Thunder Loan Initial Audit Report

Version 0.1

Cyfrin.io

January 20, 2024

Thunder Loan Audit Report

Duncan DuMond

January 20, 2024

Thunder Loan Audit Report

Prepared by: Duncan DuMond Lead Security Researcher: Duncan DuMond

- DuncanDuMond

Assisting Auditors:

- None

Table of contents

See table

About Duncan DuMond

A security researcher with a passion for blockchain security and learning the ins and outs of smart contract security from the best in the industry.

Disclaimer

The Duncan DuMond 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

Audit Details

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

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

Protocol Summary

Puppy Rafle is a protocol dedicated to raffling off puppy NFTs with variying rarities. A portion of entrance fees go to the winner, and a fee is taken by another address decided by the protocol owner.

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

Issues found

Severity	Number of issues found
High	2
Medium	2
Low	3
Info	1
Gas	2
Total	10

Findings

High

[H-1] `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees thaan it really does, which blocks redemption and incorrectly sets the exchange rate.

Description: In `ThunderLoan::deposit`, the `updateExchangeRate` function is called after the `transferFrom` call. This means that the protocol thinks it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate.

The funciton is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it is responsible for keeping track of how many fees to give to liquidity providers. However, the `deposit` function, updates this rate without collecting any fees!

```
1     function deposit(IERC20 token, uint256 amount) external
          revertIfZero(amount) revertIfNotAllowedToken(token) {
2         AssetToken assetToken = s_tokenToAssetToken[token];
3         uint256 exchangeRate = assetToken.getExchangeRate();
```

```
4           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
5           emit Deposit(msg.sender, token, amount);
6           assetToken.mint(msg.sender, mintAmount);
7   @>      uint256 calculatedFee = getCalculatedFee(token, amount);
8   @>      assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

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 getting way more or less than they should

Proof of Concept:

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

Proof of Code

Place the following code in `ThunderLoanTest.t.sol`

```
1       function testRedeemAfterLoan() public setAllowedToken hasDeposits {
2           uint256 amountToBorrow = AMOUNT * 10;
3           uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
4
5           vm.startPrank(user);
6           tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
7           thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
                amountToBorrow, "");
8           vm.stopPrank();
9
10          uint256 amountToRedeem = type(uint256).max;
11          vm.startPrank(liquidityProvider);
12          thunderLoan.redeem(tokenA, amountToRedeem);
13      }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from the `deposit` function.

```
1       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token];
```

```
3           uint256 exchangeRate = assetToken.getExchangeRate();
4           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
5           emit Deposit(msg.sender, token, amount);
6           assetToken.mint(msg.sender, mintAmount);
7 -         uint256 calculatedFee = getCalculatedFee(token, amount);
8 -         assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

[H-2] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution

Description:

```
1       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
2           AssetToken assetToken = s_tokenToAssetToken[token];
3           uint256 exchangeRate = assetToken.getExchangeRate();
4           uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
5           emit Deposit(msg.sender, token, amount);
6           assetToken.mint(msg.sender, mintAmount);
7           uint256 calculatedFee = getCalculatedFee(token, amount);
8           assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

Impact: Users can deposit and immediately withdraw more funds. Since exchange rate is increased on deposit, they will withdraw more funds then they deposited without any flash loans being taken at all.

Recommended Mitigation: It is recommended to not update exchange rate on deposits and updated it only when flash loans are taken, as per documentation.

```
1   function deposit(IERC20 token, uint256 amount) external revertIfZero(
        amount) revertIfNotAllowedToken(token) {
2       AssetToken assetToken = s_tokenToAssetToken[token];
3       uint256 exchangeRate = assetToken.getExchangeRate();
4       uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
            ) / exchangeRate;
5       emit Deposit(msg.sender, token, amount);
6       assetToken.mint(msg.sender, mintAmount);
7 -     uint256 calculatedFee = getCalculatedFee(token, amount);
8 -     assetToken.updateExchangeRate(calculatedFee);
9       token.safeTransferFrom(msg.sender, address(assetToken), amount);
```

```
10 }
```

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 { }
```

[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 Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
1   function getPriceInWeth(address token) public view returns (
       uint256) {
```

```
2     address swapPoolOfToken = IPoolFactory(s_poolFactory).
          getPool(token);
3 @>      return ITSwapPool(swapPoolOfToken).
      getPriceOfOnePoolTokenInWeth();
4   }
```

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 a Uniswap TWAP fallback oracle.

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
1 File: src/protocol/ThunderLoan.sol
2
3 261:     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):

```
1 File: src/protocol/OracleUpgradeable.sol
2
3 11:     function __Oracle_init(address poolFactoryAddress) internal
     onlyInitializing {
```

```
1 File: src/protocol/ThunderLoan.sol
2
3 138:     function initialize(address tswapAddress) external initializer
      {
4
5 138:     function initialize(address tswapAddress) external initializer
      {
6
7 139:         __Ownable_init();
8
```

```
9  140:         __UUPSUpgradeable_init();
10
11 141:         __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.

```
1  +    event FlashLoanFeeUpdated(uint256 newFee);
2  .
3  .
4  .
5      function updateFlashLoanFee(uint256 newFee) external onlyOwner {
6          if (newFee > s_feePrecision) {
7              revert ThunderLoan__BadNewFee();
8          }
9          s_flashLoanFee = newFee;
10 +        emit FlashLoanFeeUpdated(newFee);
11     }
```

Informational

[I-1] Poor Test Coverage

```
1  Running tests...
2  | File                                | % Lines        | % Statements
      | % Branches    | % Funcs        |
3  | ----------------------------------- | -------------- | --------------
      | ------------- | -------------- |
4  | src/protocol/AssetToken.sol         | 70.00% (7/10)  | 76.92% (10/13)
      | 50.00% (1/2)  | 66.67% (4/6)   |
5  | src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
      | 100.00% (0/0) | 80.00% (4/5)   |
6  | src/protocol/ThunderLoan.sol        | 64.52% (40/62) | 68.35% (54/79)
      | 37.50% (6/16) | 71.43% (10/14) |
```

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);
```