Kamino LIMO

Security Assessment

November 7th, 2024 — Prepared by OtterSec

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Kamino Finance engaged OtterSec to assess the `limo` program. This assessment was conducted between October 25th and November 4th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, we identified an issue concerning the lack of verification to ensure that the permission account is unique to each order, creating a vulnerability where the same permission may be re-utilized across multiple bids (OS-KLO-ADV-00), and the need to validate that all associated token account (ATA) references (OS-KLO-ADV-01).

We also made recommendations to ensure adherence to coding best practices (OS-KLO-SUG-01) and suggested utiutilizinglize a two-step verification process to confirm a change in the admin authority (OS-KLO-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Kamino-Finance/limo. This audit was performed against commit 0f4e77e.

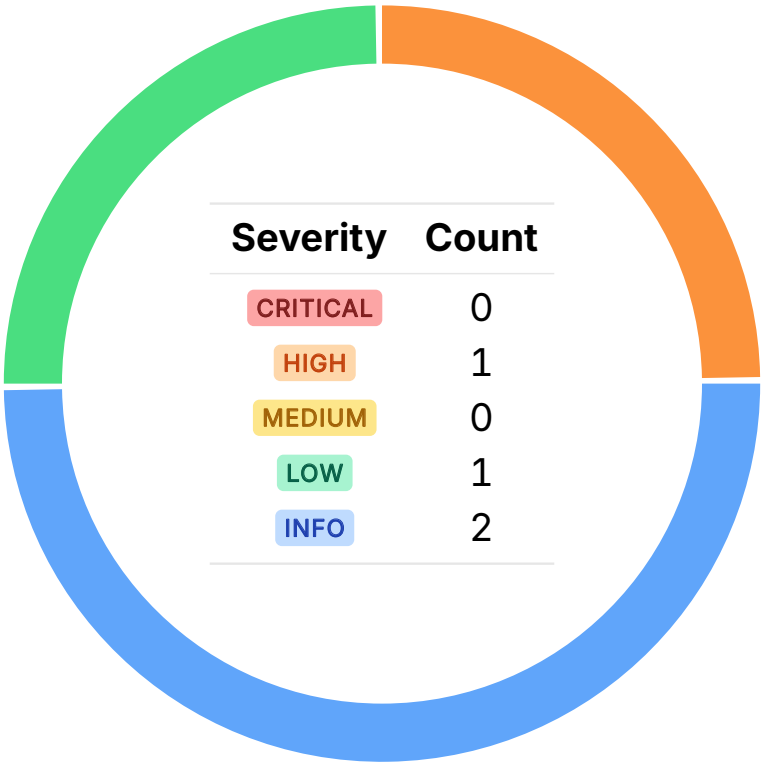
A brief description of the program is as follows:

Name	Description
limo	The system includes contracts defining the Kamino liquidity integration and the matching orders decentralized finance protocol.

03 — Findings

Overall, we reported 4 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	1
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-KLO-ADV-00	HIGH	RESOLVED	`TakeOrder` and `FlashTakeOrder` lack verification to ensure that the `permission` account is unique to each `order`, creating a vulnerability where the same permission may be re-utilized across multiple bids.
OS-KLO-ADV-01	LOW	RESOLVED	The program does not validate that all associated token account (ATA) references, especially `maker_output_ata`, are legitimate ATAs.

Permission Reuse in Orders HIGH OS-KLO-ADV-00

Description

There is a lack of validation for the `permission` account against the `order` account in both `TakeOrder` and `FlashTakeOrder`. Each `TakeOrder` or `FlashTakeOrder` transaction is intended to be executed with a specific order and associated `permission` account. Since there is no validation binding the `permission` account directly to the `order`, the same `permission` account may be misused with multiple orders, enabling repeated utilization of a single permission credential across unrelated orders.

>_ *programs/limo/src/handlers/flash_take_order.rs* RUST

```rust
#[account(mut,
    has_one = global_config,
    has_one = input_mint,
    has_one = output_mint
)]
pub order: AccountLoader<'info, Order>,

/// CHECK: this is the permission key, checked by CPI to ExpressRelay program
pub permission: AccountInfo<'info>,
```

Remediation

Modify `TakeOrder` and `FlashTakeOrder` to check that the `permission` account is explicitly associated with the `order` account before proceeding with any token transfers.

Patch

Fixed in PR#27.

Failure to Verify Associated Token Accounts LOW OS-KLO-ADV-01

Description

In the current implementation it is assumed that accounts with names ending in `*_ata`, especially `maker_output_ata`, are indeed associated token accounts (ATAs) for specific tokens and owners, without performing any explicit validation. This is especially important for the `maker_output_ata`, as if this account is not verified as an actual ATA for the specified maker and token, tokens intended for the maker may instead be sent to any arbitrary account owned by them.

>_ *programs/limo/src/handlers/flash_take_order.rs* RUST

```rust
#[account(mut,
    token::mint = output_mint,
    token::authority = maker
)]
pub maker_output_ata: Box<InterfaceAccount<'info, TokenAccount>>,
```

Remediation

Verify that each `*_ata` accounts are valid associated token accounts and are correctly linked to the expected token mint and owner, preventing the diversion of assets to a non-designated account.

Patch

Fixed in PR#30.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-KLO-SUG-00	Utilize a two-step verification process to confirm a change in the admin authority.
OS-KLO-SUG-01	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Double Verification For Admin Change OS-KLO-SUG-00

Description

The current process to change the global configuration admin authority is a simple single-step process; there is no confirmation step. Once the transaction is submitted, the admin authority change is irreversible. This may result in a denial of service if the current admin accidentally sends an unintended input as a parameter while executing an admin change.

Remediation

Utilize a two-step process to change the global configuration admin authority.

Code Maturity OS-KLO-SUG-01

Description

1. Deduplicate the code between the `take_order` and `flash_pay_order_output` in `operations`, to significantly improve maintainability and reduce redundancy. Since both functions share similar logic for updating the order state, tip calculations, and handling the effects of the order, it is possible to extract the common parts into a helper function, centralizing the duplicated logic.
2. To ensure consistency in `operations::flash_withdraw_order_input`, it would be appropriate to check that `order.is_flash_ix` is equal to zero at the beginning of the function. This check will verify that the function is only called when the order is not already involved in a flash operation.

>_ programs/limo/src/operations.rs RUST

```rust
pub fn flash_withdraw_order_input(
    order: &mut Order,
    input_amount: u64,
    output_amount: u64,
) -> Result<TakeOrderEffects> {
    [...]
    order.is_flash_ix = 1; // Within flash operation - everything besides flash_pay is
        ↪  blocked
    Ok(TakeOrderEffects {
        input_to_send_to_taker,
        output_to_send_to_maker,
    })
}
```

3. The `system_program` account in `UpdateGlobalConfig` instructions seems unnecessary and may be removed.
4. From a design standpoint, it would be more appropriate to directly utilize `minimum_output_to_send_to_maker` than to allow users to specify `output_amount` and possibly overpay.

Remediation

Implement the above-mentioned suggestions.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.