Solidity
시큐어 코딩 가이드

2018. 5.

CONTENTS

제1장 시큐어 코딩 가이드

제1절 데이터 검증

데이터를 그대로 받아들여 사용하면 많은 보안위험에 노출될 수 있다. 이러한 약점을 예방하기 위해서 유효한 데이터만 사용이 가능하도록 프로그래밍 하여 보안약점을 제거해야 한다.

1. 정수형 오버플로우와 언더플로우

가. 정의

오버플로우란 프로세스가 데이터를 저장할 때 최대 크기를 초과하는 데이터를 입력하면 정상적인 경우에는 사용되지 않아야 할 영역에 데이터가 덮어씌워 오류를 일으키거나 프로그램 보안 취약점을 야기시키는 결함이다. 언더플로우는 최소 크기보다 작은 데이터를 입력하는 경우이다. 이 결함은 수 십 년간 개발자를 괴롭혀온 오류이기 때문에 소프트웨어 보안에서 매우 중요하다. 솔리디티는 부호 없는 정수형 사용 시 최대 256비트 데이터를 처리할 수 있으므로 데이터 값을 증가시킬 때 2^{256} -1보다 클 경우 0값이 되고 감소시킬 때 0보다 작을 경우 2^{56}-1값이 되어 올바르지 않은 값을 사용할 수 있다.

나. 안전한 프로그래밍 기법

오버, 언더플로우를 피하기 위한 첫 번째 방법은 프로그래밍 할 때 데이터 경계 검사를 하는 것이다. 두 번째는 결함이 발생하지 않도록 만들어진 안전한 라이브러리를 사용하는 것이다. 이러한 방법을 통해 트랜잭션마다 경계 검사를 함으로써 보안 약점을 제거할 수 있다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: Contract OverflowUnderflow {
2:     uint public zero = 0;
3:     uint public max = 2**256-1;
4:     function increment() public {
5:         max += 1;
6:     }
7:     function decrement() public {
8:         zero -= 1;
9:     }
10:}
```

오버, 언더플로우에 대비하지 않고 데이터를 그대로 사용하고 있기 때문에 악의적인 공격에 매우 취약할 수 있는 코드이다.

■ 안전한 코드의 예

```
1: Contract OverflowUnderflow {
2:     uint public zero = 0;
3:     uint public max = 2**256-1;
4:     function increment() public {
5:         assert(max < (2**256-1));
6:         max += 1;
7:     }
8:      function decrement() public {
9:         assert(zero > 0);
10:        zero -= 1;
11:    }
12:}
```

데이터를 사용하기 전에 예외 발생 함수 assert()를 통해 경계 검사를 함으로써 결함이 발생할 가능성이 존재하는 트랜잭션을 무효로 만들어 보안 약점을 제거할 수 있다.

■ 안전한 코드의 예

```
1: library SafeMath {
2:     ……
3:     function add(uint a, uint b) internal pure returns (uint) {
4:       uint c = a + b;
5:       assert(c >= a);
6:       return c;
7:     }
8:     function sub(uint a, uint b) internal pure returns (uint) {
9:       assert(b <= a);
10:      return a - b;
11:    }
12:}
13:
14:contract SafeOverflowUnderflow {
15:    using SafeMath for uint;
16:    uint public zero = 0;
17:    uint public max = 2**256-1;
18:
19:    function increment() public {
20:          max = max.add(1);
21:    }
22:    function decrement() public {
23:          zero = zero.sub(1);
24:    }
25:}
```

오픈소스 프레임워크인 OpenZeppelin의 SafeMath 라이브러리를 통해 결함이 발생할 가능성이 있다면 트랜잭션을 무효화 시킨다. add()함수에서 두 개의 매개변수의 합이 첫 번째 매개변수 값 보다 작을 경우(오버플로우), 혹은 sub()함수에서 두 번째 매개변수 값이 첫 번째 매개변수 값보다 클 경우(언더플로우) 예외를 발생시켜 트랜잭션을 무효화 한다.

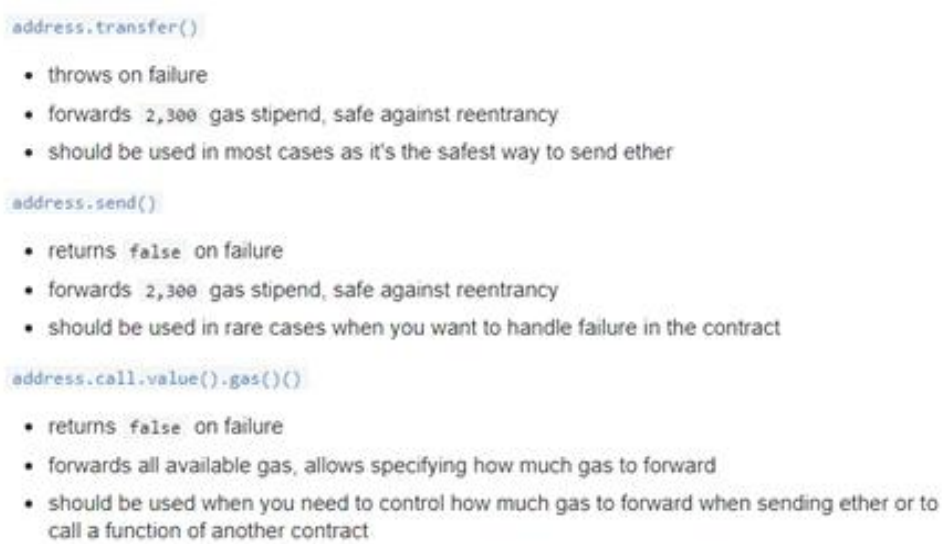
제2절 재진입성

신뢰할 수 없는 계약을 호출하면 예상치 못한 위험이나 오류가 발생 할 수 있다. 외부 호출은 그 계약이나 그 계약에 의존하는 다른 계약에 의해 악의적인 코드를 실행할 수 있다. 따라서 모든 외부 호출을 잠재적인 보안위험으로 간주하고 프로그래밍 해야 한다.

1. 이더 전송시 잘못된 함수 사용

가. 정의

이더를 전송하고자 하는 contract의 fallback 함수 내부에 외부 호출 코드가 작성되어 있을 경우 address.call.value().gas()(), address.send(), address.transfer() 각각의 특징을 고려하지 않고 사용하여 예상치 못한 외부 호출을 할 수 있다. address.call.value().gas()()는 fallback 함수의 명령어를 실행하기 위한 가스를 지정하기 때문에 악의적인 코드가 작성되어 있을 경우 예상하지 못한 코드를 실행할 수 있다. address.send()는 fallback 함수가 명령어실행을 위해 요구되는 가스량을 2300으로 제한한다. 이더 전송이 실패할 경우 false를 반환한다. 마지막으로 address.transfer() 또한 가스량을 2300으로 제한하지만, 이더 전송이 실패 시 예외를 발생시켜 트랜잭션을 무효화 시킨다.



나. 안전한 프로그래밍 기법

fallback 함수의 명령어를 실행 시키기 위해서는 address.call.value().gas()()를 통해 가스량을 설정해주고, fallback 함수가 전송 받은 이더만 처리할 수 있도록(2300 가스) 제한할 때 address.send()이나 address.transfer() 사용을 한다. 하지만 fallback 함수 내에서 외부 호출의 위험이 있기 때문에 전송 가스를 2300으로 제한하는 것을 지향한다. 이더 전송이 실패할 경우 address.transfer()는 예외를 발생으로 데이터 보호에 있어 address.send() 보다 안전하기 때문에 이더 전송 함수는 address.transfer()을 사용하도록 한다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract Bank {
2:     uinttotalDeposit;
3:     mapping(address => uint) balance;
4:     ……
5:     function withdraw(uint amount) public payable {
6:         require(amount <= balance[msg.sender]);
7:         balance[msg.sender] -= amount;
8:         totalDeposit -= amount;
9:         msg.sender.call.value(amount).gas(3000)();
10:    }
11:}
```

address.call.value().gas()() 사용시 사용할 수 있는 가스량을 설정해주기 때문에 fallback 함수에서 예상치 못한 외부 호출이 발생할 수 있다. 외부 호출은 그 계약이나 그 계약에 의존하는 다른 계약에 의해 악의적인 코드를 실행할 수 있으므로 잠재적인 보안 위험이다.

■ 안전한 코드의 예

```
1: contract Bank {
2:     uinttotalDeposit;
3:     mapping(address => uint) balance;
4:     ……
5:     function withdraw(uint amount) public payable {
6:         require(amount <= balance[msg.sender]);
7:         balance[msg.sender] -= amount;
8:         totalDeposit -= amount;
9:         msg.sender.transfer(amount);
10:    }
11:}
```

공격자가 fallback 함수의 악의적인 코드를 삽입하고 계약을 배포하더라도, 이더를 전송하는 사용자가 address.transfer()를 사용하여 fallback 함수가 현재 전송받은 이더만 처리할 수 있도록 한다면 예상하지 못한 보안 위험이 존재하더라도 그것을 예방할 수 있다.

2. fallback 함수 기능으로 인한 위험

가. 정의

fallback 함수는 호출하는 함수가 contract에 존재하지 않거나 이더를 전송 받았을 때 대체 기능으로 자동 호출되는 함수로서, fallback 함수를 구현할 때 외부 호출 코드를 삽입한다면 공격자에 의해 예상하지 못한 위험을 야기할 수 있다.

나. 안전한 프로그래밍 기법

외부 호출 코드로 인해 계약이 위험에 빠질 수 있기 때문에 기본적으로 컨트랙트 내에 함수에서 외부 호출 코드 삽입을 지양한다. 만일 외부 호출이 불가피할 경우 mutex를 이용할 수 있다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract SimpleDAO {
2:     mapping (address => uint) public credit;
3:     function queryCredit(address to) public returns (uint) { return credit[to]; }
4:     function withdraw(uint amount) public {
5:         if(credit[msg.sender] >= amount) {
6:             msg.sender.transfer(amount);
7:             credit[msg.sender] -= amount;
8:         }
9:     }
10:}
11:
12:contract Mallory {
13:     SimpleDAO public dao
14:     = SimpleDAO(0xca35b7d915458ef540ade6068dfe2f44e8fa733c);
15:     address owner;
16:     function Mallory() { owner = msg.sender; }
17:     function() public { dao.withdraw(dao.queryCredit(this)); }
18:}
```

Mallory contract에 이더를 전송할 경우 fallback 함수가 자동 호출되고 내부에서 SimpleDAO 객체 dao의 withdraw()를 통해 외부 호출을 하게 된다. withdraw()에는 호출한 contract주소로 이더를 전송하여 다시 Mallory contract의 fallback 함수가 호출되는 재진입성의 위험이 발생하게 된다.

■ 안전한 코드의 예

```
1: contract SimpleDAO {
2:     bool lockCredit;
3:     mapping (address => uint) public credit;
4:     function queryCredit(address to) public returns (uint) { return credit[to]; }
5:     function withdraw(uint amount) public {
6:         require(!lockCredit);
7:         lockCredit = true;
8:         if(credit[msg.sender] >= amount) {
9:             msg.sender.transfer(amount);
10:            credit[msg.sender] -= amount;
11:        }
12:        lockCredit = false;
13:    }
14: }
15:
16:contract Mallory {
17:    SimpleDAO public dao
18:    = SimpleDAO(0xca35b7d915458ef540ade6068dfe2f44e8fa733c);
19:    address owner;
20:    function Mallory() { owner = msg.sender; }
21:    function() public { dao.withdraw(dao.queryCredit(this)); }
22:}
```

프로그램 코드 작성 시 외부 호출 코드 삽입이 불가피할 경우, mutex를 이용할 수 있다. 위 프로그램에서 withdraw()함수는 lockCredit값이 true일 경우 require()를 통해 예외를 발생시켜 함수가 실행되지 않도록 하고 함수가 종료되기 직전에 lockCredit 값을 false로 설정한다. 만일 호출된 함수가 실행도중 재진입을 하게 될 경우 lockCredit값이 true이기 때문에 재진입을 막을 수 있다. 호출된 설정예외 발생 함수 require()으로 bool 변수 lockCredit 값을 확인하여 외부 호출로 인한 재진입성을 예방한다.

3. 잘못된 접근지정자 사용

가. 정의

솔리디티는 기본적으로 contract내부 변수의 디폴트 접근지정자는 internal이며, 함수는 public이다. 접근지정자를 default로 하거나 잘못 사용할 경우 예상하지 못한 외부 접근이 이루어 질 수 있다.

나. 안전한 프로그래밍 기법

접근지정자를 명시적으로 표기해주지 않으면 디폴트 접근지정자가 적용되며, 함수의 경우 public이기 때문에 누구나 접근할 수 있어 예상하지 못한 위험에 빠질 수 있다. 그렇기 때문에 프로그램 코드 작성 시 접근지정자를 명시적으로 표기해야 한다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract WalletLibrary {
2:     // ...
3:     function initWallet(address[] _owners, uint _required, uint _daylimit) {
4:         //...
5:         initMultiowned(_owners, _required);
6:     }
7:
8:     function initMultiowned(address[] _owners, uint _required) {
9:         //...
10:    }
11:}
```

Parity Wallet 해킹 사건의 주요 원인으로서 WalletLibrary에서 initWallet()은 지갑의 주인만 호출할 수 있어야 하는데 접근지정자를 명시적으로 표기하지 않아 public으로 간주되어 누구나 지갑을 생성하고 주인을 바꿀 수 있다.

■ 안전한 코드의 예

```
1: contract WalletLibrary {
2:     // ...
3:     function initWallet(address[] _owners, uint _required, uint _daylimit) internal{
4:         //...
5:         initMultiowned(_owners, _required);
6:     }
7:
8:     function initMultiowned(address[] _owners, uint _required) internal{
9:         //...
10:    }
11:}
```

함수의 접근지정자가 public일 경우 누구나 접근가능하며 internal일 경우 같은 컨트랙트내에 함수만 접근이 가능하다. external로 접근지정자를 설정할 경우 컨트랙트 내의 함수가 아닌 다른 컨트랙트에서 호출이나 사용자 계정의 직접적인 트랜잭션으로 접근이 가능하다. 위 프로그램은 외부에서 접근이 불가능하도록 하기 위해 internal을 사용하고 있다.

제3절 예외 발생

예외 발생은 잘못된 트랜잭션에 의한 데이터를 보호하는 데 있다. 하지만 공격자는 예외 발생 코드를 이용하여 계약이 본 기능을 하지 못하도록 할 수 있다.

1. DoS (서비스 거부) 공격

가. 정의

시스템을 악의적으로 공격해 해당 시스템의 자원을 부족하게 하여 원래 의도된 용도로 사용하지 못하게 하는 공격으로서, 예외를 계속 발생시켜 공격할 수 있다.

나. 안전한 프로그래밍 기법

예외가 발생하게 되면 모든 트랜잭션이 무효가 된다. 만일 어떠한 목적을 위한 트랜잭션이 항상 무효가 된다면 원래 의도된 용도로 사용할 수 없게 된다. DoS공격에 취약한 부분을 별도의 트랜잭션으로 분리한다면 예방 할 수 있다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract Auction {
2:     address currentLeader;
3:     uinthighestBid;
4:
5:     function bid() payable external{
6:         require(msg.value > highestBid);
7:         if(currentLeader != 0){
8:             currentLeader.transfer(highestBid);
9:         }
10:        currentLeader = msg.sender;
11:        highestBid = msg.value;
12:    }
13:}
```

최고입찰자가 변경되었을 때, 기존의 currentLeader의 주소로 이더를 환불시 transfer()가 실패하면(입찰 후 계정 삭제 등) 트랜잭션이 무효가 되어 입찰기능을 사용할 수 없게 된다.

■ 안전한 코드의 예

```
1: contract Auction {
2:     address currentLeader;
3:     uinthighestBid;
4:     mapping(address => uint) refunds;
5:
6:     function bid() payable external{
7:         require(msg.value >highestBid);
8:         if(currentLeader != 0){
9:             refunds[currentLeader] += highestBid;
10:        }
11:        currentLeader = msg.sender;
12:        highestBid = msg.value;
13:    }
14:
15:    function withdrawRefund() external {
16:        uint refund = refunds[msg.sender];
17:        refunds[msg.sender] = 0;
18:        msg.sender.transfer(refund);
19:    }
20:}
```

Mapping을 이용하여 refunds에 addres인덱스로 uint값을 저장한다. Bid()함수에서 입찰가가 현재 최고입찰가 보다 크지 않을 경우 require()를 통해 무효화 시킨다. 그렇지 않을 경우 if문을 통해 이전 최고입찰자에게 환불할 이더를 refunds에 저장하고 있다. 이 후 최고입찰자의 주소값을 currentLeader에 할당하고 최고입찰가도 변경한다. 최고입찰자가 변경되었을 경우 자동으로 이전 최고입찰자에게 이더를 환불하는 것이 아니라 bid()와는 다르게 withdrawRefund()를 만들어 환불 받을 수 있도록 하였다. 이렇게 트랜잭션을 분리하여 DoS공격을 예방할 수 있다.

제4절 특수 변수 사용

솔리디티에서는 블록체인에 대한 정보를 제공하는 특수 변수가 전역 네임 스페이스에 항상 존재한다. 이러한 변수를 통해 필요한 데이터를 쉽게 얻을 수 있는 장점이 있지만 보안 약점도 존재한다.

1. tx 원점 공격

가. 정의

트랜잭션에 관한 데이터를 가지고 있는 tx를 이용하여 필요한 데이터를 쉽게 얻을 수 있다. tx.gasprice, tx.origin이 대표적인 예이며 gasprice는 거래의 가스 가격 값을 가지고 origin은 거래의 최초 생성자 주소 값을 가진다. 하지만 이러한 데이터를 이용한 악의적인 계약을 배포하여 예상하지 못한 위험을 일으킬 수 있다.

나. 안전한 프로그래밍 기법

사용자를 인증하기 위해 tx.origin와 컨트랙트 호출자를 비교하지만 악의적인 컨트랙트에 의해 위험(정보 유출)에 빠질 수 있다. 그렇기 때문에 tx.origin 사용을 지양한다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract MyWallet {
2:     address owner;
3:
4:     function MyWallet() public {
5:         owner = msg.sender;
6:     }
7:
8:     function sendTo(address to, uint amount) public {
9:         require(tx.origin == owner);  to.transfer(amount);
10:    }
11:}
```

```
12:
13: contract AttackingContract {
14:     address attacker;
15:
16:     function AttackingContract() public {
17:         attacker = msg.sender;
18:     }
19:
20:     function() public {
21:         MyWallet(msg.sender).sendTo(attacker, msg.value);
22:     }
22: }
23:
24:
```

사용자가 지갑에서 이더를 AttackingContract로 전송 했을 때, fallback함수가 자동으로 실행되고 MyWallet().sendTo()를 호출하며 생성자 주소인 attacker(공격자 주소)와 현재 트랜잭션의 이더를 매개변수로 넘긴다. MyWallet에서 owner의 값은 사용자의 주소값이 되고 tx.origin값 또한 사용자의 주소값이 된다. 결국 tx.origin과 MyWallet의 owner가 일치하게 된다. 그렇기 때문에 require()호출에도 예외가 발생하지 않고 to에게 이더를 전송하게 된다. to는 attacker의 주소로서 사용자는 자신의 전송 이더를 잃어버릴 수 있다.

■ 안전한 코드의 예

```
1: contract MyWallet {
2:     address owner;
3:
4:     function MyWallet() public {
5:         owner = msg.sender;
6:     }
7:
8:     function sendTo(address to, uint amount) public {
9:         require(to == owner);  to.transfer(amount);
10:    }
11:}
```

```
12:
13: contract AttackingContract {
14:     address attacker;
15:
16:     function AttackingContract() public {
17:         attacker = msg.sender;
18:    }
19:
20:     function() public {
21:         MyWallet(msg.sender).sendTo(attacker, msg.value);
22:     }
22: }
23:
24:
```

tx.origin을 통하여 컨트랙트를 배포한 생성자와 사용자를 비교하려고 하였지만 컨트랙트가 컨트랙트를 생성할 경우 예상하지 못한 오류가 발생할 수 있다. 이러한 오류를 예방하기 위해서 tx.origin사용을 지양하고 만일 원점 주소를 알아야 할 때는 처음 컨트랙트를 배포한 주소를 매개변수로 연결하여 해결할 수 있다.

제4절 OOP 특성

솔리디티는 객체지향 특성을 가지고 있다. C++, Java 등 객체지향 언어에서 class를 사용하는 것처럼 솔리디티는 contract 지정어를 통해 계약을 추상화시키며, 정보 은닉이 가능하다. 또한 상속을 통해 코드의 재사용이 가능하고 중복 및 재정의를 통한 다형성을 보장한다. 하지만 이러한 특성으로 인해 예상하지 못한 오류가 발생할 수 있다.

1. 다중 상속으로 인한 문제

가. 정의

다중 상속이란 OOP 특징 중 하나이며, 어떤 클래스가 하나 이상의 상위 클래스로부터 여러 가지 행동이나 특징을 상속받을 수 있는 것을 말한다. 만일 다중 상속 관계에서 같은 이름의 멤버 함수가 있다면 모호성이 발생할 수 있다.

나. 안전한 프로그래밍 기법

솔리디티는 다중 상속 관계에서 모호성이 발생할 경우 가장 최근에 상속받은 컨트랙트를 처리하게 된다. 이것은 의도하지 않은 결과일 수 있으며 자바에서는 모호성 문제로 인해 클래스의 다중 상속을 지원하지 않는다. 솔리디티에서는 모호성을 해결하기 위해 다중 상속을 지양한다.

다. 예제

■ 안전하지 않은 코드의 예

```
1: contract ownedContract {
2:     uint a;
3:     function get() public constant returns(uint) {
4:         return a;
5:     }
6: }
7:
```

```
 8: contract BaseNo1 is ownedContract {
 9:     function set() public {
10:         a = 1;
11:     }
12: }
13:
14: contract BaseNo2 is ownedContract {
15:     function set() public {
16:         a = 2;
17:     }
18: }
19:
20: contract Final is BaseNo1, BaseNo2{}
```

Final 컨트랙트를 생성 후 set()을 호출하였을 때, C++에서는 컴파일오류가 발생하지만 솔리디티는 BaseNo1과 BaseNo2에서 재정의된 set() 중 BaseNo2의 set()을 호출하게 되어 get()을 호출할 경우 a의 값은 2가 된다. 만일 BaseNo1 컨트랙트에 중요한 코드가 들어가 있다면 문제가 될 수 있다.

■ 안전한 코드의 예

```
1: contract ownedContract {
2:     uint a;
3:     function get() public constant returns(uint) {
4:         return a;
5:     }
6: }
7:
8: contract Base is ownedContract {
9:     function set() public {
10:         a = 1;
11:     }
12:}
13:
15:contract Final is Base { }
```

솔리디티에서 상속을 사용할 때 컴파일러가 오류로 인지하지 않기 때문에 의도하지 않은 오류를 막기 위해서 다중 상속은 지양한다.

참고 문헌

[1] Ethereum Smart Contract Security Best Practices,
https://consensys.github.io/smart-contract-best-practices/

[2] Recommendations for Smart Contract Security in Solidity,
https://consensys.github.io/smart-contract-best-practices/recommendations/

[3] Known Attacks,
https://consensys.github.io/smart-contract-best-practices/known_attacks/

[4] How to Secure Your Smart Contracts: 6 Solidity Vulnerabilities and how to avoid them,
https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-1-c33048d4d17d

[5] More Ethereum Attacks: Race-To-Empty is the Real Deal,
https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/

[6] Contract Safety and Security Checklist,
http://www.kingoftheether.com/contract-safety-checklist.html

[7] Solidity: Tx Origin Attacks,
https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514

[8] Solidity Inheritance,
https://www.bitdegree.org/learn/solidity-inheritance/