OpenZeppelin | security

Spline Audit

September 30, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-15 To 2025-09-23
Languages	Cairo

Total Issues	5 (1 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	1 (0 resolved)
Notes & Additional Information	3 (0 resolved)

Scope

OpenZeppelin audited the SplineFinance/spline-v0 repository at commit 080c245.

In scope were the following files:

```
src
├── lib.cairo
├── lp.cairo
├── math.cairo
├── profile.cairo
├── shared_locker.cairo
├── token.cairo
└── profiles
    ├── bounds.cairo
    ├── cauchy.cairo
    └── symmetric.cairo
```

System Overview

Spline is a novel DeFi protocol deployed on Starknet and built on top of Ekubo. It harnesses Ekubo's concentrated-liquidity AMM design and extension hook capabilities to optimize liquidity allocation. By concentrating depositor liquidity around a specific tick price following a Cauchy distribution curve, Spline enables highly efficient, like-like token swaps. This concentrated liquidity structure significantly reduces slippage near the trading price, creating a more capital-efficient and user-friendly swap experience. Ultimately, Spline seeks to establish itself as the foundational layer for precise, low-slippage liquidity provisioning for like-like tokens in the Starknet ecosystem.

Core Operations

Pool Creation

Pool creation in Spline is integrated directly with Ekubo through the extension contract (`lp.cairo`). Only this contract has the permission to create Ekubo pools with the Spline extension, ensuring that pool deployment remains secure and controlled. Furthermore, the ability to execute the pool creation function is restricted to the contract owner. This permissioning is critical, as it safeguards the integrity of the Cauchy distribution parameters and the choice of tokens, which are sensitive to the protocol's efficiency and security.

Liquidity Provisioning

Liquidity provisioning in Spline is fully permissionless. Any user can deposit liquidity into a pool, and the extension contract automatically allocates this liquidity across tick prices according to the Cauchy distribution defined by the pool deployer. This ensures that liquidity is highly concentrated around the target trading price, enhancing capital efficiency. In return, liquidity providers receive share tokens, representing their proportional ownership of the pool. These tokens entitle holders to withdraw their share of the underlying liquidity along with accrued trading fees, which are compounded into the pool during any protocol interaction. Both liquidity addition and removal are protected by built-in slippage checks, preventing malicious actors from exploiting price changes to reduce LP value during interactions.

Liquidity Removal

Just like provisioning, liquidity removal is permissionless. Liquidity providers can redeem their share tokens at any time to withdraw their proportional share of the pool's liquidity and fees. Withdrawals are executed with the same slippage protection mechanism, ensuring that LPs are not disadvantaged by sudden or manipulative price movements at the time of removal.

Token Sweeping

Spline also includes a token sweeping function. Sweeping is permissioned exclusively to the contract owner, who can extract ERC-20 tokens that have been mistakenly sent directly to the extension contract. This ensures that no unintended balances remain locked in the system while maintaining clear boundaries for protocol-controlled funds.

Cauchy Liquidity Profile

The Cauchy Liquidity Profile contract governs liquidity allocation using a Cauchy probability distribution. It serves as the main profile in the Spline system, producing smooth liquidity curves that emphasize a central price while preserving presence across the full spectrum. It implements a Cauchy-based formula for its distribution logic, augmented with a constant term, to model liquidity density. The continuous function is discretized into positions using the Symmetric Liquidity Profile, which sets tick boundaries and computes liquidity deltas per range. This also ensures a minimum layer of liquidity from `MIN_TICK` to `MAX_TICK`, providing availability for large or unexpected price swings. This design combines concentrated liquidity efficiency near the target price with stability from full-range support.

Symmetric Liquidity Profile

The Symmetric Liquidity Profile is a modular building block that transforms continuous liquidity distributions into discrete tick ranges. It establishes the geometric structure that profiles, such as the Cauchy Liquidity Profile, use to allocate liquidity positions. It divides the tick range into exponentially widening segments, offering high precision near the center and coarser coverage toward the edges. Within each segment, resolution is controlled by subdividing into bins of size `segment_width / resolution`, ensuring consistent granularity. The component also exposes a `get_bounds_for_liquidity_updates` function, which supplies bounds arrays that guide liquidity placement. Finally, it applies a symmetric design around a configurable center (`tick_start`), mirroring ranges on both sides to maintain balanced distribution across the price spectrum.

Security Model and Trust Assumptions

As noted earlier, the `lp.cairo` contract is central to liquidity management and pool initialization in Spline. However, certain design and implementation choices introduce risks that may impact the economic viability and robustness of deployed pools. These risks should be carefully understood by both developers and liquidity providers before relying on the system in production.

- **Initial share supply economic viability**: In the extension, initial shares are minted based on `initial_liquidity_factor` without any validation. These shares are then permanently locked to prevent inflation attacks. With small values for this factor, the share calculation can open a window to malicious users to execute first depositor attacks, potentially leading to a loss of funds for legitimate users once deployed.
- **Missing gamma validation**: A risk arises from gamma parameter in the Cauchy distribution logic. The profile only validates the sign of gamma but not its magnitude. As a result, a zero-gamma value passes validation but leads to a division-by-zero panic.
- **Owner denial of service**: Protocol fee is transferred to the pool owner during each fee compounding, assuming that the owner can always receive tokens. If the designated owner contract cannot accept tokens, for example, due to incompatible logic, then all pool operations involving fee transfers will fail. This inadvertently creates a denial-of-service scenario for the entire pool, even if liquidity provisioning would otherwise function.
- **Initial tick validation**: The initial tick validation is incomplete. The `create_and_initialize_pool` function does not enforce alignment of the initial tick with the defined tick spacing. This misalignment risk means that pools could be initialized at unintended or invalid prices, introducing subtle inconsistencies in price discovery and liquidity distribution.
- **Token compatibility assumptions**: The protocol is designed to only interact with standard ERC-20 tokens and does not account for tokens with additional behaviors such as transfer fees or rebasing mechanics. Moreover, it is strictly intended for like-like token pairs. Using incompatible tokens or mismatched pairs may result in unpredictable behavior or breakage.
- **Risks associated with Ekubo's integrations**: Since Ekubo has not undergone on-chain verification and its source code is not publicly available, Spline is operating under the assumption that Ekubo's core contracts work as described in their documentation. Should there be any vulnerabilities or unforeseen issues within Ekubo, it could have a direct effect on the functionality of Spline.

High Severity

H-01 Non-Compounded Fees Can Be Stolen

When someone adds or removes liquidity in a pool, fees should be compounded before the operation takes effect. This is indeed how these two operations (i.e., adding and removing liquidity) work. However, compounding fees is a no-op when any of the pool's reserves drops down to 0. A malicious actor can leverage this to drive the pool into a state where fees are not compounded, add a large amount of liquidity, and later withdraw the added funds along with part of the accumulated fees. The attack path would go as follows:

1. A pool with a sufficiently large tick spacing exists and has some fees accumulated that have not yet been compounded.
2. A malicious user (attacker) notices it and takes a flash loan from one of the pool tokens.
3. The attacker executes a sufficiently large swap to move the price tick outside the widest liquidity position, moving the reserve of one of the tokens down to 0.
4. At this stage, the attacker creates a huge addition of liquidity without triggering the compounding of fees due to one of the reserves being 0.
5. The attacker moves the pool's price tick to its previous position by executing a swap in the opposite direction.
6. The attacker withdraws the liquidity that they had deposited. At this point, since both reserves are non-zero, fees are compounded before the removal of funds. Hence, the proportional amount of fees will be extracted by the attacker.
7. The attacker returns the flash loan.

Notice that this attack needs the tick spacing to meet the following requirements:

- **It should be different from 1**: Pools with a tick spacing of 1 have the widest liquidity position with `MAX_TICK` and `MIN_TICK` as bounds. Hence, it is not possible to move the pool's price tick outside of the widest position.
- **It should be sufficient**: As explored in the fork tests, moving the price of a pool to an extreme case can be really gas expensive. Depending on the granularity of the tick spacing, if it is not big enough, the loop can run out of gas and make the attack unfeasible.

Consider compounding fees in the `before_swap` hook in order to prevent the above-described attack.

Update: *Resolved in commit 4edcef5.*

Low Severity

L-01 Incomplete Liquidity Coverage Due to Missing Final Bounds Segment

In the implementation of the `get_bounds_for_liquidity_updates` function, there is an oversight that results in the omission of the final liquidity segment when `ticks.upper > tick_max + dt`. Specifically, the following sequence occurs:

1. The upper tick value is adjusted to `tick_max + dt`
2. The loop is prematurely terminated with a break statement, without appending the adjusted ticks to the bounds list.
3. Consequently, the final liquidity segment defined by `ticks.lower, tick_max + dt` is not included in the bounds.

This oversight leads to a scenario where the liquidity intended to cover the upper price range of the pool is not fully accounted for, resulting in a gap in the liquidity coverage. This gap could potentially affect the efficiency and effectiveness of liquidity utilization within the pool, especially in scenarios where the price approaches or exceeds the upper boundary.

Consider ensuring that the adjusted bounds are appended to the bounds list before the loop is exited.

Update: *Acknowledged, not resolved.*

Notes & Additional Information

N-01 Missing Documentation

Throughout the codebase, multiple instances of missing documentation were identified.

Missing docstrings hinder reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions and events, including their parameters and return values, that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well.

Update: *Acknowledged, not resolved.*

N-02 Missing Event Emissions

Critical operations lack event emissions, preventing off-chain monitoring. Emitting events provides a convenient way for external entities to observe and react to changes in the contract's state. The following instances have been identified as relevant for emitting events:

- `set_liquidity_profile`
- `create_and_initialize_pool`
- `sweep`

Consider emitting events for all state-affecting operations.

Update: *Acknowledged, not resolved.*

N-03 Redundant Storage Read During Pool Initialization

In the pool initialization process, an ERC-20 token is created, and its address is stored in the `pool_token` variable. This step is crucial for setting up the liquidity pool's associated token. However, towards the end of the initialization function, the protocol performs an additional step where it reads the `pool_token` address from storage to mint tokens. This is unnecessary since the address is already available in memory due to the initial token deployment step.

Consider removing the storage read for `pool_token` and utilizing the value already available in memory.

Update: *Acknowledged, not resolved.*

Conclusion

Spline introduces a Cauchy-distribution–based liquidity profile leveraging Ekubo's concentrated liquidity AMM design and extension architecture to deliver capital-efficient, low-slippage swaps for like-like tokens. The design provides permissionless liquidity provisioning and removal, empowering broad participation, while built-in slippage protections strengthen user safety. The modular liquidity profiles, particularly the Cauchy and Symmetric designs, ensure that liquidity is not only concentrated near the active price but also available across the full spectrum for stability during volatile movements.

The codebase was found to be well written and well documented. Several changes were suggested to improve the clarity of the codebase and facilitate future audits, integrations, and development. That said, several areas raise important concerns, such as risks that may impact the economic viability and robustness of deployed pools due to implementation choices. In addition, reliance on Ekubo's unverified core contracts introduces an external layer of uncertainty that could directly affect Spline's reliability.

The Spline team is appreciated for being highly responsive to the questions posed by the audit team and sharing comprehensive documentation about the project.