있고, contract가 설치(deploy)될 때 실행되는 생성자와 일반 함수들로 구성되어 있다. 일반 함수 외에 트랜잭션이 특정한 함수를 지정하지 않은 경우에 실행되는 폴백(fallback)함수라는 특수함수도 포함시킬 수 있다. 본 절에서는 Solidity 언어의 구성요소 중에서 취약점과 연관성이 높은 생성자, 변경자, 폴백 함수에 대하여 설명한다.

2. 생성자(Constructor)

Solidity 언어는 객체지향언어(OOP)이기 때문에 객체지향 특징 중 하나인 생성자(Constructor)를 포함하고 있다. 생성자는 스마트 계약을 최초로 배포(deploy)할 시에 자동으로 호출되는 함수이다. 생성자는 필수적으로 작성하는 문법은 아니지만 정의할 시에 단 1개의 생성자만을 정의해야 한다. 스마트 계약의 최초 배포 시 생성자가 없는 경우 기본 생성자가 자동(default)으로 생성된다. 외부 호출 여부를 결정하기 위해서 internal이나 public의 옵션을 필수로 설정해야 한다. 그림 1.5와 1.6은 생성자 함수의 기본적인 문법과 예제 코드를 나타낸다.

```
constructor(매개변수) public {
        생성자 함수 내에서 필요한 로직
}
```

[그림 1.5] 생성자 함수의 기본 문법

```
contract example {
        mapping(address => uint256) public balanceOf;
        constructor() public {
                balanceOf[msg.sender] = 100;
        }
}
```

[그림 1.6] 생성자 함수의 기본 예제 코드

제 2 장 스마트 계약 보안 취약점 조사

제 1 절 개요

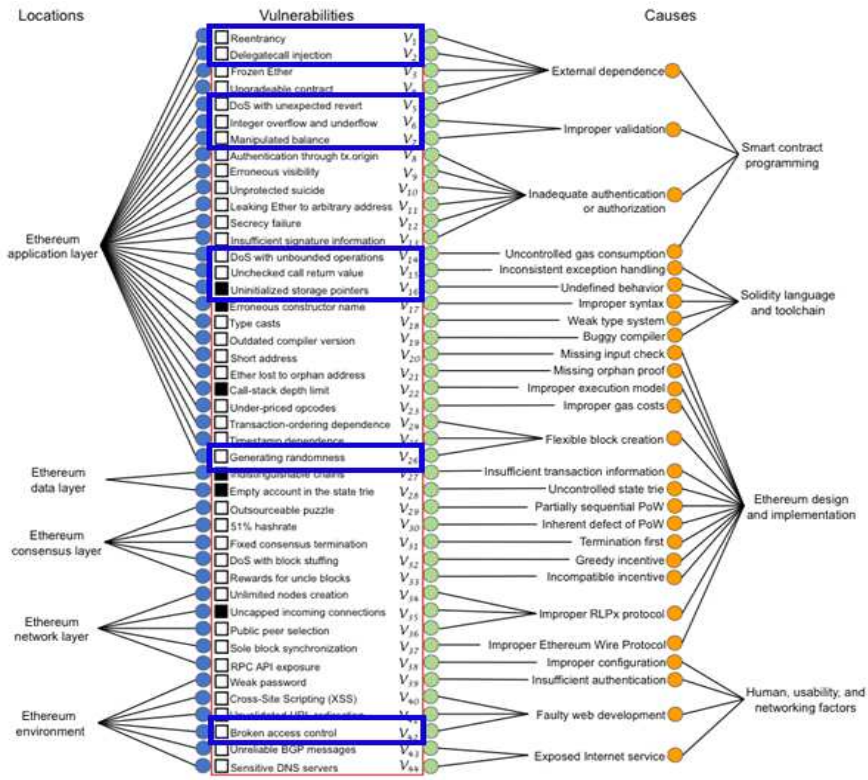
스마트 계약과 연관해서 발생할 수 있는 취약점은 스마트 계약의 언어적 문제에서 시작하여 블록체인 시스템의 설계 문제까지 연관되어 있다. 그림 3.1은 스마트 계약의 취약점이 나타나는 위치와 발생하는 원인을 연결한 그림으로 스마트 계약과 관련 취약점의 복잡한 상관관계를 보여준다. 블록체인에서 발생하는 취약점들은 블록체인 시스템의 설계상의 한계 때문에 발현되기도 하고, 사용자의 단순한 실수에 의해서 발생하기도 한다. 그림 2.1에 나와 있는 것처럼, 주로 스마트 계약의 하부 계층(consensus와 network layer)에서 발생하는 문제들이 이에 해당한다. 응용 계층(application layer)의 문제들은 올바르지 않은 스마트 계약의 버그들이나 개발자의 잘못된 접근 때문에 발생한다. 따라서 이러한 문제점들은 스마트 계약의 취약점으로 이어지고, 적절한 점검과 개발 패턴을 구성하여 예방되어야 한다.

스마트 계약의 취약점은 지속적으로 변화하고 있는 스마트 계약 플랫폼들의 특수성 및 사용 형태와 연관된다. 스마트 계약에서 발생하는 취약점을 분석하기 전에 다음과 같은 문제들을 고려해야 한다.

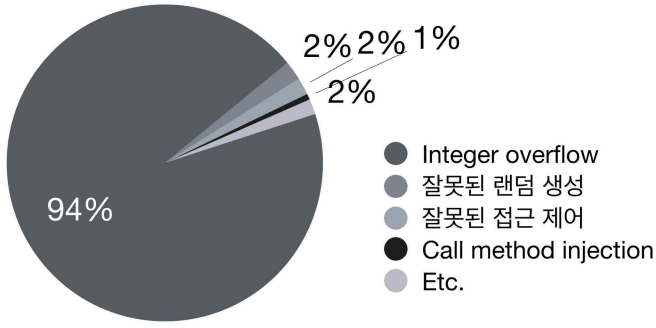
- (취약점 명명의 표준 미비) 스마트 계약의 취약점의 이름이 표준화 되어 있지 않아서 대표적이거나 전통적인 취약점을 제외하고는 분석자나 분석도구마다 서로 다른 이름을 사용한다.

- (취약점의 모호한 위험도) 스마트 계약의 발전과 함께 취약점을 공격하는 방법도 지속적으로 진화하고 있어 스마트 계약의 개별적인 위험도에 대한 정의가 어려운 상황이다.

본 보고서에서는 최근 가장 빈번하게 공격받고 있는 스마트 계약의 취약점을 중심으로 전통적으로 큰 피해를 가져오거나 스마트 계약의 개발과정에서 반복해서 발생하는 취약점을 대상으로 분석을 진행한다.



[그림 2.1] 스마트 계약 취약점 종류[53]



[그림 2.2] 주로 이용되는 스마트 계약 취약점 (2019년 7월 CVE 기준)

그림 2.2는 2019년 7월 기준으로 스마트 계약에 대해서 보고된 CVE[3]의

비율로 대부분의 취약점이 integer overflow 공격에 해당되고 그 외에 잘못된 랜덤 생성과 잘못된 접근 제어의 공격으로 구성되어 있다는 것을 알 수 있다. 현재 운영 중인 대부분의 스마트 계약은 interger overflow에 취약한 ERC-20 형태이기 때문으로 추정된다. 이러한 결과를 바탕으로 고려해야할 주요 취약점을 정리하면 표 2.1과 같다. 표준화되어 있지 않은 취약점 이름을 통일하기 위하여 표 2.1에는 스마트 취약점 분석 논문[53]의 표기를 기준으로, 본 보고서에서 사용할 한글 취약점 이름이 함께 표기되어 있다. 각 취약점에 대한 자세한 설명은 이후 절에서 이어진다.

<표 2.1> 주요 스마트 계약 취약점 및 표준 이름
(위험도는 Mythx점검도구에서 제시하는 위험도를 기준(2019.09.21.)으로 함)

표준 이름	취약점 이름	설명	위험도
Re-entracy	의도하지 않은 재진입	하나의 함수가 개발자의 의도와는 다르게 반복적으로 호출	중간 (Medium)
Integer overflow and underflow	Integer Overflow / Underflow	과도한 연산이나 올바르지 못한 값이 사용되어 정수의 범위를 벗어나는 취약점	높음 (High)
DoS with unexpected revert	논리 DoS	논리적 오류로 인하여 스마트 계약의 진행이 불가능한 경우	중간 (Medium)
DoS with unbounded operations	Gas limit을 이용한 DoS	부정확한 gas cost의 할당으로 인하여 스마트 계약의 사용이 불가능해지는 경우	중간 (Medium)
Manipulated balance	강제 잔액 변조	강제로 이더를 전송하여 스마트 계약의 잔액 정보를 변조	높음 (High)

3) 관련 링크:
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smart+contract

Uninitialized storage pointers	초기화되지 않는 스토리지 포인터	초기화되지 않은 스토리지 포인터를 이용한 변조 공격	낮음 (Low)
Unchecked call return value	Return값 미확인	실행과정의 실제 수행 조건과 결과가 중요한 함수들을 적절한 확인 없이 실행	중간 (Medium)
Generating randomness	잘못된 랜덤 생성	노출된 정보를 이용하여 유추 가능한 랜덤 결과 이용	중간 (Medium)
Broken access control	잘못된 접근 제어	권한 확인이 중요한 스마트 계약에서 비정상적으로 소유자 정보 변경이 가능	높음 (High)
미정	Call method injection	Call 명령을 통한 임의의 함수 실행	높음 (High)

제 2 절 스마트 계약 주요 보안 취약점

1. 의도하지 않은 재진입 (Re-entrancy)

가. 개요 및 특징

- The DAO의 해킹 사례를 통해 알려진 취약점
- **위험도 :** 중간(Medium)
- 트랜잭션 수신자의 종류를 고려하지 않아 스마트 계약의 예외처리 함수(fallback)를 통해서 의도치 않은 재진입으로 특정 기능 반복 수행
- **관련함수 :** (address).call.value() 함수

The DAO의 해킹 사례를 통해 유명해진 취약점으로 기존의 컴퓨팅 환경에서는 고려되지 못한 문제점을 이용하고 있으며 피해액이 크다는 점에서 스마트 계약의 대표적인 취약점에 해당한다. 이 취약점은 의도하지 않게 호출한 함수와 호출된 함수 사이에서 서로를 재귀적으로 호출하는 순환구조가 생길 수 있는 점을 이용한다. Call이란 함수를 통해서 특정한 주소로 이더를 전송하려 할 때, 수신하는 계정이 일반 사용자 계정이 아닌 스마트 계약이라면 폴백 함수를 이용하여 의도하지 않은 제어 흐름을 만들 수 있다. 특히 call.value(비용)와 같은 형태로 이더를 함께 전송하는 호출의 경우에는 이러한 재귀적인 순환구조를 통해서 의도된 비용 이상을 편취하는 공격으로 이어질 수 있다.

나. 예제 코드

그림 2.3의 함수 withdrawBalance가 포함하고있는 call.value 함수는 re-entrancy 취약점의 원인이 되는 함수이다. 일반적으로 call.value를 호출하는 과정에서 호출 대상자(여기서는 msg.sender)가 만약 스마트

계약 계정인 경우 폴백 함수가 처리를 담당한다. 이때 해당 폴백 함수가 이전에 호출했던 withdrawBalance 함수를 다시 호출하는 경우, 둘 사이의 재귀적 순환구조가 형성되고 gas가 모두 소진될 때까지 반복하여 call.value 함수가 반복적으로 수행된다. 따라서 원래의 의도와는 다르게 amountToWithdraw 만큼의 이더를 반복적으로 전송하게 되어 허가된 비용 이상을 인출할 수 있게 된다.

```
contract Vulnerability {
mapping(address => uint) private userBalance;
function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        require(msg.sender.call.value(amountToWithdraw)(""));
        userBalances[msg.sender] = 0;
} }
```

[그림 2.3] Re-entrancy의 기본 취약점 예시

다. 피해사례

이더리움 대표 프로젝트인 The DAO의 서비스 과정에서 '의도하지 않은 재진입 취약점'을 이용한 해킹 발생하여 한화로 약 750억원 가량의 피해가 발행하였다. 이 경우 call.value()를 통한 재진입으로 인해 대량의 Ether가 탈취된 것으로 확인되었으며, 세부 내용은 splitDAO 함수(그림 2.4)의 createTokenProxy.value(call.value) 함수를 통해 확인할 수 있다.

```
function splitDAO(uint _proposalID, address _newCurator)
noEther onlyTokenholders returns (bool _success) {
...
if(p.splitData[0].newDAO.createTokenProxy.value
  (fundsToBeMoved)(msg.sender) == false) throw;
...
```

[그림 2.4] The DAO 해킹 사건의 원인 코드

2. Interger Overflow / Underflow

가. 개요 및 특징

- 값이 주어진 자료형에서 표현할 수 있는 크기를 벗어나는 경우 발생

- 표현 범위를 벗어나면서 최소한계(e.g., 0)을 지나는 경우에는 underflow, 최대한계(e.g., 2^{256}-1)를 지나는 경우에는 overflow로 표현

- **위험도 :** 높음(High)

Interger Overflow/Underflow는 스마트계약 이외의 일반적인 개발에도 발생하는 전통적인 취약점으로써, 값이 주어진 자료형에서 표현할 수 있는 크기를 벗어나는 경우 발생하게 된다. 예를 들어 ERC20 표준 Token에서 잔액(balance)를 다루는 uint256(256 bit의 unsigned integer)는 0 ~ 2^{256}-1 범위의 값을 가지게 된다. 표현 범위를 벗어나면서 최소한계(e.g., 0)을 초과하는 경우는 underflow, 최대한계(e.g., 2^{256}-1)를 초과하는 경우는 overflow로 표현한다. 일반적인 컴퓨팅 환경에서는 실제 공격 코드로 이어지기 어려운 문제 때문에 치명적이지 않은 버그이지만, 스마트 계약에서는 overflow를 통해서 잔액의 정보가 부정확해질 수 있어서 치명적인 버그로 작용한다. 스마트 계약의 용도가 ERC20 표준 Token의 구현으로 많이 치우져 있는 상황에서 특히 위협적인 취약점이다.

나. 예제 코드

그림 2.5는 Integer Overflow 취약점의 예시이다. 그림 2.5 내에서 balanceOf의 value의 타입은 uint256에 해당하므로 _value의 매개변수가 256 비트의 최대 한계를 넘어설 경우 Overflow가 발생한다.

```
mapping(address => uint256) public balanceOf;
function transfer(address _to, uint256 _value) {
        require(balanceOf[msg.sender] >= _value);
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
}
```

[그림 2.5] Integer Overflow 기본 취약점 예시

그림 2.6는 Integer Underflow의 취약점의 예시이다. 그림 3.6에서 balancesOf[msg.sender]의 값이 0일 경우 _value 만큼의 뺄셈 연산을 하게되므로 Underflow가 발생한다. 이러한 취약점을 통해 balancesOf의 value 값을 조정할 수 있다.

```
mapping(address => uint) balances;
uint public totalSupply;
function transfer(address _to, uint _value) public returns (bool){
        require(balances[msg.sender] - _value >= 0);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        return true;
}
```

[그림 2.6] Integer Underflow 기본 취약점 예시

다. 피해사례

이더리움 기반의 ERC-20 표준을 구현한 SMT(SmartMesh) 토큰 코드에서 Integer Overflow가 발생하였다. 그림 2.7의 transferProxy함수의 balances[_from] < _fee + _value의 조건 체크의 부재로 인해 발생하였다. _fee와 _value의 값을 조절하여 _fee + _value의 값을 overflow시켜서 해당하는 조건을 통과하고 balances[_to] += _value를 통해 _to의 주소로 막대한 금액의 토큰 전송을 유도하였다. 해당 취약점을 이용하여 공격자는 1경개의 토큰 발행 후 탈취하였다. (그림 2.8)

```
203     function transferProxy(address _from, address _to, uint256 _value, uint256 _fee,
204         uint8 _v,bytes32 _r, bytes32 _s) public transferAllowed(_from) returns (bool){
205
206         if(balances[_from] < _fee + _value) revert();
207
208         uint256 nonce = nonces[_from];
209         bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
210         if(_from != ecrecover(h,_v,_r,_s)) revert();
211
212         if(balances[_to] + _value < balances[_to]
213             || balances[msg.sender] + _fee < balances[msg.sender]) revert();
214         balances[_to] += _value;
215         Transfer(_from, _to, _value);
216
217         balances[msg.sender] += _fee;
218         Transfer(_from, msg.sender, _fee);
219
220         balances[_from] -= _value + _fee;
221         nonces[_from] = nonce + 1;
222         return true;
223     }
```

[그림 2.7] MESH & SMT Token의 Integer Overflow 코드

[그림 2.8] MESH & SMT Token의 Integer Overflow 트랜잭션

3. 논리 DoS 공격 (DoS with unexpected revert)

가. 개요 및 특징

- 스마트 계약의 논리적 오류를 이용한 DoS(Denial of Service) 공격
- 위험도: 중간(Medium)
- 스마트 계약에서 많이 사용하는 send 함수의 사용 패턴을 이용
- 함수의 진행을 위해서 확인하는 조건을 항상 실패하도록 유도하여 상태를 고착화

스마트 계약의 논리적 오류를 이용하여 스마트 계약의 실행이 더 이상 진행되지 못하도록 고착상태에 빠트리는 공격이다. 일반적으로 스마트 계약에서 많이 사용하는 send 함수를 이용해서 고착상태에 빠트리는 경우가 많다. send 함수는 이더를 송금하고 송금의 성공 여부를 true 또는 false로 리턴하게 되는데, 컨트랙트 내의 payable 옵션이 있는 함수에 revert를 정의하는 등 송금이 항상 실패하는 조건으로 스마트 계약을 유도하여 아무도 사용할 수 없는 상황으로 유도한다. Send 함수는 require 문 등을 통해서 결과를 확인하고, 송금이 실패하는 경우 실행을 중지(revert)하는 형태의 패턴을 많이 사용하고 있어 공격을 구성하기 위한 주요 목표가 된다. 또한 스마트 계약은 변경불가(immutable)한 성질을 가지고 있기 때문에, 한번 고착화된 스마트 계약은 임의로 상태를 변경할 수 없어 회복이 불가능하다.

나. 예제 코드 및 피해 사례

그림 2.9는 논리적 버그를 이용한 DoS 공격의 예시 코드로 실제 관련된 취약점을 통해 공격을 받은 King of the Ether 서비스의 코드의 간략하게 표현하였다. King of the Ether는 더 많은 양의 이더(highestBid)를 제시한 주소가 군주가 되고 나머지의 주소들은 일정량의 이더를 군주에

게 보내야하는 게임으로, 즉 King of the Ether 내 현재 군주가 되는 변수인 currentLeader가 바뀔 경우 이전 currentLeader에게 send 함수를 통해 이더를 지급한다. 하지만, 이더를 지급 받을 계정이 이더 지급을 거부할 경우 (스마트 계약의 폴백 함수를 조정하여 만들 수 있다.) 해당 계정으로 송금하려는 send 함수의 결과가 실패하고 실행이 중지된다. 그림 2.9의 경우는 require문에서 실행이 중지되고 나면 이후에 있는 currentLeader를 변경하는 코드도 실행되지 못하여 더 이상 상태를 변경하거나 진행할 수 없는 고착 상태에 빠지게 된다.

```
contract Auction {
        address currentLeader;
        uint highestBid;
        constructor() public { currentLeader = msg.sender; }
        function bid() payable {
                require(msg.value > highestBid);
                require(currentLeader.send(highestBid));
                currentLeader = msg.sender;
                highestBid = msg.value;
        }
}
```

[그림 2.9] 논리적 버그를 이용한 DoS 취약점 예시

4. Gas Limit을 이용한 DoS 공격 (DoS with unbounded operations)

가. 개요 및 특징

- EVM에서는 무분별한 자원사용을 제한하고자 한 블록 내에서 사용할 수 있는 gas의 양을 제한 (gas limit)

- 위험도: 중간(Medium)

- 특정 함수의 실행과정에서 gas limit을 넘는 경우, 해당 함수를 더 이상 사용할 수 없음

- Gas limit을 고려하지 않은 반복문의 조건 설정 등의 이유로 문제 발생

EVM에서는 무분별한 스마트 계약의 실행을 방지하고자 한 블록에서 사용할 수 있는 gas의 양을 제한한다. 즉, 한 블록에서 실행시킬 수 있는 명령의 한계를 정의하고 있는 셈이다. 하지만 개발과정에서 이러한 블록의 gas limit을 고려하지 않고 개발하는 경우, 연산이 많은 특정 함수를 실행하는 과정에서 항상 gas limit을 넘게 되어 더 이상 해당 함수를 사용할 수 없게 되는 상황에 빠지게 된다. 일반적으로 스마트 계약에서 반복문을 사용할 때, gas limit을 고려하여 반복 회수를 적절하게 설정하지 않는 경우에 발생한다.

나. 예제 코드

그림 2.10은 Gas Limit을 이용한 DoS 공격의 예이다. 예제 코드에서 payOut 함수내에 있는 while 문이 반복되기 위해서는 두 가지 조건을 만족해야 한다. 트랜잭션에 남아있는 gas의 양(msg.gas)이 충분하다면 payees.length 만큼 반복이 일어나게 된다. 하지만, payees의 숫자가 점점 늘어나서 payees.length의 크기가 일정 이상 커지는 경우에, 전체 반복문을 모두 실행시키기 위한 gas의 양이 gas limit보다 커지게 되어 payees.length가 줄어들기 전까지는 payOut 함수가 성공적으로 실행될 수 없게 된다.

```
struct Payee {
        address addr;
        uint256 value;
}
Payee[] payees;
uint256 nextPayeeIndex;
function payOut() {
        uint256 i = nextPayeeIndex;
        while (i<payees.length && msg.gas > 200000) {
                payess[i].addr.send(payees[i].value);
                i++;
        }
        nextPayeeIndex = i;
}
```

[그림 2.10] Gas Limit을 이용한 DoS 취약점 예시

다. 피해사례

GovernMental의 잭팟 게임에서 과도한 gas 사용으로 인해 gas limit을 이용한 DoS 공격이 발생하였다. GovernMental은 일종의 Gambling 게임 서비스로서 유저에 대한 정보를 배열에 저장하는데, 유저가 많아지면서 배열이 커짐에 따라 배열을 반복하여 초기화하는 과정에서 gas limit을 넘는 문제가 발생하였다. 그림 2.11의 lendGovernmentMoney 함수는 creditorAddresses와 creditorAmounts의 배열을 지우는 과정을 수행하는데, 배열 내에 데이터가 어느 이상 존재하는 경우 배열을 지우는 과정에서 gas limit을 초과하게 된다.

```
...
address[] public creditorAddresses;
uint[] public creditorAmounts
function lendGovernmentMoney(address buddy) returns (bool){
uint amount = msg.value;
if (lastTimeOfNewCredit + TWELVE_HOURS < block.timestamp)
{
 msg.sender.send(amount);
 creditorAddresses[creditorAddresses.length-1].send
 (profitFromCrash);
 ...
 creditorAddresses = new address[](0);
 creditorAmounts = new uint[](0);
 round += 1;
 return false;
}
...
}
```

[그림 2.11] GovernMental의 Gas limit을 이용한 DoS 취약점 코드

5. 강제 잔액 변조

가. 개요 및 특징

- 스마트 계약의 잔액을 나타내는 this.balance 변수를 변조하여 발생하는 취약점

- 위험도: 높음(High)

- 현재의 잔액을 스마트 계약의 진행을 위한 핵심 정보로 사용하는 경우 취약 상황 발생

- selfdestruct를 이용하는 방법 : selfdestruct는 자신의 스마트 계약을 제거하면서 남아있는 이더를 대상 스마트 계약에 무조건 이동

- 설치 전 미리 전송 : 스마트 계약이 설치되기 전에 주소를 예측하여 이더를 미리 전송

스마트 계약의 잔액을 나타내는 this.balance 변수가 의도하지 않게 변경되는 취약점이다. 스마트 계약의 잔액을 변경하기 위해서는 스마트 계약 계정으로 이더를 송금하고 payable 속성을 갖는 함수가 이를 처리해 주어야 한다. 하지만 payable 속성의 함수를 거치지 않고 강제로 송금한 이더를 받게함으로써 this.balance 변수를 개발자 의도와는 다른 상태로 변경할 수 있다. 특히 this.balance 변수는 스마트 계약의 초기화를 확인하는 용도로 사용되었기 때문에 이러한 공격이 혼란을 가져오게 되었다. 강제로 이더를 전송하기 위해서는 selfdestruct 함수를 사용하거나 스마트 계약 설치 이전에 미리 이더를 전송하는 방법 등이 있다.

스마트 계약의 selfdestruct(address) 함수는 스마트 계약을 suicide(삭제) 처리하기 위해 구현되었다. 하지만 이 함수는 자신을 삭제하면서 인자로 넘겨준 주소로 자신의 이더를 강제 이동시킨다. 즉 해당 스마트 계약에서 payable 속성을 가진 폴백함수가 없더라도 송금을 할 수 있게 된다.

스마트 계약의 주소값은 설치되기 전이라도 그림 2.12와 같이 주소를 생성하는 공식을 통해 유추할 수 있다. 스마트 계약을 설치하는 주소 및 현재 정보 등을 이용하여 앞으로 생성될 스마트 계약의 주소를 계산하여 이더를 미리 전송하면 스마트 계약의 payable 확인 절차들을 무시하고 대상 주소의 잔액 정보를 변조할 수 있다.

```
sha3(rlp.encode([account_address,transaction_node])
```

[그림 2.12] 스마트 계약의 주소를 만들기 위한 기본 공식

나. 예제 코드

그림 2.13은 강제 잔액 변조 공격에 취약한 코드의 예제이다. play 함수 내에 현재 잔액(this.balance)를 기반으로 한 currentBalance를 동작의 조건으로 사용하고 있기 때문에, 외부에서 강제로 balance 값을 조정함으로써 특정 단계(milestone)의 달성 조건을 조작할 수 있다.

다. 피해사례

실제 서비스가 아닌 취약점 콘테스트(2017 Underhanded Contest)에서 처음으로 등장한 취약점이다. Underhanded Contest 등장 이후 실제 서비스 내에서의 해킹 사례는 없지만 끊임없이 위험에 대한 부분이 강조되고 있는 취약점이다. 그림 2.14의 코드 중 변경자 checkInvariant에서 this.balance를 진행 조건으로 활용하기 때문에 buyMagicBeans 함수에서 강제 잔액 변조 전송에 의해 DoS 취약점이 발생한다. 해당 취약점은 this.balance 변수가 payable 속성의 함수를 거치지 않고 변겨될 수 있다는 상황을 고려하지 않은 것이 원인이다.

```
uint public payoutMileStone1 = 3 ether;
uint public mileStone1Reward = 2 ether;
uint public payoutMileStone2 = 5 ether;
uint public mileStone2Reward = 3 ether;
mapping(address => uint) redeemableEther;
function play() public payable {
  require(msg.value == 0.5 ether);
  uint currentBalance = this.balance + msg.value;
  require(currentBalance <= finalMileStone);
  if(currentBalance == payoutMileStone1){
 redeemableEther[msg.sender] += mileStone1Reward;
  }
  else if(currentBalance == payoutMileStone2) {
 redeemableEther[msg.sender] += mileStone2Reward;
  }
  depotedWei += msg.value;
  return;
}
```

[그림 2.13] 강제 잔액 변조의 취약점 예시

```
contract ICO {
mapping(address => uint) magicBeans;
    uint totalSupply;
    address company;
    uint constant fundingGoal = 100000 ether;
    uint endOfFundingPeriod;
    modifier checkInvariant() {
        _;
        if (this.balance != totalSupply) throw;
    }
    function ICO(uint _endOfFundingPeriod ) {
        company = msg.sender;
        endOfFundingPeriod = _endOfFundingPeriod;
    }
    function() {throw;}
    function buyMagicBeans() payable checkInvariant() {
        if (now > endOfFundingPeriod) throw;
        magicBeans[msg.sender] += msg.value;
        totalSupply += msg.value;
    }
}
```

[그림 2.14] 강제 잔액 변조 취약점의 사례 코드

6. 초기화되지 않은 스토리지 포인터 (Uninitialized storage pointers)

가. 개요 및 특징

- Solidity에서 사용되는 메모리와 스토리지를 가리키는 포인터가 혼동되어 사용될 경우 발생하는 취약점

- 위험도: 낮음(Low)

- struct를 가리키는 포인터가 적절히 초기화되지 않은 경우에 스토리지 공간을 참조하는 문제

초기화되지 않은 스토리지 포인터 버그는 메모리와 스토리지에 저장된 변수를 참조하는 과정에서, 의도와는 다른 영역을 참조하게 되어 발생하는 취약점이다[23][29]. Solidity에서 변수는 메모리와 스토리지의 두 가지 형태로 저장될 수 있다. 메모리는 비영구적인 데이터 저장소이다. 일종의 RAM의 역할을 하는 것으로 특정 함수에서 정의된 변수나 연산 후 값을 일시적으로 저장하기 위해 사용된다. 스토리지는 영구적인 데이터 저장소이다[11]. 실제로 저장되는 곳은 블록체인 내 블록으로 저장된다.

주목할 점은 스토리지 공간은 전역 state 변수가 선언된 순서대로 할당이 된다는 것이다. 함수 내에서 정의되는 변수는 모두 메모리로 할당하지만 예외적으로 struct를 정의할 경우, 적절히 초기화 하지 않는다면 디폴트로 스토리지 포인터를 참조하는 문제가 생긴다. 일반적인 전역 변수 상태에서는 문제가 없지만, 함수 내에서 스토리지 포인터를 새로 참조할 경우 이전에 사용된 변수들의 값을 참조하는 포인터에 영향을 주는 문제가 생긴다[53].

나. 예제 코드

그림 2.17의 코드에서 NameRegister란 스마트 계약의 전역 state 변수가 정의된 순서대로 스토리지에 할당된다고 가정하면 그림 2.15와 같은 형태로 할당이 된다. 이러한 스토리지 구성에서 문제가 되는 부분은 register 함수 부분의 newRecord struct 변수가 선언된 곳이다. struct는 디폴트 상태에서 스토리지 포인터를 참조하게 된다. 해당하는 함수에서 newRecord는 storage 순서가 다시 초기화되므로 그림 2.16과 같은 스토리지 포인터 참조 구조가 된다. 그림 2.16 구조에서 NameRecord의 name 부분이 bytes 단위이므로 1의 값을 바이트 단위로 할당할 경우 실제 unlock 변수의 값이 false 값에서 true 값으로 변경 가능하다.

```
[0]:bool public unlocked = false;
[1]:mapping(address=>NameRecord) public registeredNameRecord;
[2]:mapping(bytes32=>address) public resolve;
```

[그림 2.15] 예시 코드의 전역 변수 스토리지 포인터의 구조

```
[0]: NameRecord -> name
[1]: NameRecord -> mappedAddress
```

[그림 2.16] 예시 코드의 register 함수의 변수 스토리지 포인터의 구조

```
contract NameRegister{
bool public unlocked = false;
struct NameRecord{
        bytes32 name;
        address mappedAddress;
}
mapping(address => NameRecord) public registeredNameRecord;
mapping(bytes32 => address) public resolve;
function register(bytes32 _name, address _mappedAddress) public
{
NameRecord newRecord;
newRecord.name = _name;
newRecord.mappedAddress = _mappedAddress;
resolve[_name] = _mappedAddress;
registeredNameRecord[msg.sender] = newRecord;
require(unlocked);
}
}
```

[그림 2.17] 초기화되지 않은 스토리지 포인터의 취약점 예시

다. 피해사례

OpenAddressLottery 허니팟에 사용된 초기화되지 않은 스토리지 포인터 취약점이다. 주로 공격 가능한 실제 서비스의 취약점이 아닌 공격자들에게 공격이 가능한 부분처럼 보이게 하여 공격자를 속이는 허니팟 기법에 쓰인다. 해당 스마트 계약의 컨트랙트 주소는 0x741F1923974464eFd0Aa70e77800BA5d9ed18902이다. 그림 2.18 내

초기화되지 않은 구조체 SeedComponents를 forceReseed()에 정의하면서 luckyNumber를 s.component4 = tx.gasprice * 7 문을 통하여 변수 값을 덮어쓰는 취약점이 발생하였다.

```
contract OpenAddressLottery{
    struct SeedComponents{
        uint component1;
...
        uint component4;
    }
    address owner;
    uint private secretSeed;
    uint private lastReseed;
    uint LuckyNumber = 7;
    mapping (address => bool) winner;
    function forceReseed() {
        require(msg.sender==owner);
        SeedComponents s;
        s.component1 = uint(msg.sender);
        s.component2 =
        uint256(block.blockhash(block.number - 1));
        s.component3 = block.difficulty*(uint)(block.coinbase);
        s.component4 = tx.gasprice * 7;
        reseed(s);
    }
}
```

[그림 2.18] 초기화되지 않은 스토리지 포인터의 취약점 사례 코드

7. Return값 미확인 (Unchecked call return value)

가. 개요 및 특징

- Solidity 내의 함수 call()과 send()는 트랜잭션 전송에 실패하는 예외 상황에서 상태를 되돌리는 실행 중지(revert) 작업을 하지 않고 false를 리턴

- 위험도: 중간(Medium)

- call과 send의 결과값을 적절히 확인하지 않는 경우에는, 예외에 대한 처리가 되지 않아 악용될 수 있음

Solidity 내의 함수 call과 send를 사용할 경우 발생하는 취약점이다. 일반적으로 transfer 함수는 사용할 경우 전송 과정에서 문제가 생긴다면 실행을 중지(revert)하고 해당하는 상태를 함수가 실행되기 전으로 돌려놓는다[23][53]. 하지만 call과 send의 경우 전송에 에러가 생기면 false 값을 리턴하고 별도의 revert 처리를 하지 않는다. 따라서 이러한 예외에 경우에 대해서 개발자가 직접 별도의 리턴값을 확인하는 코드를 추가하여 처리해야 한다. 하지만 각 함수의 특성을 고려하지 않고 실수로 예외 처리가 누락되는 경우에 이를 악용한 공격이 가능하다[32][53].

나. 예제 코드

그림 2.19는 Return값 미확인 취약점에 대한 예제 코드이다. 코드의 withdrawLeftOver() 함수의 msg.sender.send(this.balance) 형태의 호출 구문에서 send 함수의 결과 값을 확인하지 않기 때문에 실제 전송이 되지 않은 경우에도 문제를 확인하지 못하고 프로그램은 정상 종료된다. 특히 이러한 경우 내부적인 정보와 실제 잔액간의 불일치가 발생하여 프로그램의 오동작이 유발될 수 있다.

```
contract Lotto {
        bool public payedOut = false;
        address public winner;
        uint public winAmount;
        function sendToWinner() public {
                require(!payedOut);
                winner.send(winAmount);
                payedOut =true;
        }
        function withdrawLeftOver() public {
                require(payedOut);
                msg.sender.send(this.balance);
        }
}
```

[그림 2.19] Return값 미확인 취약점 예시

다. 피해사례

Etherpot(contract lottery) 서비스에서 Return값 미확인 취약점이 발생하였다. 일종의 Lottery 종류의 서비스이며, Cash Out을 하는 과정에서 send 함수에 대한 Return값에 대한 확인 절차 부재가 원인이다. 그림 2.20 내 cash 함수에서 winner.send(subpot)의 send 함수가 false 상태일 경우, winner는 다시 보상을 받을 수 없는 고착 상황에 빠지게 된다.

```
...
   function cash(uint roundIndex, uint subpotIndex){
            var subpotsCount = getSubpotsCount(roundIndex);
            if(subpotIndex>=subpotsCount)
                 return;
            var decisionBlockNumber =
             getDecisionBlockNumber(roundIndex,subpotIndex);
            if(decisionBlockNumber>block.number)
                 return;
            if(rounds[roundIndex].isCashed[subpotIndex])
                 return;
            var winner = calculateWinner(roundIndex,subpotIndex);
            var subpot = getSubpot(roundIndex);
            winner.send(subpot);
            rounds[roundIndex].isCashed[subpotIndex] = true;
}
...
```

[그림 2.20] Etherpot에서의 return값 미확인 취약점 사례 코드

8. 잘못된 랜덤 생성 (Generating randomness)

가. 개요 및 특징

- 블록체인은 참여자가 같은 정보를 동일하게 유지해야하는 특성상 자체적으로 예측 불가능한 random 값을 만들기 어려움

- 위험도: 중간(Medium)

- 블록의 타임스탬프 등을 활용하는 기존의 블록체인 서비스에서 취약점 발생

- 공개된 블록체인 정보를 바탕으로 공격자가 랜덤 값을 유추하여 원하는 결과가 나오도록 공격

블록체인은 특성상 랜덤한 값을 생성하기가 어렵다. 다음 값이 예측 불가능하면서도 참여하는 모든 노드가 확인할 수 있는 결정론적 랜덤 값을 만들어야하기 때문이다[36][53]. 이러한 한계에 대응하여 최근 생성된 블록의 타임스탬프나 블록 해시 정보와 같이 미리 예측하기 힘든 값을 이용해서 랜덤 값을 만드는 시도들이 있지만, 블록체인 내의 모든 트랜잭션 및 블록 정보는 공개되기 때문에, 공개되는 시점에서 빠르게 연산하여 랜덤값을 먼저 예측하는 공격이 가능하다. 특히 대부분 블록체인 게임에서 블록 타임스탬프를 활용하고 있는데, 이는 어느 정도 예측이 가능하기에 문제가 주로 발생한다. 이러한 해결을 위하여 Oraclize와 같은 외부 Seed 값 활용을 통해 Random Number를 생성하는 방법들이 적용되고 있다[23][53].

나. 예제 코드

그림 2.21의 예제 코드에서는 play란 함수에서 랜덤 값을 만들기 위해서 블록의 타임스탬프를 사용한다. 연산 과정에서 uint(sha3(block.timestamp)) % 2와 같은 수식을 통하여 블록의 타임스탬프에 기반을 둔 연산을 수행하지만, sha3도 결정적인(deterministic)한 함수이므로 블록의 타임스탬프만 알면 바로 현재 random 값에 들어있는 정보를 유추할 수 있다. 따라서 공격자는 새로운 블록이 나오기 전에 random값을 계산하여 유추하고, 스마트 계약이 원하는 상태(random == 0)에 있는 경우에는 트랜잭션을 보내서 상금을 획득할 수 있다.

```
function play() payable {
        assert(msg.value == TICKET_AMOUNT);
        pot += msg.value;
        var random = uint(sha3(block.timestamp)) % 2;
        if(random == 0) {
                bank.transfer(FEE_AMOUNT);
                msg.sender.transfer(pot - FEE_AMOUNT);
                pot = 0;
        }
}
```

[그림 2.21] 잘못된 랜덤 생성의 기본 취약점 예시

다. 피해 사례

블록체인 RPG 게임인 Cryptosaga에서 발생된 잘못된 랜덤 생성 취약점이다. 그림 2.22 내의 random 함수에서 block.blockhash를 활용하여 랜덤값을 생성하는 과정에서 Random Number를 예측할 수 있다. 특정 능력을 가진 캐릭터 생성 시에 랜덤 값을 사용하므로 블록 해시를 기반으로 생성하는 Random Numebr를 예측하여 원하는 조건을 가진 캐릭터를 생성하기 위한 시점을 확인할 수 있다.

```
for(uint i=0; i<_amount; i++){
        var _randomValue = random(10000 , 0);
        uint8 _heroRankToMint = 0;
if(_randomValue < 5000 ) {
        _heroRankToMint = 1;
} else if(_randomValue < 9550 ) {
        _heroRankToMint = 2;
} else if(_randomValue < 9950 ) {
        _heroRankToMint = 3;
} else {
        _heroRankToMint = 4;
}
}
function random(uint32 _upper, uint32 _lower) private
returns(uint32) {
require( _upper > _lower );
seed =
uint32(keccak256(keccak256(block.blockhash(block.number,seed),
now)
return seed % (_upper - _lower) + _lower;
}
```

[그림 2.22] Cryptosaga의 잘못된 랜덤 생성 실제 사례 코드

9. 잘못된 접근 제어(Broken Access Control)

가. 소개

- 특정 함수를 사용하여 권한을 확인하는 접근 제어 코드에서의 실수로 발생하는 취약점

- 위험도: 높음(High)

- 일반적으로 접근제어에 사용하는 변경자(modifier)나 require문 등에서 발생

함수를 사용하기 위한 접근 제어 코드에서 실수로 인하여 발생하는 취약점이다[23][32]. 주로 권한을 가진 소유자를 확인하는 권한 관리 코드에서 발생한다. 로직의 실수에 해당되기 때문에 스마트 계약의 언어적 특성과는 상관없이 어디서나 적용된다. 일반적으로 접근 제어에 사용되는 변경자(modifier), require 문 등에서 발생한다[53].

나. 예제 코드

그림 2.23은 대표적인 접근 제어 과정에서의 취약점 예시로 require의 용도에 대한 혼동에서 발생한다. 예제 코드의 onlyOwner에 변경자에 있는 require문은 트랜잭션의 송신자(msg.sender)가 저장되어 있는 소유자(owner)의 주소와 같은지를 확인하려는 용도였으나, 결과적으로는 조건을 반대로 설정하여 접근 제어에 실패하는 경우를 보여주고 있다. 올바른 조건은 msg.sender != owner이 아닌 msg.sender == owner이다.

```
modifier onlyOwner() {
        require(msg.sender != owner); {}
        _;
}
function Coinlancer() public onlyOwner {
        owner = msg.sender;
        balance[owner] = _totalSupply;
}
```

[그림 2.23] 잘못된 접근 제어의 취약점 예시

다. 피해 사례

Parity Multisig Wallet의 지갑 동결 사건이다. Parity 사에서 제공하는 Multisig Wallet의 Ownership에 대한 접근 제어의 부주의한 실수로 인해 51만 ether가 동결되었다. 핵심 원인은 Ownership에 대한 초기화 관리 부재이다. 그림 2.24 내 only_unitialized의 조건을 통해 initWallet이 초기화 되지 않을 때만 사용할 수 있도록 설계하였지만, 설치와 동시에 초기화를 시키지 않은 실수로 인해 공격자가 initWallet 호출 이후 지갑을 동결 시키게 되었다.

```
modifier only_uninitialized {
        if(m_numOwners > 0) throw; _;
}
function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
        initDaylimit(_daylimit);
        initMultiowned(_owners, _required);
}
```

[그림 2.24] Parity Multisig Wallet의 잘못된 접근 제어 취약점 예시

10. Call method injection 공격 (Evilreflex)

가. 개요 및 특징

- call 함수에 bytes 타입의 입력을 전달할 수 있는 경우 발생하는 취약점
- 위험도: 높음(High)
- call 함수에 전달되는 값을 calldata 구성방식과 동일하게 구성하여 임의의 다른 함수를 호출 가능

Peckshield 사에서 최초로 발견한 취약점으로 EvilReflex 취약점으로도 불린다[38][53]. call 함수에 bytes 타입의 인자를 받아서 사용하는 패턴이 있는 경우, 입력을 트랜잭션의 calldata의 형태로 재구성하여 원하는 함수를 대신 호출할 수 있도록 하는 공격이다. 그림 2.25와 같이 call 함수에 임의로 생성한 함수서명과 매개변수를 모두 전달하여 함수서명에 해당하는 함수를 호출할 수 있게 된다. 공격 조건이 만족하는 경우에는 call 함수에 공격 코드를 주입하는 방식이기 때문에, 스마트 계약의 제어권을 장악하는 강력한 공격이 가능하다[23][40].

```
call(bytes4(keccak256("해당함수(매개변수 형식)")),매개변수값)
```

[그림 2.25] Call 함수를 이용한 코드 주입 방법

나. 예제 코드

그림 2.26은 call method injection 취약점을 가진 예제 코드이다. 코드의 approveAndCallcode 함수 내 _spender.call(_extraData)문은 bytes 타입인 외부 매개변수 _extraData를 call에 직접 전달하여 사용한다. 이때 변수 _extraData를 통한 call Injection으로 transfer와 같은 전송 함수를 호출할 경우 스마트 계약이 보유하고 있는 토큰을 탈취할 수 있다.

```
  function approveAndCallcode(address _spender, uint256 _value,
bytes _extraData) returns (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        if(!_spender.call(_extraData)){ revert(); }
        return true;
  }
```

[그림 2.26] Call method injection의 취약점 사례 예시

다. 피해사례

GVE(Globalvillage Ecosystem) Coin(그림 2.27)의 Access Control 취약점으로 인한 후오비 거래소 지급 정지 사례이다. 피해 당시 PeckShield 사에서 최초 발견 이후 후오비 및 해당 업체에 공지하였다. 그림 2.26의 예제 코드 내 _spender.call 함수 내의 _extraData 매개변수로 인해 취약점이다.

[그림 2.27] Globalvillage ecosystem(GVE) 정보