Neutron Slinky STD

Security Assessment

April 7th, 2025 — Prepared by OtterSec

Yordan Stoychev anatomic@osec.io

James Wang james.wang@osec.io

Table of Contents

TABLE OF CONTENTS

01 — Executive Summary

Overview

Thesis engaged OtterSec to assess the `neutron slinky vault` program. This assessment was conducted between February 20th and March 21st, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 13 findings throughout this audit engagement.

In particular, we identified a vulnerability where the deposit value is incorrectly calculated by adding the deposit value of token zero twice, overestimating the deposit value (OS-NSV-ADV-04), and the incorrect calculation of the LP tokens to mint by utilizing the total contract balance instead of the newly deposited amount (OS-NSV-ADV-00). Additionally, the value of total shares is never updated after initialization, resulting in every deposit to be treated as the first, allowing users to mint shares disproportionately (OS-NSV-ADV-01). Also, the total shares are not properly updated after burning LP tokens during withdrawals, resulting in an inflated value of total shares, which affects share calculations during minting (OS-NSV-ADV-03).

We also made recommendations to ensure adherence to coding best practices (OS-NSV-SUG-05) and suggested accounting for potential rounding effects, as currently, the vault rounds down token amounts when withdrawing DEX shares, resulting in minor losses (OS-NSV-SUG-01). We further advised incorporating a check to ensure that both tokens have the same quote (OS-NSV-SUG-04), and also to modify the square root algorithm such that it returns an error on overflow, rather than assuming success (OS-NSV-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/neutron-org/slinky-vault. This audit was performed against commit b2f8274.

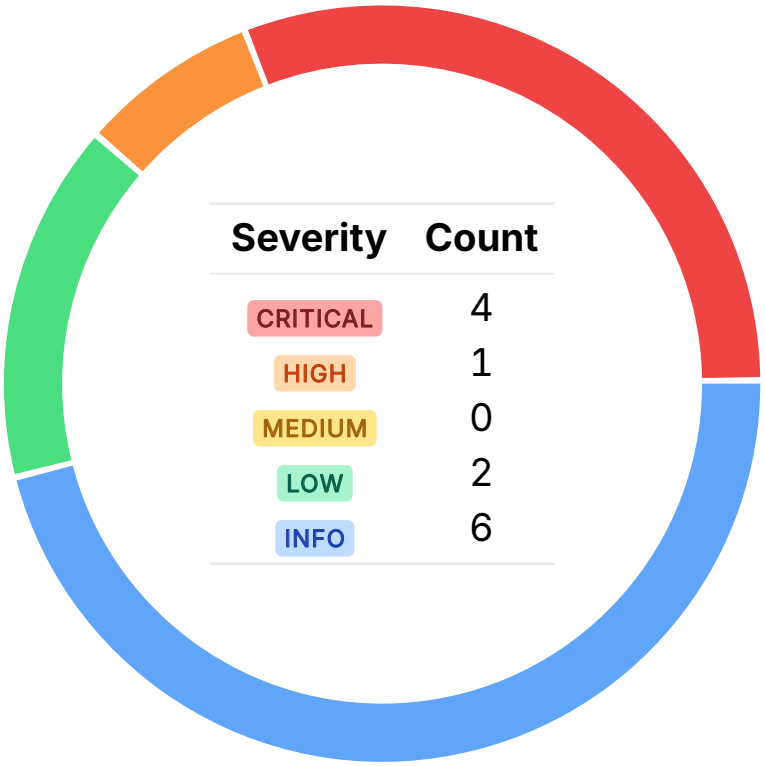
A brief description of the programs is as follows:

Name	Description
mmvault	It functions as a managed liquidity vault that interacts with a DEX to facilitate automated market making and liquidity management.
precDec	A high-precision fixed-point arithmetic library which defines a decimal type with 27 decimal places.

03 — Findings

Overall, we reported 13 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	4
HIGH	1
MEDIUM	0
LOW	2
INFO	6

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-NSV-ADV-00	CRITICAL	RESOLVED	`deposit` incorrectly calculates LP tokens to mint by utilizing the total contract balance instead of the newly deposited amount.
OS-NSV-ADV-01	CRITICAL	RESOLVED	The `total_shares` variable is never updated after initialization, resulting in every deposit to be treated as the first, allowing users to mint shares disproportionately.
OS-NSV-ADV-02	CRITICAL	RESOLVED	While calculating the adjusted tick index, the function assumes `PrecDec` supports negative values. However, since it is unsigned, if `value_token_0` is smaller than `value_token_1`, a subtraction error occurs, preventing the proper imbalance calculation.
OS-NSV-ADV-03	CRITICAL	RESOLVED	`total_shares` is not properly updated after burning LP tokens during withdrawals, resulting in an inflated `total_shares` value, which affects share calculations during minting.
OS-NSV-ADV-04	HIGH	RESOLVED	The deposit value is incorrectly calculated by adding `deposit_value_0` twice, overestimating the deposit value.

OS-NSV-ADV-05	LOW	RESOLVED	`get_mint_amount` checks for zero before casting to `Uint128`, but the casting itself may truncate small values to zero, resulting in unintended zero-minting.
OS-NSV-ADV-06	LOW	RESOLVED	The `imbalance` parameter in `ConfigUpdateMsg` is included in the message structure but is not processed in `update_config`.

Inaccurate LP Token Minting CRITICAL OS-NSV-ADV-00

Description

In `mmvault`, `execute::deposit` incorrectly computes the value of the current deposit by calling `get_token_value` with the total balance of the contract instead of the deposited amount. `get_token_value` is supposed to determine the value of the deposited tokens. However, instead of passing the actual deposit amounts (`token0_deposited` and `token1_deposited`), it utilizes `total_amount_0` and `total_amount_1`, which represent the entire balance of the contract, including past deposits. As a result, when `get_mint_amount` is called, it receives the total contract balance as input rather than the new deposit, resulting in inaccurate LP token minting.

>_ mmvault/src/execute.rs RUST

```rust
pub fn deposit(deps: DepsMut, env: Env, info: MessageInfo) -> Result<Response, ContractError> {
    [...]
     // Get the value of the tokens in the contract
    let (deposit_value_0, deposit_value_1) =
        get_token_value(prices.clone(), total_amount_0, total_amount_1)?;
    [...]
    // get the amount of LP tokens to mint
    let amount_to_mint = get_mint_amount(
        config.clone(),
        prices,
        deposit_value_0,
        deposit_value_1,
        total_amount_0,
        total_amount_1,
    )?;
    [...]
}
```

Remediation

Modify the call to `get_token_value` to utilize only the newly deposited amounts rather than the total contract balance.

Patch

Resolved in commit 7e45894.

Uninitialized Shares Resulting in Uncontrolled Share Issuance CRITICAL OS-NSV-ADV-01

Description

The vulnerability concerns how `total_shares` is handled within `utils::get_mint_amount`. `total_shares` is declared inside the `if` block but is not assigned back to any external variable. This implies that when the `if` block executes, the calculated `total_shares` is immediately discarded. Thus, even after the first deposit, the function never updates the `total_shares` state in `config`. Since `config.total_shares` remains `Uint128::zero()`, every subsequent deposit satisfies the `if` condition and is handled as an initial deposit, disregarding the proportionate share calculation for existing liquidity providers.

>_ *mmvault/src/utils.rs* RUST

```rust
pub fn get_mint_amount(
    config: Config,
    prices: CombinedPriceResponse,
    deposited_value_token_0: PrecDec,
    deposited_value_token_1: PrecDec,
    total_amount_0: Uint128,
    total_amount_1: Uint128,
) -> Result<Uint128, ContractError> {
    [...]
    if config.total_shares == Uint128::zero() {
        // Initial deposit - set shares equal to deposit value
        let total_shares =
            deposit_value_incoming.checked_mul(PrecDec::from_ratio(SHARES_MULTIPLIER, 1u128))?;
    }
    [...]
}
```

Remediation

Declare `total_shares` outside the `if` block so its value persists.

Patch

Resolved in commit 483e392.

Inability to Represent Negative PrecDec Value CRITICAL OS-NSV-ADV-02

Description

In `utils::calculate_adjusted_tick_index`, during the calculation of imbalance ratio, `PrecDec` is an unsigned decimal type, meaning it cannot represent negative values. If `value_token_0` is smaller than `value_token_1`, the subtraction (`value_token_0 - value_token_1`) will underflow and result in an error instead of yielding a negative imbalance value. This will affect proper imbalance calculation and tick adjustment.

>_ mmvault/src/utils.rs RUST

```rust
pub fn calculate_adjusted_tick_index(
    base_tick_index: i64,
    fee: u64,
    value_token_0: PrecDec,
    value_token_1: PrecDec,
) -> Result<i64, ContractError> {
    [...]
    // Calculate the imbalance ratio (-1.0 to 1.0)
    // -1.0 means token1 completely dominates
    // 1.0 means token0 completely dominates
    // 0.0 means perfectly balanced
    let imbalance = value_token_0
        .checked_sub(value_token_1)?
        .checked_div(total_value)?;
    [...]
}
```

Remediation

Ensure the function properly handles signed imbalances, such that the tick index may move in both directions based on actual token value differences.

Patch

Resolved in commit 483e392.

Failure to Update Total Shares CRITICAL OS-NSV-ADV-03

Description

The `total_shares` value is not correctly updated during liquidity withdrawals, resulting in inaccurate LP share calculations in `get_mint_amount`. When DEX deposits exist, `total_shares` should be updated in `handle_withdrawal_reply`, but this update is missing despite the fact that LP tokens are burned in `get_withdrawal_messages`. As a result, `config.total_shares` remains higher than expected, inflating LP minting calculations and over-minting LP tokens. This results in incorrect tracking of available LP shares and miscalculations during minting in `get_mint_amount`.

>_ *mmvault/src/utils.rs* RUST

```rust
pub fn get_mint_amount([...]) -> Result<Uint128, ContractError> {
    [...]
    else {
        // Calculate proportional shares based on the ratio of deposit value to total value
        total_shares = deposit_value_incoming
            .checked_mul(PrecDec::from_ratio(config.total_shares, 1u128))
            .map_err(|_| ContractError::ConversionError)?
            .checked_div(total_value_existing)
            .map_err(|_| ContractError::ConversionError)?;
    }
    [...]
}
```

Remediation

Reduce `total_shares` after burning LP tokens in `handle_withdrawal_reply`.

Patch

Resolved in commit aa5429a.

Inflation of Deposit Value via Double Counting HIGH OS-NSV-ADV-04

Description

In `mmvault`, `execute::deposit` calculates the total value of the deposit (`deposit_value`) by adding the values of `token_0` (`deposit_value_0`) and the value of `token_1` (`deposit_value_1`). However, on the subsequent line, when checking if the total deposit value exceeds the deposit cap, the function adds `deposit_value_0` again to the previously calculated `deposit_value`, effectively doubling the value of `deposit_value_0` in this check. This may allow the deposit to exceed the actual cap because the contract is considering the deposit as larger than it actually is.

>_ *mmvault/src/execute.rs* RUST

```rust
pub fn deposit(deps: DepsMut, env: Env, info: MessageInfo) -> Result<Response, ContractError> {
    [...]
    // calculate the total deposit value
    let deposit_value = deposit_value_0.checked_add(deposit_value_1)?;
    // check if they exceed the cap
    let exceeds_cap = deposit_value.checked_add(deposit_value_0)?
        > PrecDec::from_atomics(config.deposit_cap, 0).unwrap();
    [...]
}
```

Remediation

Remove the duplicate addition of `deposit_value_0` during the limit check.

Patch

Resolved in commit 4caab71.

Unintended Zero-Minting Due to Missing Check LOW OS-NSV-ADV-05

Description

There is a potential edge case where `amount_to_mint` may become 0 due to how casting to `Uint128` is performed. `get_mint_amount` calculates the LP tokens to be minted based on deposited tokens and the liquidity pool balance. A previous check in `deposit` was removed under the assumption that `get_mint_amount` prevents zero minting. However, `get_mint_amount` checks for zero before casting to `Uint128`. If the calculated amount is rounded down to 0 upon casting, it will bypass the original zero check, resulting in zero minting.

Remediation

Ensure the `amount_to_mint.is_zero` check is performed after casting.

Patch

Resolved in commit b87286f.

Lack of Utilization of Imbalance Field LOW OS-NSV-ADV-06

Description

The `ConfigUpdateMsg` structure includes an optional imbalance field, but this field is not utilized in `execute::update_config`. Thus, even if the `imbalance` field is supplied in an `UpdateConfig` message, it will not be applied, and the contract will not actually update the `imbalance` value.

Remediation

Ensure that `update_config` explicitly checks for and updates the imbalance field in the contract's stored configuration.

Patch

Resolved in commit 483e392.

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-NSV-SUG-00	The `sqrt` function in `PrecDec` assumes that multiplication at precision level 0 will not overflow. However, due to an uneven 27 decimal places, a factor of 10 is always included, which may result in unexpected overflows and trigger a panic.
OS-NSV-SUG-01	The vault rounds down token amounts when withdrawing DEX shares, resulting in minor losses.
OS-NSV-SUG-02	`handle_withdrawal_reply` redeposits leftover funds during a withdrawal, effectively allowing withdrawal actions to also act as privileged `DexDeposit` calls.
OS-NSV-SUG-03	Utilizing `f64` for price, tick, and value calculations may introduce precision loss at extremely high token prices.
OS-NSV-SUG-04	The current validation of token pairs allows one token to have a `USD` quote while the other has a different quote, resulting in potential pricing inconsistencies.
OS-NSV-SUG-05	Suggestions regarding ensuring adherence to coding best practices.

Overflow Due to Improper Precision Scaling OS-NSV-SUG-00

Description

`PrecDec::sqrt` aims to compute the square root of a high-precision decimal without overflowing. However, due to the uneven number of decimal places (27), a bug exists that causes a panic due to an overflow at precision level 0 when a sufficiently large number is provided. The algorithm iterates over precision levels from 13 down to 0. At each precision level, it calls `sqrt_with_precision(i)` to attempt a square root computation until it manages to perform a square root operation without overflowing. At each precision level, `sqrt_with_precision(i)` scales the input number up before computing the square root.

>_ *neutron-std/src/types/neutron/util/precdec.rs* RUST

```rust
#[must_use = "this returns the result of the operation, without modifying the original"]
pub fn sqrt(&self) -> Self {
    // We start with the highest precision possible and lower it until
    // there's no overflow.
    (0..=Self::DECIMAL_PLACES / 2)
        .rev()
        .find_map(|i| self.sqrt_with_precision(i))
            // The last step (i = 0) is guaranteed to succeed because `isqrt(u256::MAX) * 10^13`
                ↪  does not overflow
        .unwrap()
}

#[must_use = "this returns the result of the operation, without modifying the original"]
fn sqrt_with_precision(&self, precision: u32) -> Option<Self> {
    let inner_mul = Uint256::from(10u128).pow(precision * 2 + 1);
    self.0.checked_mul(inner_mul).ok().map(|inner| {
        let sq = inner.isqrt();
        let outer_mul = Uint256::from(10u128).pow(Self::DECIMAL_PLACES / 2 - precision);
        Self(inner.isqrt().checked_mul(outer_mul).unwrap())
    })
}
```

Due to the imbalance in decimal places, the inner multiplication always includes a factor of 10, even at precision level 0. The algorithm incorrectly assumes that at this level, the inner product is simply the atomics value and will not overflow. However, this assumption is flawed. Consequently, when `sqrt_with_precision(0)` fails, `sqrt` unwraps the `None`, triggering a panic.

Remediation

Modify the square root algorithm so that the `sqrt()` algorithm returns an error on overflow, rather than assuming success. This will imply that not all values within the `PrecDec` range are guaranteed to be square-rootable by design.

Loss of Value Due to Improper Rounding OS-NSV-SUG-01

Description

In the current implementation, rounding down the token amounts attributed to DEX shares owned by the vault creates a small, systematic loss of value over time during both simulated and actual withdrawals. This may be exploited, allowing an attacker to slightly increase their vault shares during mints and impose minor losses on the vault during withdrawals. While the cost of such an attack currently outweighs the profit, it is advisable to harden the contract to mitigate these losses.

>_ mmvault/src/execute.rs RUST

```rust
pub fn deposit(deps: DepsMut, env: Env, info: MessageInfo) -> Result<Response, ContractError> {
    [...]
    // get total contract balance, including value in active deposits.
    let (total_amount_0, total_amount_1) =
        get_virtual_contract_balance(env.clone(), &deps, config.clone())?;
    [...]
}

pub fn withdraw([...]) -> Result<Response, ContractError> {
   [...]
    // Add all withdrawals from existing deposits
    for deposit in res.deposits.iter() {
        let msg_withdrawal = MsgWithdrawal {
            [...]
        };
        messages.push(msg_withdrawal.into());
    }
    [...]
}
```

Remediation

Ensure to account for potential rounding effects.

Privileged Redeposit via Withdrawal OS-NSV-SUG-02

Description

Currently, the withdrawal operation in `execute::handle_withdrawal_reply` also performs a redeposit of leftover funds into the DEX, effectively mimicking the behavior of a `DexDeposit` operation. This dual behavior introduces a potential issue, as it gives users who are withdrawing funds the ability to interact with the DEX's liquidity pool in a privileged way.

>_ *mmvault/src/execute.rs* RUST

```rust
pub fn handle_withdrawal_reply([...]) -> Result<Response, ContractError> {
    match msg_result {
        SubMsgResult::Ok(_result) => {
            [...]
            // Create deposit messages
            let deposit_msgs = get_deposit_messages(
                &env,
                config,
                tick_index,
                prices,
                balances[0].amount - withdraw_amount_0,
                balances[1].amount - withdraw_amount_1,
            )?;[...]
        }[...]
    }
}
```

Remediation

Verify whether this behavior is intentional.

Precision Loss in Calculations

OS-NSV-SUG-03

Description

The code relies on `f64` to handle calculations involving `PrecDec`, which represents high-precision decimal values. However, `f64` has a limited precision. This implies that operations involving token prices, tick values, and deposited value calculations may suffer from precision loss if the token prices reach unrealistically high values. Thus, even though this occurs in only extreme cases, it may be prudent to appropriately handle such edge cases.

Remediation

Ensure to address the above edge cases to avoid any inaccuracies due to precision loss.

Incomplete Quote Validation OS-NSV-SUG-04

Description

The issue revolves around the validation of token pairs, specifically the validation of the quote field of tokens during contract instantiation. The existing validation only enforces that if both tokens share the same quote, that quote must be `USD`. However, there is an implicit gap in the validation. One token may have a valid `USD` quote, but the other token may have a completely different quote.

>_ *mmvault/src/msg.rs* RUST

```rust
pub fn validate(&self) -> ContractResult<()> {
    [...]
    if self.token_a.pair.quote == self.token_b.pair.quote && self.token_b.pair.quote != "USD" {
        return Err(ContractError::OnlySupportUsdQuote {
            quote0: self.token_a.pair.quote.clone(),
            quote1: self.token_b.pair.quote.clone(),
        });
    }
    Ok(())
}
```

Remediation

Ensure that both tokens have the same quote.

Code Maturity OS-NSV-SUG-05

Description

1. In the current implementation of `execute`, the functionality to stop the contract by checking `config.paused` is only present in `dex_deposit` and not in the `deposit` call. Verify if this behavior is intentional.

   >_ *mmvault/src/execute.rs* RUST

   ```rust
   pub fn dex_deposit(deps: DepsMut, env: Env, info: MessageInfo) -> Result<Response,
       ↪  ContractError> {
       [...]
       CONFIG.save(deps.storage, &config)?;

       if config.paused {
           return Err(ContractError::Paused {});
       }
       [...]
   }
   ```

2. It is recommended to block all LP token-related functionalities until `create_token` is called. While the current implementation causes these APIs to revert at some stage, relying on implicit guards can be error-prone.

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.