ThunderLoan Audit Report

Version 1.0

Predator

November 8, 2024

ThunderLoan Audit Report

Predator

November 11, 2024

Prepared by: [Predator] Lead Security Researcher: - Predator

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - The findings are described in this document corresponf the following commit:
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Mixing up varuable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
    - [H-2] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which block redemption and incorrectly sets the exchange rate

    - [H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
    - [H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals
  - Medium
    - [M-1] Centralization risk for trusted owners
      - Impact:
      - 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
    - [M-3] Fee on transfer, rebase, etc
  - 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:

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 PREDATOR 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

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

```
- Commit Hash:  8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

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

We spent around 3 hours with team to identify all vulnerabilities.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	3
Info	3
Gas Optimizations	4
Total	14

Findings

High

[H-1] Mixing up varuable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

IMPACT: High -> fee is going be janked up for upgrade/storage collision is real bad! Likelihood: Medium/Low (only if protocol upgrade)

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

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

However, the upgraded contract ThunderLoanUpgraded.sol has them in a different order:

```
    uint256 private s_flashLoanFee;
    uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity storage works, after the upgrade s_flashLoanFee will have the value of s_feePrecision. You cannot adjust the position of storage variables, and removing storage variables, breaks the storage location as well.

Impact After the upgrade, s_flashLoanFee will have the value of s_feePrecision. This means the users who take out the flash loans right after upgrade, will be charged with wrong fee.

More importantly, ThunderLoan::s_currentlyFlashLoaning mapping with storage in the wrong storage slot.

Proof of Concepts

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 upgrade = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgrade), "");
        uint256 feeAfterUpgarde = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("Fee before upgrade", feeBeforeUpgrade);
        console2.log("Fee after upgrade", feeAfterUpgarde);

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

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 storage variable, leave it as blank to not mess up the storage slot.

```
-    uint256 private s_flashLoanFee;
-    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 block redemption and incorrectly sets the exchange rate

Description In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetsTokens and underlying tokens. In a way, it's responsible for kepping track of how many fees to give to liquidity providers.

However, the `deposit` function, updates the rate, without collecting the 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 of this bug 1. The `redeem` function is blocked because the protocol think 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. User takes out a flash loan 3. It's now impossible for LP to redeem

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);
        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];
        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] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

[H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

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

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 Concepts The following happens in 1 transaction: 1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged with original fee `fee1`. During the flash loan, they do the following: 1. User sells 1000 `tokenA`, tanking the price. 2. Instead of repaying right now, the user takes another flash loan for another 1000 `tokenA`. 1. Due to the fact that the way `ThundeLoan` 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.
```

I have created a proof of code located in my `audit-data` folder. It is too large to include here.

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

[M-3] Fee on transfer, rebase, etc

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);
```