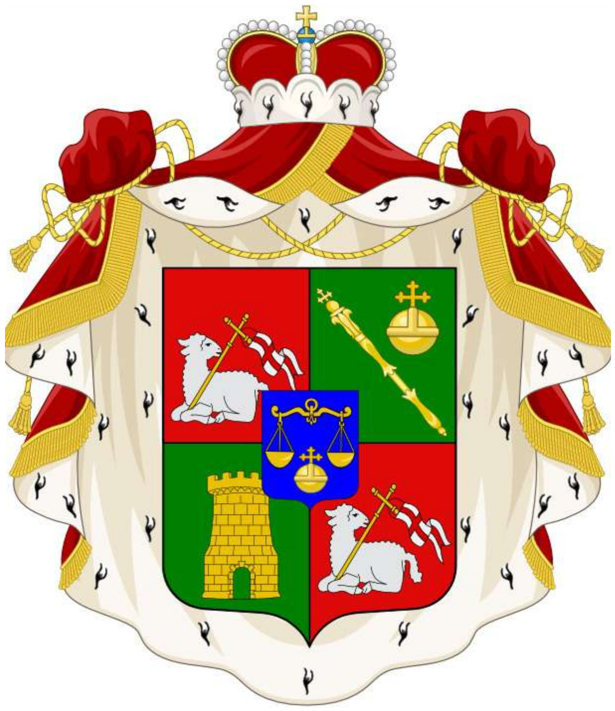

ThunderLoan Audit Report

Version 1.0

Zurab Anchabadze

November 19, 2024

ThunderLoan Audit Report

Zurab Anchabadze

November 19, 2024

Prepared by: Zurab Anchabadze

Table of Contents

- Medium
  - [M-1] Using TSwap as price oracle leads to users gets much cheaper fees than expected (price and oracle manipulation attacks)
- Low
- Informational
- Gas

Protocol Summary

Protocol does X, Y, Z ______________________________

Disclaimer

Zurab Anchabadze 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
	Low	M	M/L	L

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

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	Numbers of issues found
High	3
Medium	0
Low	0
Gas	0
Info	0
Total	4

Findings

High

[H-1] Erroneous `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 the ThunderLoan system, the `exchangeRate` is responsible for calculationg the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, updates this rate, without collecting any fees.

```
    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, bacause the protocol thinks the owed tokens 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 `ThunderLoan.test.t.sol`

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

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

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

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

```
    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] All the flashloaned funds can be stolen if the flash loan is returned using deposit()

Description: An attacker can acquire a flash loan and deposit funds directly into the contract using the deposit(), enabling stealing all the funds.

Impact: All funds are stolen

Proof of Concept: 1. Take a flash loan. 2. Return the borrowed funds not through `repay()`, but via `deposit()`. 3. The flash loan function checks that the funds have returned to the contract and approves the transaction. 4. We withdraw the funds back using `redeem()`, since we deposited them with `deposit()`.

You can find PoC in ThunderLoanTest.t.sol

Recommended Mitigation: Add a check in deposit() to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in flashloan() and checking it in deposit().

[H-3] Mixing up variable location causes stroage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: The `ThunderLoanUpgrade.sol` has two variables in the following order:

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

    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
```

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

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

    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
```

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 storage variables for constant variables, breaks the storage location as well.

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

More importantly, the `s_currentlyFlashLoaning` mapping will start in the wrong storage slot.

Proof of Concept:

PoC

Place the following into `ThunderLoanTest.t.sol`

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

You can also see the storage layout differences by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

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

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

     mapping(IERC20 token => bool currentlyFlashLoaning) private
         s_currentlyFlashLoaning;
```

Medium

[M-1] Using TSwap as price oracle leads to users gets much cheaper fees than expected (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:

    i. User sells 1000 tokenA, tanking the price.
    ii. Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA.

        a. 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
   ();
     }
```

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

I have created a proof of code located in `ThunderLoanTest.t.sol`. It is too large to include here.

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