Haedal HMM

Security Assessment

February 18th, 2025 — Prepared by OtterSec

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Haedal engaged OtterSec to assess the `haedal-pmm` 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 3 findings throughout this audit engagement.

In particular, we identified a vulnerability where rounding the additional quote purchase amount to zero allows a user to receive base tokens without paying any quote tokens (OS-HPM-ADV-00).

We also made recommendations to address several rounding issues. In particular, if the primitive price is very low and spare quote value is one, rounding down the fair amount to zero results in no penalty (OS-HPM-SUG-01). Traders may exploit fee rounding by selling minimal amounts, receiving one unit of quote coin, and avoiding fees due to rounding down of LP and protocol fee quote values (OS-HPM-SUG-00). While the team acknowledges these vulnerabilities, they remain unfixed on the grounds that the potential monetary exploits are insignificant.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/haedallsd/haedal-pmm-contract. This audit was performed against 72edcec.

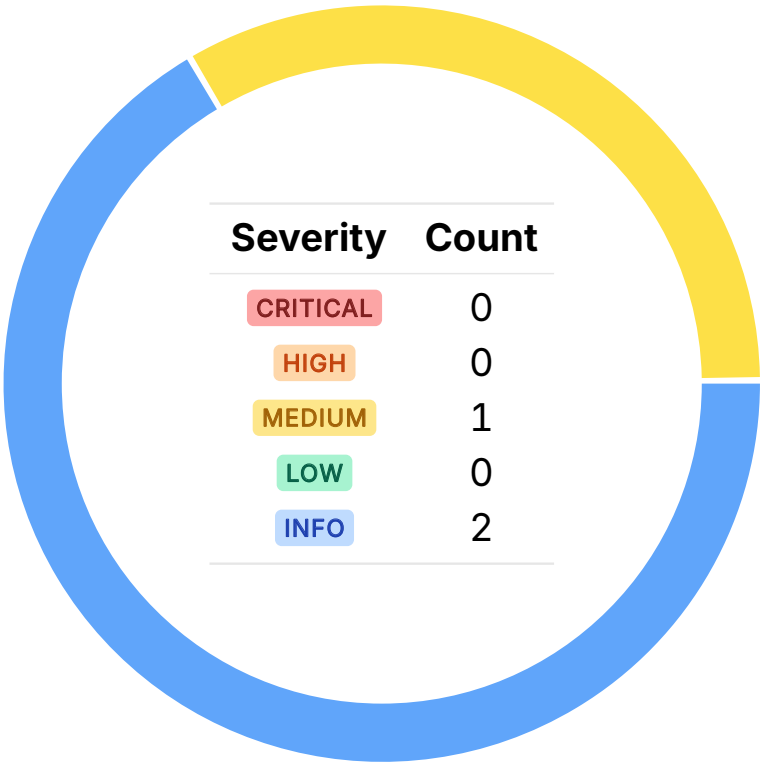
A brief description of the program is as follows:

Name	Description
haedal-pmm	A decentralized trading platform powered by the Proactive Market Maker (PMM) algorithm.

03 — Findings

Overall, we reported 3 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	0
MEDIUM	1
LOW	0
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-HPM-ADV-00	MEDIUM	RESOLVED	A user may receive extra base tokens without paying additional quote if a small purchase results in `r_one_buy_base_coin` to return 0, resulting in underpayment.

Receiving Additional Base Tokens at Zero Additional Quote MEDIUM OS-HPM-ADV-00

Description

In `query_buy_base_coin_by_r_below_one` within `oracle_driven_pool`, when the buy request results in `R` to cross from below 1 to above 1, the quote payment is split between restoring `R = 1` and paying for the excess base tokens. If the extra base amount (`buy_base_amount - back_to_one_receive_base`) is very small, the `r_one_buy_base_coin` calculation may return 0 due to rounding. This allows the user to receive additional base tokens without paying any extra quote, effectively underpaying.

>_ haedal-pmm-contract/sources/oracle_driven_pool.move RUST

```
fun query_buy_base_coin_by_r_below_one<CoinTypeBase, CoinTypeQuote>(
    pool: &Pool<CoinTypeBase, CoinTypeQuote>,
    primitive_price:u64,
    buy_base_amount:u64,
    new_base_target:u64,
    new_quote_target:u64,
):(u64, RStatus){
    [...]
    else {
        // case 3.3: R status changes to ABOVE_ONE
        (
            back_to_one_pay_quote + r_one_buy_base_coin(
                primitive_price,
                pool.get_k(),
                buy_base_amount - back_to_one_receive_base,
                new_base_target,
            ),
            RStatus::ABOVE_ONE,
        )
    }
}
```

Remediation

Check that `r_one_buy_base_coin > 0`.

Patch

Resolved in e9bb15d.

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-HPM-SUG-00	Traders may exploit fee rounding by selling minimal amounts, receiving one unit of quote coin, and avoiding fees due to rounding down of `lp_fee_quote` and `protocol_fee_quote`.
OS-HPM-SUG-01	When the `primitive_price` is very low and `spare_quote` is one, rounding down `fair_amount` to zero results in no penalty.

Fee Evasion Due to Rounding Errors OS-HPM-SUG-00

Description

In `query_sell_base_coin` within `oracle_driven_pool`, traders may exploit fee rounding by selling very small amounts of base coins, resulting in a `receive_quote` value of one. Due to the way liquidity provider (LP) fees and protocol fees (`lp_fee_quote` and `protocol_fee_quote`) are calculated, this may allow traders to avoid paying any fees because of rounding behavior. Both `lp_fee_quote` and `protocol_fee_quote` are rounded down due to integer arithmetic. If `receive_quote` is one, the result of both multiplications is zero, implying no fees are deducted.

>_ sources/oracle_driven_pool.move RUST

```
public fun query_sell_base_coin<CoinTypeBase, CoinTypeQuote>(
    pool: &Pool<CoinTypeBase, CoinTypeQuote>,
    primitive_price:u64,
    amount:u64,
):(u64, u64, u64, RStatus, u64, u64){
    let (new_base_target, new_quote_target) = pool.get_expected_target(primitive_price);
    [...]
    // count fees
    let lp_fee_quote = safe_math::mul(receive_quote, pool.get_lp_fee_rate());
    let protocol_fee_quote = safe_math::mul(receive_quote, pool.get_protocol_fee_rate());
    receive_quote = receive_quote - lp_fee_quote - protocol_fee_quote;
    [...]
}
```

Remediation

Ensure proper rounding direction to avoid paying of the fees.

Patch

Resolved in b5924ce.

Failure to Impose Penalty Due to Improper Rounding OS-HPM-SUG-01

Description

In `get_withdraw_quote_penalty_by_r_below_one`, the multiplication in the calculation of `fair_amount` (`safe_math::mul(spare_base, price)`) may result in an edge case where the `fair_amount` is rounded down to zero if `primitive_price` is less than 10^9 and `spare_quote` is one, resulting in a scenario where no penalty is applied. Thus, a trader may withdraw quote coins without paying any penalty, even though the pool's reserve ratio is below one.

>_ *sources/oracle_driven_pool.move* RUST

```rust
fun get_withdraw_quote_penalty_by_r_below_one<CoinTypeBase,CoinTypeQuote>(
    pool: &Pool<CoinTypeBase,CoinTypeQuote>,
    price:u64,
    amount:u64,
) :(u64) {
    let quote_balance = pool.get_quote_balance();
    let spare_base = pool.get_base_balance() - pool.get_target_base_coin_amount();
    let fair_amount = safe_math::mul(spare_base, price);
    [...]
}
```

Remediation

Utilize `mul_ceil` to round up the product.

Patch

Resolved in 8ecf616.

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.