Hop Aggregator

Security Assessment

October 16th, 2024 — Prepared by OtterSec

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Hop Aggregator engaged OtterSec to assess the `fun` program. This assessment was conducted between August 24th and October 9th, 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 a vulnerability in the functionality responsible for asserting the configuration version, where the two values in the expression are the same, resulting in the condition always evaluating to true (OS-HPA-ADV-00). Furthermore, we highlighted the possibility where the pool closure may be prevented due to improper rounding logic (OS-HPA-ADV-01).

We also made recommendations to incorporate a validation check to ensure that the swap fee is not set to an excessively high value (OS-HPA-SUG-00) and suggested logging the migration time (OS-HPA-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/hopaggregator/fun. This audit was performed against commit 674c090.

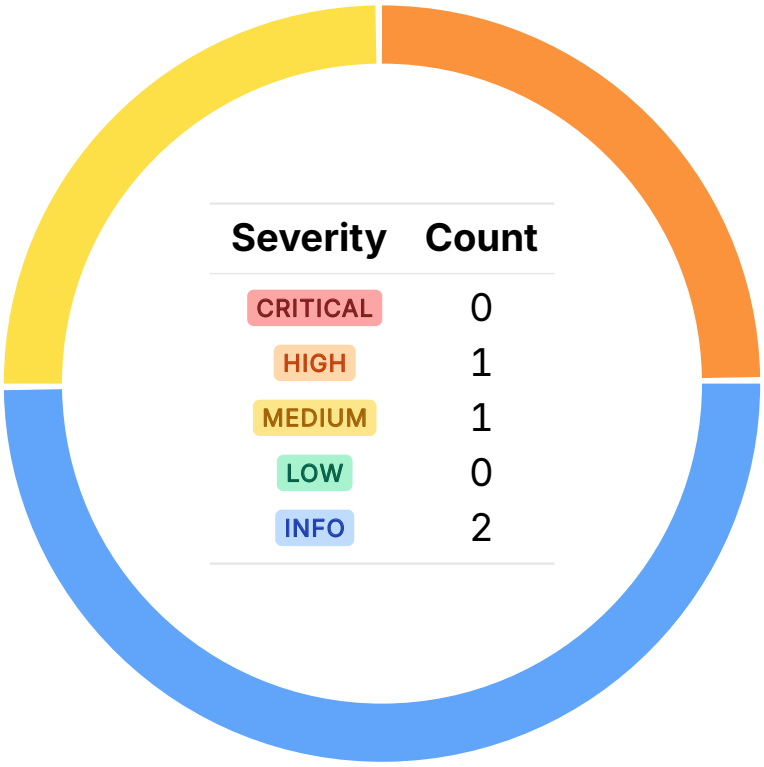
A brief description of the program is as follows:

Name	Description
fun	It enables liquidity provision through a bonding curve mechanism, where the price of tokens adjusts based on supply and demand dynamics.

02 — 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	1
LOW	0
INFO	2

03 — 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-HPA-ADV-00	HIGH	RESOLVED	`enforce_config_version` performs a meaningless check by asserting `config.version >= config.version`, which is always true.
OS-HPA-ADV-01	MEDIUM	RESOLVED	The rounding down of both `max_amount_in` and `amount_out` in `buy` may prevent the full drainage of the pool, rendering it perpetually in the `POOL_STATUS_OPEN` state.

Flawed Version Validation Check HIGH OS-HPA-ADV-00

Description

There is a logical flaw in `meme::enforce_config_version`. The current function does not enforce any version check because it compares `config.version` with itself. Consequently, this expression will always evaluate to true for any configuration value. As a result, this renders the check meaningless, as it does not enforce any actual version comparison.

>_ *contracts/hopfun/sources/meme.move* RUST

```rust
fun enforce_config_version(config: &MemeConfig) {
    assert!(config.version >= config.version, EBadVersion);
}
```

Since the function is supposed to enforce a version check, its current behavior will allow outdated or incompatible configurations to be utilized.

Remediation

Rewrite the version check to compare `config.version` to a valid threshold to ensure that only configurations meeting the required version are accepted.

Patch

Fixed in 0923bc9.

Prevention of Pool Closure Due to Rounding MEDIUM OS-HPA-ADV-01

Description

`meme::get_amount_in` calculates the maximum input (`max_amount_in`) that may be provided to the pool to purchase tokens. This calculation rounds down the result to ensure that the input amount does not exceed the pool's limits. After determining the actual input amount (`amount_in`), `get_amount_out` computes how many tokens the user will receive from the pool. This result is also rounded down to ensure that the user does not receive more tokens than the system allows.

>_ *contracts/hopfun/sources/meme.move* RUST

```rust
fn earn_emissions(&mut self, vault: &Vault) {
    // TODO - consider negative rates
    let sy_balance = self.total_sy_balance(vault);
    for (index, emission) in vault.emissions.iter().enumerate() {
        let e = &mut self.emissions[index];
        let earned_emission =
            calc_share_value(e.last_seen_index, emission.final_index, sy_balance);
        e.inc_staged(earned_emission);
        e.last_seen_index = emission.last_seen_index;
    }
}
```

Because both `max_amount_in` and `amount_out` are rounded down, a situation may arise where the pool's `max_amount_out` may never be fully extracted. This may result in the pool remaining perpetually in the `POOL_STATUS_OPEN` state even though the majority of its tokens have been sold.

Remediation

Ensure that rounding occurs only in one step.

Patch

Fixed in 0923bc9.

04 — 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-HPA-SUG-00	`update_swap_fee_bps` lacks a limit check, allowing excessively high fees that may negatively impact users and the system.
OS-HPA-SUG-01	Suggestion to record the time at which the migration was reached.

Unchecked Swap Fee Value

OS-HPA-SUG-00

Description

`meme::update_swap_fee_bps` allows an admin to update the `swap_fee_bps` parameter of the `MemeConfig` structure. However, it currently lacks any check to limit the maximum value of `swap_fee_bps`. Thus, it is possible to set it to more than 10000 bps (more than 100%). If an admin sets the swap fee to an excessively high value, users attempting to perform swaps will receive no tokens in return, resulting in a loss of their funds.

>_ contracts/hopfun/sources/meme.move RUST

```
public fun update_swap_fee_bps(
    _cap: &AdminCap,
    config: &mut MemeConfig,
    swap_fee_bps: u64,
) {
    config.swap_fee_bps = swap_fee_bps;
}
```

Remediation

Add a validation check in `update_swap_fee_bps` to ensure that `swap_fee_bps` does not exceed the predefined threshold of 10000 bps.

Setting Migration Time

OS-HPA-SUG-01

Description

Set `curve.reached_migration_at` when the curve has reached migration within `buy` to record the timestamp at which migration was reached.

Remediation

Implement the above suggestion.

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.