Lyra Accounts Smart Contract Audit

Lyra, 25 April 2023

Confidential
17 Tyrwhitt Avenue, Johannesburg, 2196, South Africa
hello@iosiro.com

1. Introduction

iosiro was commissioned by Lyra (https://www.lyra.finance/) to conduct a smart contract audit of Lyra's smart contracts. The audit was performed by 2 auditors between 22 March and 31 March 2023, using 14 resource days.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.
- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to understand the smart contracts' risk exposure better and as a guide to improving the security posture of the smart contracts by remediating the issues identified. The results of this audit reflect the in-scope source code reviewed at the time of the audit.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts function according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was outside of the scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered high risk from cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle, and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of an audit performed by iosiro on specific peripheral smart contracts for the Lyra Protocol.

Audit findings

The most significant finding of the audit was a high-risk vulnerability related to a potential denial-of-service attack, which could be triggered by an anonymous user adding an incompatible asset.

Additionally, three low-risk issues were identified, including missing return values, potential for undesirable reentrancy, and the ability to create an account with an incompatible manager. Lastly, an informational finding highlighted the inability to remove broken or malicious managers from the smart contract.

Recommendations

At a high level, the security posture of the Lyra Protocol could be further strengthened by:

- Remediating the issues identified in this report and performing a review to ensure that the issues were correctly addressed.
- Performing additional audits at regular intervals, as security best practices, tools, and knowledge change over time. Additional audits throughout the project's lifespan ensure the longevity of the codebase.
- Continue their bug bounty program encouraging the responsible disclosure of security vulnerabilities in the system.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security vulnerabilities for the purposes of this audit.

3.1.1 Smart contracts

Project name: Accounts
Commits: 3da99d1
Files: Accounts.sol, Allowances.sol, SecurityModule.sol, CashAsset.sol

3.2 Methodology

The audit was conducted using a variety of techniques described below.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high-risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and discover security issues that could be exploited.

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. Static analysis results were reviewed manually and any false positives were removed. Any true positive results are included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk:** The issue could result in a loss of funds for the contract owner or system users.
- **Medium risk:** The issue resulted in the code specification being implemented incorrectly.
- **Low risk:** A best practice or design issue that could affect the security of the contract.
- **Informational:** A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed:** The issue was identified during the audit and has since been satisfactorily addressed, removing the risk it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. This specification is based on the implementation in the codebase. Any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

4.1 Accounts Contract

The Accounts contract, represented by Accounts, is the base layer for account management in Lyra. Users can create accounts, which can have managers assigned to them, and keep track of balances of assets and sub IDs. In the context of Lyra, sub IDs refer to encoded options that an account has purchased in the Lyra ecosystem.

Accounts are represented as ERC-721 NFTs, and allows account owners to control allowances of assets and sub IDs of assets of the account.

4.2 Allowances Contract

The Allowances contract provides the Accounts contract with the functionality to set and spend allowance balances for accounts. The accounts contract inherits the allowances contract, which does not use any external or public methods.

4.3 Security Module

The security module registers itself as an account in the accounts contract, and can be used by whitelisted modules to bail out, or restore, insolvent accounts. The CashAsset.sol contract is used as the asset for the security module.

4.4 Cash Asset

The CashAsset contract, which implements the IAsset interface, serves as an asset to be used in the accounts contract. Its primary purpose is to allow users to deposit USDC into their accounts. As this asset is used by the security module to restore insolvent accounts, depositors' balances accrue interest.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

5.1.1 Denial-of-service

Accounts.sol#L484

Description

Any address calling `assetAdjustment` with `adjustment.asset == msg.sender` could add an invalid asset entry, causing any subsequent calls to `changeManager` to revert. This was possible because none of the existing Manager contracts checked that the asset was valid in their `handleAdjustment` hook.

The following unit test was developed to illustrate the issue:

```
JavaScript
diff --git a/test/account/unit-tests/ChangeManager.t.sol
b/test/account/unit-tests/ChangeManager.t.sol
index 6eeacfd..8d26ae5 100644
--- a/test/account/unit-tests/ChangeManager.t.sol
+++ b/test/account/unit-tests/ChangeManager.t.sol
@@ -44,6 +44,18 @@ contract UNIT_ChangeManager is Test,
AccountTestBase {
      vm.clearMockedCalls();
   }

+  function testCanMigrateIfInvalidAsset() public {
+     account.assetAdjustment(AccountStructs.AssetAdjustment({
+     acc: aliceAcc,
+     asset: IAsset(address(this)),
+     subId: 0,
+     amount: 1,
+     assetData: ""
+     }), false, "");
+
+     testCanMigrateIfOldManagerAgree();
+  }
+
   function testCannotMigrateIfAssetDisagree() public {
```

```
        dumbManager.setRevertHandleManager(true);
        // alice has usdc in her wallet
```

Recommendation

The Manager contracts should implement standardized checks that ensure the asset being added is valid when the `handleAdjustment()` hook is invoked. The checks could include validation that the asset conforms to the correct interface as defined in EIP-165.

5.2 Medium risk

No identified medium-risk issues were open at the conclusion of the review.

5.3 Low risk

5.3.1 Missing return values

Accounts.sol#L143, Accounts.sol#L567, Accounts.sol#L587

Description

Manager and asset hooks do not return a value to indicate successful execution. This can, under specific circumstances, result in logical inconsistencies.

As an example, if a Manager or Asset address implemented a fallback function, it would not be evident that the hook was not executed.

Recommendation

As a defense-in-depth measure, all hooks should return a boolean result that can be asserted to be `true` by the calling contract.

5.3.2 Potential reentrancy

SecurityModule.sol#L72

Description

If the `stableAsset` configured for the `SafetyModule` were to have a `beforeTokenTransfer()` hook it would be possible to reenter the `deposit()` function and manipulate the shares minted, as state modifications are made after an external call.

The risk is considered low, as the USDC token did not have a `beforeTokenTransfer()` hook at the time of the audit.

Recommendation

In `SecurityModule::deposit()` the `safeTransferFrom()` should be performed before `_mint()` and `cashAsset.deposit()`.

5.3.2 Incompatible managers

Accounts.sol#L117

Description

When creating an account, there is no check to ensure that the manager specified is valid. If an invalid manager is specified, the account would be unusable, and changing the manager would not be possible.

Recommendation

The `manager` should be validated when creating an account using either EIP-165 or with a registry of approved managers.

5.4 Informational risk

5.4.1 Design comments

Actions to improve the functionality and readability of the codebase are outlined below.

Broken or malicious managers

[Accounts.sol#L137]

Removing a malicious or broken manager is impossible, possibly due to a failed upgrade, as the manager can cause `changeManager()` to revert.
In such instances, allowing a manager's forceful removal might be useful.
A possible solution would be to wrap `oldManager.handleManagerChange()` in a `try/catch`, as shown below:

```
JavaScript
bool success;
try oldManager.handleManagerChange(accountId, newManager)
returns (bool success_)
{
     success = success_
} catch {}

if (!success && !force) revert AC_ManagerChangeError();
```