Thunder Loan Protocol Security Review

Reviewer: Muhammad Faran **Date:** July 4, 2025 **Repo:** Cyfrin/6-thunder-loan-audit **Commit Hash:** `026da6e73fde0dd0a650d623d0411547e3188909`

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	High	Medium	Low
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium-Low
Low	Medium	Medium-Low	Low

Audit Details

Commit Hash Reviewed: `1ec3c30253423eb4199827f59cf564cc575b46db`

Scope:

```
├── interfaces
|   ├── IFlashLoanReceiver.sol
|   ├── IPoolFactory.sol
|   ├── ITSwapPool.sol
|   |── IThunderLoan.sol
├── protocol
|   ├── AssetToken.sol
|   ├── OracleUpgradeable.sol
|   |── ThunderLoan.sol
|── upgradedProtocol
|   |── ThunderLoanUpgraded.sol
```

Roles/Actors:

- 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.

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!

Executive Summary

Severity	Number Of Issues Found
High	**3**
Medium	**0**
Low	**0**
Informational	**4**

[I-1] Front-Running Risk in `initialize()` Function

Scope `ThunderLoan.sol`

Description

The `initialize()` function sets important protocol parameters such as the oracle address and fee structure. It is meant to be called only once via the OpenZeppelin `initializer` modifier.

However, since the contract uses the UUPSUpgradeable pattern and the initializer is `external`, a race condition could occur during deployment or upgrade if it's not executed by a trusted initializer contract. If the contract is deployed without a proper upgrade path or proxy admin, **a malicious actor could front-run the initializer and set malicious parameters**.

```
//@audit initialize function, can be front run
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15; // 0.3% ETH fee
}
```

Risk

Likelihood:

- When the contract is deployed and no proxy admin or secure deployment flow is enforced
- When developers forget to initialize right after deployment

Impact:

- Permanent takeover of the contract
- Setting a malicious oracle to control pricing logic

Proof of Concept

If the proxy contract is deployed and `initialize()` is not immediately called by the deployer, attacker can divert the protocol towards his own Oracle:

```
thunderLoan.initialize(attackerOracleAddress);
```

This gives the attacker control over pricing and flashloan fee manipulation.

Recommended Mitigation

Add a constructor-like `_disableInitializers()` (already present) and **ensure secure deployment via proxy only**. Additionally:

```diff
- function initialize(address tswapAddress) external initializer {
+ function initialize(address tswapAddress) external initializer onlyProxy {
```

[H-1] Incorrect Reward Accrual on `deposit()`

Scope `ThunderLoan.sol`

Description

Normally, the `updateExchangeRate()` function should be called after **actual fee income has been realized** (e.g flashloan fee). However, the current implementation updates the exchange rate **immediately after deposit**, based on a fee calculated from the deposit amount, which is incorrect.

This **results in inflated rewards** for liquidity providers because the fee has not yet been earned.

```
//@audit Reward Manipulation, exchange rate updated on deposit, it should be updated on fee earned
...
assetToken.mint(msg.sender, mintAmount);
uint256 calculatedFee = getCalculatedFee(token, amount);
@> assetToken.updateExchangeRate(calculatedFee); // unjustified exchange rate bump
token.safeTransferFrom(msg.sender, address(assetToken), amount);
```

Risk

Likelihood:

- Every time a liquidity provider deposits, this code is triggered.

Impact:

- Misleading exchange rate
- Unsustainable inflation of `AssetToken` values
- Protocol loses control over supply-value equilibrium

Proof of Concept

Liquidity provider deposits 1 token → `getCalculatedFee()` returns a fake "fee" → `updateExchangeRate()` applies it → user's `AssetToken` is overvalued.

No actual flashloan occurred, no real fee was earned.

Recommended Mitigation

Only call `updateExchangeRate()` in places where the protocol **receives external revenue**, such as:

- Flashloan fees

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

Remove this line from `deposit()` and restrict `updateExchangeRate()` usage to fee-generating scenarios.

[H-2] Oracle Manipulation in `getCalculatedFee()`

Scope `ThunderLoan.sol`

Description

The function `getCalculatedFee()` relies on `getPriceInWeth(token)` from the oracle contract to determine the value of a borrowed token. If this price is manipulated (e.g., via an untrusted oracle), the protocol will charge lower or zero fees for large flashloans.

```
//@audit Oracle Manipulation
...
uint256 valueOfBorrowedToken = (amount * getPriceInWeth(address(token))) /
s_feePrecision;
@> uint256 fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
```

Risk

Likelihood:

- When oracle source is manipulatable (e.g., DEX TWAP, low liquidity pool)

Impact:

- Attacker can reduce flashloan fees to near-zero
- Loss of funds if attacker takes huge loans without paying fair value

Proof of Concept

If the oracle returns a price of `0`:

```
getCalculatedFee(token, 1_000_000e18); // returns 0
```

Now a large flashloan is issued with **no fee** due to manipulated pricing.

Recommended Mitigation

Use robust oracles like Chainlink.

Here's the markdown audit report for the **storage slot collision** introduced in the updated `ThunderLoanUpgraded` contract due to the introduction of the `FEE_PRECISION` `constant`:

[H-3] Storage Slot Collision Due to Introduction of Constant

Scope `ThunderLoanUpgraded.sol`

Description

In upgradeable contracts using proxy patterns (such as UUPS or Transparent Proxy), the **storage layout must remain consistent** across all upgrades to prevent overwriting existing state variables. In the original implementation of `ThunderLoan`, both `s_feePrecision` and `s_flashLoanFee` were declared as `uint256` state variables in storage.

In the upgraded `ThunderLoanUpgraded` contract, the `s_feePrecision` variable was **replaced with a `constant FEE_PRECISION`**, which does not occupy a storage slot (as constants are inlined at compile time). This causes a **shift in the storage layout**, resulting in all subsequent storage variables being written to the wrong slots—**leading to dangerous storage collision**.

```
// Before (Original Implementation)
uint256 private s_feePrecision;
uint256 private s_flashLoanFee;

// After (Upgraded Implementation)
uint256 private s_flashLoanFee;
uint256 public constant FEE_PRECISION = 1e18;
```

@> By removing `s_feePrecision` as a state variable and replacing it with a constant, `s_flashLoanFee` shifts into the original slot of `s_feePrecision`, potentially corrupting proxy storage.

Risk

Likelihood:

- Will occur immediately upon deploying or upgrading to the new implementation using the `ThunderLoanUpgraded` logic behind a proxy.
- Solidity stores state variables in the order of declaration. The new layout no longer matches the proxy storage layout.

Impact:

- `s_flashLoanFee` will overwrite the slot that originally held `s_feePrecision`.
- Subsequent mappings and storage variables (e.g., `s_currentlyFlashLoaning`) will also become misaligned.
- May result in corrupted state, inaccessible data, broken functionality, and unrecoverable funds.

Recommended Mitigation

Do not change the storage layout across upgradeable contract versions.

To fix this:

```diff
- uint256 public constant FEE_PRECISION = 1e18;
+ uint256 private s_feePrecision;
```

And reintroduce the `s_feePrecision` variable at its original position to preserve the storage layout:

```
uint256 private s_feePrecision; // Must remain for compatibility
uint256 private s_flashLoanFee;
```

Another fix if you want to keep `s_feePrecision` constant because of gas issues could be as; preserve the storage layout by inserting a dummy variable in place of the removed one:

```diff
- uint256 private s_flashLoanFee;
- uint256 public constant FEE_PRECISION = 1e18;

+ uint256 private __gap_s_feePrecision; // reserved slot
+ uint256 private s_flashLoanFee;
+ uint256 public constant FEE_PRECISION = 1e18;
```

```
// New upgraded version
uint256 private __gap_s_feePrecision; // reserve slot 0
uint256 private s_flashLoanFee;
uint256 public constant FEE_PRECISION = 1e18;
```

This __gap_s_feePrecision acts as a placeholder for the old s_feePrecision slot. It ensures that s_flashLoanFee and all subsequent variables remain in their correct positions.

[I-2] Upgradable USDC Contract May Break Invariants

Description

The protocol relies on **USDC**, which is a proxied contract (i.e., upgradable). While USDC is widely adopted and trusted, **its implementation can change**, possibly introducing new behaviors, breaking assumptions, or disabling features (e.g., ERC-20 hooks or `transferFrom()` behavior).

Any protocol assuming USDC has a stable interface and behavior across time is making a **risky assumption** in a decentralized context.

Risk

Likelihood:

- Occurs when Circle upgrades USDC's logic — which has happened in the past.

Impact:

- Could break integrations that assume specific behaviors (e.g., return values, approval logic, gas usage).
- Silent failures (e.g., `transferFrom` always reverting).

Recommended Mitigation

- Monitor USDC upgrade announcements.
- Include fallback mechanisms or pause features for critical upgrades.

[I-3] Proxy-Based Design Introduces Centralization Risk

Description

The entire protocol is built using **UUPS proxies** with upgradeable logic. While this supports agility and fixes, it introduces **centralization risk**, especially if `onlyOwner` or privileged access is not trust-minimized.

Risk

Likelihood:

- Always present if `onlyOwner` is not replaced with a DAO, timelock, or governance mechanism.

Impact:

- Users must **trust the upgrade keyholder**.
- Undermines decentralization guarantees.
- Can be exploited in case of compromised or malicious owner.

Recommended Mitigation

- Introduce a multi-sig or timelocked upgrade path.
- Communicate upgradeability clearly in documentation.
- Minimize upgrade surface once protocol stabilizes.

Here is a complete markdown report for the **unused import issue** you've identified in the interface file, written in the security audit format:

[I-4] Unused Import Increases Codebase Complexity and Audit Surface

Scope `IFlashLoanReceiver.sol`

Description

The following line is an unused import found in the interface file `IFlashLoanReceiver.sol`:

```
// import { IThunderLoan } from "./IThunderLoan.sol";
```

@> The imported `IThunderLoan` contract is **never used** in this file. It exists solely to make the file compilable in `test/mocks/MockFlashLoanReceiver.sol`, which is a test/mock contract.

Including unused imports in production-facing or core files (e.g., interfaces or contracts) solely for test support is considered **bad engineering practice**. It creates **tight coupling between test logic and protocol code**, bloats the codebase, and introduces unnecessary compilation dependencies — especially problematic in larger or security-sensitive systems.

Risk

Likelihood:

- Always present during compilation and maintenance
- Can go unnoticed by automated tools if not explicitly flagged

Impact:

- Increases cognitive load for developers and auditors
- Introduces unnecessary dependencies that may change or break over time
- Creates a false impression of dependencies between unrelated components
- Slightly increases bytecode size and audit scope (in rare compiler edge cases)

Recommended Mitigation

- Remove the unused import from the interface file.

- `IThunderLoan` is required in a mock for testing purposes, **import it in the test file instead** (`MockFlashLoanReceiver.sol`) — not in the interface.
- Keep interface contracts clean, minimal, and purpose-specific.

Suggested Fix

```diff
- import { IThunderLoan } from "./IThunderLoan.sol"; // unused in this file
```

And in `test/mocks/MockFlashLoanReceiver.sol`:

```diff
+ import { IThunderLoan } from "../../path/to/IThunderLoan.sol"; //use only where required
```
