ThunderLoan Security Audit Report by MK (15th DEC 2024)

High Risk Findings

[H-1] Storage Collision on Upgrade

Description: Storage collision because of changing the places/slots of the variables.

Impact:

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

Proof of Concept:

1. When there is a collision the user/admin will assume he sets a certain variable but another the contract will set the variable that is reserved in that slot and that will lead to unwanted behaviour.

```
import {ThunderLoanUpgraded} from
"../../src/upgradedProtocol/ThunderLoanUpgraded.sol";

 function upgradeThunderloan() internal {
        thunderLoanUpgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(thunderLoanUpgraded));
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));
    }

 function testSlotValuesBeforeAndAfterUpgrade() public setAllowedToken {
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint precision = thunderLoan.getFeePrecision();
        uint fee = thunderLoan.getFee();
        bool isflanshloaning = thunderLoan.isCurrentlyFlashLoaning(tokenA);
        /// 4 slots before upgrade
        console.log("????SLOTS VALUE BEFORE UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(asset));
        console.log("slot 1 for s_feePrecision =>", precision);
        console.log("slot 2 for s_flashLoanFee =>", fee);
        console.log("slot 3 for s_currentlyFlashLoaning =>", isflanshloaning);
        //upgrade function
        upgradeThunderloan();

        //// after upgrade they are only 3 valid slot left because precision is
now set to constant
        AssetToken assetUpgrade = thunderLoan.getAssetFromToken(tokenA);
        uint feeUpgrade = thunderLoan.getFee();
        bool isflanshloaningUpgrade = thunderLoan.isCurrentlyFlashLoaning(
            tokenA
```

```
        );

        console.log("????SLOTS VALUE After UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(assetUpgrade));
        console.log("slot 1 for s_flashLoanFee =>", feeUpgrade);
        console.log(
            "slot 2 for s_currentlyFlashLoaning =>",
            isflanshloaningUpgrade
        );
        assertEq(address(asset), address(assetUpgrade));
        //asserting precision value before upgrade to be what fee takes after
upgrades
        assertEq(precision, feeUpgrade); // #POC
        assertEq(isflanshloaning, isflanshloaningUpgrade);
    }
```

Recommended Mitigation:

1. Keep the location of the declaration of the variables.
2. If a new variable needs to be implemented it can be implemented after the existing ones.

```
//
 function upgradeThunderloanFixed() internal {
        thunderLoanUpgraded = new ThunderLoanUpgraded();
        //getting the current fee;
        uint fee = thunderLoan.getFee();
        // clear the fee as
        thunderLoan.updateFlashLoanFee(0);
        // upgrade to the new implementation
        thunderLoan.upgradeTo(address(thunderLoanUpgraded));
        //wrapped the abi
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));
        // set the fee back to the correct value
        thunderLoanUpgraded.updateFlashLoanFee(fee);
    }



function testSlotValuesFixedfterUpgrade() public setAllowedToken {
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint precision = thunderLoan.getFeePrecision();
        uint fee = thunderLoan.getFee();
        bool isflanshloaning = thunderLoan.isCurrentlyFlashLoaning(tokenA);
        /// 4 slots before upgrade
        console.log("????SLOTS VALUE BEFORE UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(asset));
        console.log("slot 1 for s_feePrecision =>", precision);
        console.log("slot 2 for s_flashLoanFee =>", fee);
        console.log("slot 3 for s_currentlyFlashLoaning =>", isflanshloaning);
        //upgrade function
        upgradeThunderloanFixed();
```

```
        //// after upgrade they are only 3 valid slot left because precision is
now set to constant
        AssetToken assetUpgrade = thunderLoan.getAssetFromToken(tokenA);
        uint feeUpgrade = thunderLoan.getFee();
        bool isflanshloaningUpgrade = thunderLoan.isCurrentlyFlashLoaning(
            tokenA
        );

        console.log("????SLOTS VALUE After UPGRADE????");
        console.log("slot 0 for s_tokenToAssetToken =>", address(assetUpgrade));
        console.log("slot 1 for s_flashLoanFee =>", feeUpgrade);
        console.log(
            "slot 2 for s_currentlyFlashLoaning =>",
            isflanshloaningUpgrade
        );
        assertEq(address(asset), address(assetUpgrade));
        //asserting precision value before upgrade to be what fee takes after
upgrades
        assertEq(fee, feeUpgrade); // #POC
        assertEq(isflanshloaning, isflanshloaningUpgrade);
    }
```

[H-2] Cannot redeem because of the update exchange rate

Description: The `deposit` function calls `getCalculatedFee` on the depositor's inputs of token and amount and then calls `assetToken.getExchangeRate(calculatedFee)`. This is acting as if the depositor is being charged a fee that is then accruing to assetToken, but in fact the only people who are supposed to be charged fees are the people taking flash loans. The depositor does not pay a fee. Therefore, this is throwing off the calculation of the exchange rate.

Impact: This incorrectly drives up the exchange rate by acting as if fees are earned from deposits when they are not. This can result in depositors getting more or less tokens than they should when they deposit and redeem.

Proof of Concept:

1. The `deposit` function calculates a fee using the depositor's inputs and then updates the exchange rate using that calculated fee:

```
    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);
    }
```

Recommended Mitigation:

1. The deposit function shouldn't call getCalculatedFee, so I recommend removing the line. Also, I recommend removing the call to `updateExchangeRate` as well. I explained in another finding why I recommend not using the exchange rate concept for purposes of calculating how many tokens/asset tokens depositors get for deposits and redemptions.

[H-3] Fee is 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.

Impact: Incorrect calculations of fees and exchange rates, leading to less interest rates and less fees imposed on flash loan borrower.

Proof of Concept: The Below behaviour has 2 implications:

1. a flashloan with a token that has very small value to WETH will have a insignificant fee, that will not be 3% (as the initial state of the contract) from the borrowed amount
2. a flashloan with a token that has a big value to WETH(let's assume 1 token = 100 WETH) will require a extremely high fee. In this case, if we borrow 1 token, the returned fee is 3 -> valueOfBorrow = (1 * 100) / 1; fee = (100 * 3) / 100 = 3. So, after flashloaning 1 token, the user will have to return 4 tokens(1amount + 3fee)

```
    function getCalculatedFee(IERC20 token, uint256 amount) public view returns
(uint256 fee) {
        // converts the value of the tokens in WETH
        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
s_feePrecision;
        // calculate the fee percentage from the value in WETH
        fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
    }
```

Recommended Mitigation: Drop the conversion to WETH, and use following implemention:

```
    function getCalculatedFee(uint256 amount) public view returns (uint256 fee) {
        fee = (amount * s_flashLoanFee) / s_feePrecision;
    }
```

[H-4] Funds can be drained by `ThunderLoan::flashloan()`

Description: `ThunderLoan::flashloan()` function can be exploited by attacker and funds can be drained. The revert condition check `endingBalance < startingBalance + fee` is not sufficient, as it can be pretended by attacker that funds are repaid by calling the ThunderLoan::deposit() function and amount equivalent to (flash loan + fees) can be deposited in ThunderLoan, this will increase the balance to the required repayment amount and the condition `endingBalance < startingBalance + fee` will becomes false and it will not revert, but it should have reverted as the funds were not paid they were only deposited. As a result of which funds can be later withdrawn by calling the ThunderLoan::redeem() function.

Impact: All funds in the protocol can be drained.

Proof of Concept: User can maliciously mint the tokens using the flash loans.In the flash loan function the check of the repayment is done by checking the token balance of the assetToken contract this can be increased just by depositing the tokens so by doing this we can mint the assetTokens and no need of paying the flashloan back because of the increase in token balance of the assetToken.

```
uint256 endingBalance = token.balanceOf(address(assetToken));
if (endingBalance < startingBalance + fee)
```

Recommended Mitigation:

1. To mitigate this vulnerability, users should not be allowed to deposit funds while taking flash loans. Modify the function ThunderLoan::deposit()

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(amount)
revertIfNotAllowedToken(token) {
+   if (s_currentlyFlashLoaning[token]) {
+       revert ThunderLoan__CantDepositDuringFlashLoan();
+   }

    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);
}
```

Medium Risk Findings

[M-1] Centralized owners can brick redemptions by unallowing a token

Description: The redeem function is used by liquidity providers to get their underlying tokens (plus fees they earned) back. The redeem function has a check that the token must be an allowed token or the redeem will not work. But the setAllowedToken function (which is used to both add and remove allowed tokens) can just change a token from allowed to not allowed and there is no provision for liquidity providers to get their tokens back (unless the owner switches the token back to allowed, but that might not be an attractive option if you don't want people to continue being able to flash loan that token for whatever reason).

Impact: LP's liquidity will be stuck in the contract at least temporarily. You can fix it by making the token allowed again but even if you do this it is possible that some people won't get the news that they need to redeem, so unless you are okay with having the token permanently allowed, some people's tokens will be locked. You probably had a reason to no longer have the token as allowed so you won't want to keep it allowed long term, most likely.

Proof of Concept:

1. The redeem function will revert if the token is no longer an allowed token:

```
function redeem(
        IERC20 token,
        uint256 amountOfAssetToken
    ) external revertIfZero(amountOfAssetToken) revertIfNotAllowedToken(token)
```

2. This portion of setAllowedToken makes a token not allowed and it is silent as to what should happen with the tokens currently put in as liquidity.

```
        } else {
            AssetToken assetToken = s_tokenToAssetToken[token];
            delete s_tokenToAssetToken[token];
            emit AllowedTokenSet(token, assetToken, allowed);
            return assetToken;
        }
```

3. Here is a test I wrote to prove that you can't redeem an unapproved token even if you still have deposits....it uses the setAllowedToken and hasDeposits modifiers. hasDeposits means that liquidityProvider has deposited DEPOSIT_AMOUNT of tokenA.

```
        function testCantRedeemOnceTokenUnapproved()
        public
        setAllowedToken
        hasDeposits
    {
        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);
        vm.stopPrank();
        vm.startPrank(liquidityProvider);
        vm.expectRevert();
        thunderLoan.redeem(tokenA, DEPOSIT_AMOUNT);
```

```
        vm.stopPrank();
    }
```

Recommended Mitigation:

1. You could send all the tokens back to liquidity providers as part of the function to change a token to not allowed (although this may not be ideal if you just want to turn off a token's allowed state briefly). Alternatively you can make it so that tokens that were approved but were later disapproved can still use the redeem function through the following changes:

2. Add an array of asset token addresses that will track all addresses ever approved:

```
IERC20 public approvedTokenAddresses[];
```

3. In the setAllowedToken function, push every asset token address that is ever approved to the array (you won't remove the address from this array even if its approval is revoked):

```
approvedTokenAddresses.push(token);
```

4. Add a function to check if a particular token was ever an approved token:

```
function wasEverAllowedToken(IERC20 token) public {
       for(i = 0, i < approvedTokenAddresses.length, i++) {
           if(approvedTokenAddresses[i] == token) {
              return true;}
                 continue;
           }
}
```

5. Add a modifier revertIfNeverAllowedToken as follows and then apply this new modifier to the redeem function in place of revertIfNotAllowedToken:

```
    modifier revertIfNeverAllowedToken(IERC20 token) {
        if (!wasEverAllowedToken(token)) {
            revert ThunderLoan__NotAllowedToken(token);
        }
        _;
    }
```

[M-2] Weak Oracle: Not using decentralized price feed

Description: Not using an oracle network to get price of asset

Impact: Getting the price from a single source of truth, like a liquidity pool, makes the protocol vulnerable to a price oracle attack. The liquidity pool price can be manipulated in the attacker's favor. See https://chain.link/education-hub/flash-loans for details on price oracle attacks and how to mitigate against them using Chainlink price feeds.

Proof of Concept:

1. The protocol is using a liquidity pool on some exchange to return the price of an asset in WETH.

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

2. This is using a single source of truth instead of using a decentralized oracle network to get the price.

Recommended Mitigation:

1. Implement chainlink price feeds in the protocol.
2. Do not use a liquidity pool to get prices.

[M-3] `ThunderLoan::deposit` Doesn't Work Well With ERC20 Tokens With A Fee On Transfer

Description: Accepted tokens like USDT, STA, PAXG charge a fee on transfer, that isn't accounted during the minting of `AssetToken` when a user provides liquidity to the protocol.

Impact: Liquidity providers can receive more `AssetToken` than the their actual deposits, giving them a bigger ownership of the pool that they actually own.

Proof of Concept:

1. Let's assume a user provides 10,000 USDT as liquidity, according to the logic in `deposit` that same amount is used to mint `AssetToken`:

```
uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()) /
exchangeRate;
```

2. But since a fee is charged the actual quantity deposited in `AssetToken` will be less than `amount`. So the liquidity provider will receive an amount of `AssetToken` greater than the actual deposit.

Recommended Mitigation:

1. Read the balance of token in `AssetToken` before and after the deposit to calculate a delta that represents the actual amount deposited by the user to know how many tokens to mint and use the delta to calculate the fee.

```
diff --git a/src/protocol/ThunderLoan.sol.orig b/src/protocol/ThunderLoan.sol
index 1031975..efeb6cc 100644
--- a/src/protocol/ThunderLoan.sol.orig
+++ b/src/protocol/ThunderLoan.sol
@@ -163,12 +163,17 @@ contract ThunderLoan is Initializable, OwnableUpgradeable,
UUPSUpgradeable, OracleUpgradeable

     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);
+
+        uint256 balanceBefore = token.balanceOf(address(assetToken));
+        token.safeTransferFrom(msg.sender, address(assetToken), amount);
+        uint256 balanceAfter = token.balanceOf(address(assetToken));
+        uint256 amountDeposited = balanceAfter - balanceBefore;
+
+        uint256 mintAmount = (amountDeposited *
assetToken.EXCHANGE_RATE_PRECISION()) / exchangeRate;
+        emit Deposit(msg.sender, token, amountDeposited);
         assetToken.mint(msg.sender, mintAmount);
-        uint256 calculatedFee = getCalculatedFee(token, amount);
+        uint256 calculatedFee = getCalculatedFee(token, amountDeposited);
         assetToken.updateExchangeRate(calculatedFee);
-        token.safeTransferFrom(msg.sender, address(assetToken), amount);
     }
```

Low Risk Findings

[L-1] `getCalculatedFee` can be Zero

Description: In the `getCalculatedFee()` method the fee is calculated with `amount * price / fee_precision`. If `amount * price` is smaller than `fee_precision` then the resulting fee will be smaller than 1. This will cause the set fee to be 0 while the actual cost is above 0.

Impact:

1. The ETH held in the smart contract.

Proof of Concept:

1. Any value up to 333 for "amount" can result in 0 fee based on calculation

```
    function testFuzzGetCalculatedFee() public {
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
```

```
        uint256 calculatedFee = thunderLoan.getCalculatedFee(
            tokenA,
            333
        );

        assertEq(calculatedFee ,0);

        console.log(calculatedFee);
    }
```

Recommended Mitigation:

1. A minimum fee can be used to offset the calculation, though it is not that important.

[L-2] `updateFlashLoanFee()` missing event

Description: `ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` does not emit an event, so it is difficult to track changes in the value `s_flashLoanFee` off-chain.

Impact:

1. In Ethereum, events are used to facilitate communication between smart contracts and their user interfaces or other off-chain services. When an event is emitted, it gets logged in the transaction receipt, and these logs can be monitored and reacted to by off-chain services or user interfaces.
2. Without a `FeeUpdated` event, any off-chain service or user interface that needs to know the current `s_flashLoanFee` would have to actively query the contract state to get the current value. This is less efficient than simply listening for the `FeeUpdated` event, and it can lead to delays in detecting changes to the `s_flashLoanFee`.
3. The impact of this could be significant because the `s_flashLoanFee` is used to calculate the cost of the flash loan. If the fee changes and an off-chain service or user is not aware of the change because they didn't query the contract state at the right time, they could end up paying a different fee than they expected.

Proof of Concept:

1. It is observed.

```
function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > FEE_PRECISION) {
            revert ThunderLoan__BadNewFee();
        }
@>        s_flashLoanFee = newFee;
    }
```

Recommended Mitigation:

1. Emit an event for critical parameter changes.

```
+ event FeeUpdated(uint256 indexed newFee);

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

[L-3] Mathematic Operations Handled Without Precision in `getCalculatedFee()` Function in `ThunderLoan.sol`

Description: In a manual review of the ThunderLoan.sol contract, it was discovered that the mathematical operations within the getCalculatedFee() function do not handle precision appropriately. Specifically, the calculations in this function could lead to precision loss when processing fees. This issue is of low priority but may impact the accuracy of fee calculations.

Impact:

1. This issue is assessed as low impact. While the contract continues to operate correctly, the precision loss during fee calculations could affect the final fee amounts. This discrepancy may result in fees that are marginally different from the expected values.

Proof of Concept:

1. The identified problem revolves around the handling of mathematical operations in the getCalculatedFee() function. The code snippet below is the source of concern:

```
uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
s_feePrecision;
fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

2. The above code, as currently structured, may lead to precision loss during the fee calculation process, potentially causing accumulated fees to be lower than expected.

Recommended Mitigation:

1. To mitigate the risk of precision loss during fee calculations, it is recommended to handle mathematical operations differently within the getCalculatedFee() function. One of the following actions should be taken:
2. Change the order of operations to perform multiplication before division. This reordering can help maintain precision. Utilize a specialized library, such as math.sol, designed to handle mathematical operations without precision loss.

3. By implementing one of these recommendations, the accuracy of fee calculations can be improved, ensuring that fees align more closely with expected values.