hashlock.

2024

# Security Audit

## Fluidity (DeFi)

July 2024

info@hashlock.com.au

Hashlock Pty Ltd

# Table of Contents

## CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

# Executive Summary

The Longtail team partnered with Hashlock to conduct a security audit of their Longtail AMM (Seawater) smart contract. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

# Project Context

Longtail is a DeFi-native Layer-3 blockchain built on the Arbitrum stack, designed to incentivize user participation by rewarding them for using the platform. It features a novel on-chain order book called the Super Book, which facilitates faster execution speeds and shared, permissionless liquidity for all decentralized applications (dApps) on the chain.

Overall, Longtail aims to create a dynamic and sustainable DeFi ecosystem by interweaving liquidity and utility into its core operations, providing a seamless and rewarding experience for both users and developers.

**Project Name**: Longtail
**Compiler Version:** N/A
**Website:** https://long.so

# Audit scope

We at Hashlock audited the solidity code within the Longtail project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

| **Description** | **Longtail Protocol Smart Contracts** |
|---|---|
| **Platform** | **Arbitrum / Rust** |
| **Audit Date** | **July, 2024** |
| **Contracts** | Longtail AMM (Seawater) |
| **GitHub Commit Hash** | 1989b8055fab34387f96c04ad0451ad35dd14210 |

# Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.


*The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.*

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

We initially identified some significant vulnerabilities that have since been addressed.

**Hashlock found:**

3 Low-severity vulnerabilities

2 QA suggestions

**Caution:** *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

# Intended Smart Contract Behaviours

| Claimed Behaviour | Actual Behaviour |
|---|---|
| - **./src/erc20.rs** - wrapper for erc20<br>- **./src/error.rs** - errors definitions and some logical macros definitions<br>- **./src/eth_serde.rs** - utilities for encoding and decoding ethereum calldata<br>- **./src/events.rs** - import of events<br>- **./src/immutable.rs** - constant values related to deployment<br>- **./src/lib.rs** - the main entry point exposing logic of the protocol, contains all public functions that can be called from outside<br>- **./src/main.rs** - main function definition<br>- **./src/migrations.rs** - empty contract as of now<br>- **./src/permit2_types.rs** - definition of permit struct<br>- **./src/pool.rs** - the contract containing pool related logic. Allows functions that are responsible for initialization of a pool, creation and modification of positions and collecting fees from the protocol. Defined StoragePool struct which contains the core logic of AMM functions.<br>- **./src/position.rs** - contains impl for StoragePositions, which defines the logic to manage a single position such as update, collect fees or create new | **Contract achieves this functionality.** |

| | |
|---|---|
| position.<br>- **./src/tick.rs** - Contains structures and functions to related to a pool's ticks like update, get_fee_growth_inside, Updates a tick's fee information when the tick is crossed or delete a tick from the map<br>- **./src/types.rs** - Re-exports and extension traits for stylus' bigint types.<br>- **./src/wasm_erc20.rs** - logic for encoding of ERC20 functions, including on calling them on other contracts, encode functions such as: transfer, transfer from, call optional return but also define non-solidity native functions give, take, exchange | |
| - **./src/maths/bit_math.rs**<br>- **./src/maths/full_math.rs**<br>- **./src/maths/liquidity_math.rs**<br>- **./src/maths/mod.rs**<br>- **./src/maths/sqrt_price_math.rs**<br>- **./src/maths/swap_math.rs**<br>- **./src/maths/tick_bitmap.rs**<br>- **./src/maths/tick_math.rs**<br>- **./src/maths/unsafe_math.rs**<br>- **./src/maths/utils.rs**<br><br>All of those functions define mathematical operations related to protocol core logic, define atomic mathematical operations used in functions such as sqrt price calculations, compute single steps of a swap, tick calculations. | **Contract achieves this functionality.** |

# Code Quality

This audit scope involves the smart contracts for the Longtail project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

# Audit Resources

We were given the Longtail projects smart contract code in the form of GitHub access.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help us understand the overall architecture of the protocol.

# Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.

# Severity Definitions

| Significance | Description |
|---|---|
| **High** | High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community. |
| **Medium** | Medium-level difficulties should be solved before deployment, but won't result in loss of funds. |
| **Low** | Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future. |
| **Gas** | Gas Optimisations, issues, and inefficiencies |

# Audit Findings

## Low

### [L-01] pool.rs - Off by one error in create_position function

**Description**

The `create_position` function in `pool.rs:81-82` contains a potential off-by-one error in its tick range validation. The function uses exclusive comparisons (> and <) when checking if the low and up tick values are within the valid range defined by `min_tick` and `max_tick`.

However the `max` and `min` values should still be allowed.

**Impact**

This can lead at best to excluding some price range, and inconvenience to users, or in worst case, lead to some unexpected behaviour later on, since those values should still be included.

**Recommendation**

Use `>=` and `<=` instead.

**Status**

Resolved

### [L-02] pool.rs - Production code should not panic

**Description**

In `pkg/seawater/src/pool.rs` in line `298` a `debug_assert!` is used to check condition outcome.

It should be noted that when using panic, all gas allocated to the transaction is consumed when a panic occurs which increases user gas costs.

**Impact**

Increased gas consumption and worst contract usage experience for users, if an error happens.

**Recommendation**

We recommend using assert and proper error handling instead.

**Status**

Acknowledged

### [L-03] lib.rs - `exchange` function might be error prone

**Description**

In `pkg/seawater/src/wasm_erc20.rs`, the `exchange` function purpose seems to be to handle both `give` and `take` operations from the user depending on users input amount. This might indeed reduce code complexity.

However it is easily error prone, since it requires the user to use the proper value of a signed integer to decide if the operation should take from or give to the user. If a user incorrectly implements the amount, which when dealing with such large numbers is not that improbable, a financial loss can happen.

**Impact**

Users may lose funds or be unable to transfer, but due to being a very edge case, the severity is low.

**Recommendation**

Consider implementing another switch like `give: bool` and work with a regular `uint` instead, so the user will consciously choose to give or take a token when calling the function.

**Status**

Acknowledged

# QA

## [Q-01] pool.rs - Pools created as enabled may pose additional risk

### Description

`Init` function in `pkg/seawater/src/pool.rs:59` is available only for admins which means this operation is a sensitive one. On the other hand, if there is any error in pool initialization, like incorrect, unfavourable pricing, there will be no time to correct it as the pool will go public. Assume the pool is created with incorrect price, and first user noticing that can take advantage of that fact to arbitrage against the pool.

### Impact

Increased risk of exploitation of admin mistakes. However, the likelihood of this happening is low (required mistake of a trusted role).

### Recommendation

Consider creating a pool as disabled by default, and the admin should enable it only when it's considered properly initialised as an extra safeguard.

### Status

Resolved

## [Q-02] wasm_erc20.rs - Unreachable code part

### Description

In `pkg/seawater/src/wasm_erc20.rs` in the `exchange` function there is a tripe `if` clause, depending if the provided number is negative or positive and a third clause for another case which presumably is to handle zero. However as per the Rust documentation below:

(https://docs.rs/num/latest/num/trait.Signed.html#tymethod.is_negative) zero is counted as negative, therefore the third clause is redundant.

**Impact**

Missing best practice, unnecessary code gives worst development experience.

**Recommendation**

Consider removing the third clause for clarity of the code.

**Status**

Resolved

# Centralisation

The Longtail project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

# Conclusion

After Hashlocks analysis, the Longtail project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

# Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

**Manual Code Review:**

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

**Vulnerability Analysis:**

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

**Documenting Results:**

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

**Suggested Solutions:**

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

# Disclaimers

## Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

## Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

# About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

**Website:** hashlock.com.au
**Contact:** info@hashlock.com.au


#Hashlock.