---

title: Thunder Loan Audit Report author: Ritik Agarwal date: Auguest 20, 2024

Thunder Loan Audit Report

Prepared by: Ritik Agarwal Lead Auditors:

- Ritik Agarwal

Assisting Auditors:

- None

Disclaimer

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	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	4
Medium	1
Low	2
Gas	2
Total	9

Findings

[H-1] Erronoues `updateExchangeRate` in the desposit function causes protocol to think it has more fees than it really does, which blocks redemption and incprrectly sets the exchange rate.

Description In the thunderLoan system , the `exchangerate` is reponisble for calculating the exchagne rate between assestToken and underlying tokens.In a way it is responsible for keeping track of how many fees to give to liqudity providers. However, the `deposite` function, update the rate, without collecting any fees! This update should be removed.

```
    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 function Impacts toh thiss bug. 1. The `redeem` function is blocked, because the protoccol thinks the owed token is mroe than ithas. 2. Rewards are incorrectly calculated, leading liquidity providers potentially getting way more of less then deserved.

Proof of Concept

1. LP deposite
2. User takes out the falsh loans.
3. It is now impossible for LP to redeem.

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 incorrect update of the exchange rate in the `deposit` function.

```
    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] 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

```
// 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`:

```diff
-    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-3] 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.

```
    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.
```

Proof of Concept:

```
function testOracle() public {
    thunderLoan = new ThunderLoan();
       tokenA = new ERC20Mock();
       proxy = new ERC1967Proxy(address(thunderLoan), "");
       BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
           address(weth)
       );
       address tswapPool = poolFactory.createPool(address(tokenA));
       thunderLoan = ThunderLoan(address(proxy));
       thunderLoan.initialize(address(poolFactory));

       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();

       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();

       uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
       console2.log("Normal Fee Cost: ", normalFeeCost);
       // 0.296147410319118389
       uint256 amountToBorrow = 50e18;
       MFlashLoan mFlashLoan = new MFlashLoan(
           address(tswapPool),
           address(thunderLoan),
           address(thunderLoan.getAssetFromToken(tokenA))
       );
```

```
        vm.startPrank(user);
        tokenA.mint(address(mFlashLoan), 100e18);
        thunderLoan.flashloan(address(mFlashLoan), tokenA, amountToBorrow, "");
        vm.stopPrank();
        uint256 attackFee = mFlashLoan.feeOne() + mFlashLoan.feeTwo();
        console2.log("Attack Fee: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed 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: The `deposit` function in `ThunderLoan.sol` allows users to deposit funds into the protocol. However, the `deposit` function does not check if the user has taken out a flashloan. This allows users to take out a flashloan and then deposit the funds back into the protocol, effectively stealing all the funds from the protocol.

```
    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: This bug allows users to steal all funds from the protocol.

Proof of concept

```
    function testUseDepositeInsteadOfRepayToStealFunds()
        public
        setAllowedToken
        hasDeposits
    {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay depositOverRepay = new DepositOverRepay(
            address(thunderLoan)
        );
        tokenA.mint(address(depositOverRepay), fee);
        thunderLoan.flashloan(
            address(depositOverRepay),
            tokenA,
            amountToBorrow,
            ""
        );
        depositOverRepay.redeemMoney();
        vm.stopPrank();
        assert(tokenA.balanceOf(address(depositOverRepay)) > 50e18 + fee);
    }
```

[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.

```
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

[L-1] 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 of the contract.

```
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-2] 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.

```diff
+    event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

[G-1] 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;
```

[G-2] 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`.

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