uint) {
50         return g[1] + x.a + uint(j);
51     }
52
53     function ret() public returns (int){
54         return this.e();
55     }
56 }
57
58 }
```

마. 참고문헌

- [1] Unused local variables warnings discussion
- [2] Shadowing of inherited state variables discussion

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

5. 예상치 못한 이더 잔액(Unexpected Ether balance)

가. 정의 (Description)

계약은 특정 Ether 잔액을 엄격하게 가정할 때 잘못 동작할 수 있다. 언제든지 Ether를 계약으로 풀백 기능을 트리거하지 않고 강제 전송, 자체 소멸을 사용하거나, 계정에 마이닝하여 보낼 수 있다. 최악의 경우 DOS 조건으로 인해 계약을 사용할 수 없게 될 수 있다.

나. 관련 보안약점

- CWE-667: Improper Locking

다. 시큐어 코딩기법

- 계약의 Ether 잔액에 대한 엄격한 동일성 검사를 피해야 한다.

라. 예제

N/A

마. 참고문헌

- [1] Consensys Best Practices: Forcibly Sending Ether
- [2] Sigmaprime: Unexpected Ether
- [3] Gridlock (a smart contract bug)

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V		V	V	V

6. 여러 가변 길이 인수가 있는 해시 충돌(Hash Collisions With Multiple Variable Length Arguments)

가. 정의 (Description)

여러 가변 길이 인수와 함께 abi.encodePacked()를 사용하면 특정 상황에서 해시 충돌이 발생할 수 있다. abi.encodePacked() 배열의 일부인지에 대한 여부와 관계없이 모든 요소를 순서대로 압축 하므로 요소를 이동이 가능하다. abi.encodePacked()는 모든 요소가 동일한 순서에 있으면 동일한 인코딩을 반환한다. 서명 확인 상황에서 공격자는 권한 부여를 효과적으로 우회하기 위해 이전 함수 호출에서 요소의 위치를 수정하여 이를 악용할 수 있다.

나. 관련 보안약점

- CWE-294: Authentication Bypass by Capture-replay

다. 시큐어 코딩기법

- 서로 다른 매개변수를 사용하여 일치하는 서명을 얻을 수 없도록 하는 것이 중요하다. 그렇게 하려면 사용자가 예 사용된 매개변수에 액세스하지 못하도록 abi.encodePacked()하거나 고정 길이 배열을 사용하거나 또는 간단히 abi.encode()으로 대신 사용해야 한다.

라. 예제

```

1  /************************************************************************/
2  /*                         Insecure code                          */
3  /************************************************************************/
4
5 pragma solidity ^0.5.0;
6
7 import "./ECDSA.sol";
8
9 contract AccessControl {
10     using ECDSA for bytes32;
11     mapping(address => bool) isAdmin;
12     mapping(address => bool) isRegularUser;
13     // Add admins and regular users.
14     function addUsers(
15         address[] calldata admins,
16         address[] calldata regularUsers,
17         bytes calldata signature
18     )
19         external
20     {
21         if (!isAdmin[msg.sender]) {
22             // Allow calls to be relayed with an admin's signature.
23             bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
24             address signer = hash.toEthSignedMessageHash().recover(signature);
25             require(isAdmin[signer], "Only admins can add users.");
26         }
27         for (uint256 i = 0; i < admins.length; i++) {

```

```

28         isAdmin[admins[i]] = true;
29     }
30     for (uint256 i = 0; i < regularUsers.length; i++) {
31         isRegularUser[regularUsers[i]] = true;
32     }
33 }
34 }
35
36
37 /*#####
38 /*          Secure code
39 /*#####
40
41 pragma solidity ^0.5.0;
42
43 import "./ECDSA.sol";
44
45 contract AccessControl {
46     using ECDSA for bytes32;
47     mapping(address => bool) isAdmin;
48     mapping(address => bool) isRegularUser;
49     // Add a single user, either an admin or regular user.
50     function addUser(
51         address user,
52         bool admin,
53         bytes calldata signature
54     )
55         external
56     {
57         if (!isAdmin[msg.sender]) {
58             // Allow calls to be relayed with an admin's signature.
59             bytes32 hash = keccak256(abi.encodePacked(user));
60             address signer = hash.toEthSignedMessageHash().recover(signature);
61             require(isAdmin[signer], "Only admins can add users.");
62         }
63         if (admin) {
64             isAdmin[user] = true;
65         } else {
66             isRegularUser[user] = true;
67         }
68     }
69 }
70
71
72 /*#####
73 /*          Secure code
74 /*#####
75
76 pragma solidity ^0.5.0;
77
78 import "./ECDSA.sol";
79
80 contract AccessControl {

```

```

81  using ECDSA for bytes32;
82  mapping(address => bool) isAdmin;
83  mapping(address => bool) isRegularUser;
84  // Add admins and regular users.
85  function addUsers(
86      // Use fixed length arrays.
87      address[3] calldata admins,
88      address[3] calldata regularUsers,
89      bytes calldata signature
90  )
91  external
92  {
93      if (!isAdmin[msg.sender]) {
94          // Allow calls to be relayed with an admin's signature.
95          bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
96          address signer = hash.toEthSignedMessageHash().recover(signature);
97          require(isAdmin[signer], "Only admins can add users.");
98      }
99      for (uint256 i = 0; i < admins.length; i++) {
100          isAdmin[admins[i]] = true;
101      }
102      for (uint256 i = 0; i < regularUsers.length; i++) {
103          isRegularUser[regularUsers[i]] = true;
104      }
105  }
106 }
```

마. 참고문헌

- [1] Solidity Non-standard Packed Mode
- [2] Hash Collision Attack

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

7. 하드 코딩된 가스량 (Message call with hardcoded gas amount)

가. 정의 (Description)

transfer() 및 send() 기능은 고정된 양의 2,300 가스를 전달한다. EVM 명령의 가스 비용은 고정 된 가스 비용을 지불하는 계약 시스템을 깨뜨릴 수 있는 하드 포크 동안 변경될 수 있다

나. 관련 보안약점

- CWE-655: Improper Initialization

다. 시큐어 코딩기법

- transfer() 및 send() 기능을 사용해서는 안된다. 통화를 수행할 때 정해진 양의 가스를 지정하지 않고, .call.value(...)("")를 사용을 해야한다. 또한, 재진입 공격을 방지하기 위해 검사-효과-상호작용 패턴 및/또는 재진입 잠금을 사용을 해야한다.

라. 예제

```

1  ****
2  /*           Insecure code           */
3  ****
4
5 contract Vulnerable {
6     function withdraw(uint256 amount) external {
7         // This forwards 2300 gas, which may not be enough if the recipient
8         // is a contract and gas costs change.
9         msg.sender.transfer(amount);
10    }
11 }
12
13 ****
14 /*           Secure code           */
15 /*
16 ****
17
18 contract Fixed {
19     function withdraw(uint256 amount) external {
20         // This forwards all available gas. Be sure to check the return value!
21         (bool success, ) = msg.sender.call.value(amount)( "");
22         require(success, "Transfer failed.");
23     }
24 }
```

마. 참고문헌

- [1] ChainSecurity - Ethereum Istanbul Hardfork: The Security Perspective
- [2] Steve Marx - Stop Using Solidity's transfer() Now
- [3] EIP 1884

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V

8. 효과가 없는 코드(Code With No Effects)

가. 정의 (Description)

Solidity에서는 의도한 효과를 생성하지 않는 코드를 작성할 수 있다. 현재, solidity 컴파일러는 효과가 없는 코드에 대한 경고를 반환하지 않다. 이로 인해 의도한 작업을 제대로 수행하지 않는 dead 코드가 존재할 수 있다.

나. 관련 보안약점

- CWE-1164: Irrelevant Code

다. 시큐어 코딩기법

- 계약이 의도한 대로 작동하는지 주의 깊게 확인하는 것이 중요하다. 코드의 올바른 동작을 확인하기 위해 단위 테스트를 실시 해야 한다.

라. 예제

```

1 *******/
2 /*          Insecure code          */
3 ******/
4
5 pragma solidity ^0.5.0;
6
7 contract Wallet {
8     mapping(address => uint) balance;
9
10    // Deposit funds in contract
11    function deposit(uint amount) public payable {
12        require(msg.value == amount, 'msg.value must be equal to amount');
13        balance[msg.sender] = amount;
14    }
15
16    // Withdraw funds from contract
17    function withdraw(uint amount) public {
18        require(amount <= balance[msg.sender], 'amount must be less than balance');
19
20        uint previousBalance = balance[msg.sender];
21        balance[msg.sender] = previousBalance - amount;
22
23        // Attempt to send amount from the contract to msg.sender
24        msg.sender.call.value(amount);
25    }
26 }
27
28
29 *******/
30 /*          Secure code          */
31 ******/
32

```

```

33 pragma solidity ^0.5.0;
34
35 contract Wallet {
36     mapping(address => uint) balance;
37
38     // Deposit funds in contract
39     function deposit(uint amount) public payable {
40         require(msg.value == amount, 'msg.value must be equal to amount');
41         balance[msg.sender] = amount;
42     }
43
44     // Withdraw funds from contract
45     function withdraw(uint amount) public {
46         require(amount <= balance[msg.sender], 'amount must be less than balance');
47
48         uint previousBalance = balance[msg.sender];
49         balance[msg.sender] = previousBalance - amount;
50
51         // Attempt to send amount from the contract to msg.sender
52         (bool success, ) = msg.sender.call.value(amount)("");
53         require(success, 'transfer failed');
54     }
55 }
```

마. 참고문헌

- [1] Issue on Solidity's Github - raise an error when a statement can never have side-effects
- [2] Issue on Solidity's Github - msg.sender.call.value(address(this).balance); should produce a warning

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
V	V			V

9. 암호화 되지 않은 개인 데이터 온체인(Unencrypted Private Data On-Chain)

가. 정의 (Description)

계약이 게시되지 않은 경우에도 공격자는 계약 트랜잭션을 확인하여 계약 상태에 저장된 값을 확인할 수 있다. 이러한 이유로 암호화되지 않은 개인 데이터는 계약 코드 또는 상태에 저장되지 않는 것이 중요하다.

나. 관련 보안약점

- CWE-767 : Access to Critical Private Variable via Public Method

다. 시큐어 코딩기법

- 모든 개인 데이터는 오프체인에 저장하거나 신중하게 암호화해야 한다.

라. 예제

```

1  *****
2  /*                               Insecure code                         */
3  *****
4
5 pragma solidity ^0.5.0;
6
7 contract OddEven {
8     struct Player {
9         address addr;
10        uint number;
11    }
12
13    Player[2] private players;
14    uint count = 0;
15
16    function play(uint number) public payable {
17        require(msg.value == 1 ether, 'msg.value must be 1 eth');
18        players[count] = Player(msg.sender, number);
19        count++;
20        if (count == 2) selectWinner();
21    }
22
23    function selectWinner() private {
24        uint n = players[0].number + players[1].number;
25        (bool success, ) = players[n%2].addr.call.value(address(this).balance)("");
26        require(success, 'transfer failed');
27        delete players;
28        count = 0;
29    }
30}
31
32*****
33 /*                               Secure code                         */
34 /*

```

```

35 /*#####*/
36
37 pragma solidity ^0.5.0;
38
39 contract OddEven {
40     enum Stage {
41         FirstCommit,
42         SecondCommit,
43         FirstReveal,
44         SecondReveal,
45         Distribution
46     }
47
48     struct Player {
49         address addr;
50         bytes32 commitment;
51         uint number;
52     }
53
54     Player[2] private players;
55     Stage public stage = Stage.FirstCommit;
56
57     function play(bytes32 commitment) public payable {
58         // Only run during commit stages
59         uint playerIndex;
60         if(stage == Stage.FirstCommit) playerIndex = 0;
61         else if(stage == Stage.SecondCommit) playerIndex = 1;
62         else revert("only two players allowed");
63
64         // Require proper amount deposited
65         // 1 ETH as a bet + 1 ETH as a bond
66         require(msg.value == 2 ether, 'msg.value must be 2 eth');
67
68         // Store the commitment
69         players[playerIndex] = Player(msg.sender, commitment, 0);
70
71         // Move to next stage
72         if(stage == Stage.FirstCommit) stage = Stage.SecondCommit;
73         else stage = Stage.FirstReveal;
74     }
75
76     function reveal(uint number, bytes32 blindingFactor) public {
77         // Only run during reveal stages
78         require(stage == Stage.FirstReveal || stage == Stage.SecondReveal, "wrong stage");
79
80         // Find the player index
81         uint playerIndex;
82         if(players[0].addr == msg.sender) playerIndex = 0;
83         else if(players[1].addr == msg.sender) playerIndex = 1;
84         else revert("unknown player");
85
86         // Check the hash to prove the player's honesty

```

```

87     require(keccak256(abi.encodePacked(msg.sender, number, blindingFactor)) == players[playerIndex]
88         .commitment, "invalid hash");
89
90     // Update player number if correct
91     players[playerIndex].number = number;
92
93     // Move to next stage
94     if(stage == Stage.FirstReveal) stage = Stage.SecondReveal;
95     else stage = Stage.Distribution;
96 }
97
98 function distribute() public {
99     // Only run during distribution stage
100    require(stage == Stage.Distribution, "wrong stage");
101
102    // Find winner
103    uint n = players[0].number + players[1].number;
104
105    // Payout winners winnings and bond
106    players[n%2].addr.call.value(3 ether)("");
107
108    // Payback losers bond
109    players[(n+1)%2].addr.call.value(1 ether)();
110
111    // Reset the state
112    delete players;
113    stage = Stage.FirstCommit;
114 }
```

마. 참고문헌

- [1] Keeping secrets on Ethereum
- [2] A Survey of Attacks on Ethereum Smart Contracts (SoK)
- [3] Unencrypted Secrets
- [4] Stack Overflow - Decrypt message on-chain

바. 도구

Slither	Remix plugin	Security 2.0	SmartCheck	Odin
				V