




이더리움과 스마트 컨트랙트 보안: DAO 해킹과 재진입 공격 실습



목차 table of contents

- 1 스마트 컨트랙트 보안의 중요성
 - 2 재진입 공격 실습: 취약 컨트랙트 분석
 - 3 보안 적용 실습: 방어 코드와 실패 테스트
 - 4 실무 보안 적용
-



III
ADVENTURES...
AND LESSONS
LEARNED

Part 1

스마트 컨트랙트 보안의 중요성

스마트 컨트랙트란?

스마트 컨트랙트 보안의 중요성



왜 보 안 이 중 요 한 가 ?

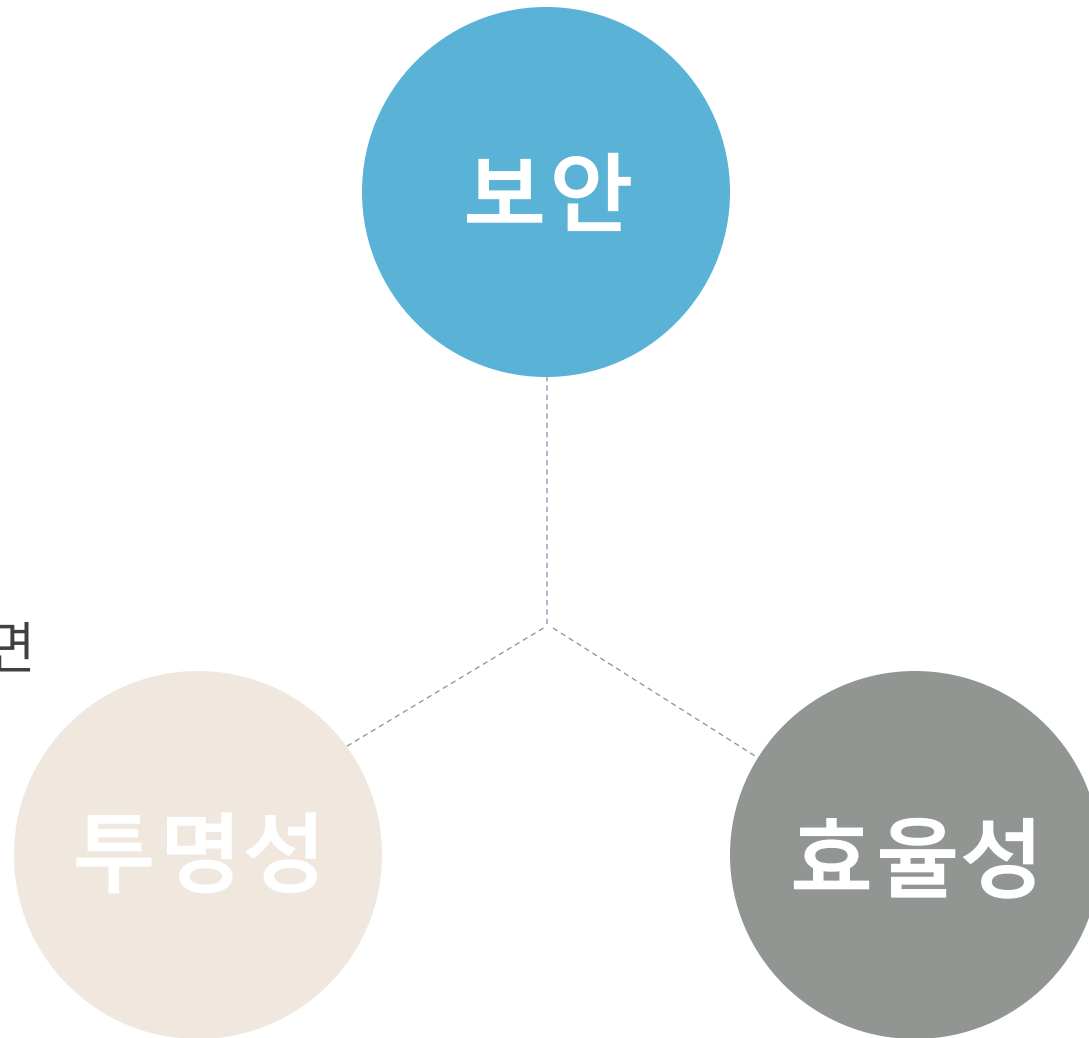
스마트 컨트랙트 보안의 중요성

모든 노드가 공유함

= 조작하려면 모든 노드의 데이터를 조작해야 함

= 조작하기 어려움

이행 검증 시 문제점이 있으면
계약 취소됨



제3자가 필요 없음

왜 보 안 이 중 요 한 가 ?

스마트 컨트랙트 보안의 중요성

모든 노드가 공유함

= 조작하려면 모든 노드의 데이터를 조작해야 함

= 조작하기 어려움

보안

한번 배포되면

수정도 불가, 삭제도 불가

= 코드에 작은 실수 하나만 있어도
수십억 원의 피해로 이어질 수 있음

이행 검증 시 문제점이 있으면
계약 취소됨

투명성

효율성

제3자가 필요 없음

The DAO 해킹 사건

개발자들이 이더리움 위에서 스마트 컨트랙트를 작성하고, 조직을 만듦.

이 조직의 경영에 참여할 수 있는 권한인 토큰(DAO token)을 발행해서 판매하고, 조직 자금 마련함.
투자자 입장에서 이더를 주고 토큰을 받는 방식.

여기서 사용할 수 있는 기능 중 하나가 이더를 환불받는 기능.

해커는 환불신청(split)해서 이더를 먼저 받고, 토큰을 주기 전에 이더를 다시 환불받는,
무제한 환불 공격으로 해킹.

당시 가치로 약 750억 원 해킹됨.



Part 2

재진입 공격 실습: 취약 컨트랙트 분석

어떻게 공격이 가능했는가?

재진입 공격 실습: 취약 컨트랙트 분석

```
/// @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 {}
}
```

출금을 한 후에
남은 금액 balance를 0으로 조정

```
describe("★ Reentrancy Attack Test", function () {
  let deployer, victim, attacker;
  let vulnerableBank, attackerContract;

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

    const VulnerableBank = await ethers.getContractFactory("VulnerableBank", deployer);
    vulnerableBank = await VulnerableBank.deploy();
    await vulnerableBank.waitForDeployment();

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

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

    it("🐛 공격자가 재진입 공격으로 자금을 탈취할 수 있어야 한다", async () => {
      await attackerContract.connect(attacker).attack({ value: ethers.parseEther("1") });

      const balance = await ethers.provider.getBalance(await attackerContract.getAddress());
      console.log("공격자 컨트랙트 잔액:", ethers.formatEther(balance), "ETH");

      expect(balance).to.be.gt(ethers.parseEther("10"));
    });
  });
});
```

(다른 사람이 넣은 금액 10)

1만 입금하고
10을 출금할 수 있음



Part 3

보안 적용 실습: 방어 코드와 실패 테스트



어떤 방식으로 방어했는가?

보안 적용 실습: 방어 코드와 실패 테스트

0으로 바꾼 후 출금하도록 변경

```
/// @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 {}
}
```

```
/// @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 {}
}
```

공격해도 출금 트랜잭션 거절됨

```
describe("✱ Reentrancy Attack Test", function () {
  let deployer, victim, attacker;
  let vulnerableBank, attackerContract;

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

    const VulnerableBank = await ethers.getContractFactory("VulnerableBank", deployer);
    vulnerableBank = await VulnerableBank.deploy();
    await vulnerableBank.waitForDeployment();

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

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

    it("공격자가 재진입 공격으로 자금을 탈취할 수 있어야 한다", async () => {
      await attackerContract.connect(attacker).attack({ value: ethers.parseEther("1") });

      const balance = await ethers.provider.getBalance(await attackerContract.getAddress());
      console.log("공격자 컨트랙트 잔액:", ethers.formatEther(balance), "ETH");

      expect(balance).to.be.gt(ethers.parseEther("10"));
    });
  });
});
```

```
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;
    });
  });
});
```

기본적인 거래가 잘 되는 것을 확인할 수 있음

```
describe("✅ SecureBank - 정상 입출금 테스트", function () {  
  let deployer, user;  
  let secureBank;  
  
  beforeEach(async () => {  
    [deployer, user] = await ethers.getSigners();  
  
    const SecureBank = await ethers.getContractFactory("SecureBank", deployer);  
    secureBank = await SecureBank.deploy();  
    await secureBank.waitForDeployment();  
  });  
  
  it("사용자가 입금 후 출금하면 잔액이 0이 되어야 한다", async () => {  
    const depositAmount = ethers.parseEther("1");  
  
    await secureBank.connect(user).deposit({ value: depositAmount });  
  
    let contractBalance = await ethers.provider.getBalance(await secureBank.getAddress());  
    expect(contractBalance).to.equal(depositAmount);  
  
    await secureBank.connect(user).withdraw();  
  
    contractBalance = await ethers.provider.getBalance(await secureBank.getAddress());  
    expect(contractBalance).to.equal(0);  
  });  
});
```

Balance를 0으로
먼저 변경



Part 4

실무 보안 적용



1

상태 먼저 변경

```
// 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");
```

2

외부 호출은 나중에

3

배포 전에는 보안 도구로 체크

모든 노드가 공유함
= 조작하려면 모든 노드의 데이터를 조작해야 함
= 조작하기 어려움

보안

한번 배포되면
수정도 불가, 삭제도 불가
= 코드에 작은 실수 하나만 있어도
수십억 원의 피해로 이어질 수 있음

Q&A



감사합니다

모든 내용은 아래 깃헙에 있습니다.

<https://github.com/errorinusermane/dao-rekt-security>