Thunder Loan Initial Audit Report

Version 1.0

Umesh.io

October 20, 2025

Thunder Loan Audit Report

Umesh1145

October 18, 2023

Prepared by: Umesh1145

Table of contents

See table

    - [H-3] By calling a flashloan and then `ThunderLoan::deposit` instead of `ThunderLoan::repay` users can steal all funds from the protocol
    - [H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals
  - 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
    - [M-4] Fee on transfer, rebase, etc
  - 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

Disclaimer

The YOUR_NAME_HERE 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

Audit Details

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

```
026da6e73fde0dd0a650d623d0411547e3188909
```

Scope

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

Protocol Summary

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.

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	2
Medium	2
Low	3
Info	1
Gas	2
Total	10

Findings

High

[H-1] 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 Concept:

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.upgradeToAndCall(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;
```

[H-2] Unnecessary `updateExchangeRate` in `deposit` function incorrectly updates `exchangeRate` preventing withdraws and unfairly changing reward distribution

Description: The `deposit` flow mutates the interest accounting by calling `assetToken.updateExchangeRate(calculatedFee)` on every deposit. The exchange rate represents how many underlying tokens back one `AssetToken`. Tying exchange rate updates to deposit calls allows any depositor to arbitrarily move the exchange rate, which: - shifts value between depositors depending on when they deposit, - can make withdrawals revert if the updated exchange rate makes the user's redeem math underflow or fail slippage checks, - decouples protocol fee accrual from real economic events (e.g., actual fees collected from flash loans).

```
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); // Problematic:
    mutates global price based on a user deposit
     token.safeTransferFrom(msg.sender, address(assetToken), amount);
}
```

Impact: - Withdrawals can fail or return less-than-expected amounts due to exchange rate drift caused by arbitrary user deposits. - Depositors can be diluted or unfairly advantaged based on transaction ordering (MEV surface). - Accounting no longer reflects real earnings; the protocol may overstate fee income and misallocate rewards.

Proof of Concept: Consider two users A then B depositing sequentially. B's deposit updates the exchange rate upward via `calculatedFee`, minting fewer `AssetToken` per unit, diluting A or B depending on timing rather than true earnings. A simple test can highlight the skew:

```
// Pseudocode test
vm.prank(owner);
thunderLoan.setAllowedToken(tokenA, true);

// User A deposits
tokenA.mint(userA, 100e18);
vm.startPrank(userA);
tokenA.approve(address(thunderLoan), 100e18);
thunderLoan.deposit(tokenA, 100e18);
vm.stopPrank();

// User B deposits; deposit mutates exchange rate
tokenA.mint(userB, 100e18);
vm.startPrank(userB);
tokenA.approve(address(thunderLoan), 100e18);
thunderLoan.deposit(tokenA, 100e18);
vm.stopPrank();

// Now attempt to redeem proportionally and observe mismatched returns
    due to exchange rate drift
```

Recommended Mitigation: - Remove `assetToken.updateExchangeRate(calculatedFee)` from `deposit`.

```
-    assetToken.updateExchangeRate(calculatedFee);
```

- Accrue protocol fees based on realized earnings only (e.g., on flash-loan repayment) and update exchange rate in a deterministic place tied to those earnings, or use a share-based model where the exchange rate derives from `totalUnderlying/totalShares` without manual bumps.
- If exchange rate updates are necessary, gate them behind trusted hooks (e.g., only ThunderLoan can update after fees actually arrive) and base on balance-delta snapshots.

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

Description: During a flash loan, the receiver controls execution. The protocol expects the receiver to approve and call `repay`. However, a malicious receiver can instead call `deposit(token, amount + fee)` sending funds to the `AssetToken` rather than the loan accounting, receiving newly minted `AssetToken` in return while the flash-loan internal invariant "loan must be repaid" is never enforced against actual ThunderLoan-held balances.

This converts debt into a credited deposit, effectively minting value out of thin air, breaking solvency.

Impact: - Loss of funds; the attacker can drain protocol liquidity by converting loaned tokens into deposit receipts and walking away with claim on the pool. - Insolvency; accounting no longer matches held balances.

Proof of Concept: In a custom receiver's `executeOperation`, replace the expected repay path with a deposit:

```
function executeOperation(address token, uint256 amount, uint256 fee,
    address, bytes calldata)
    external
    returns (bool)
{
    IERC20(token).approve(address(thunderLoan), amount + fee);
    // Malicious path: deposit instead of repay
    thunderLoan.deposit(IERC20(token), amount + fee);
    // No call to thunderLoan.repay(...)
    return true;
}
```

Unless ThunderLoan enforces a post-execution invariant (e.g., snapshot-based balance check or explicit `repaid` flag), the flash loan completes while the attacker holds freshly minted `AssetToken`.

Recommended Mitigation: - Enforce end-of-loan invariant: after callback, require `assetToken.balanceOf(ThunderLoan)` or the relevant underlying balance increased by at least `amount +`

`fee` compared to a pre-loan snapshot. - Make `repay` the only valid repayment path by: - Tracking an internal "expected repayment" and zeroing it only in `repay`, - Reverting in the flash-loan function if outstanding expected repayment is non-zero after callback. - Do not credit deposits during an active flash loan for the borrowed `token` (reentrancy guard per-token or loan-context lock). - Alternatively, transfer repayments directly to ThunderLoan, not to `AssetToken`, and only later sweep to the asset contract.

[H-4] getPriceOfOnePoolTokenInWeth uses the TSwap price which doesn't account for decimals, also fee precision is 18 decimals

Description: The price returned from TSwap pools is computed assuming 18-decimal precision inputs, but listed assets may use different decimals (e.g., 6 for USDC). ThunderLoan uses this price for fee computation with `FEE_PRECISION = 1e18`. Without normalizing for token decimals, fees are materially mispriced for non-18-decimal assets.

Additionally, quoting "price of one token in WETH" by simulating `1e18` input ignores per-token decimals and can skew pricing and fee logic in edge reserves.

Impact: - Under- or over-charging fees by orders of magnitude for tokens with decimals != 18. - Potential economic exploits where attackers prefer assets with favorable decimal mismatches to pay near-zero fees or cause accounting errors.

Proof of Concept: For a 6-decimal token, treating `1e18` as "one token" scales the notionally priced amount by `1e12`. Fees based on this incorrect notional are off by `1e12`, either massively overcharging or undercharging compared to intent.

Recommended Mitigation: - Normalize all amounts to 18 decimals when computing prices and fees: read `decimals()` once per token, store it, and scale inputs/outputs with `10**(18 - tokenDecimals)` or vice-versa. - Prefer robust oracles: Chainlink feeds for base price and a TWAP (e.g., Uniswap V3 TWAP) as fallback; do not rely on single-block AMM spot quotes. - Make fee precision configurable per-asset or compute using share-based accounting that is decimals-agnostic. ## 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.

[M-4] Fee on transfer, rebase, etc

Description: Some ERC20 tokens implement transfer fees, rebasing, or deflationary mechanics. Assuming `amount` sent equals `amount` received leads to accounting drift. Minting `AssetToken` based on requested transfer size without verifying actual received balance causes over-crediting. Similarly, repayments based on nominal values may underfund the protocol.

Impact: - Over-minting `AssetToken` on deposit if fewer tokens arrive than expected. - Under-repayment on flash loans when tokens skim fees on transfer. - Long-term insolvency and inability to satisfy withdrawals.

Recommended Mitigation: - Use balance-delta accounting: before/after balance snapshots around transfers to determine the actual received amount and mint based on that. - For repayments, check that the ThunderLoan-held underlying balance increased by at least `amount + fee` (not just allowances/approvals). - Consider explicitly disallowing fee-on-transfer/rebasing tokens, or maintain per-token adapters that normalize behavior safely.

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) |
```

[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`.

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