CS438 Project Presentation

Group Members:
Emre Koçer
Borhan Javadian
Neval Yaprak
Mert Karaçavuşoğlu

Overview

Overview

Our Project is a Decentralized Application (Dapp) that implements a simple Decentralized Finance (De-Fi) ecosystem in which users can perform basic financial operations such as swapping, borrowing and lending tokens.

These operations are implemented in Smart Contracts that can be deployed by users on demand.

Technical Details

Technical Details

The Smart Contracts are written in Solidity and can be deployed by users on demand to be used for De-Fi functionalities. For testing and demonstration purposes we deploy the contracts on the Sepolia testnet.

Technical Details

The application is consists of three modules that handle the required functionalities. These modules are implemented as separate Smart Contracts in individual Solidity files.

These modules are:

- AMM.sol
- DeFi.sol
- Core.sol

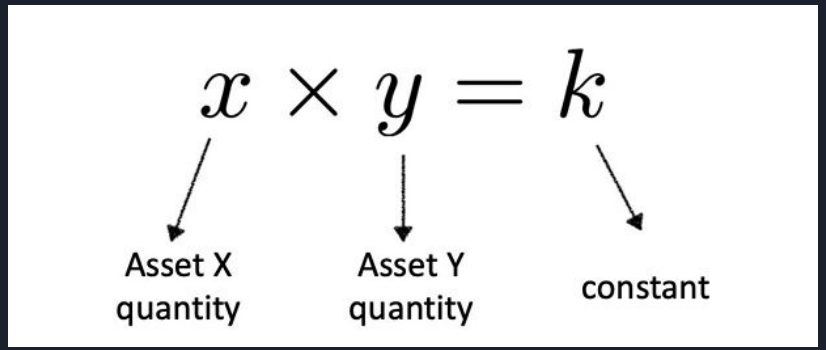
AMM.sol

AMM.sol contains the liquidity pool implementation that allows users to create a pool of two tokens, add and remove liquidity to the pool and swap tokens through the pool.

The tokens in the pool can be any ERC20 compliant token deployed on the blockchain.

AMM.sol

The pool is implemented as a Constant Product Automatic Market Maker (AMM) where the ratio of the number of tokens in the pool is determined by a constant product.



(Image Source: CS438 Lecture Slides)

AMM.sol Functionalities

Users can create a pool of two ERC20 compliant tokens and initialize it with a given liquidity.

Users can swap tokens by adding a single type of token to the pool and withdrawing a number of tokens of the other type. The number of tokens that can be swapped for each other is determined by the tokens value.

The tokens value in terms of each other is determined by the ratio of the tokens in the pool.

AMM.sol Functionalities

Users can add liquidity into existing pools from tokens in their possession. Users can both add liquidity by adding both types of tokens or only a single type of token. These users are called liquidity providers (LPs).

A fee of 0.3% is charged to users for each swap. These fees accumulate in the pool.

When a liquidity provider withdraws from the pool they also are rewarded by a share of the accumulated fees proportional to the amount of liquidity they contributed to the pool.

DeFi.sol

DeFi.sol contains the implementation of the borrowing and lending functionalities.

This Smart Contract functions on top of a deployed pool and allows users to lend and borrow tokens of the types in the pool.

DeFi.sol Functionalities

Users can present one type of token as collateral and borrow tokens of the other type from the pool. These loans are then kept track of until they are repaid.

Since prices can fluctuate radically in a De-Fi ecosystem, the value of the collateral can decrease very quickly. We use the health factor of a loan to determine whether the collateral is sufficient for the loan amount.

DeFi.sol Functionalities

Loans that are considered healthy can only be repaid by the user that initially took the loan. Whereas loans that are unhealthy can be paid for by liquidators who will then receive the collateral.

Liquidators can check the active loans and manually attempt to liquidate a loan. If the loan is unhealthy then the necessary amount of tokens to cover the loan will be transferred from the liquidator to the pool and the initial collateral will be transferred to the liquidator.

In either case of a loan being covered, the loan will get removed from the active list of loans.

DeFi.sol Functionalities

Users can also lend their tokens to the pool and then withdraw them later. As loans are taken and repaid by users, interest will be paid and accumulate in the pool.

When a lender withdraws from the pool they will also receive a reward from the total accumulated interest proportional to the amount that they had loaned.

Core.sol

Core.sol is used for the creation of new AMM pools. When users want to deploy a new pool, they pass the addresses of the tokens they wish to have in the pool and their initial liquidity to the Core.sol Smart Contract which then creates the pool by calling the necessary functions from the AMM.sol Smart Contract. The address of the deployed pool on the blockchain is then returned to the user.

The rest of the operations are then performed through the functions of the other Smart Contracts to which the address of the deployed pool is passed.

Logging Framework

Logs of the state changes and operations performed by the Smart Contracts are kept in the Blockchain. The AMM and DeFi modules both have a set of events defined and these events are logged when appropriate as the different functions making up the application are called.

This allows keeping track of what is being performed by the users and since Blockchain provides data integrity, the integrity of the logs is also guaranteed.

Frontend User Interface

The users can interact with the Smart Contracts through a simple HTML Frontend UI.

The users can specify the operations they would like to perform through the interface which will then be processed and performed by the smart contracts on the blockchain.

The results of the operation are then also displayed to the users through the Frontend UI.

Bonus Module: Arbitrage

The Arbitrage Module is an extra module that can be used by Arbitrageur Actors to perform Arbitrage between different deployed pools. This module was not demanded in the project but was implemented as a bonus.

This module first checks if there exists a valid arbitrage path between deployed pools by simulating swaps, and if valid paths exist, performs arbitrage through the most optimal path.

Security

Security

Since our application is meant for Decentralized Finance user security is of utmost importance to our project.

While developing our project, we applied certain security principles, modules and tools to make sure user assets are sufficiently protected from security breaches.

Security Modules

We use the SafeERC20 and ReentrancyGuard modules from OpenZeppelin to ensure a more secure implementation.

SafeERC20 is a ERC20 compliant token implementation that implements the safeTransfer() and safeTransferFrom() functions. These functions throw errors on failure and makes sure that functions revert if the transfer fails.

Since we make our token transfers at the end of our function calls to comply with the Checks-Effects-Interactions principle, the use of the safeTranfer functions make sure that any changes made to the contracts state will be reverted if the transfer fails.

Reentrancy Attack Prevention

A Reentrancy Attack is a Smart Contract hack where a function of a Smart Contract is called multiple times in a single transaction to manipulate the Smart Contract more times than a user should be able to.

Such attacks are commonly used by attackers to steal tokens and other digital currencies by using withdraw or transfer functions of Smart Contracts multiple times.

Reentrancy Attack Prevention

Reentrancy attacks are performed by exploiting security vulnerabilities in Smart Contracts that allow a function to be called before a previous call to it has terminated.

If the function had not updated the state of the Smart Contract before it was called again, this can allow attackers to manipulate the behavior of the Smart Contract to perform malicious actions such as withdrawing or transferring tokens multiple times.

Reentrancy Attack Prevention

Reentrancy attacks are performed using attacker contracts that are deployed by malicious users.

The malicious user forces the victim contract to make an external call to their attacker contract from which they will call the victim contract again before the first call to it has completed.

Checks-Effects-Interactions Pattern

The Checks-Effects-Interactions pattern is a principle in Smart Contract programming where a function of a smart contract first performs necessary checks for the function to proceed, then updates the state of the smart contract and finally performs interactions with other smart contracts.

This pattern is used to protect the smart contract from security vulnerabilities and potential security breaches such as reentrancy attacks.

Since this pattern ensures that the state of the Smart Contract will be properly updated before any external calls are made, attackers will not be able to call the contract again before it has updated its state to manipulate its behavior.

We tried to follow the Checks-Effects-Interactions pattern as much as possible while implementing our smart contracts.

Checks-Effects-Interactions Pattern Example

Checks:

Perform necessary checks for valid function execution

Effects:

Update the state of the Smart Contract variable

Interactions:

Interaction with other contracts (Token Transfer)

```
function swap(address tokenIn, uint256 amountIn) external nonReentrant  {    infinite gas
    require(tokenIn == tokenA || tokenIn == tokenB, "Invalid tokenIn");
    address tokenOut = tokenIn == tokenA ? tokenB : tokenA;

    uint256 liquidityIn = tokenIn == tokenA ? liquidityA : liquidityB;
    uint256 liquidityOut = tokenIn == tokenA ? liquidityB : liquidityA;

    uint256 amountInWithFee = (amountIn * FEE_NUMERATOR) / FEE_DENOMINATOR;
    uint256 amountOut = (liquidityOut * amountInWithFee) / (liquidityIn + amountInWithFee);

    require(amountOut > 0, "Insufficient output");

    if (tokenIn == tokenA) {
        liquidityA += amountIn;
        liquidityB -= amountOut;
    }
    else {
        liquidityB += amountIn;
        liquidityA -= amountOut;
    }

    IERC20(tokenIn).safeTransferFrom(msg.sender, address(this), amountIn);
    IERC20(tokenOut).safeTransfer(msg.sender, amountOut);
}
```

Use of ReentrancyGuard and Non Reentrant Functions

The use of the ReentrancyGuard module allows us to declare functions as non reentrant. A function that is declared as non reentrant cannot be called again while a previous call to it has not completed.

This ensures that malicious user cannot call a function multiple times before the state of the Smart Contract has been updated.

```
function swap(address tokenIn, uint256 amountIn) external nonReentrant {
```

Reentrancy Attack Prevention

In the implementation of our De-Fi Application, another way in which we avoid potential reentrancy attacks by avoiding external calls to other smart contracts. The only external function calls from our Smart Contracts are to the OpenZeppelin ERC20 transfer() and transferFrom() functions.

Since these functions themselves do not make any external function calls, malicious users cannot call a function multiple times through an attacker contract in a single transaction since they cannot force our Smart Contracts to make a call to their attacker contract.

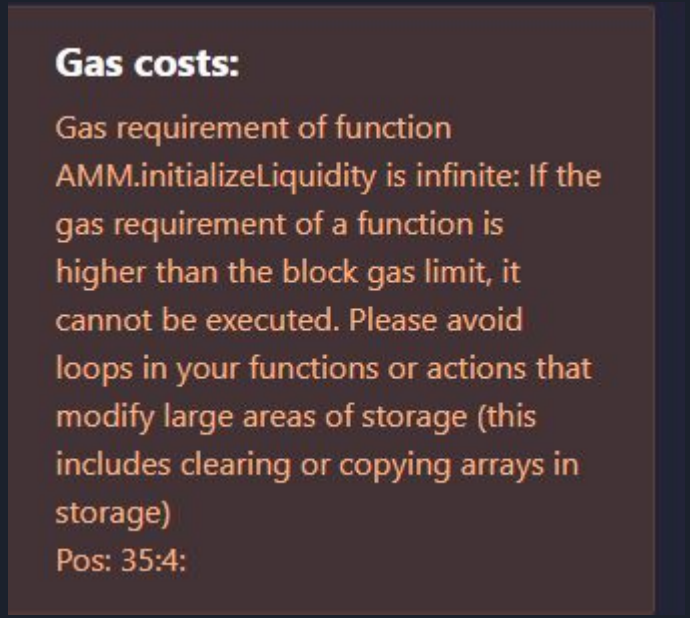
Static Code Analysis

We used the Static Code Analyzer plug-in of Remix to analyze our Smart Contracts for potential security issues.

When checked with the analyzer, our final implementation still returns warnings however these warnings do not actually indicate security flaws.

The following slides include explanations of the analyzer warnings and how they do not indicate security flaws.

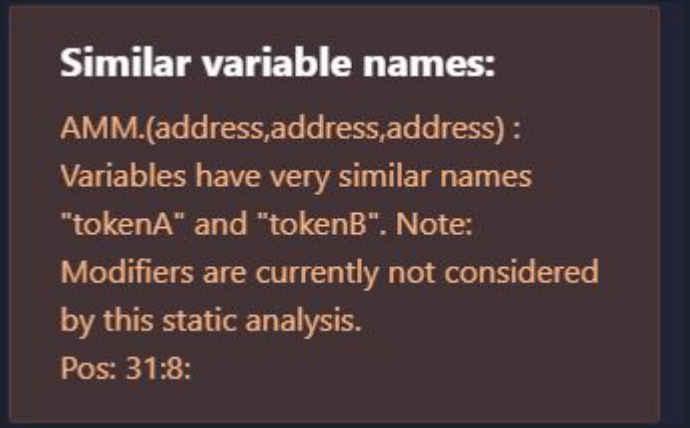
Gas Cost Warning



This warning is raised when Remix is not able to compute a maximum gas cost for a function.

Since we are able to deploy our contracts and call all our functions on the Sepolia Testnet we can conclude that this warning is inaccurate.

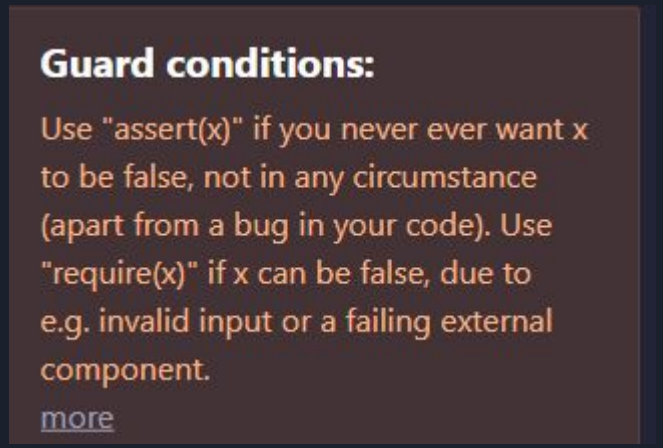
Similar Variable Names Warning



This warning is raised when Remix considers variable names to be too similar to each other.

This warning is meant to encourage code readability and has no significance in regards to security.

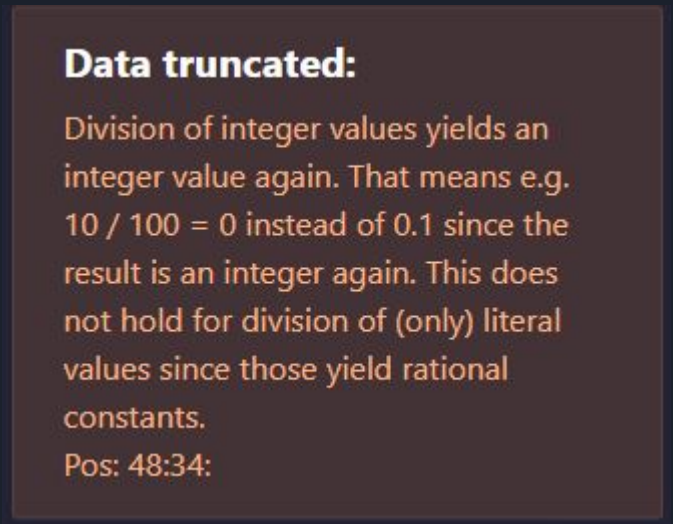
Guard Conditions Warning



This warning is related to which guard condition should be used when checking for a requirement.

In our case we use require(x) and we check for user inputted values which can be false, hence our use of the guard condition is correct.

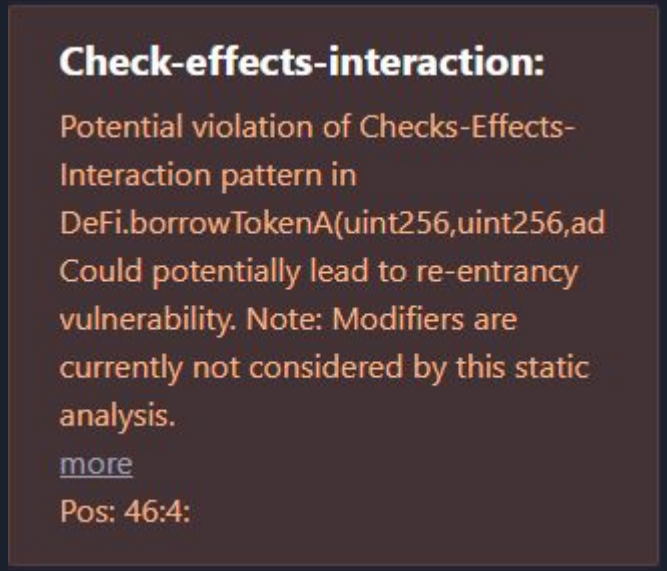
Data truncated Warning



This warning is to do with how results are computed during integer division and is not related to contract security.

Since Solidity does not support floating point numbers, we have to perform all our computations using integers.

Checks-Effects-Interactions Warning



This warning is an actual security concern because it detects a potential violation of Checks-Effects-Interactions which can lead to a reentrancy attack.

However after checking the flagged functions we see that the functions do not violate Checks-Effects-Interactions and that the warning is a false flag.

Checks-Effects-Interactions Warning

The Checks-Effects-Interactions violation warning is raised for 6 functions all of which are in DeFi.sol. These functions are:

- DeFi.borrowTokenA()
- DeFi.borrowTokenB()
- DeFi.repayLoan()
- DeFi.liquidate()
- DeFi.lendTokenA()
- DeFi.lendTokenB()
- DeFi.withdrawTokenA()
- DeFi.withdrawTokenB()

The following slides contain explanations of how these warnings are false flags.

Checks-Effects-Interactions Warning

```
function borrowTokenA(uint256 collateralAmount, uint256 borrowAmount, address poolAddress) external nonReentrant {
    require(loans[msg.sender].borrowedAmount == 0, "Loan already active");

    AMM pool = AMM(poolAddress);
    address tokenA = pool.token0();
    address tokenB = pool.token1();
    uint256 liquidityA = pool.liquidity0();
    uint256 liquidityB = pool.liquidity1();

    uint256 collateralValueInA = (liquidityA * collateralAmount) / liquidityB;

    require(collateralValueInA * 100 >= borrowAmount * 150, "Insufficient collateral: min 150% required");
    require(totalTokenAStaked[poolAddress] >= borrowAmount, "Insufficient TokenA liquidity");

    loans[msg.sender] = Loan({
        collateralAmount: collateralAmount,
        borrowedAmount: borrowAmount,
        loanType: LoanType.TokenA
    });

    totalTokenAStaked[poolAddress] -= borrowAmount;
    totalTokenBStaked[poolAddress] += collateralAmount;

    IERC20(tokenB).safeTransferFrom(msg.sender, address(this), collateralAmount);
    IERC20(tokenA).safeTransfer(msg.sender, borrowAmount);

    emit LoanCreated(msg.sender, poolAddress, collateralAmount, borrowAmount, LoanType.TokenA);
    emit CollateralValueCalculated(msg.sender, poolAddress, collateralValueInA, liquidityA * 1e18 / liquidityB);
}
```

Checks

Effects

Interactions

Checks-Effects-Interactions Warning

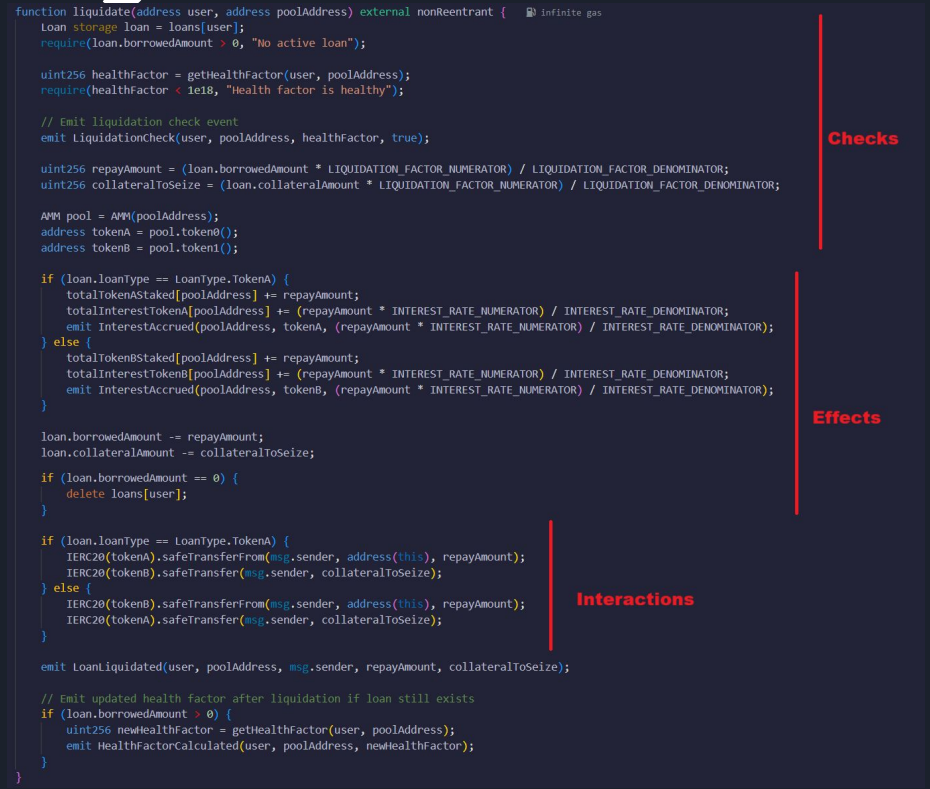
```
function borrowTokenB(uint256 collateralAmount, uint256 borrowAmount, address poolAddress) external nonReentrant {
    require(loans[msg.sender].borrowedAmount == 0, "Loan already active");

    AMM pool = AMM(poolAddress);
    address tokenA = pool.token0();
    address tokenB = pool.token1();
    uint256 liquidityA = pool.liquidity0();
    uint256 liquidityB = pool.liquidity1();

    uint256 collateralValueInB = (liquidityB * collateralAmount) / liquidityA;

    require(collateralValueInB * 100 >= borrowAmount * 150, "Insufficient collateral: min 150% required");
    require(totalTokenBStaked[poolAddress] >= borrowAmount, "Insufficient TokenB liquidity");

    loans[msg.sender] = Loan({
        collateralAmount: collateralAmount,
        borrowedAmount: borrowAmount,
        loanType: LoanType.TokenB
    });

    totalTokenBStaked[poolAddress] -= borrowAmount;
    totalTokenAStaked[poolAddress] += collateralAmount;

    IERC20(tokenA).safeTransferFrom(msg.sender, address(this), collateralAmount);
    IERC20(tokenB).safeTransfer(msg.sender, borrowAmount);

    emit LoanCreated(msg.sender, poolAddress, collateralAmount, borrowAmount, LoanType.TokenB);
    emit CollateralValueCalculated(msg.sender, poolAddress, collateralValueInB, liquidityB * 1e18 / liquidityA);
}
```

Checks

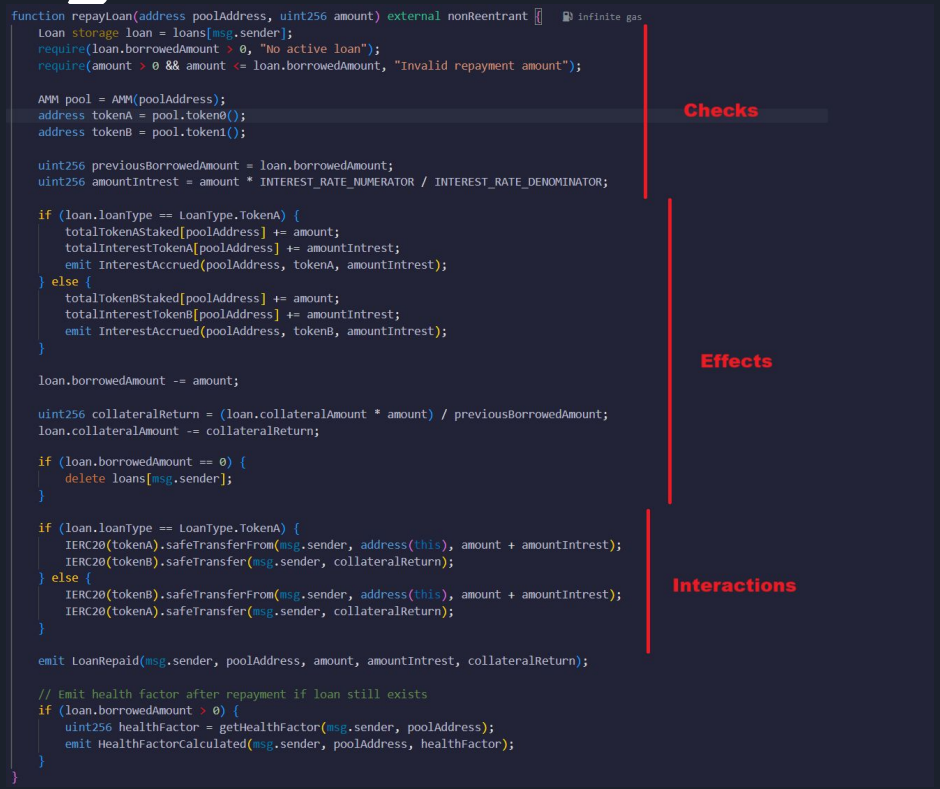
Effects

Interactions

Checks-Effects-Interactions Warning



Checks-Effects-Interactions Warning



Checks-Effects-Interactions Warning

```
function lendTokenA(uint256 amount, address poolAddress) external nonReentrant {
    require(amount > 0, "Cannot lend 0");

    address tokenA = AMM(poolAddress).token0();
    totalTokenAStaked[poolAddress] += amount;

    shares[msg.sender][poolAddress].tokenA_share += amount;
    IERC20(tokenA).safeTransferFrom(msg.sender, address(this), amount);

    emit TokenLent(msg.sender, poolAddress, tokenA, amount);
}
```

Checks

Effects

Interactions

Checks-Effects-Interactions Warning

```
function lendTokenB(uint256 amount, address poolAddress) external nonReentrant {
    require(amount > 0, "Cannot lend 0");

    address tokenB = AMM(poolAddress).token1();
    totalTokenBStaked[poolAddress] += amount;

    shares[msg.sender][poolAddress].tokenB_share += amount;
    IERC20(tokenB).safeTransferFrom(msg.sender, address(this), amount);

    emit TokenLent(msg.sender, poolAddress, tokenB, amount);
}
```

Checks

Effects

Interactions

Checks-Effects-Interactions Warning

```
function withdrawTokenA(address poolAddress) external nonReentrant {    infinite gas
    Share storage share = shares[msg.sender][poolAddress];
    uint256 userShare = share.tokenA_share;
    require(userShare > 0, "Nothing to withdraw");

    uint256 totalStaked = totalTokenAStaked[poolAddress];
    uint256 interest = totalInterestTokenA[poolAddress];
    uint256 userInterest = (userShare * interest) / totalStaked;

    uint256 totalAmount = userShare + userInterest;

    totalTokenAStaked[poolAddress] -= userShare;
    totalInterestTokenA[poolAddress] -= userInterest;
    share.tokenA_share = 0;

    address tokenA = AMM(poolAddress).token0();
    IERC20(tokenA).safeTransfer(msg.sender, totalAmount);

    emit TokenWithdrawn(msg.sender, poolAddress, tokenA, totalAmount, userShare, userInterest);
}
```

Checks

Effects

Interactions

Checks-Effects-Interactions Warning

```
function withdrawTokenB(address poolAddress) external nonReentrant {    infinite gas
    Share storage share = shares[msg.sender][poolAddress];
    uint256 userShare = share.tokenB_share;
    require(userShare > 0, "Nothing to withdraw");

    uint256 totalStaked = totalTokenBStaked[poolAddress];
    uint256 interest = totalInterestTokenB[poolAddress];
    uint256 userInterest = (userShare * interest) / totalStaked;

    uint256 totalAmount = userShare + userInterest;

    totalTokenBStaked[poolAddress] -= userShare;
    totalInterestTokenB[poolAddress] -= userInterest;
    share.tokenB_share = 0;

    address tokenB = AMM(poolAddress).token1();
    IERC20(tokenB).safeTransfer(msg.sender, totalAmount);

    emit TokenWithdrawn(msg.sender, poolAddress, tokenB, totalAmount, userShare, userInterest);
}
```

Checks

Effects

Interactions

Checks-Effects-Interactions Warning

As is visible from these screenshots, the checks, effects and interactions components of the functions are all in the correct order. This means that the checks-effects-interactions pattern is not actually violated and that the analyzer warnings are false flags.

Project Github Repository

Link: https://github.com/Borhanxj/defi-blockchain-app

QUESTIONS