Kamino Kvault

Security Assessment

December 9th, 2024 — Prepared by OtterSec

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

© 2024 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Kamino Finance engaged OtterSec to assess the `kamino-vault` program. This assessment was conducted between November 12th and November 28th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a mismatch between the invested allocations array and the array for the vault's target allocation strategy due to the filtering of empty allocations. This filtering resulted in sequence misalignment and failures in (OS-KVL-ADV-00). Additionally, when refreshing target allocations, it is possible for the token target allocation of a reserve to exceed the token allocation cap (OS-KVL-ADV-01).

We also made recommendations to ensure adherence to coding best practices (OS-KVL-SUG-01) and suggested removing certain instances of unutilized code to improve readability and maintainability (OS-KVL-SUG-02). Additionally, we advised implementing a mechanism to enable the admin to withdraw crank funds from the vault (OS-KVL-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Kamino-Finance/klend-private. This audit was performed against commit 8a4b1fe.

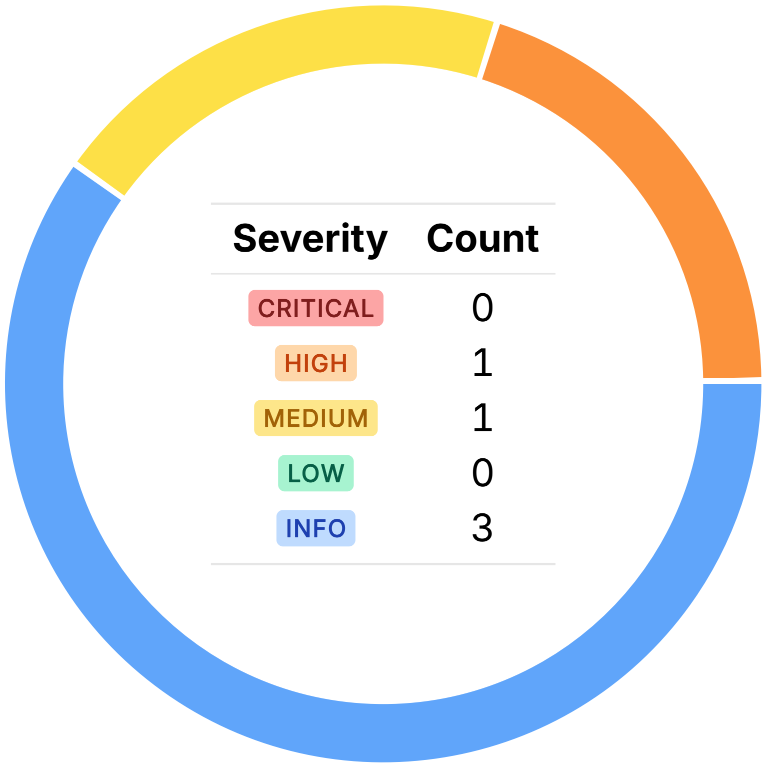
A brief description of the program is as follows:

Name	Description
kamino-vault	It manages and optimizes user funds across multiple reserves, tracks assets under management, charges periodic management and performance fees, and ensures proportional allocations to target investment strategies.

03 — Findings

Overall, we reported 5 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	1
LOW	0
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities 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-KVL-ADV-00	HIGH	RESOLVED ✓	`amounts_invested` creates a mismatch between `invested.allocations` and `vault_allocation_strategy` by filtering out empty allocations, resulting in sequence misalignment and failures in `refresh_target_allocations`.
OS-KVL-ADV-01	MEDIUM	RESOLVED ✓	The current implementation of `refresh_target_allocations` allows `token_target_allocation` to exceed `token_allocation_cap`.

Sequence Misalignment in Allocations Array HIGH OS-KVL-ADV-00

Description

There is a sequence mismatch between the `vault_allocation_strategy` and `invested.allocations` arrays in `amounts_invested`. `vault_allocation_strategy` represents the vault's target allocation strategy and contains all allocations, including active and inactive ones (inactive entries have `Pubkey::default` as the `reserve` key). `invested.allocations`, on the other hand, reflects the current state of investments in the vault.

>_ *operations/vault_operations.rs* RUST

```rust
pub fn amounts_invested<'info, T>(
    vault: &VaultState,
    mut reserves_iter: impl Iterator<Item = T>,
    slot: Slot,
) -> Result<Invested>
where
    T: AnyAccountLoader<'info, Reserve>,
{
    let mut invested = Invested::default();
    let mut total = Fraction::ZERO;
    for (index, allocation) in vault
        .vault_allocation_strategy
        .iter()
        .filter(|a| a.reserve != Pubkey::default())
        .enumerate()
    { [...]
    }
    invested.total = total;
    Ok(invested)
}
```

Since `amounts_invested` filters out inactive entries from `vault_allocation_strategy` while building `invested.allocations`, it includes only active entries, without placeholders for inactive ones. As a result, the indices of the `reserves` in `invested.allocations` no longer align with their corresponding indices in `vault_allocation_strategy`.

However, in `invest`, when `refresh_target_allocations` is invoked, it relies on accurate alignment between `vault_allocation_strategy` and `invested.allocations` to determine which reserves are part of the allocation strategy and calculate the actual from target allocations for each reserve. Consequently, due to the mismatch in index sequences between `vault_allocation_strategy` and `invested.allocations`, the operations in `refresh_target_allocations` will not succeed, causing `invest` to fail.

Remediation

Preserve the original sequence of `vault_allocation_strategy` in `invested.allocations` by including placeholders in `invested.allocations` for inactive entries.

Patch

Fixed in PR#5.

Improper Enforcement of Allocation Cap MEDIUM OS-KVL-ADV-01

Description

In the current implementation of `state::refresh_target_allocations`, there is a potential issue where the `token_target_allocation` of a reserve may exceed its `token_allocation_cap`. This occurs because the allocation logic does not account for the existing allocation `token_target_allocation` when determining whether the ideal allocation `reserve_target_ideal` exceeds the cap.

>_ src/state.rs RUST

```rust
pub fn refresh_target_allocations(&mut self, invested: &Invested) -> Result<()> {
    [...]
        let reserve_target_capped =
            if reserve_target_ideal >= Fraction::from(allocation.token_allocation_cap) {
                a_cap_was_reached = true;
                // Remove the weight from the total
                remaining_weight_to_allocate -= reserve_weight;
                Fraction::from(allocation.token_allocation_cap)
            } else {
                reserve_target_ideal
            };
        remaining_tokens_to_allocate -= reserve_target_capped;
        *token_target_allocation += reserve_target_capped;
    [...]
}
```

The `reserve_target_ideal >= Fraction::from(allocation.token_allocation_cap)` comparison checks only if the current iteration's ideal allocation exceeds the cap, ignoring any previously accumulated allocation (`token_target_allocation`). If the `token_target_allocation` from previous iterations already brings the total close to the cap, the function may allocate additional tokens beyond the cap.

Remediation

Check that the sum of `token_target_allocation + reserve_target_ideal` is greater than or equal to `token_allocation_cap`, and then set `reserve_target_capped` to `token_allocation_cap - token_target_allocation`.

Patch

Fixed in PR#5.

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-KVL-SUG-00	There is no mechanism to withdraw crank funds from the vault.
OS-KVL-SUG-01	Suggestions to ensure adherence to coding best practices.
OS-KVL-SUG-02	The protocol contains multiple cases of irrelevant code, which can be removed.

Inability to Withdraw Crank Funds

OS-KVL-SUG-00

Description

The vault currently does not have any functionality that allows the admin to extract the crank funds from the vault. As a result, these funds may be permanently locked in the contract.

Remediation

Add an instruction for the admin to withdraw `vault.available_crank_funds`.

© 2024 Otter Audits LLC. All Rights Reserved.

Code Maturity OS-KVL-SUG-01

Description

1. In `klend_operations::cpi_refresh_reserves`, prevent unnecessary Cross-Program Invocations when `reserve_account_infos_iter` is empty by returning early. If the iterator is empty, it is not necessary to proceed with the Kamino Lending program invocation.

   >_ operations/klend_operations.rs RUST

```rust
pub fn cpi_refresh_reserves<'a, 'info>(
    cpi: &mut CpiMemoryLender,
    reserve_account_infos_iter: impl Iterator<Item = &'a AccountInfo<'info>>,
) -> Result<()>
where
    'info: 'a,
{
    [...]
    for (account_meta, reserve_account_info) in accounts_metadatas
        .chunks_mut(2)
        .zip(reserve_account_infos_iter)
    {
        account_meta[0] = AccountMeta::new(*reserve_account_info.key, false);
       [...]
    }
    cpi.program_invoke(
        &kamino_lending::id(),
        &accounts_metadatas[..num_reserves * 2],
        &kamino_lending::instruction::RefreshReservesBatch {
            skip_price_updates: true,
        }
        .data(),
    )
    .map_err(Into::into)
}
```

2. It would be appropriate to error out early in cases where `ctx.remaining_accounts.len() < reserves_count`, before proceeding with the logic of refreshing reserves in all instructions, to optimize the execution flow and prevent unnecessary operations.

Remediation

Implement the above-mentioned suggestions.

Redundant/Unutilized Code OS-KVL-SUG-02

Description

1. In the `Deposit` instruction, the `instruction_sysvar_account` appears unnecessary and can be removed.
2. The `_reserves_iter` value computed in the `Withdraw` instruction is not utilized and should be removed.
3. In the `WithdrawPendingFees` instruction, when `invested_to_disinvest_ctokens > 0`, the `cpi_mem` variable defined earlier in the code may be re-utilized instead of creating a new one.
4. `lending_operations::socialize_loss` performs two separate checks: one verifies that the obligation deposits are not empty (`!obligation.deposits_empty()`), while the other checks if both the deposits and borrows are empty (`obligation.deposits_empty() && obligation.borrows_empty()`). If the latter condition is true, it raises an error indicating that the obligation has no deposits or borrows. These checks should be combined into a single, unified conditional block for clarity and efficiency.

>_ lending_market/lending_operations.rs RUST

```rust
pub fn socialize_loss<'info, T>([...]) -> Result<Fraction>
where
    T: AnyAccountLoader<'info, Reserve>,
{
    [...]
    if !obligation.deposits_empty() {
        msg!("Obligation hasn't been fully liquidated!");
        return Err(LendingError::CannotSocializeObligationWithCollateral.into());
    }
    if obligation.deposits_empty() && obligation.borrows_empty() {
        msg!("Obligation has no deposits or borrows");
        return Err(LendingError::ObligationEmpty.into());
    }
    [...]
}
```

5. Currently, `handler_flash_borrow_reserve_liquidity::process` calls `lending_checks::flash_borrow_reserve_liquidity_check` and `lending_operations::flash_borrow_reserve_liquidity`, which both check if `reserve.config.fees.flash_loan_fee_sf == u64::MAX` to determines whether flash loans are disabled for the reserve. Ensure to de-duplicate this redundant check.

Remediation

Remove the above-mentioned instances of unutilized/duplicate code.

© 2024 Otter Audits LLC. All Rights Reserved.

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.

© 2024 Otter Audits LLC. All Rights Reserved.