Haedal haSUI

Security Assessment

August 1st, 2025 — Prepared by OtterSec

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Haedal engaged OtterSec to assess the `Haedal haSUI` program. This assessment was conducted between January 30th and February 11th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 1 finding throughout this audit engagement.

In particular, we made a recommendation to modify the getter function to include the uncollected protocol fees when retrieving the SUI vault amount (OS-HFR-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/haedallsd/haedal-protocol. This audit was performed against 510212e.

A brief description of the program is as follows:

Name	Description
Haedal haSUI	Haedal is the leading liquid staking protocol on Sui, boosting yield with automated vaults and more DeFi initiatives.

03 — 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-HFR-SUG-00	The getter function responsible for retrieving the SUI vault amount always returns zero because `uncollected_protocol_fees` are not deposited into the vault.

Incorrect Protocol Fee Accounting OS-HFR-SUG-00

Description

In the current implementation of `staking`, `get_protocol_sui_vault_amount` returns only the balance in `protocol_sui_vault`, but `uncollected_protocol_fees` are never deposited into this vault until explicitly collected. As a result, the function always returns zero, even if protocol fees have accumulated.

>_ haedal-protocol/sources/staking.move RUST

```
public fun get_protocol_sui_vault_amount(staking: &Staking): u64 {
    vault::vault_amount(&staking.protocol_sui_vault)
}
```

Remediation

Update the existing getter function to include uncollected fees.

Patch

Resolved in 6fabffc.

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.