plainshift

February 14, 2025

Ekubo Protocol

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk. This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at https://plainshift.io.

1.1. Executive Summary

Plainshift was tasked with reviewing Ekubo core contracts, TWAMM, limit orders extension, and oracle extension from January 6th, 2025 to February 14th, 2025. We ultimately found and confirmed 3 medium and 4 low severity issues.

Initially, we reviewed the prior audit findings and confirmed they were all correctly mitigated. Following a thorough audit and detailed drafts of potential attack vectors, we set up a custom testing suite to verify PoCs for our leads. We should note that, as per our account, the in-scope codebase is secure and during our audit it was evident that considerations were made for almost all edge cases. We recommend adding detailed documentation regarding the math powering the AMM down the line.

1.2. Project Timeline

Date	Phase
January 6th, 2025	Kickoff
February 14th, 2025	Audit End
February 14th, 2025	Delivery

1.3. Scope

Repositories	https://github.com/EkuboProtocol/contracts
Version	`b66c4d95a8876442447e81d4c343c37140518055`
Contracts	`src/**/*.cairo`
Type	Cairo
Platform	StarkNet

1.4. Overview of Findings

Our comprehensive review yielded 3 medium and 4 low severity issues.

Severity/Impact Level	Count
High	0
Medium	3
Low	4

2. Findings

2.1. [M1] Partial swaps break multihop routes due to unhandled deltas

Target	src/router.cairo
Severity	Medium
Category	Vulnerability

2.1.1. Description

The `Router` contract's `multi_multihop_swap()` function only handles token deltas for the first and last tokens in a multihop swap route. However, if any intermediate swap in the route results in a partial fill due to hitting a `sqrt_ratio_limit`, those intermediate deltas must also be handled or the core contract will revert.

This occurs because each swap in the route can potentially be constrained by its `sqrt_ratio_limit` parameter, resulting in a partial fill that causes some amount of input tokens to that particular node to remain in the core contract. These tokens need to be transferred but are not being accounted for.

The key issue is in the delta handling logic:

src/router.cairo

```
199         // execute deltas now if it's not a simulation
200         if (!simulate) {
201             let first = first_swap_amount.unwrap();
202             handle_delta(core, token_amount.token, -token_amount.amount, recipient);
203             handle_delta(core, first.token, first.amount, recipient);
204         }
```

The core contract requires all deltas to be properly accounted for via the `account_delta()` function, which is called during `swap()`. If any delta is not properly handled, the core will detect this in its final state check:

src/core.cairo

```
524        assert(self.get_nonzero_delta_count(id) == 0, 'NOT_ZEROED');
```

2.1.2. Impact

- Any multihop swap that includes a `sqrt_ratio_limit` on intermediate hops will revert if that limit is hit
- This effectively makes `sqrt_ratio_limit` unusable for any nodes except the first one in a route
- Users cannot properly control price impact on intermediate swaps in a route
- Complex trading strategies that rely on limiting price impact across multiple hops are impossible to implement

2.1.3. Recommendation

The router should track and handle deltas for all intermediate swaps in the route. An option would be to check the `delta.amount1` in case of `is_token1` and vice versa against `token_amount.amount` and withdraw any difference from the core.

2.2. [M2] ERC721 implementation does not comply with advertised interface IDs

Target	src/interfaces/src5.cairo, src/interfaces/erc721.cairo
Severity	Medium
Category	Spec Deviation

2.2.1. Description

The contract is using interface IDs from OpenZeppelin's ERC721 implementation but does not fully implement the corresponding interfaces. Specifically:

1. For `SRC5_ERC721_METADATA_ID`, the OpenZeppelin interface defines `token_uri()` to return a `felt252`, while the current implementation returns an `Array<felt252>`.
2. For `SRC5_ERC721_ID`, the OpenZeppelin interface includes a `safe_transfer_from()` function that is not implemented in the current contract.

According to SNIP-5, interface IDs are calculated as the XOR of all function selectors in the interface. The specification states:

> *"For some "standard interfaces" like the ERC-721 token interface, it is sometimes useful to query whether a contract supports the interface and if yes, which version of the interface, to adapt how the contract is to be interacted with."*

By advertising support for these interface IDs without fully implementing them, the contract violates the specification's intent and could cause integration issues.

2.2.2. Impact

1. Contracts attempting to integrate with this implementation by checking the interface IDs may fail when trying to call functions that either:
   - Don't exist (`safe_transfer_from`)
   - Return unexpected types (`token_uri`)

2. This could lead to failed transactions and broken integrations, particularly in systems that rely on interface detection for dynamic behavior.

3. The mismatch between advertised and actual interfaces violates the core purpose of SRC-5, which is to provide reliable interface detection.

2.2.3. Recommendation

Update the implementation to use the latest OpenZeppelin interface IDs and ensure full compatibility by implementing all functions defined in the interface. If `safe_transfer_from` is not intended to be supported, do not advertise this interface ID or revert with a custom error message.

2.3. [M3] Fee accumulation overflow can lead to undefined protocol behavior

Target	src/core.cairo
Severity	Medium
Category	Vulnerability

2.3.1. Description

The `accumulate_as_fees()` function in `Core.cairo` allows pool extensions to accumulate fees by adding to the pool's `FeesPerLiquidity` state. However, the `FeesPerLiquidity` struct stores fee values as raw `felt252` values which can overflow during addition.

The key issue occurs in this code path:

1. Extensions call `accumulate_as_fees()` with token amounts
2. These are converted to `FeesPerLiquidity` values via `fees_per_liquidity_new()`
3. The new values are added to existing pool fees using the `AddFeesPerLiquidity` trait
4. The addition happens directly on the `felt252` values without overflow checks

While the `to_fees_per_liquidity()` function includes overflow protection for individual fee calculations:

src/types/fees_per_liquidity.cairo

```
37          (u256 { low: 0, high: amount } /
            liquidity.into()).try_into().expect('FEES_OVERFLOW')
38      }
```

The subsequent addition of these values can still overflow since `felt252` addition wraps around. This will, in turn, cause the calculation of the change in fees per liquidity in `Position::fees()` to underflow.

This is concerning because:

1. Extensions can be registered without permission
2. Extensions have full authority to accumulate fees

3. The protocol was presumably not designed with fee overflow scenarios in mind

An intentional overflow of the fee rate can be achieved for any token by creating a fresh pool with low liquidity and registering an extension that repeatedly accumulates and recuperates the fees.

2.3.2. Impact

The overflow in fee accumulation creates undefined behaviour in the protocol's accounting system. While direct theft of funds has not been demonstrated, the presence of undefined behaviour in core accounting logic carries the risk undiscovered exploit paths with the potential for fee rate manipulation affecting all pools containing a given token, with downstream effects on trading pairs and the protocol as a whole.

2.3.3. Recommendation

Implement strict bounds on fee accounting to prevent overflow scenarios.

2.4. [L1] Oracle tick addition can produce invalid results

Target	src/extensions/oracle.cairo
Severity	Low
Category	Vulnerability

2.4.1. Description

The `get_average_tick_over_periods()` function calculates cross-pool ticks by adding the ticks from two pools that share the oracle token. While this mathematical approach is sound for valid tick ranges, the function does not validate that the resulting tick remains within the protocol's valid tick bounds.

When calculating:

src/extensions/oracle.cairo

```
395                 while let Option::Some(t_quote) = t_quotes.pop_front() {
396                     results.append(*t_quote + *t_bases.pop_front().unwrap());
397                 };
```

...the addition of two valid ticks could produce a result outside the valid tick range. While this is mitigated in price conversion functions since `tick_to_sqrt_ratio()` will revert for invalid ticks, external integrations may assume all returned ticks are valid and fail to handle out-of-bounds values.

2.4.2. Impact

External integrations that consume tick values directly (without converting to prices) and assume they are within valid bounds could experience unexpected behavior.

2.4.3. Recommendation

Add validation after the tick addition to ensure the result remains within valid bounds.

2.5. [L2] OwnedNFT SRC-5 implementation returns true for legacy ERC165 interface IDs

Target	src/owned_nft.cairo, src/interfaces/src5.cairo
Severity	Low
Category	Spec Deviation

2.5.1. Description

The `owned_nft` contract's implementation of SRC-5 returns `true` for ERC165 interface IDs (e.g., `ERC165_ERC721_ID: 0x80ac58cd`), which deviates from the SRC-5 specification. According to SNIP-5:

> *"The interface ID is calculated as the XOR of all function selectors in the interface, where function selectors are calculated as `starknet_keccak(function_name)`."*

The specification explicitly states that implementations must return `false` for any interface ID not calculated according to this method. ERC165 interface IDs use a different calculation method and therefore should return `false` according to the spec.

While this implementation choice provides backwards compatibility with ERC165, it technically violates the SRC-5 specification's requirements.

2.5.2. Impact

As the ecosystem moves away from ERC165 towards SRC-5 (e.g. OpenZeppelin fully migrated from ERC165 interface IDs to SRC-5 here), maintaining this backwards compatibility may become unnecessary technical debt.

2.5.3. Recommendation

Modify the `supportsInterface` implementation to only return `true` for SRC-5 calculated interface IDs.

2.6. [L3] `tick_spacing` bounds check uses incorrect maximum value

Target	src/types/bounds.cairo
Severity	Low
Category	Spec Deviation

2.6.1. Description

In the `max_bounds()` function, the tick spacing validation check uses `MAX_TICK_MAGNITUDE` instead of `MAX_TICK_SPACING` :

src/types/bounds.cairo

```
14        assert(tick_spacing <= tick_constants::MAX_TICK_MAGNITUDE,
          'MAX_BOUNDS_TICK_SPACING_LARGE');
```

This is incorrect as `MAX_TICK_MAGNITUDE` is much larger than the intended maximum tick spacing.

2.6.2. Impact

The incorrect bounds check allows initializing positions with tick spacings larger than `MAX_TICK_SPACING` but smaller than `MAX_TICK_MAGNITUDE` . However, this has no security impact since:

1. Core pool initialization enforces valid tick spacing
2. TWAMM extension requires exactly `MAX_TICK_SPACING`
3. Position updates using invalid tick spacing will revert

2.6.3. Recommendation

Update the check in `max_bounds()` to use the correct constant.

2.7. [L4] Valid `i129` values can revert when stored

Target	src/types/i129.cairo
Severity	Low
Category	Vulnerability

2.7.1. Description

The `StorePacking` implementation for `i129` enforces that the magnitude must be less than 2^{127} when packing values for storage, even though the `i129` type can technically represent values up to 2^{128}. This constraint was intentionally added for compatibility with the native `i128` type.

In the `pack()` function, the following check is enforced:

src/types/i129.cairo

```
107             assert(value.mag < 0x80000000000000000000000000000000, 'i129_store_overflow');
```

This means that while the `i129` type can produce larger values in computations, attempting to store values with magnitudes $\geq 2^{127}$ will revert.

2.7.2. Impact

Some valid `i129` values that can naturally occur in internal logic will revert when attempting to write them to storage. This could potentially cause certain operations to be permanently impossible if they invariably result in `i129` values larger than 2^{127} that need to be stored.

While no concrete scenarios have been identified where this limitation causes issues in the current codebase, it represents an inconsistency between the type's capabilities and its storage behaviour that could lead to unexpected reverts.

2.7.3. Recommendation

Consider one of the following approaches:

1. Update the documentation to clearly communicate this storage limitation
2. Modify the type to enforce the 2^{127} magnitude limit consistently throughout all operations
3. If compatibility with `i128` is not strictly required, remove the magnitude restriction in storage

The simplest solution would be adding clear documentation about this storage constraint to prevent confusion for developers using the type.