

Be as proud of Sogang as Sogang is proud of you

블록체인 스마트컨트랙트 취약점



서강대학교
SOGANG UNIVERSITY

- 스마트컨트랙트 취약점
- 보안을 고려한 스마트컨트랙트 패턴

- 스마트 컨트랙트(Smart Contracts)
 - 컴퓨터 프로그램으로 작성된 디지털 계약서
 - 블록체인에 의해 신뢰성 보장(거래 중개자 불필요)
 - 튜링 완전한 EVM 바이트코드 프로그램
 - 솔리디티 프로그래밍 언어로 작성



■ 솔리디티

- 이더리움 블록체인에서 실행되는 스마트 계약을 작성하기 위해 특별히 설계됨.
- C++, Python 및 JavaScript와 유사한 문법
- 이더리움가상머신(Ethereum Virtual Machine,EVM)에서 실행 가능한 이더리움 바이트코드로 컴파일됨
- **계정 및 주소:** 스마트 계약과 외부 계정 간의 상호작용을 위한 계정 및 주소 개념을 포함
 - 이더 보유
 - 주소를 통해 이더를 다른 계정으로 전송
- **매핑:** 키-값 쌍을 저장하기 위한 매핑(mapping) 자료형 제공
- **이벤트:** 블록체인에 기록되며 프론트엔드 애플리케이션이 이를 수신할 수 있도록 하는 이벤트 기능 제공

■ 스마트 컨트랙트의 한계와 취약점

■ 컨트랙트 코드 변경이 어렵다

- 수정 및 삭제가 불가능한 블록체인의 특징
- 바이트코드와 ABI 형태로 컴파일 된 후 블록체인 상에 올라가면 그 코드를 수정할 수 없다 → 스마트 컨트랙트 코드에 문제가 있거나, 기존 기능을 업그레이드해야 할 때는 수정된 코드를 반영한 새로운 스마트 컨트랙트를 배포해서 사용

■ 보안이 취약

- 코드를 공개할 경우, 공격자가 코드의 허점을 파악할 수 있기 때문에 컨트랙트가 공격당할 수도 있다

■ 스마트컨트랙트 보안 취약 사례

- DAO 공격: 약 6천만달러 손실
- Parity 해킹: 약 3천만 달러 손실, 약 2억 8천만 달러 동결

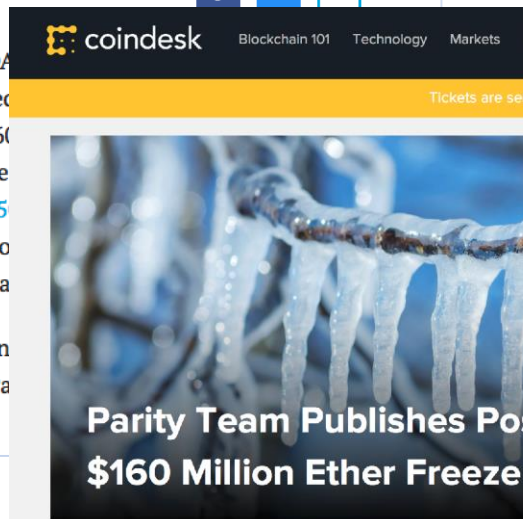
BLOCKCHAIN

Blockchain-based Venture Capital Fund Hacked for \$60 Million

David Z. Morris
Jun 18, 2016

News emerged Friday that The DAO, a venture capital fund operating through a decentralized autonomous organization, had been robbed of more than \$60 million in Ethereum currency, or about 1/3 of its value. The DAO, which raised more than \$150 million in Ethereum, intended as a showcase for the potential of the blockchain platform for cloud-based financial services.

The nature of the hack was outlined by a tweet from the attacker, posted to Parity's GitHub page:



Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:

<https://paritytech.io/blog/security-alert.html>

"This means that currently no funds can be moved out of the [ANY Parity] multisig wallets"

A lot of people/companies/ICOs are using Parity-generated multisig wallets. About \$300M is frozen and (probably) lost forever.

Disclaimer: I lost little money (about \$1000) but my friends lost about \$300K.

■ The DAO

- DAO(Decentralized Autonomous Organization): 블록체인 기반의 협동 조직
 - 탈중앙화된 네트워크에 참여하는 익명의 참여자들의 투표에 의해 조직에 필요한 의사결정을 하는 새로운 형태의 협동 조직
- The DAO
 - 최초의 이더리움 기반 투자 조직
 - 2016년 4월 30일 이더리움의 이더를 DAO 토큰으로 교환하는 방식으로 모금을 진행했으며 그 결과 약 1억 5,000만 달러(한화 약 1조 7천억 원)를 조달

■ The DAO 해킹 사건

- 공격자는 The DAO로부터 30만 이더리움을 인출
- 모든 코드 깃허브(소스코드 공유 서비스)에 공유 → 코드를 보고 허점을 파악
- The DAO 스마트 컨트랙트의 `withdraw()` 함수를 재귀적으로 호출해, 스마트 컨트랙트에 사용자의 잔액을 업데이트하기 전에 재귀적으로 이더를 반환하도록 함 → 재진입 공격(Reentrancy Attack)
- 이더리움 네트워크 자체의 문제로 인해 발생한 것이 아니라, 사람이 작성한 스마트 컨트랙트 코드의 허점 때문에 발생
- 공격 지점이 없는 견고한 스마트 컨트랙트를 작성하는 것은 블록체인 네트워크 자체의 보안만큼 중요

■ DAO 공격

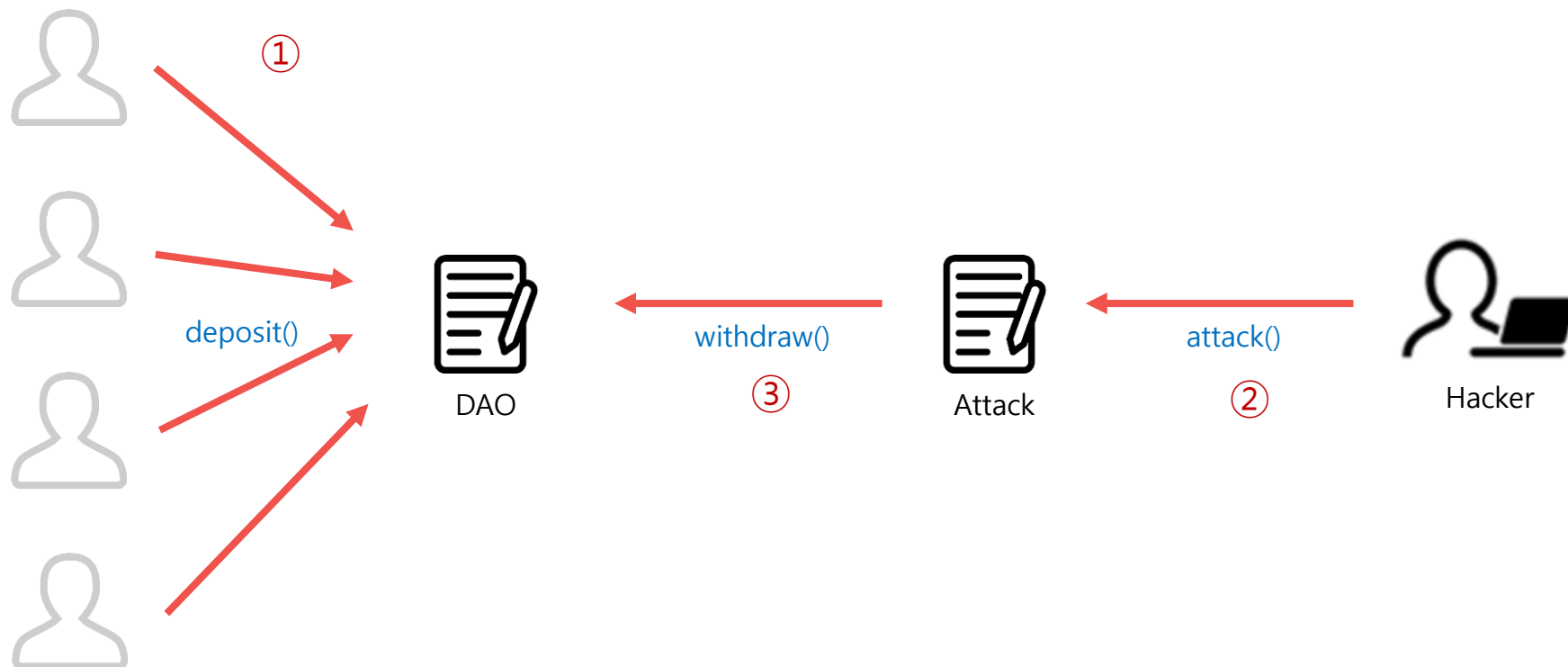
```
contract SimpleDAO {  
  
    mapping(address => uint) public balances;  
  
    function deposit() public payable {  
        balances[msg.sender] += msg.value;  
    }  
    function withdraw() public {  
        uint bal = balances[msg.sender];  
        require(bal > 0);  
  
        (bool sent, ) = msg.sender.call{value: bal}("");  
        require(sent, "Failed to send Ether");  
  
        balances[msg.sender] = 0;  
    }  
    function getBalance() public view returns (uint) {  
        return address(this).balance;  
    }  
}
```

```
contract Attack {  
  
    SimpleDAO public dao;  
    uint256 constant public AMOUNT = 1 ether;  
  
    constructor(address _daoAddress) {  
        dao = SimpleDAO(_daoAddress);  
    }  
    receive() external payable {  
        if (address(dao).balance >= AMOUNT) {  
            dao.withdraw();  
        }  
    }  
    function attack() external {  
        dao.withdraw();  
    }  
    function deposit() external payable {  
        require(msg.value >= AMOUNT);  
        dao.deposit{value: AMOUNT}();  
    }  
}
```

Reentrancy
vulnerability



■ DAO 공격



■ DAO 공격 순서

- SimpleDAO 컨트랙트 소유자, 기부자1, 기부자2, ... , 공격자(Attack소유자) 설정

1. SimpleDAO 컨트랙트 소유자: SimpleDAO 컨트랙트 배포
2. 공격자(Attack소유자): 배포한 SimpleDAO 컨트랙트 주소를 인자로 Attack컨트랙트 배포
3. 각 기부자들: 기부.... 즉 SimpleDAO 컨트랙트의 deposit() 호출
4. 공격자(Attack소유자): Attack 주소로 기부(즉, Attack의 deposit 함수 호출)
5. 공격자(Attack소유자): Attack의 attack() 함수를 호출
 - dao의 withdraw를 호출
 - dao의 withdraw에서 Attack에 송금하면서 receive함수 호출
 - dao 의 withdraw를 호출

...

■ 재진입성 취약점

- 사용자가 컨트랙트에 송금한 이더를 사용자별로 관리하고, 사용자는 자신의 잔액을 전액 인출하는 처리 순서가 다음과 같을 때

1. 잔액 확인
2. 잔액 전액 인출(호출한 쪽으로 송금)
3. 잔액을 0으로 설정

→ 2 에서 재진입성 문제가 발생할 가능성 존재

- 송금 받을 대상(사용자)가 컨트랙트인 경우 다시 인출 함수를 호출하는 방법으로 아직 0이 되지 않은 잔액을 재차 송금 받을 수 있기 때문


■ 경매 – King of the Ether Throne

- 왕좌를 갖기 위해 금액을 송금했을 때 현재 왕좌금액보다 크면, 이전 소유자에게 자신이 송금했던 금액(현재 왕좌금액)을 약간의 수수료를 제외하고 돌려주고 새로운 소유자가 되게 하는 프로그램
- send가 정상적으로 실행되지 않았을 경우 이전 소유자는 자신의 투자금도 받지 못하고 왕좌도 잃게 됨
 - send는 해당 금액을 송금하는 것으로 call 처럼 처리하는데 gas가 정해져 있음
 - send 실행이 정상 종료했는지 확인하지 않아서 일어나는 오류
 - Exception disorder vulnerability

■ 경매 – King of the Ether Throne

```
contract KotET {  
    address public king;  
    uint public claimPrice = 100;  
    address owner;  
    function KotET() {  
        owner = msg.sender;  
        king = msg.sender;  
    }  
    function sweepCommission(uint amount) {  
        owner.send(amount);  
    }  
    function() {  
        if (msg.value < claimPrice)  
            throw;  
        uint compensation = calculateCompensation();  
        king.send(compensation);  
        king = msg.sender;  
        claimPrice = calculateNewPrice();  
    }  
}
```

Exception
disorder
vulnerability



■ 오버플로우 & 언더플로우

```
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
+ 0x00000000000000000000000000000001
-----
= 0x00000000000000000000000000000000
```

```
0x00000000000000000000000000000000
- 0x00000000000000000000000000000001
-----
= 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

```
pragma solidity 0.4.18;
```

```
contract OverflowUnderFlow {
```

```
    uint public min = 0;
```

```
    uint public max = 2**256-1;
```

```
    function underflow() public {
```

```
        min -= 1;
```

```
    }
```

```
    function overflow() public {
```

```
        max += 1;
```

```
    }
```

```
}
```

▪ SmartMesh token contract (CVE-2018-10376)

- from에서 to로 value만큼 전송하고 msg.sender에게 수수료(fee)만큼 전송

```
1 function transferProxy (address from, address to, uint value, uint fee) {  
2     if (balance[from] < fee + value)                // prevent integer underflow at line 11  
3         revert();  
4  
5     if (balance[to] + value < balance[to] ||          // prevent integer overflows at lines 9 and 10  
6         balance[msg.sender] + fee < balance[msg.sender])  
7         revert();  
8  
9     balance[to] += value;  
10    balance[msg.sender] += fee;  
11    balance[from] -= value + fee;  
12 }
```

CVE(Common Vulnerabilities and Exposures):

공개적으로 알려진 사이버 보안 취약성에 대한 항목 목록 (각각 식별 번호, 설명 및 하나 이상의 공개 참조 포함)

■ SmartMesh token contract (CVE-2018-10376)

초기상태

`balance[from]=0, balance[to]=0, balance[msg.sender]=0`

호출 조건

`value=0x8ff...ff, fee=0x700...01`

`from = to ≠ msg.sender`

방어코드 Revert 실패

`fee+value = 0x8ff...ff + 0x700...01 = 0`

`balance[to] = balance[msg.sender] = 0`

호출 결과

`balance[to] = 0x8ff...ff`

`balance[msg.sender] = 0x700...01`

■ SmartMesh token contract (CVE-2018-10376)

- 취약점을 이용하여 전송이 아니라 엄청난 토큰 생성

From: [0xd6a098DB29e1EafA92a30373c44b09E2e2e0651E](#)

Interacted With (To): [0x55F93985431Fc9304077687a35A1BA103dC1e081](#) (SmartMesh: Token Sale) ✓

ERC-20 Tokens Transferred: 2

All Transfers

Net Transfers

From [0xDF31A4...B34Eb46F](#) To [0xDF31A4...B34Eb46F](#) For
65,133,050,195,990,359,925,758,679,067,386,948,167,464,366,374,422,817,272,194.891004451135422463
[1.58189941672006E+56](#) SmartMesh... (SMT...)

From [0xDF31A4...B34Eb46F](#) To [0xd6a09B...e2e0651E](#) For
50,659,039,041,325,835,497,812,305,941,300,959,685,805,618,291,217,746,767,262.693003461994217473
[1.23036621300449E+56](#) SmartMesh... (SMT...)

<https://etherscan.io/tx/0x1abab4c8db9a30e703114528e31dee129a3a758f7f8abc3b6494aad3d304e43f>

■ batchOverflow Bug (CVE-2018-10299)

- 여러 계정으로 같은 금액 전송

```
function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
        Transfer(msg.sender, _receivers[i], _value);
    }
    return true;
}
```

■ Parity Wallet Bug

- Parity Wallet은 단일 서명을 통해 지갑에서 암호화폐를 출금하던 방식과 달리, 다중 서명을 통해 출금이 가능하도록 한 스마트 컨트랙트 지갑(Contract Wallet)
- 로직을 WalletLibrary로 제공하고 delegatecall 이용



■ Parity Wallet Bug

- 지갑 설정 시 한 번만 호출되어야 하는 initWallet() 함수를 fallback 함수를 통해 우회하여 delegateCall한 다음, 소유자 목록에 본인의 주소를 추가
- 그런 다음 fallback 함수를 통해 pay() 함수를 delegateCall하여 인출

```
contract WalletLibrary {
    address[256] owners;
    mapping ( bytes => uint256 ) approvals;
    function confirm ( bytes32 _op ) internal bool {
        /* logic for confirmation */
    }
    function initWallet ( address [] _owners ) {
        /* initialize the wallet owners */
    }
    function pay ( address to , uint amount ) {
        if ( confirm ( keccak256 ( msg.data ) ) )
            to.transfer( amount );
    }
}
```

```
contract Wallet {
    address library = 0xAABB . . . ;

    // constructor
    function Wallet ( address[] _owners ) {
        library.delegatecall( "initWallet" , _owners )
    }

    function() payable {
        library.delegatecall( msg.data );
    }
}
```

■ TOD(Transaction-Ordering Dependence)

- 트랜잭션 순서에 따라 결과값이 달라질 수 있는 경우, 공격자가 자신의 트랜잭션 순서를 조정해 이득을 취할 수 있다.
- 트랜잭션이 블록에 저장되는 순서는 채굴자가 결정
- 더 높은 가스비로 트랜잭션 발생시켜 먼저 실행되게 하는 공격 가능
 - MarketPlace 컨트랙트
 - 판매자가 구매 트랜잭션을 보고
 - 가격을 더 높게 변동시키는 트랜잭션을 높은 가스비로 실행시킨다
 - 그러면 구매자가 의도한 가격보다 높은 가격에 판매 가능

■ Timestamp Dependence

- 블록의 Timestamp 값에 의존하는 처리를 공격하는 취약점 존재
 - block.timestamp: 블록의 생성시간
 - 채굴자가 블록의 생성시간을 어느정도 통제 가능하여 악의적인 의도를 갖고 결정 가능
 - 예: block.timestamp에 의존하는 추첨 프로그램

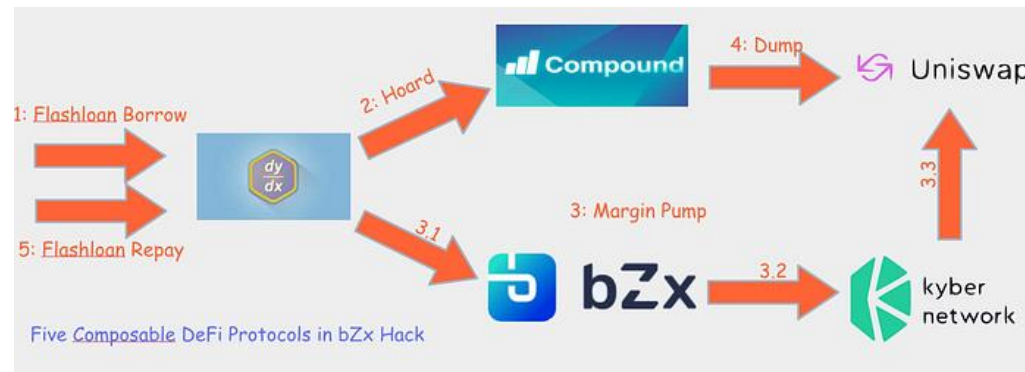
```
// 추첨
function hold() public onlyOwner {
    require(numApplicants == 3);    // 응모자가 3명인지 확인
    timestamp = block.timestamp;    // 타임스탬프 값 설정

    winnerInd = timestamp % 3;      // 추첨
    winnerAddress = applicants[winnerInd];
}
```

■ bZx 해킹 사건 (2020년): 플래시 론(Flash Loan) 공격

- 플래시 론은 한 트랜잭션 내에서 대출을 받고, 사용하고, 상환하는 것을 허용하는 디파이(DeFi) 서비스

- 플래시 론 받기: 공격자는 dYdX와 같은 플래시 론 제공자로부터 자금 대출
- 매집
- 가격 조작: 빌린 자금을 사용해 특정 자산의 가격을 조작
- 이익 실현: 가격 조작으로 인해 이익을 볼 수 있는 거래를 실행
- 대출 상환: 빌린 자금을 상환하고, 나머지 이익을 획득



■ 개인정보 등 민감한 정보 처리

- 트랜잭션은 암호화되지 않으므로 private으로 설정해도 이 정보는 트랜잭션 정보에서 확인 가능
- 상태정보는 블록체인에 영구히 저장되어 확인할 수 있다

■ 보안 고려 사항

- ➔ 개인정보 및 민감한 정보는 스마트 컨트랙트에 저장하지 말고, 필요시 오프체인 데이터베이스에서 관리하도록 설계
- ➔ 변수나 함수의 공개범위는 최소한으로, 공개하는 경우에도 적절한 접근제한을 두어야 한다
- ➔ 스마트 컨트랙트 리뷰 및 감사: 작성된 코드를 전문 감사 기관에 의뢰하거나, 자동화된 도구(Slither, MythX 등)를 사용해 취약점을 점검
- ➔ 업그레이드 가능한 스마트 컨트랙트 설계

■ 스마트 컨트랙트의 알려진 보안 취약점

- Reentrancy(The DAO attacks)
- Call to the unknown(The DAO attacks)
 - 지정한 함수가 없거나 다른 계약으로 ether를 전송하는 경우 fallback함수가 호출됨
- Exception disorders(King of the Ether Throne, GovernMental)
- TOD(Transaction-Ordering Dependence)
- Timestamp Dependence
- 오버플로우 & 언더플로우
- Gasless send(King of the Ether Throne)
 - send 함수를 사용할 때 out-of-gas 예외 발생 가능
- Ether lost in transfer: 실제 사용되지 않고 있는 주소로 전송

■ 스마트 컨트랙트 보안 취약점 참고 논문

- Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. **A Survey of Attacks on Ethereum Smart Contracts**. In Proceedings of the 6th International Conference on Principles of Security and Trust , New York, NY, USA, 164-186.
- Oyente tool 논문: Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: **Making smart contracts smarter**. In: ACM CCS (2016)
- So, S., Lee, M., Park, J., et al. **VERISMART: A highly precise safety verifier for ethereum smart contracts**, In IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, May 2020, pp. 1678–1694
- Frank, J., Aschermann, C., and Holz, T. **ETHBMC: A bounded model checker for smart contracts**, In 29th USENIX Security Symposium, USENIX Association, August 2020
- S. Sayeed, H. Marco-Gisbert and T. Caira, **Smart Contract: Attacks and Protections**, in IEEE Access, vol. 8, pp. 24416-24427, 2020

■ Condition-Effects-Interaction 패턴

- Condition: 함수를 실행하는 조건을 확인하고 조건이 유효하지 않은 경우에는 처리를 중단
- Effects: 상태를 업데이트
- Interaction: 다른 컨트랙트에 메시지를 보낸다

```
function withdraw(uint amount) {  
    if (credit[msg.sender] >= amount) {  
        credit[msg.sender] -= amount;  
        msg.sender.call.value(amount)();  
    }  
}
```

■ ReentrancyGuard 컨트랙트 활용

- ReentrancyGuard 컨트랙트를 상속받아 방어할 함수에 모디파이어를 적용
 - 함수의 진입, 종료 각각의 시점에서 상태변수를 업데이트하여 락을 거는 원리

```
contract ReentrancyGuard {  
    bool private locked;  
  
    modifier noReentrancy() {  
        require(!locked, "Reentrancy detected!");  
        locked = true;  
        _;  
        locked = false;  
    }  
}
```

```
contract EtherStoreGuard is ReentrancyGuard {  
    function withdraw() public noReentrancy { // 방어할 함수에 모디파이어 적용  
        ...  
    }  
}
```

■ Withdraw 패턴

- 스마트 컨트랙트에서 자금을 안전하게 출금하는 방법을 다루는 디자인 패턴
- 이더를 송금할때 소유자가 설정한 시점에 다른 사람에게 보내는 대신(push 방식) 송금을 받을 사람이 언제든지 인출하도록(pull 방식) 하는 패턴
 - push 방식은 송금받는 계정이 컨트랙트인 경우 receive함수에서 악의적인 처리가 가능
 - 자금 출금을 호출자 (특정 사용자 또는 계정)가 직접 수행

■ Withdraw 패턴

■ push형 송금 패턴

- 입찰(bid)을 원하는 사용자가 현재 최고 입찰액보다 많은 이더를 송금하면 최고 입찰자 정보를 변경하고 기존 최고 입찰자에게 자신의 입찰금 반환
- 받는 컨트랙트(기존 최고 입찰자)가 revert()하면 다른 계정에서 입찰 불가

/// 입찰 처리 함수

```
function bid() public payable {
```

```
    require(msg.value > highestBid);
```

```
    uint refundAmount = highestBid;
```

```
// 입찰액이 최고 입찰액보다 높은지 확인
```

```
// 기존 최고 입찰자에게 반환할 액수 설정
```

```
    address currentHighestBidder = highestBidder; // 최고 입찰자 어드레스 업데이트
```

```
    highestBid = msg.value;
```

```
// 스테이트 값 업데이트
```

```
    highestBidder = msg.sender;
```

```
    if(!currentHighestBidder.send(refundAmount)) { // 이전 최고액 입찰자에게 입찰금 반환  
        throw;
```

```
    }
```

```
}
```

■ Withdraw 패턴

■ pull형 송금 패턴

- 입찰금 반환에 push형 송금 대신 전용함수를 사용하여 사용자가 직접 인출 (withdraw)하게 함
- bid와 입찰금 반환 처리 부분 분리

```
mapping(address => uint) public pendingReturns;    // 반환할 액수를 관리하는 매핑
function bid() public payable {
    ...
    pendingReturns[highestBidder] += highestBid; // 기존 최고 입찰자에게 반환할 금액 설정
}
function withdraw() public{
    require(pendingReturns[msg.sender] > 0);        // 반환할 액수가 0보다 큰지 확인
    uint refundAmount = pendingReturns[msg.sender]; // 반환할 액수를 구함
    pendingReturns[msg.sender] = 0;                 // 반환액 업데이트
    if(!msg.sender.send(refundAmount)) {             // 입찰금 반환 처리
        throw;
    }
}
```


■ Access Restriction 패턴

■ 함수에 대한 접근 제어(Modifier)를 이용

- 예: 컨트랙트를 생성한 소유자만이 특정 함수 실행 가능
 - 거의 모든 계약에서 필요한 기능이므로 상속을 통해 재사용

```
contract Owned {
    address public owner;
    constructor() {
        owner = msg.sender; // 컨트랙트 생성자를 소유자로 설정
    }

    modifier onlyOwner {
        require(msg.sender == owner, "Only the owner can call this function.");
        _;
    }
    // 소유자 변경: 현재 소유자만이 호출 가능
    function transferOwnership(address _newOwner) public onlyOwner {
        owner = _newOwner;
    }
}
```

■ Access Restriction 패턴

- 함수에 대한 접근 제어를 이용

```
contract AccessRestriction is Owned{
    string public someState;

    constructor() {
        // someState에 초기값 설정
        someState = "initial";
    }

    // someState의 값을 변경하는 함수
    function updateSomeState(string memory _newState) public onlyOwner {
        someState = _newState;
    }
}
```

■ Access Restriction 패턴

- 접근제한 여부와 상관없이 public으로 두는 함수를 최소화
- 특정 계정만 호출해야 하는 함수는 반드시 해당 계정만 접근하게 제한
- 접근 제한을 modifier로 구현
- 소유자 변경 함수 필요
- 추상화한 후, 상속하여 사용

■ Circuit Breaker 패턴

- 문제가 발생했을 때 임시 기능 정지하는 수단 적용

```
contract CircuitBreaker {  
    bool public emergencyStop = false; // 값이 true이면 긴급 정지가 발동된 상태  
    address public owner;  
  
    modifier isActive {                                // 중단 여부 확인 모디파이어  
        require(!emergencyStop, "Contract is currently paused.");  
        _;  
    }  
  
    modifier onlyOwner {  
        require(msg.sender == owner, "Only the owner can call this function.");  
        _;  
    }  
}
```

■ Circuit Breaker 패턴

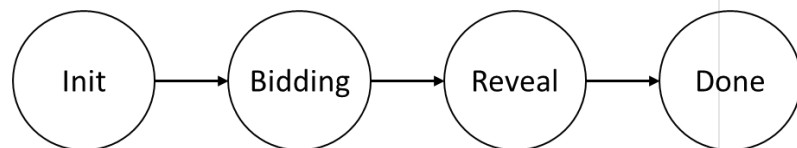
- 문제가 발생했을 때 임시 기능 정지하는 수단

```
constructor() {  
    owner = msg.sender;  
}  
  
// 소유자만 비상 중지를 활성화하거나 비활성화 가능  
function toggleEmergencyStop() external onlyOwner {  
    emergencyStop = !emergencyStop;  
}  
  
// 비상 중지 상태인 경우 실행되지 않도록 isActive 제한자를 사용  
function someFunction() external isActive {  
    // 실행될 로직들...  
}  
}
```

■ State Machine 패턴

- 시스템의 상태 전이를 명확하게 관리하는 디자인 패턴
 - 컨트랙트가 다르게 동작하거나 다른 기능을 호출하는 특정 단계가 있음
 - 컨트랙트의 상태를 추적하고, 특정 상태에서만 실행 가능한 기능을 정의
- 상태를 모델링하고 스마트컨트랙트의 잘못된 사용을 방지하기 위해 함수 수 정자를 사용
- 예) 블라인드 옥션(입찰 단계, 입찰 공개 단계)

```
enum Phase {Init, Bidding, Reveal, Done}  
Phase public currentPhase = Phase.Init;  
  
modifier validPhase(Phase phase) {  
    require(currentPhase == phase, "phaseError");  
    _;  
}
```




■ State Machine 패턴

```
enum Phase {Init, Bidding, Reveal, Done}
Phase public currentPhase = Phase.Init;
modifier validPhase(Phase phase) {
    require(currentPhase == phase, "phaseError");
    _;
}
function advancePhase() onlyOwner public {
    if (currentPhase == Phase.Init) {
        currentPhase = Phase.Bidding;
    } else if (currentPhase == Phase.Bidding) {
        currentPhase = Phase.Reveal;
    } else if (currentPhase == Phase.Reveal) {
        currentPhase = Phase.Done;
    } else {
        currentPhase = Phase.Init;
    }
}
function bid(bytes32 blindBid) public payable validPhase(Phase.Bidding) { ... }
function reveal(uint value, bytes32 secret) public validPhase(Phase.Reveal) { ... }
function auctionEnd() public validPhase(Phase.Done) { ... }
```

ETH - Blind Auction

May the highest bidder win!



Bid on this antique trophy!

Enter blinded bid:

Enter deposit amount:

Reveal your bid:

Enter the OTP:

Current phase: Auction Ended

■ 프록시 패턴(Proxy Pattern)

- 실제 로직 컨트랙트(Logic Contract)을 대리 역할을 하는 프록시 컨트랙트
- 사용자는 프록시 컨트랙트와 상호 작용하지만, 실제 실행되는 로직은 별도의 논리 컨트랙트에서 수행
- 프록시 패턴을 이용한 업그레이더블 컨트랙트 구현

■ Role-Based Access Control 패턴

- 사용자들의 역할에 기반하여 시스템 내에서 접근 권한을 관리하는 보안 모델
- 시스템에서 관리자는 모든 기능에 접근할 수 있지만, 일반 사용자는 일부 기능만 접근할 수 있도록 설정하는 방식

```
contract RBAC is AccessControl {
    constructor() {
        _setupRole(ADMIN_ROLE, msg.sender); // 컨트랙트 배포자에게 ADMIN 역할 부여
    }

    function addUser(address account) public onlyRole(ADMIN_ROLE) {
        grantRole(USER_ROLE, account); // AccessControl 컨트랙트의 역할을 부여하는 함수
    }
    function adminFunction() public onlyRole(ADMIN_ROLE) { // Admin 역할을 가진 계정만 할 수 있는 기능
    }
    function userFunction() public onlyRole(USER_ROLE) { // User 역할을 가진 계정만 할 수 있는 기능
    }
}
```

■ 스마트 컨트랙트의 보안 방법

- 철저한 코드 검토와 감사
- 보안 패턴(예: Checks-Effects-Interactions) 사용
- 외부 라이브러리와 상호작용 최소화
- 상태 변경 후 외부 호출 수행
- 스마트 컨트랙트의 초기화 및 접근 제어 철저

→ 스마트 컨트랙트의 보안은 사용자의 신뢰와 직결되므로 지속적인 관리와 최신 보안 기법 적용이 필요

■ 특정 목적의 스마트 컨트랙트 표준 API 관련

■ ERC-20 (Ethereum Request for Comment 20)

- 이더리움 토큰 스마트 계약을 위한 표준 API
- 표준 API 토큰들을 월렛(Wallet)이나 분산 거래소(Decentralized exchanges) 등에서 공통으로 관리하기 위해 도입

■ 토큰 스마트 계약의 표준 API 적합성 평가 방법 부재

- ERC-20 API 동작에 대한 표준이 모호
 - 함수 이름, 인자와 타입, 리턴 타입(API)만 정의
 - 동작에 대한 불완전한 설명
- API 해석 차이(버그)로 인한 재정 손실 가능