Ekubo BoostedFees + SavedBalancesWrapper + FreePositions Audit Report

Prepared by Riley Holterhus

February 2, 2026

Contents

1 Introduction

1.1 About Ekubo

Ekubo is a high-performance, gas-optimized AMM. It supports several configuration options for pools, including choices between concentrated and stableswap liquidity, hookable extensions, and parameters such as fees and tick spacing. Ekubo is deployed on both Starknet and EVM-based chains.

1.2 About the Auditor

Riley Holterhus is an independent security researcher who focuses on Solidity smart contracts. Other than conducting independent security reviews, he works as a Lead Security Researcher at Spearbit, and also searches for vulnerabilities in live codebases. Riley can be reached by email at rileyholterhus@gmail.com, by Telegram at @holterhus and on Twitter/X at @rileyholterhus.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an “as-is” basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

From January 31, 2026 to February 1, 2026, Riley Holterhus conducted a manual security review of a subset of Ekubo's `evm-contracts`. The goal was to identify potential vulnerabilities and logic issues in the reviewed components.

The audit was performed on the codebase found in the `EkuboProtocol/evm-contracts` GitHub repository. The audit started on commit fdd1cae. Only a portion of the codebase was considered in scope, specifically the `SavedBalancesWrapper`, `FreePositions` and `BoostedFees` contracts, along with the related `ManualPoolBooster`, `BoostedFeesDataFetcher`, and `BoostedFeesLib` contracts.

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities aim to capture the impact and likelihood of each potential issue.

In total, **2 findings** were identified, both being informational findings.

3 Findings

3.1 Informational Findings

3.1.1 Revert with `int128.min` in `SavedBalancesWrapper`

Description: The `SavedBalancesWrapper` contract accepts an arbitrary `int128 amount` as input from the user and then executes the following logic:

```
if (amount > 0) {
    _mint({to: counterparty, id: id, amount: uint256(uint128(amount)), data: ""});
} else if (amount < 0) {
    _burn({by: original.addr(), from: counterparty, id: id, amount: uint256(uint128(-amount))});
}
```

One edge case worth highlighting is when `amount == type(int128).min`. In this scenario, the `amount < 0` branch is taken. However, because `type(int128).min` has a greater absolute value than the maximum representable positive `int128`, the expression `-amount` overflows and causes a revert.

This revert occurs because the code is not wrapped in an `unchecked` block. If it were, the behavior would be different. The current behavior does not lead to any issues since a revert is an appropriate result in this scenario.

Recommendation: No changes necessary. This finding is provided for informational purposes only.

Ekubo: Acknowledged.

Auditor: Acknowledged.

3.1.2 State can be zero due to timestamp wraparound

Description: At the start of `maybeAccumulateFees()`, the `BoostedFees` contract performs the following check:

```
TwammPoolState state = _getPoolState(poolId);

if (TwammPoolState.unwrap(state) == bytes32(0)) {
    if (poolKey.config.extension() != address(this) || !CORE.poolState(poolId).isInitialized()) {
        revert PoolNotInitialized();
    }
}
```

This logic checks whether the extension's own storage is non-zero for the pool, and if it isn't, it then requires that the pool is configured for this extension and that the pool state in the Ekubo core contract is initialized.

Note that the extension state is comprised of the last timestamp synced to (stored as a `uint32`), and the two rates of the tokens being streamed as fees. In general, the code stores the timestamp as a `uint32`, but it is capable of converting it into the appropriate actual timestamp if the `uint32` container is exceeded.

Technically, it is possible that the extension state is all zero, but it has actually been initialized, simply because the `uint32` casting wraps exactly back to zero. The first time this would happen is in February 2106, and the second time this would happen is in March 2242, so this is a very niche scenario.

If this were to happen, the code would still work correctly, since the state within the core contract would be initialized and would therefore not hit the `PoolNotInitialized` error above. The rest of `maybeAccumulateFees()` would be able to handle the timestamp the same as any other timestamp. So, this behavior appears to be handled correctly in the code.

Recommendation: No changes necessary. This finding is provided for informational purposes only. It's possible this was the intended behavior with the above check in the first place.

Ekubo: Acknowledged.

Auditor: Acknowledged.