# 이더리움과 스마트 컨트랙트 보안 이슈 및 해결방안

**이화체인 베이직 스터디**

**1985084 임수지**

## 1. 서론: 스마트 컨트랙트 보안의 중요성

스마트 컨트랙트는 블록체인 위에서 자동으로 실행되는 프로그램으로, 탈중앙화된 환경에서 신뢰 없이 계약을 이행할 수 있게 한다. 특히 이더리움 네트워크에서는 수많은 디앱(DApp)이 스마트 컨트랙트를 기반으로 운영되며, 이들 중 상당수는 자산을 직접 다루기 때문에, 보안 취약점은 곧 막대한 금전적 피해로 직결된다. Luu et al. (2016)의 "Making Smart Contracts Smarter"에서는 약 8859개의 이더리움 컨트랙트를 분석한 결과, 재진입(Reentrancy), 시간 의존성(Time Dependency), 예외 취약성(Unhandled Exception) 등 다양한 보안 문제들이 발견되었음을 보여준다. 이는 스마트 컨트랙트가 가진 구조적 특성상 보안 결함에 특히 민감함을 시사한다.

2016년 발생한 DAO 해킹 사건은 그 대표적인 사례다. 스마트 컨트랙트의 코드상 취약점을 악용한 공격자는 당시 약 6000만 달러에 해당하는 이더를 탈취하는 데 성공했으며, 이 사건은 이더리움 역사상 가장 큰 위기를 초래했다. 해당 사건은 뉴욕타임즈 기사(2016)에서도 상세히 다루어진 바 있으며, 해킹 발생 이후 커뮤니티 내 격렬한 논의 끝에 이더리움은 하드포크를 선택하게 된다. 이처럼 스마트 컨트랙트의 보안은 단순한 개발 문제를 넘어서 블록체인 생태계 전체의 신뢰와 직결된다.

## 2. 사례 분석: DAO 해킹과 재진입 공격 (Reentrancy Attack)

DAO(Decentralized Autonomous Organization)는 탈중앙화된 자율조직으로, 투자자들이 스마트 컨트랙트를 통해 자금을 모으고, 제안에 투표하여 투자 결정을 내릴 수 있도록 설계되었다. 당시 DAO는 이더리움 전체 유통량의 약 14%에 해당하는 자금을 유치하며 큰 주목을 받았다. 그러나 DAO의 스마트 컨트랙트에는 구조적인 결함이 존재했고, 이는 결국 대규모 해킹으로 이어졌다.

DAO 해킹의 핵심은 재진입 공격(Reentrancy Attack) 이라는 보안 취약점이다. 이는 스마트 컨트랙트에서 외부 컨트랙트에 이더를 전송한 뒤 상태값을 변경하는 방식으로 코드를 작성할 경우, 외부 호출 중에 공격자가 다시 같은 함수(예: withdraw())를 호출할 수 있게 되어, 상태값이 업데이트되기 전에 반복적으로 자금을 출금하는 공격이 가능하다는 점에 착안한 것이다.

당시 DAO의 withdraw() 함수는 다음과 같은 구조를 갖고 있었다:

```
/// @title VulnerableBank - 재진입 공격에 취약한 예제 컨트랙트
contract VulnerableBank {
    mapping(address => uint256) public balances;

    /// @notice 입금 함수. 사용자 잔액을 기록
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    /// @notice 출금 함수. 재진입 공격에 취약함
    function withdraw() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");

        // ⚠️ 문제점: 외부 호출을 먼저 실행한 후, 상태값을 나중에 변경
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");

        // 공격자는 이 전에 다시 withdraw()를 호출 가능
        balances[msg.sender] = 0;
    }

    /// @notice 수신 전용 함수 (fallback 아님)
    receive() external payable {}
}
```

위 코드의 문제는, msg.sender.call.value() 호출이 공격자의 컨트랙트로 제어권을 넘긴다는 데 있다. 공격자가 자신의 컨트랙트에 fallback() 또는 receive() 함수를 구현해두고, 이

함수에서 다시 withdraw()를 호출하면, 잔액이 아직 차감되지 않은 상태에서 출금이 다시 발생하게 된다. 이 과정을 재귀적으로 반복할 수 있기 때문에, 공격자는 자신의 잔액보다 훨씬 많은 금액을 인출할 수 있었다.

이 공격으로 인해 DAO는 전체 보유 자금의 약 1/3에 해당하는 360만 ETH를 탈취당했고, 이더리움 커뮤니티는 이를 복구하기 위해 하드포크를 단행하였다. 이 사건은 이후 스마트 컨트랙트 보안의 중요성을 대두시키는 계기가 되었다.

해당 공격을 재현한 실습 예제는 아래와 같은 구성으로 이루어진다.

- 피해 컨트랙트: VulnerableBank.sol
- 공격 컨트랙트: Attacker.sol
- 실행 프레임워크: Hardhat
- 핵심 메커니즘: 잔액을 나중에 차감하는 구조 + 외부 호출 중 함수 재호출

재진입 공격을 방지하기 위한 핵심은, **외부 호출보다 먼저 상태값을 변경**하거나, **중복 호출 자체를 차단**하는 방식이다. 이 두 가지 방식은 다음과 같은 코드로 구현할 수 있다.

방법 1. 상태값을 먼저 변경 (Checks-Effects-Interactions 패턴)

```solidity
/// @title SecureBank - 재진입 공격을 방어하는 보안 컨트랙트
contract SecureBank is ReentrancyGuard {
    mapping(address => uint256) public balances;

    /// @notice 입금 함수. 사용자 잔액 증가
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    /// @notice 안전한 출금 함수. 재진입 공격 방지
    /// @dev nonReentrant modifier로 중첩 호출 차단
    function withdraw() public nonReentrant {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");

        // 상태 먼저 변경 (Checks-Effects-Interactions)
        balances[msg.sender] = 0;

        // 이후 외부 호출
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }

    receive() external payable {}
}
```

이 방식은 가장 기본적인 보안 패턴으로, 상태값을 먼저 변경함으로써 재진입 호출이 발생해도 더 이상 출금이 이루어지지 않는다.

방법 2. ReentrancyGuard 사용 (OpenZeppelin 라이브러리)

```
describe("🛡️ SecureBank – 재진입 공격 방어 테스트", function () {
  let deployer, victim, attacker;
  let secureBank, attackerContract;

  beforeEach(async () => {
    [deployer, victim, attacker] = await ethers.getSigners();

    const SecureBank = await ethers.getContractFactory("SecureBank", deployer);
    secureBank = await SecureBank.deploy();
    await secureBank.waitForDeployment();

    const Attacker = await ethers.getContractFactory("Attacker", attacker);
    attackerContract = await Attacker.deploy(await secureBank.getAddress());
    await attackerContract.waitForDeployment();

    await secureBank.connect(victim).deposit({ value: ethers.parseEther("10") });
  });

  it("공격자가 공격을 시도해도 트랜잭션이 revert되어야 한다", async () => {
    await expect(
      attackerContract.connect(attacker).attack({ value: ethers.parseEther("1")
    ).to.be.reverted;
  });
});
```

nonReentrant modifier를 붙이면, 해당 함수가 실행 중일 때 동일 함수가 중첩 호출되는 것을 차단한다. 공격자 컨트랙트에서 재귀적으로 withdraw()를 호출하려 하면, 런타임에서 자동으로 revert된다.

요약하면, 공격자가 SecureBank 컨트랙트를 대상으로 재진입 공격을 시도했을 때 트랜잭션은 실행 도중 reverted 되며 중단되었다. 이로 인해 스마트 컨트랙트의 자산에는 아무런 변화가 없었고, 입금된 10 ETH는 고스란히 남아 있었다. 실험 코드를 통해 이러한 보안 조치가 실제로 효과를 발휘함을 확인할 수 있었으며, 단순한 코드 수정만으로도 재진입 공격을 완전히 차단할 수 있음을 입증하였다.

## 4. 결론 및 실무 적용 방안

DAO 해킹 사례는 스마트 컨트랙트의 작은 실수 하나가 치명적인 피해로 이어질 수 있음을 보여준다. 특히 재진입 공격은 초보 개발자가 실수하기 쉬운 구조에서 자주 발생하기 때문에, 코딩 시 다음의 원칙을 반드시 지켜야 한다.

- 상태값 갱신은 항상 외부 호출보다 먼저 수행할 것
- OpenZeppelin의 보안 라이브러리를 적극 활용할 것
- 배포 전에는 Slither, Mythril 등 보안 감사 툴로 정적 분석을 수행할 것

본 실습을 통해 단순히 기능을 구현하는 것을 넘어, 보안까지 고려한 스마트 컨트랙트 개발이 얼마나 중요한지 체감할 수 있었다. 특히 작은 코드 수정만으로도 대규모 해킹을 막을 수 있다는 점에서, 스마트 컨트랙트 보안은 단순한 옵션이 아닌, 블록체인 개발의 핵심 요소임을 분명히 깨달을 수 있었다.