Security Review Report
NM-0536 Spline

NETHERMIND
SECURITY

(July 1, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for Spline. The engagement focused on Spline's liquidity profile framework, which is built as an extension to the Ekubo DEX protocol. Unlike standard AMM implementations, Spline implements a Cauchy distribution-based liquidity profile to create smooth liquidity curves across price ranges. The **LiquidityProvider** contract serves as the main interface, allowing users to create pools, add/remove liquidity, and manage positions using the Cauchy mathematical distribution.

The **CauchyLiquidityProfile** contract implements the core liquidity distribution logic, using the Cauchy probability distribution to determine how liquidity is allocated across different price ticks. The system uses the component **SymmetricLiquidityProfileComponent** to handle discrete tick range management, while the Cauchy profile applies the mathematical distribution formula to calculate actual liquidity amounts at each tick.

The audit comprises 897 lines of Cairo code. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 5 points of attention, where one is classified as `Critical`, one is `Medium`, one is `Low`, and two are `Informational` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

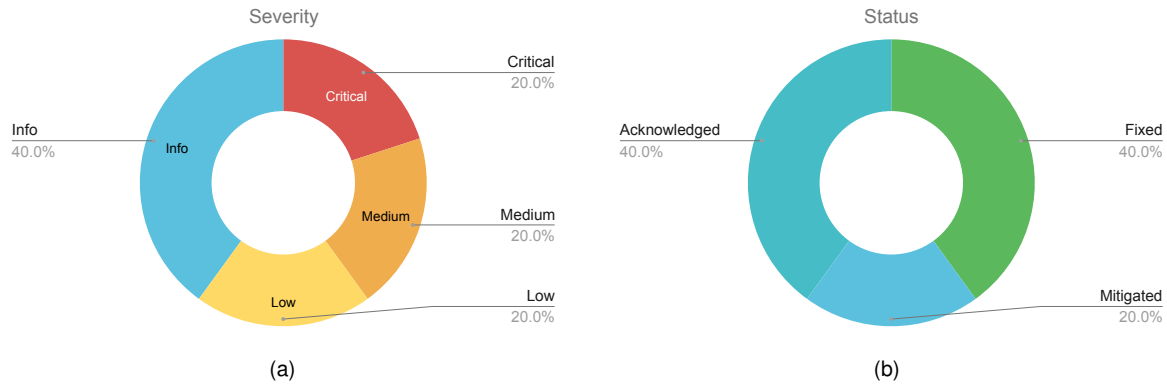

Fig. 1: Distribution of issues: Critical (1), **High** (0), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (0). **Distribution of status: Fixed** (2), **Acknowledged** (2), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 2, 2025
Final Report	July 1, 2025
Repositories	spline-v0
Initial Commit	ec3337d
Final Commit	d116e66
Documentation	HackMD
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/lp.cairo	553	82	14.8%	102	737
2	src/shared_locker.cairo	18	1	5.6%	1	20
3	src/lib.cairo	19	0	0.0%	0	19
4	src/sweep.cairo	29	6	20.7%	4	39
5	src/math.cairo	8	2	25.0%	3	13
6	src/token.cairo	39	13	33.3%	11	63
7	src/profile.cairo	21	8	38.1%	4	33
8	src/profiles/cauchy.cairo	138	22	15.9%	23	183
9	src/profiles/symmetric.cairo	67	23	34.3%	12	102
10	src/profiles/bounds.cairo	5	1	20.0%	0	6
	Total	**897**	**158**	**17.6%**	**160**	**1215**

3 Summary of Issues

	Finding	Severity	Update
1	Initial locked liquidity can be withdrawn by anyone which causes permanent DoS for the pool	Critical	Fixed
2	Lack of slippage protection in `add_liquidity` and `remove_liquidity`	Medium	Fixed
3	Not enough validation on `params` for `_set_grid_for_bounds(...)`	Low	Mitigated
4	Pool initialization `initial_tick` may mismatch profile's `tick_start`	Info	Acknowledged
5	The `try_call_core_with_callback(...)` function reverts on deserialization failure, defeating its purpose of safely handling call errors	Info	Acknowledged

4 System Overview

The Spline protocol is a sophisticated liquidity management framework engineered as an extension to the Ekubo DEX protocol. At its core, the `LiquidityProvider` contract deviates from standard AMM liquidity provision models. Instead of using discrete price ranges or uniform distributions, it functions as a coordinator that delegates liquidity distribution logic to specialized mathematical profile contracts. These interactions are orchestrated through a standardized `ILiquidityProfile` interface.

The current implementation focuses on Cauchy distribution-based liquidity profiles, which create smooth, continuous liquidity curves that concentrate around a center price while maintaining liquidity across the full price spectrum.

The system operates as an Ekubo extension. The `LiquidityProvider` contract maintains pool state, manages ERC20 LP tokens for each pool, and handles fee collection and compounding, while delegating the mathematical complexity of liquidity distribution to the profile contracts.

The following diagram provides a high-level illustration of the Spline architecture.

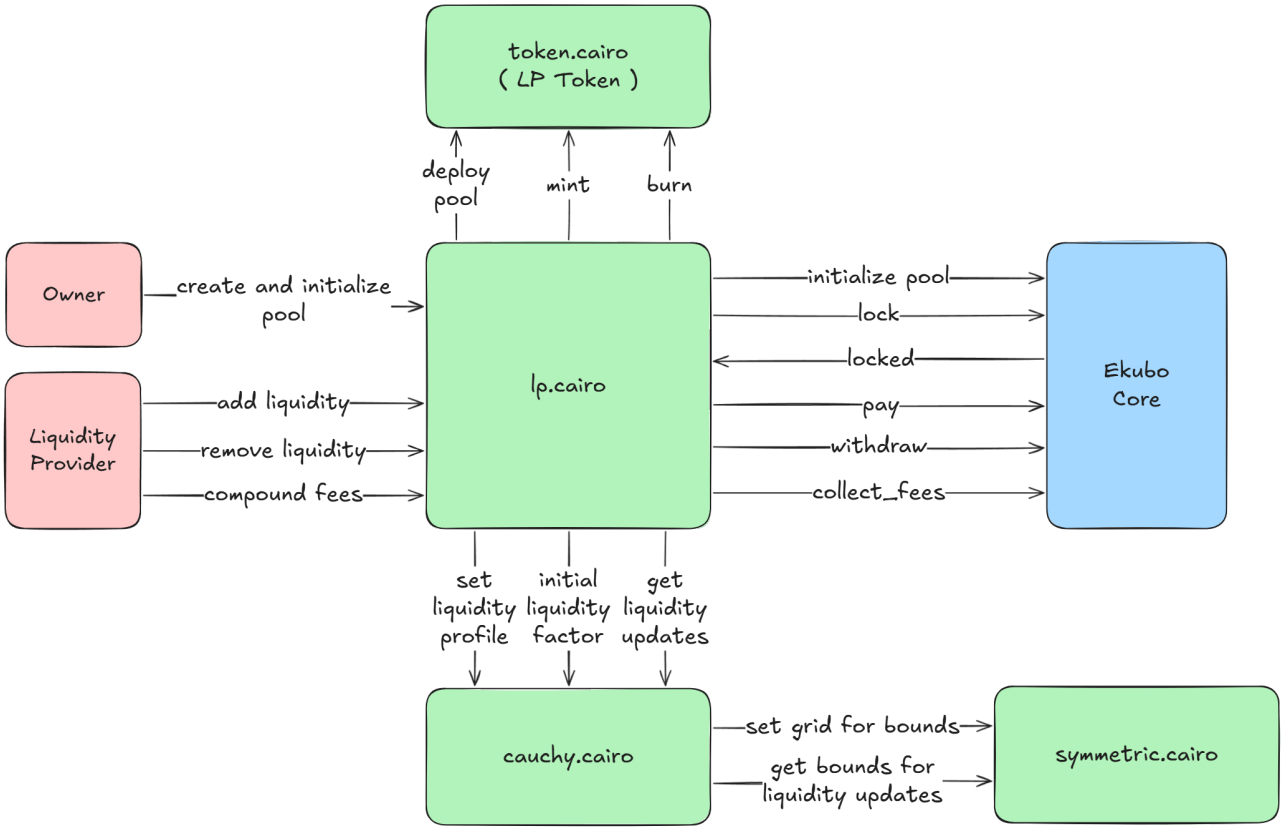

Fig. 2: Spline overview

4.1 System Components

`LiquidityProvider`

The `LiquidityProvider` contract serves as the main entry point and coordinator for all liquidity management operations within the Spline protocol. Operating as an Ekubo extension, it manages the complete lifecycle of liquidity pools while delegating mathematical distribution logic to specialized profile contracts.

- **Pool Management**: Creates and initializes pools on Ekubo, deploying dedicated ERC20 LP tokens for each pool and managing pool state including liquidity factors and reserves.
- **Liquidity Operations**: Handles adding and removing liquidity by converting between user-provided liquidity factors and ERC20 LP token shares, with automatic fee compounding on each operation.
- **Profile Integration**: Routes all liquidity distribution calculations to the configured `ILiquidityProfile` implementation, which determines how liquidity is allocated across price ranges.
- **Fee Management**: Automatically collects swap fees from Ekubo positions, retains 50% as protocol fees, and compounds the remainder back into the liquidity factor to benefit LP token holders.

- **Access Control**: Restricts pool creation to the contract owner while allowing permissionless liquidity provision and removal for existing pools.

`CauchyLiquidityProfile`

The `CauchyLiquidityProfile` contract implements the mathematical logic for distributing liquidity according to a Cauchy probability distribution. It serves as the primary profile implementation in the current Spline system, creating smooth liquidity curves that concentrate around a center price while maintaining some liquidity across the full price spectrum.

- **Mathematical Distribution**: Implements the Cauchy distribution formula supplemented by a constant base layer of full range liquidity to calculate liquidity amounts at specific ticks, creating heavy-tailed distributions with concentrated center liquidity.

$$l_{\gamma,\rho}(\tau) = \frac{L}{\pi\gamma}\left[\frac{1}{1+(\tau/\gamma)^2} + \frac{1}{1+(\rho/\gamma)^2}\right]$$

  Converts the continuous mathematical distribution into discrete liquidity positions by leveraging the `SymmetricLiquidityProfileComponent` to determine tick boundaries and calculating appropriate liquidity deltas for each range.

- **Full-Range Base Liquidity**: Provides a constant base liquidity layer across the entire tick range (from `MIN_TICK` to `MAX_TICK`) to ensure some liquidity is always available for large price movements.
- **Access Control**: Restricts profile parameter configuration to the associated `LiquidityProvider` contract through the `pool_key.extension` validation mechanism.

`SymmetricLiquidityProfileComponent`

The `SymmetricLiquidityProfileComponent` serves as a reusable component that handles the discretization of continuous liquidity distributions into manageable tick ranges. It provides the geometric framework that the Cauchy profile uses to determine where to place liquidity positions.

- **Exponential Segmentation**: Partitions the full tick range into segments with exponentially increasing widths: $[0, s)$, $[s, 2s)$, $[2s, 4s)$, $[4s, 8s)$, etc., where each subsequent segment doubles in size. This creates finer granularity near the center and progressively coarser granularity toward the edges.
- **Configurable Resolution**: Each segment is subdivided into a fixed number of bins equal to the `resolution` parameter, so the step size within each segment is $\frac{\text{segment\_width}}{\text{resolution}}$.
- **Bounds**: Provides the `get_bounds_for_liquidity_updates` function that returns an array of `Bounds` structures, which the Cauchy profile uses to determine where to place liquidity positions.
- **Symmetric Design**: Creates symmetric tick ranges around a configurable center point (`tick_start`), ensuring balanced liquidity distribution on both sides of the expected price range by mirroring the segmentation pattern in both positive and negative directions.

4.2 Security Assumptions

The security posture of the Spline protocol relies on the following assumptions:

- **Ekubo Protocol Integrity**: As Ekubo is a closed-source protocol, Spline assumes that the Ekubo core contracts function as intended and as described in their documentation. Any vulnerabilities or unexpected behaviors in Ekubo could directly impact Spline's operations.
- **Poor Swap Execution at Low Liquidity**: At extremely low pool liquidity levels, swaps may receive significantly reduced output amounts when price movements reach the tail ends of the Cauchy distribution, effectively creating regions where theoretical liquidity exists but practical execution yields poor results.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Initial locked liquidity can be withdrawn by anyone which causes permanent DoS for the pool

File(s): src/sweep.cairo

Description: When a new pool is created, initial liquidity is provided. The corresponding LP shares representing this initial liquidity are minted to the LP contract itself and are intended to be locked indefinitely. This is shown in the create_and_initialize_pool(...) function:

```
fn create_and_initialize_pool(...) {
    // ...
    let shares = initial_liquidity_factor.try_into().unwrap();
    call_core_with_callback::<
        (PoolKey, i129, i129, u256, ContractAddress), (),
    >(core, @(pool_key, liquidity_factor_delta, Zero::<i129>::zero(), shares, caller));

    // lock initial minted lp tokens forever in this contract
    let pool_token = self.pool_tokens.read(pool_key);
    // @audit-issue Initial LP tokens, representing the first liquidity, are minted to this contract.
    ILiquidityProviderTokenDispatcher { contract_address: pool_token }
        .mint(get_contract_address(), shares);
}
```

This initial locked liquidity ensures that the pool's total_factor (total liquidity factor) is non-zero. The total_factor is crucial for share calculations, as demonstrated in the calculate_shares(...) function, which asserts total_factor > 0:

```
fn calculate_shares(...) -> u256 {
    // @audit-issue This assertion requires total_factor to be positive. If it becomes '0', subsequent liquidity
    ↪  additions will fail.
    assert(total_factor > 0, 'Total factor is 0');
    let denom: u256 = total_factor.try_into().unwrap();
    let num: u256 = factor.try_into().unwrap();
    let shares: u256 = muldiv(total_shares, num, denom);
    shares
}
```

The problem arises because the LP contract embeds a SweepableComponent which exposes a sweep(...) function. This function allows anyone to withdraw any ERC20 tokens held by the LP contract.

```
#[embeddable_as(Sweepable)]
impl SweepableImpl<...> of super::ISweepable<ComponentState<TContractState>> {
    fn sweep(...) {
        let balance = IERC20Dispatcher { contract_address: token }
            .balance_of(get_contract_address());
        // @audit-issue The function checks for minimum balance but does not restrict which token can be swept or who
        ↪  can call it.
        assert(balance >= amount_min, 'Insufficient balance');
        IERC20Dispatcher { contract_address: token }.transfer(recipient, balance);
    }
}
```

Crucially, the sweep(...) function does not prevent the pool's own LP tokens (those representing the initial locked liquidity) from being swept. An attacker can call sweep(...), specifying the pool's LP token address as the token and their own address as the recipient. This allows them to take ownership of the initially locked LP shares.

Once the attacker possesses these LP shares, they can call remove_liquidity(...) to withdraw the actual underlying initial liquidity from the pool. This action reduces the pool's liquidity_factor_delta and subsequently the total_factor to 0. As a result, the assert(total_factor > 0, 'Total factor is 0') in calculate_shares(...) will fail for all future attempts to add liquidity, effectively causing a permanent Denial of Service (DoS) for the affected pool. All subsequent operations that depend on adding liquidity will become impossible.

The following Proof of Concept demonstrates the attack:

```
#[test]
#[fork("mainnet")]
#[should_panic(expected: ('Total factor is 0',))]
fn test_poc_foo() {
    let (pool_key, lp, _, _, default_profile_params, token0, token1) = setup_add_liquidity();
    let initial_liquidity_factor = lp.pool_liquidity_factor(pool_key);
    assert_eq!(initial_liquidity_factor, 100000000000000000);

    // @audit Attacker (bob) identifies the pool's LP token and its balance held by the LP contract.
    let pool_token = IERC20Dispatcher { contract_address: lp.pool_token(pool_key) };
    let locked_shares = pool_token.balance_of(lp.contract_address);
    let lp_sweep = ISweepableDispatcher { contract_address: lp.contract_address };

    // @audit Attacker calls sweep(...) to transfer the locked LP shares to themselves.
    cheat_caller_address(lp.contract_address, 123.try_into().unwrap(), CheatSpan::TargetCalls(1));
    lp_sweep.sweep(pool_token.contract_address, 123.try_into().unwrap(), locked_shares);

    // @audit Attacker calls remove_liquidity(...) with the stolen LP shares, draining initial liquidity.
    cheat_caller_address(lp.contract_address, 123.try_into().unwrap(), CheatSpan::TargetCalls(1));
    lp.remove_liquidity(pool_key, locked_shares);

    // @audit A legitimate user attempts to add liquidity to the pool.
    let step = *default_profile_params[2];
    let n = *default_profile_params[3];
    let factor = 100000000000000000000; // 100 * 1e18
    let amount: u128 = (step.mag * n.mag * (factor)) / (1900000);
    token0.transfer(lp.contract_address, amount.into());
    token1.transfer(lp.contract_address, amount.into());

    // @audit-issue This call will now panic because total_factor is '0', causing a DoS.
    lp.add_liquidity(pool_key, factor);
}
```

Recommendation(s): Consider preventing the sweeping of the initial, locked LP shares specific to any pool managed by this contract.

Status: Fixed

Update from the client: Fixed in e423418, 1993c7f and 7b390f5

Update from the Nethermind Security Team: The `safe_transfer_from(...)` in `shared_locker` doesn't handle the case for the tokens which returns `false` and doesn't revert in the case of error.

Update from the client: Fixed in 29844dc

6.2 [Medium] Lack of slippage protection in `add_liquidity` and `remove_liquidity`

File(s): src/lp.cairo

Description: The `add_liquidity(...)` function enables users to deposit assets into the underlying liquidity pool. In return, users receive pool shares proportional to their contributed liquidity relative to the total pool size.

```
fn add_liquidity(ref self: ContractState, pool_key: PoolKey, factor: u128) -> u256 {
    // compound fees if possible. also checks pool key and pool initialized
    self.compound_fees(pool_key);

    // calculate shares to mint
    let liquidity_factor_delta = i129 { mag: factor, sign: false };
    self.check_liquidity_factor_delta(pool_key, liquidity_factor_delta);

    let pool_token = self.pool_tokens.read(pool_key);
    let total_shares = IERC20Dispatcher { contract_address: pool_token }.total_supply();

    let liquidity_factor = self.pool_liquidity_factors.read(pool_key);
    let shares = self.calculate_shares(total_shares, factor, liquidity_factor);

    // add amount to liquidity factor in storage
    let new_liquidity_factor = liquidity_factor + factor;
    self.pool_liquidity_factors.write(pool_key, new_liquidity_factor);

    // obtain core lock. should also effectively lock this contract for unique pool key
    let core = self.core.read();
    let caller = get_caller_address();
    call_core_with_callback::<
        (PoolKey, i129, i129, u256, ContractAddress),
        ()
    >(
        core,
        @(
            pool_key,
            liquidity_factor_delta,
            Zero::<i129>::zero(),
            shares,
            caller
        )
    );

    // mint pool token shares to caller
    ILiquidityProviderTokenDispatcher { contract_address: pool_token }.mint(caller, shares);

    shares
}
```

The current implementation lacks any form of slippage protection. In the `add_liquidity(...)` function, the number of pool `shares` a user receives is determined by the `factor` they provide and the pool's state (specifically `total_shares` and current `liquidity_factor`) at the time of execution via the `self.calculate_shares(...)` call. However, this state can change due to other transactions (e.g., deposits or withdrawals by other users) that are processed between the user submitting their transaction and its actual execution on-chain. This timing difference exposes the user to potential slippage: they might receive fewer pool `shares` than they expected for their deposited liquidity if the pool's total liquidity or composition changed unfavorably in the interim.

A similar vulnerability exists in the `remove_liquidity(...)` function. When a user removes liquidity, the amount of underlying assets they receive is calculated based on the pool's state at that moment. If other users' transactions alter the pool composition or value just before the withdrawal is processed, the withdrawing user might receive fewer underlying assets than anticipated for their redeemed pool `shares`.

Recommendation(s): Consider allowing users to specify a minimum acceptable number of shares (for `add_liquidity(...)`) or a minimum amount of underlying assets (for `remove_liquidity(...)`) they are willing to receive. The functions should then revert if the calculated amount falls below this user-defined threshold due to market movements.

Status: Fixed

Update from the client: Fixed in d3ff60e

6.3 [Low] Not enough validation on `params` for `_set_grid_for_bounds(...)`

File(s): src/profiles/symmetric.cairo

Description: When a pool is created and initialized, the `_set_grid_for_bounds(...)` is used to set the parameters for calculating bounds for a symmetric liquidity profile.

```
fn _set_grid_for_bounds(
    ref self: ComponentState<TContractState>, pool_key: PoolKey, params: Span<i129>,
) {
    assert(params.len() == 4, 'Invalid params length');
    assert(!*params[0].sign, 'Invalid grid s');
    assert(!*params[1].sign, 'Invalid grid resolution');

    let (s, res, tick_start, tick_max) = (
        *params[0].mag, *params[1].mag, *params[2], *params[3],
    );

    // @audit-issue It doesn't ensure res to be power of 2 rather it ensures res to be an even number.
    assert((res > 0 && (res % 2 == 0)), 'resolution must be power of 2');
    assert(tick_start < tick_max, 'tick_start must be < tick_max');
    assert(s > 0 && (s % pool_key.tick_spacing == 0), 's must divide by tick_spacing');
    // @audit-issue step is not 2*s/res rather step is s/res, so this check is incorrect.
    assert((2 * s) / res != 0, 'step must be non-zero');

    self.grid.write(pool_key, (s, res, tick_start, tick_max));
}
```

The `_set_grid_for_bounds(...)` function contains several asserts to ensure that the `params` are valid. However, some of these asserts are incorrect, and additional validations are missing.

Specifically, the following asserts are incorrect:

- To assert that "resolution must be power of 2", the current check `res > 0 (res % 2 == 0)` only ensures `res` is an even number greater than `0`. A correct check for `res` being a power of `2` and greater than `0` would be `res > 0 (res (res - 1)) == 0`;
- To assert that "step must be non-zero", the current check uses `(2 * s) / res != 0`. The actual step in the calculation is `s / res`. Therefore, even if the current assert passes, the true step (`s / res`) could still be `0`, leading to incorrect behavior;

Furthermore, the following crucial conditions are not asserted:

- The `tick_start` and `tick_max` values should be valid ticks, meaning they must be divisible by `tick_spacing`;
- The bounds calculated by `get_bounds_for_liquidity_updates(...)` should always fall within the valid tick range of `[MIN_TICK, MAX_TICK]`, i.e., `[-88722883, 88722883]`. The current logic does not prevent a combination of `s / res` (step), `tick_start`, and `tick_max` from resulting in bounds outside this valid range;
- The `s` should be divisible by `res` else the segments won't be divided into equally spaced bins resulting in the case where the check `ticks.upper == next.upper` can be missed in the `get_bounds_for_liquidity_updates(...)`;
- The step, which is `s / res`, should also be divisible by `tick_spacing` to ensure that the calculated bounds align with valid tick positions. If it's not, the bounds could point to invalid ticks;

Recommendation(s): Consider fixing the incorrect asserts and adding the missing asserts to ensure the `params` lead to a valid grid configuration.

Status: Mitigated

Update from the client: Fixed in 5796e4e

Update from the Nethermind Security Team: There are 2 specific checks to ensure that `tick_start` and `tick_max` are within the range of `[MIN_TICK, MAX_TICK]`. These 2 checks aren't enough. It should be ensured that all the ticks calculated in the `get_bounds_for_liquidity_updates(...)` by adding/subtracting with steps in the while loop are valid ticks.

Update from the client: Fixed in ae0ed13

Update from the Nethermind Security Team: This will revert even for certain valid configurations. For example, if the configurations are as follows:

- TICK RANGE = [-460, 460];
- dt = 4;
- tick_start = -100;
- tick_max = 280;
- s = 32;
- res = 8;

Then, the last bound will be [-452, 256]. But `tick_min` will be `-480`, which is less than `-460`. Thus, it will revert even though the lower tick of the last bound, i.e., -452, is a valid tick.

Update from the client: This is fine since it wouldn't revert when the pool is live, so basically just excludes certain configs.

6.4 [Info] Pool initialization `initial_tick` may mismatch profile's `tick_start`

File(s): `src/lp.cairo`

Description: The `create_and_initialize_pool(...)` function is responsible for setting up a new liquidity pool. It accepts `initial_tick` and `profile_params` as parameters. The `initial_tick` is used to initialize the pool on the Ekubo core. The `profile_params` array is used to configure the liquidity profile associated with the pool, where `profile_params[2]` represents `tick_start` and `profile_params[5]` represents `mu` for the Cauchy distribution.

The `cauchy::set_liquidity_profile(...)` function, called during pool creation, correctly asserts that `mu` (`profile_params[5]`) is equal to `tick_start` (`profile_params[2]`).

```
fn set_liquidity_profile(ref self: ContractState, pool_key: PoolKey, params: Span<i129>) {
    // ...
    // @audit This asserts that mu (params[5]) equals tick_start (params[2]).
    assert(
        *params[5].mag == *params[2].mag && *params[5].sign == *params[2].sign,
        'mu != symmetric::tick_start',
    );
    // ...
}
```

However, there is no corresponding validation in `lp::create_and_initialize_pool(...)` to ensure that the `initial_tick` (passed as a direct argument) is equal to `tick_start` (i.e., `profile_params[2]`).

```
fn create_and_initialize_pool(
    ref self: ContractState,
    pool_key: PoolKey,
    initial_tick: i129,         // @audit Used to initialize the pool on core.
    profile_params: Span<i129>, // @audit profile_params[2] is tick_start.
) {
    // ...
    // @audit profile_params are set, which includes tick_start (profile_params[2]).
    profile.set_liquidity_profile(pool_key, profile_params);
    // ...
    // @audit Pool is initialized with initial_tick.
    // @audit-issue No check ensures initial_tick matches profile_params[2] (tick_start).
    core.initialize_pool(pool_key, initial_tick);

    // ...
}
```

This omission means that the pool can be initialized on the Ekubo core with an `initial_tick` that is different from the `tick_start` value defined in its liquidity profile parameters.

Recommendation(s): Consider adding a validation step within the `lp::create_and_initialize_pool(...)` function to ensure that the provided `initial_tick` argument is equal to `profile_params[2]` (the `tick_start` value from the profile parameters).

Status: Acknowledged

Update from the client: The initial tick mismatch with tick start should be fine since it leaves flexibility to initialize pool not at center of liquidity distribution (although unlikely we'll ever do that).

6.5 [Info] The `try_call_core_with_callback(...)` function reverts on deserialization failure, defeating its purpose of safely handling call errors

File(s): src/shared_locker.cairo

Description: The `try_call_core_with_callback(...)` function is intended to make a call to an external `core` contract and handle potential errors without reverting the entire transaction. It uses a `match` statement on the `call_result` to differentiate between successful calls (`Result::Ok`) and failed calls (`Result::Err`). In the case of a failed call, it correctly returns `Option::None`.

However, if the external call to `core.contract_address` is successful (i.e., `Result::Ok`), the function then attempts to deserialize the `output_span` using `Serde::deserialize(ref output_span).expect('DESERIALIZE_RESULT_FAILED')`. If this deserialization fails for any reason (e.g., unexpected data format returned by the `core` contract), the `.expect(...)` call will cause the entire transaction to revert with the message `'DESERIALIZE_RESULT_FAILED'`. This behavior contradicts the function's apparent goal of catching errors from the external call and its aftermath without causing a full revert.

The problem lies in this line:

```
pub fn try_call_core_with_callback<TInput, TOutput, +Serde<TInput>, +Serde<TOutput>>(
    core: ICoreDispatcher, input: @TInput,
) -> Option<TOutput> {
    let data = serialize(input).span();
    let mut calldata = serialize(@data).span();
    // TODO: is this valid for ekubo core with callback into lp? to bypass full tx reverts
    let call_result = call_contract_syscall(core.contract_address, selector!("lock"), calldata);
    match call_result {
        Result::Ok(mut output_span) => {
            // @audit-issue If Serde::deserialize fails, the .expect() call will revert the transaction.
            Option::Some(Serde::deserialize(ref output_span).expect('DESERIALIZE_RESULT_FAILED'))
        },
        Result::Err(_) => { Option::None(()) },
    }
}
```

This means that even if the call to the `core` contract itself does not revert, a subsequent failure in processing its return data can still lead to a revert, which might be an unintended behavior for a "try_call" type of function.

Recommendation(s): Consider modifying the deserialization step to handle potential errors gracefully.

Status: Acknowledged

Update from the client: The try/call reverting on deserialization should be ok for spline purposes given more so intended usage is to catch any rounding issues on harvest fees. Deserialization errors on our end should revert as a bug.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Spline documentation

The `Spline` team has provided a comprehensive walkthrough of the project in the kick-off call. The HackMD page and the code comments include the explanation of the intended functionalities. Moreover, the team addressed the questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
> scarb build
   Compiling spline_v0 v0.1.4 (/.../Scarb.toml)
warn: Unused import: `spline_v0::lp::LiquidityProvider::IMathLibDispatcher`
 --> /.../src/lp.cairo:67:9
       IMathLibDispatcher, IMathLibDispatcherTrait, dispatcher as math_lib_dispatcher,
       ^^^^^^^^^^^^^^^^^^

warn: Usage of deprecated feature `"corelib-internal-use"` with no `#[feature("corelib-internal-use")]` attribute.
 ↪  Note: "Use `core::num::traits::WideMul` instead"
 --> /.../src/math.cairo:1:21
use core::integer::{u256_wide_mul, u512, u512_safe_div_rem_by_u256};
                   ^^^^^^^^^^^^

warn: Unused import: `spline_v0::test::test_wrapped_token::TestWrappedToken::IERC20MetadataDispatcher`
 --> /.../src/test/test_wrapped_token.cairo:14:9
       IERC20MetadataDispatcher, IERC20MetadataDispatcherTrait,
       ^^^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0::test::test_wrapped_token::TestWrappedToken::IERC20MetadataDispatcherTrait`
 --> /.../src/test/test_wrapped_token.cairo:14:35
       IERC20MetadataDispatcher, IERC20MetadataDispatcherTrait,
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    Finished `dev` profile target(s) in 63 seconds
```

8.2 Tests Output

```
> snforge test
[WARNING] Package snforge_std version does not meet the recommended version requirement ^0.43.1, it might result in
 ↪  unexpected behaviour
   Compiling test(spline_v0_unittest) spline_v0 v0.1.4 (/.../Scarb.toml)
warn: Unused import: `spline_v0::lp::LiquidityProvider::IMathLibDispatcher`
 --> /.../src/lp.cairo:67:9
       IMathLibDispatcher, IMathLibDispatcherTrait, dispatcher as math_lib_dispatcher,
       ^^^^^^^^^^^^^^^^^^

warn: Usage of deprecated feature `"corelib-internal-use"` with no `#[feature("corelib-internal-use")]` attribute.
 ↪  Note: "Use `core::num::traits::WideMul` instead"
 --> /.../src/math.cairo:1:21
use core::integer::{u256_wide_mul, u512, u512_safe_div_rem_by_u256};
                   ^^^^^^^^^^^^

warn: Unused import: `spline_v0::test::test_wrapped_token::TestWrappedToken::IERC20MetadataDispatcher`
 --> /.../src/test/test_wrapped_token.cairo:14:9
       IERC20MetadataDispatcher, IERC20MetadataDispatcherTrait,
       ^^^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0::test::test_wrapped_token::TestWrappedToken::IERC20MetadataDispatcherTrait`
 --> /.../src/test/test_wrapped_token.cairo:14:35
       IERC20MetadataDispatcher, IERC20MetadataDispatcherTrait,
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

   Compiling test(spline_v0_integrationtest) spline_v0_integrationtest v0.1.4 (/.../Scarb.toml)
warn: Unused import: `spline_v0_integrationtest::debug_test::ICoreDispatcherTrait`
 --> /.../tests/debug_test.cairo:3:48
use ekubo::interfaces::core::{ICoreDispatcher, ICoreDispatcherTrait, UpdatePositionParameters};
                                               ^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::UpdatePositionParameters`
 --> /.../tests/debug_test.cairo:3:70
use ekubo::interfaces::core::{ICoreDispatcher, ICoreDispatcherTrait, UpdatePositionParameters};
                                                                     ^^^^^^^^^^^^^^^^^^^^^^^^
```

```
warn: Unused import: `spline_v0_integrationtest::debug_test::PositionKey`
 --> /.../tests/debug_test.cairo:4:35
use ekubo::types::keys::{PoolKey, PositionKey};
                                  ^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::ContractClass`
 --> /.../tests/debug_test.cairo:7:5
    ContractClass, ContractClassTrait, DeclareResultTrait, EventSpyAssertionsTrait, declare,
    ^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::ContractClassTrait`
 --> /.../tests/debug_test.cairo:7:20
    ContractClass, ContractClassTrait, DeclareResultTrait, EventSpyAssertionsTrait, declare,
                   ^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::EventSpyAssertionsTrait`
 --> /.../tests/debug_test.cairo:7:60
    ContractClass, ContractClassTrait, DeclareResultTrait, EventSpyAssertionsTrait, declare,
                                                           ^^^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::spy_events`
 --> /.../tests/debug_test.cairo:8:5
    spy_events, start_cheat_caller_address, stop_cheat_caller_address,
    ^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::debug_test::ClassHash`
 --> /.../tests/debug_test.cairo:16:16
use starknet::{ClassHash, ContractAddress, contract_address_const, get_contract_address};
               ^^^^^^^^^

warn: Usage of deprecated feature `"deprecated-starknet-consts"` with no `#[feature("deprecated-starknet-consts")]`
↪  attribute. Note: "Use `TryInto::try_into` in const context instead."
 --> /.../tests/debug_test.cairo:16:44
use starknet::{ClassHash, ContractAddress, contract_address_const, get_contract_address};
                                           ^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::cauchy_test::ILiquidityProfile`
 --> /.../tests/cauchy_test.cairo:15:5
    ILiquidityProfile, ILiquidityProfileDispatcher, ILiquidityProfileDispatcherTrait,
    ^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::cauchy_test::CauchyLiquidityProfile`
 --> /.../tests/cauchy_test.cairo:17:34
use spline_v0::profiles::cauchy::CauchyLiquidityProfile;
                                 ^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::cauchy_test::ILiquidityProviderTokenDispatcher`
 --> /.../tests/cauchy_test.cairo:19:24
use spline_v0::token::{ILiquidityProviderTokenDispatcher, ILiquidityProviderTokenDispatcherTrait};
                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::cauchy_test::ILiquidityProviderTokenDispatcherTrait`
 --> /.../tests/cauchy_test.cairo:19:59
use spline_v0::token::{ILiquidityProviderTokenDispatcher, ILiquidityProviderTokenDispatcherTrait};
                                                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warn: Usage of deprecated feature `"deprecated-starknet-consts"` with no `#[feature("deprecated-starknet-consts")]`
↪  attribute. Note: "Use `TryInto::try_into` in const context instead."
 --> /.../tests/cauchy_test.cairo:20:44
use starknet::{ClassHash, ContractAddress, contract_address_const, get_contract_address};
                                           ^^^^^^^^^^^^^^^^^^^^^^

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
       let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                          ^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:35
       let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                                 ^^^^^^
note: this error originates in the attribute macro: `fork`
```

```
warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                           ^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:35
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                                  ^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                           ^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:35
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                                  ^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                           ^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:35
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                                  ^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                           ^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/cauchy_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:28
        let (pool_key, lp, owner, cauchy, params, token0, token1) = setup_with_liquidity_provider();
                           ^^^^^
note: this error originates in the attribute macro: `fork`

warn: Usage of deprecated feature `"deprecated-starknet-consts"` with no `#[feature("deprecated-starknet-consts")]`
↪  attribute. Note: "Use `TryInto::try_into` in const context instead."
 --> /.../tests/lp_test.cairo:28:44
use starknet::{ClassHash, ContractAddress, contract_address_const, get_contract_address};
                                           ^^^^^^^^^^^^^^^^^^^^^^

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:31
        let (pool_key, lp, _, profile, default_profile_params, token0, token1) = setup();
                              ^^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:31
        let (pool_key, lp, _, profile, default_profile_params, token0, token1) = setup();
                              ^^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:58
        let (pool_key, lp, _, _, default_profile_params, token0, token1) = setup_remove_liquidity();
                                                         ^^^^^^
note: this error originates in the attribute macro: `fork`
```

```
warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:20:66
       let (pool_key, lp, _, _, default_profile_params, token0, token1) = setup_remove_liquidity();
                                                                ^^^^^^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:86:9
    let n = *default_profile_params[3];
        ^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork]:86:9
    let n = *default_profile_params[3];
        ^
note: this error originates in the attribute macro: `fork`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork][proc_should_panic]:33:58
       let (pool_key, lp, _, _, default_profile_params, token0, token1) = setup();
                                                        ^^^^^^
note: this error originates in the attribute macro: `should_panic`

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /.../tests/lp_test.cairo[proc_test][proc___internal_config_statement][proc_fork][proc_should_panic]:33:66
       let (pool_key, lp, _, _, default_profile_params, token0, token1) = setup();
                                                                ^^^^^^
note: this error originates in the attribute macro: `should_panic`

warn: Unused import: `spline_v0_integrationtest::symmetric_test::IERC20DispatcherTrait`
 --> /.../tests/symmetric_test.cairo:6:62
use openzeppelin_token::erc20::interface::{IERC20Dispatcher, IERC20DispatcherTrait};
                                                             ^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `spline_v0_integrationtest::symmetric_test::SymmetricLiquidityProfileComponent`
 --> /.../tests/symmetric_test.cairo:11:37
use spline_v0::profiles::symmetric::SymmetricLiquidityProfileComponent;
                                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    Finished `dev` profile target(s) in 61 seconds


Collected 61 test(s) from spline_v0 package
Running 61 test(s) from tests/
[PASS] spline_v0_integrationtest::cauchy_test::test_get_liquidity_updates_with_negative_liquidity_factor (l1_gas: ~0,
↪  l1_data_gas: ~2016, l2_gas: ~13841920)
[PASS] spline_v0_integrationtest::cauchy_test::test_initial_liquidity_factor (l1_gas: ~0, l1_data_gas: ~2016, l2_gas:
↪  ~1521920)
[PASS] spline_v0_integrationtest::cauchy_test::test_get_liquidity_updates_with_positive_liquidity_factor (l1_gas: ~0,
↪  l1_data_gas: ~2016, l2_gas: ~13841920)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_adds_liquidity_to_pool (l1_gas: ~0, l1_data_gas: ~6880,
↪  l2_gas: ~103399360)
[PASS] spline_v0_integrationtest::cauchy_test::test_description (l1_gas: ~0, l1_data_gas: ~1440, l2_gas: ~1161920)
[PASS] spline_v0_integrationtest::lp_test::test_after_swap_updates_pool_reserves (l1_gas: ~0, l1_data_gas: ~6784,
↪  l2_gas: ~79158400)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_emits_liquidity_updated_event (l1_gas: ~0, l1_data_gas:
↪  ~6880, l2_gas: ~103119360)
[PASS] spline_v0_integrationtest::cauchy_test::test_set_liquidity_profile_updates_storage (l1_gas: ~0, l1_data_gas:
↪  ~2016, l2_gas: ~1681920)
[PASS] spline_v0_integrationtest::lp_test::test_constructor_sets_callpoints (l1_gas: ~0, l1_data_gas: ~2272, l2_gas:
↪  ~1994560)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_emits_liquidity_updated_event (l1_gas: ~0,
↪  l1_data_gas: ~6784, l2_gas: ~46313920)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_fails_if_extension_not_liquidity_provider (l1_gas: ~0,
↪  l1_data_gas: ~6592, l2_gas: ~46675840)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_fails_if_not_initialized (l1_gas: ~0, l1_data_gas: ~2272,
↪  l2_gas: ~1954560)
```

```
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_transfers_funds_to_pool (l1_gas: ~0, l1_data_gas: ~6688,
↪  l2_gas: ~104201280)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_updates_liquidity_factor (l1_gas: ~0, l1_data_gas: ~6880,
↪  l2_gas: ~101999360)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_updates_pool_reserves (l1_gas: ~0, l1_data_gas: ~6880,
↪  l2_gas: ~119519360)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_fails_if_not_initialized (l1_gas: ~0, l1_data_gas:
↪  ~2272, l2_gas: ~1954560)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_harvest_fees_adds_liquidity_prior_with_tick_less_than
_initial (l1_gas: ~0, l1_data_gas: ~8224, l2_gas: ~861470720)
[PASS]
↪  spline_v0_integrationtest::lp_test::test_remove_liquidity_harvest_fees_adds_liquidity_prior_with_tick_greater_than
_initial (l1_gas: ~0, l1_data_gas: ~8224, l2_gas: ~855430720)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_removes_liquidity_from_pool (l1_gas: ~0, l1_data_gas:
↪  ~7072, l2_gas: ~153185280)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_transfers_funds_from_pool (l1_gas: ~0, l1_data_gas:
↪  ~7072, l2_gas: ~152665280)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_updates_liquidity_factor (l1_gas: ~0, l1_data_gas:
↪  ~7072, l2_gas: ~151905280)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_updates_pool_reserves (l1_gas: ~0, l1_data_gas: ~7072,
↪  l2_gas: ~169545280)
[PASS] spline_v0_integrationtest::math_test::test_muldiv (l1_gas: ~0, l1_data_gas: ~0, l2_gas: ~200000)
[PASS] spline_v0_integrationtest::sweep_test::test_sweep (l1_gas: ~0, l1_data_gas: ~768, l2_gas: ~1601920)
[PASS] spline_v0_integrationtest::sweep_test::test_sweep_insufficient_balance (l1_gas: ~0, l1_data_gas: ~768, l2_gas:
↪  ~960960)
[PASS] spline_v0_integrationtest::symmetric_test::test_symmetric_liquidity_profile_get_bounds_for_liquidity_updates
↪  (l1_gas: ~0, l1_data_gas: ~1728, l2_gas: ~2521920)
[PASS] spline_v0_integrationtest::token_test::test_burn_burns_funds (l1_gas: ~0, l1_data_gas: ~864, l2_gas: ~2483840)
[PASS] spline_v0_integrationtest::token_test::test_burn_fails_if_not_authority (l1_gas: ~0, l1_data_gas: ~480, l2_gas:
↪  ~400000)
[PASS] spline_v0_integrationtest::token_test::test_constructor_sets_name_and_symbol (l1_gas: ~0, l1_data_gas: ~576,
↪  l2_gas: ~600000)
[PASS] spline_v0_integrationtest::token_test::test_mint_fails_if_not_authority (l1_gas: ~0, l1_data_gas: ~480, l2_gas:
↪  ~400000)
[PASS] spline_v0_integrationtest::token_test::test_mint_mints_funds (l1_gas: ~0, l1_data_gas: ~864, l2_gas: ~1481920)
[PASS] spline_v0_integrationtest::lp_test::test_initialize_pool_fails_if_not_extension (l1_gas: ~0, l1_data_gas: ~2272,
↪  l2_gas: ~1994560)
[PASS] spline_v0_integrationtest::lp_test::test_update_position_fails_if_not_extension (l1_gas: ~0, l1_data_gas: ~6912,
↪  l2_gas: ~50235840)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_emits_liquidity_updated_event (l1_gas: ~0,
↪  l1_data_gas: ~7072, l2_gas: ~153025280)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_fails_if_extension_not_liquidity_provider
↪  (l1_gas: ~0, l1_data_gas: ~2272, l2_gas: ~1954560)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_fails_if_already_initialized (l1_gas: ~0,
↪  l1_data_gas: ~6784, l2_gas: ~45153920)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_fails_if_extension_not_liquidity_provider (l1_gas: ~0,
↪  l1_data_gas: ~6880, l2_gas: ~101599360)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_fails_if_extension_not_liquidity_provider (l1_gas: ~0,
↪  l1_data_gas: ~6592, l2_gas: ~46675840)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_adds_liquidity_to_pool (l1_gas: ~0, l1_data_gas: ~6880,
↪  l2_gas: ~103399360)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_emits_liquidity_updated_event (l1_gas: ~0, l1_data_gas:
↪  ~6880, l2_gas: ~103119360)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_fails_if_not_owner (l1_gas: ~0, l1_data_gas:
↪  ~2272, l2_gas: ~2034560)
[PASS] spline_v0_integrationtest::cauchy_test::test_harvest_fees_on_remove_liquidity_with_cauchy_profile (l1_gas: ~0,
↪  l1_data_gas: ~20224, l2_gas: ~1222970240)
[PASS] spline_v0_integrationtest::cauchy_test::test_swap_with_cauchy_profile (l1_gas: ~0, l1_data_gas: ~16288, l2_gas:
↪  ~537684160)
[PASS] spline_v0_integrationtest::lp_test::test_multiple_create_and_initialize_pool_deploys_multiple_pool_tokens
↪  (l1_gas: ~0, l1_data_gas: ~10912, l2_gas: ~88471360)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_fails_if_not_initialized (l1_gas: ~0, l1_data_gas: ~2272,
↪  l2_gas: ~1954560)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_harvest_fees_adds_liquidity_prior_with_tick_greater_than
_initial (l1_gas: ~0, l1_data_gas: ~8032, l2_gas: ~860257600)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_mints_initial_shares_to_liquidity_provider
↪  (l1_gas: ~0, l1_data_gas: ~6784, l2_gas: ~45633920)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_sets_liquidity_profile (l1_gas: ~0,
↪  l1_data_gas: ~6784, l2_gas: ~45273920)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_initializes_pool (l1_gas: ~0, l1_data_gas:
↪  ~6784, l2_gas: ~45313920)
```

```
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_transfers_funds_to_pool (l1_gas: ~0,
↪  l1_data_gas: ~6592, l2_gas: ~47555840)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_sets_initial_liquidity_factor (l1_gas: ~0,
↪  l1_data_gas: ~6784, l2_gas: ~45513920)
[PASS] spline_v0_integrationtest::lp_test::test_add_liquidity_harvest_fees_adds_liquidity_prior_with_tick_less_than
_initial (l1_gas: ~0, l1_data_gas: ~8032, l2_gas: ~866337600)
[PASS] spline_v0_integrationtest::lp_test::test_remove_liquidity_burns_shares (l1_gas: ~0, l1_data_gas: ~7072, l2_gas:
↪  ~152385280)
[PASS] spline_v0_integrationtest::lp_test::test_create_and_initialize_pool_updates_pool_reserves (l1_gas: ~0,
↪  l1_data_gas: ~6784, l2_gas: ~55073920)
[PASS] spline_v0_integrationtest::cauchy_test::test_create_and_initialize_pool_with_cauchy_profile (l1_gas: ~0,
↪  l1_data_gas: ~15712, l2_gas: ~170093440)
[PASS] spline_v0_integrationtest::cauchy_test::test_add_liquidity_with_cauchy_profile (l1_gas: ~0, l1_data_gas: ~15808,
↪  l2_gas: ~445359680)
[PASS] spline_v0_integrationtest::cauchy_test::test_remove_liquidity_with_cauchy_profile (l1_gas: ~0, l1_data_gas:
↪  ~16000, l2_gas: ~725501760)
[PASS] spline_v0_integrationtest::cauchy_test::test_harvest_fees_on_add_liquidity_with_cauchy_profile (l1_gas: ~0,
↪  l1_data_gas: ~20224, l2_gas: ~1224336320)
[PASS] spline_v0_integrationtest::debug_test::test_debug_add_then_remove_liquidity (l1_gas: ~0, l1_data_gas: ~3680,
↪  l2_gas: ~884674880)
Tests: 60 passed, 0 failed, 0 skipped, 1 ignored, 0 filtered out
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.