Security Audit Report

Neutron Q1 2025: MMVault CW Contract

Last revised 16.04.2025

Authors: Darko Deuric

Contents

Audit overview

The Project

The `mmvault` contract is a CosmWasm smart contract deployed on **Neutron**, designed to automate liquidity provisioning on a DEX using dynamic strategies. It integrates with the **Slinky Oracle** to fetch real-time price data and interacts with the Neutron **DEX** module to perform periodic withdrawals and re-deposits based on price and configuration parameters. The contract is part of the broader **Supervaults** architecture, aiming to optimize yield for users while maintaining secure and predictable fund management.

Scope of this report

This audit focused exclusively on the `mmvault` contract and its interaction with:

- Neutron DEX modules (via `MsgDeposit` and `MsgWithdrawal`)
- Slinky Oracle module (for price fetching)
- Internal LP share accounting and vault logic (deposit, withdrawal, mint/burn flows)

The scope **did not include** the implementation of external modules (DEX, Oracle, Cron) themselves, but the audit assumed their behavior as defined in Neutron documentation.

Audit plan

The audit lasted three weeks (from **March 24 to April 11**) and included the following activities:

- Kickoff call to align on expectations and clarify system boundaries
- Creating diagrams of `mmvault` flows (deposits, withdrawals, DEX interaction, admin ops)
- Definition of system properties, followed by code checks ensuring their validity
- Detailed review of critical execution paths and LP share accounting
- Continuous collaboration with the client (Nicholas) via Slack and sync meetings
- Final closure meeting to present findings and discuss potential improvements

Conclusions

- No Critical or High severity issues were found.
- The contract successfully avoids common pitfalls in LP accounting and DEX integration.
- **7 findings** were reported:
  - **2 Low** severity
  - **5 Informational**

Audit Dashboard

Target Summary

- **Type: Protocol and Implementation**
- **Platform: Rust / Cosmwasm smart contract**
- **Artifacts:** `contracts/mmvault`

Engagement Summary

- **Dates**: March 24 - April 11
- **Method**: Code review

Severity Summary

Finding Severity	Number
Critical	0
High	0
Medium	0
Low	2
Informational	5
Total	**7**

System Overview

The `mmvault` contract is a **liquidity management vault** deployed on the Neutron blockchain. Its primary purpose is to manage user deposits of two tokens (e.g., `token0` and `token1`) and efficiently allocate liquidity into the Neutron DEX (Duality) by performing automated deposits and withdrawals, driven by off-chain price data and strategic configuration parameters.

The vault contract abstracts away direct DEX interactions from end users. Instead, users interact with the vault through standardized deposit and withdrawal flows, while the vault itself takes full custody of user assets and autonomously manages liquidity provisioning via Cron-triggered rebalancing logic.

Core Flows

User Deposit

Users can deposit paired tokens into the vault using the `ExecuteMsg::Deposit` entry point. In return, they receive **LP shares** (minted using the token factory module) representing their proportional claim on the vault's total value. Shares are minted based on:

- Vault's total value (fetched using slinky price oracle),
- Existing LP supply (`config.total_shares`), and
- Whether the deposit is the first-ever or a subsequent one.

Shares are minted using the `get_mint_amount()` logic, ensuring proportional issuance. LP share tracking is maintained through `config.total_shares`, which increases on every deposit.

User Withdraw

Users can redeem their LP shares via `ExecuteMsg::Withdraw`, triggering a burn of the specified amount of LP shares. If the vault holds no active DEX positions, the withdrawal immediately transfers the user's share of vault-held tokens using `get_withdrawal_messages()`. If active DEX deposits exist, the vault issues a submessage to withdraw them (`MsgWithdrawal`), and the actual token payout happens in the reply handler, followed by a fresh `MsgDeposit` to redeploy remaining liquidity.

The withdrawal flow ensures that `config.total_shares` decreases by the exact burned amount.

Automated DEX Rebalancing via Cron module

The vault relies on Neutron's **Cron module** to trigger periodic rebalancing of its DEX positions through two privileged entry points:

1. `dex_withdrawal()`:

   Executed first each block by Cron, this function:

   - Queries all active DEX positions owned by the vault,
   - Issues `MsgWithdrawal` messages to remove liquidity from all active pools,
   - Ensures the vault regains custody of all tokens before re-depositing.

2. `dex_deposit()`:

   Executed immediately after `dex_withdrawal()`, it:

   - Verifies that all prior positions have been withdrawn,
   - Fetches fresh oracle prices via the **Slinky module**,
   - Computes the target tick index and rebalancing logic based on:
     - Fee tier allocation,

  - Imbalance between `token0`/`token1` values,
  - Configured skew/imbalance settings,
- Issues `MsgDeposit` messages to redeploy liquidity across different price ranges and fee tiers.

All deposits and withdrawals are issued on behalf of the vault (`creator == receiver == contract address`), ensuring it retains full custody of all funds and LP shares.

Diagrams

All main execution flows—including user `deposit`/`withdraw` interactions, cron-triggered DEX operations, as well as state machine transitions are illustrated in the Appendix: Diagrams section of this report.

Threat Model

System Properties

All key system properties of the `mmvault` contract have been reviewed and validated. This includes invariants related to deposit/withdrawal correctness, tick index behavior, price staleness checks, fee tier allocation, LP share accounting, and execution permissions.

General

- The system **cannot** be permanently stuck in the `Paused` state. (**liveness**)
  - The `check_staleness()` function sets:

```
config.pause_block = env.block.height;
config.last_executed = env.block.time.seconds();
```

  when:

```
(env.block.time.seconds() - config.last_executed) > config.timestamp_stale
```

  This indicates a stale price / downtime. It pauses **only** for the current block.
  - Other functions (like `deposit`, `dex_deposit`, etc.) call `check_staleness()` early on. Inside that function:

```
if config.pause_block == env.block.height {
    return Err(ContractError::BlockOnHold {});
}
```

  So, only that one block (current height) is affected.
  - On the next block, `env.block.height` will no longer match `pause_block`, and:
    * The stale condition likely won't re-trigger, since `last_executed` was updated.
    * So the system re-enters `Normal / Ready` state.
- Only whitelisted addresses **can** modify config.
- Only whitelisted addresses **may** call:
  - `dex_deposit`
  - `dex_withdrawal`
  - `UpdateConfig`
  - `CreateToken`
- The contract **must** reject operations if the oracle price is stale beyond the allowed threshold, ensuring deposits and withdrawals are based on up-to-date pricing data.
  - If the block is stale (`now - last_executed > timestamp_stale`), the contract **must**:
    * Set `pause_block = current block height`.
    * Refund sent funds (if any) and exit early.
  - While `pause_block == current block height`, the contract **must** reject execution (no deposit, withdraw, or DEX-related action).
  - Once a fresh block arrives (`pause_block != current block height` and not stale), normal execution **must** resume.

Return Value	Meaning
`Err(ContractError::BlockOnHold)`	Contract is on hold this block (via `pause_block`), reject immediately

Return Value	Meaning
`Ok(Some(response))`	Contract is stale: vault detected downtime — must reject and refund
`Ok(None)`	Normal: vault is not stale or paused, continue execution

- One vault instance is designed to provide liquidity for **exactly one** token pair (`token_0` and `token_1`).
- LP shares are **fairly** distributed:
  - New users receive LP shares **proportional** to the USD value of their deposit, not to the raw amount of tokens sent.
  - vault **doesn't** over-mint.
  - vault **doesn't** under-mint.
- LP share supply reflects net deposits and withdrawals.
  - after a successful deposit, the number of LP shares minted **must** be exactly equal to the amount added to `config.total_shares`.
  - after a successful withdrawal, the number of LP shares burned **must** be exactly equal to the amount subtracted from `config.total_shares`.
- The contract **should** rely solely on `config.total_shares`, which is updated during deposits and withdrawals, rather than querying the total chain-wide supply of the LP token.
  - **Finding**: Use `config.total_shares` Instead of Querying LP Supply from Chain.
- For both `MsgDeposit` and `MsgWithdrawal`, the `creator` and `receiver` fields **must** be set to the contract's own address.
- `execute_create_token` **must** be called before `deposit` or `withdraw`. (**assumption**)
- `price_0_to_1` must always be updated whenever `token_0_price` or `token_1_price` are changed.
- Fee tier percentage sum **must** equal 100%.

Deposit

- Accepts either `token_0`, `token_1`, or both.
- If `config.paused`, early error return.
- Staleness is enforced - `check_staleness`.
- Deposit cap is enforced (but can be bypassed by whitelisted addresses).
  - user's current deposit is included in the value that's checked against the cap.
- LP shares must be minted **value-proportionally**, as defined by the `get_mint_amount` function. The number of shares minted depends on whether the deposit is the first-ever or a subsequent one.
  - For the **first-ever deposit** (`config.total_shares == 0`): `total_shares_to_mint = deposit_value * SHARES_MULTIPLIER`
  - For **subsequent deposits**: `total_shares_to_mint = deposit_value * total_shares / total_value_existing`, where `total_value_existing = total_value - deposit_value` (i.e., the value before the current deposit).
- The total LP shares (`config.total_shares`) **must not** increase unless the user has actually transferred tokens into the contract during a deposit.
- Prevents 0 LP shares amount mint.

Dex Deposit

- Callable only by whitelisted actors.
- If `config.paused`, early error return.
- Staleness is enforced - `check_staleness`.

- Skip deposit if existing active deposits are found.
- **Fresh** price from oracle is used.
- `tick_index` is calculated **correctly**.
  - According to the docs, the price formula is `p(i)=1.0001^i => tick_index = log(p) / log(1.0001)`.
  - According to the code, `tick_index = -log(p) / log(1.0001)`.
  - It's confirmed by Neutron team that the code implementation is correct.
- The vault only places liquidity at tick indices derived from the latest Slinky price, optionally adjusted by config-defined `skew` setting.
  - All tick indices are based on the Slinky price (`dex_deposit`):

```
let prices: CombinedPriceResponse = get_prices(deps.as_ref(), env.clone())?;
let tick_index = price_to_tick_index(prices.price_0_to_1)?;
```

  - Tick index may only be adjusted via `calculate_adjusted_tick_index`, which is gated by `skew` (`get_deposit_data`):

```
let adjusted_tick_index = if skew {
    calculate_adjusted_tick_index(tick_index, fee, total_value_token_0,
↪   total_value_token_1)?
} else {
    tick_index
};
```

  - All `MsgDeposit` messages use this single tick index per `dex_deposit` execution. In `get_deposit_messages` (both for base and subsequent fee tiers):

```
tick_indexes_a_to_b: vec![deposit_data.tick_index],
```

- The `imbalance` config value **does not** affect the tick index.
  - In `get_deposit_data`, the `imbalance` value is only used to compute additional token amounts to deposit → affects amounts only, not price.
- If the imbalance is greater than 0, then more of the oversupplied token goes to the tightest spread.
  - Correct and enforced in `get_deposit_data`:
    - Imbalance is calculated from the USD value difference between tokens.
    - Only the first fee tier (tightest spread) is adjusted based on it.
    - Boosted amount for underweighted token is proportional to `imbalance%`.
- The amounts sent in `MsgDeposit` **must not** exceed the contract's available token balances (`token_0_balance`, `token_1_balance`).
  - For the first `MsgDeposit`:
    - The very last part of the logic in `get_deposit_data` explicitly caps the computed amounts.
    - This guarantees that:
      - `amount0 <= token_0_balance`
      - `amount1 <= token_1_balance`
    - The `DepositResult` uses these capped values.
  - For remaining fee tiers, each amount is computed from:

```
let total_amount0_to_distribute = remaining_amount0;
let total_amount1_to_distribute = remaining_amount1;
```

    and,

```
let remaining_amount0 = token_0_balance - deposit_data.amount0;
let remaining_amount1 = token_1_balance - deposit_data.amount1;
```

    This ensures that:
    - All secondary deposits only use what's left over after the base deposit.
    - Even if there are multiple tiers, no cumulative over-deposit can occur.
- `MsgDeposit` **must not** be emitted for any fee tier if both `amount0` and `amount1` are zero.
  - The code explicitly avoids emitting a `MsgDeposit` when both `amount0` and `amount1` are zero, for all fee tiers.
    - For the first fee tier, it only emits a message if at least one amount is non-zero.
    - For remaining fee tiers:
      - check for non-final

  - check for final
- Token amounts **must** be distributed across configured fee tiers proportionally to their configured `percentage` values.
  - In `get_deposit_messages`, this logic applies to all fee tiers after the first one (which uses `get_deposit_data` and may apply `imbalance` logic):

```
// Total to distribute across remaining fee tiers
let total_amount0_to_distribute = remaining_amount0;
let total_amount1_to_distribute = remaining_amount1;

...

// Proportional distribution for each fee tier:
let amount_0 = total_amount0_to_distribute
    .multiply_ratio(fee_tier.percentage as u128, remaining_percentages as u128);
let amount_1 = total_amount1_to_distribute
    .multiply_ratio(fee_tier.percentage as u128, remaining_percentages as u128);
```

  - `remaining_percentages` is the sum of all tier percentages excluding the first.
  - So each tier receives:

```
tier_amount = (total_remaining_amount * tier_percentage) / total_remaining_percentage
```

  - This guarantees that distribution is percentage-based.
- Final tier **must** use all remaining tokens
  - This is enforced in `get_deposit_messages`, inside the loop over remaining fee tiers:

```
for (i, fee_tier) in remaining_tiers.iter().enumerate() {
    if i == remaining_tiers.len() - 1 {
        let amount_0 = total_amount0_to_distribute - distributed_amount0;
        let amount_1 = total_amount1_to_distribute - distributed_amount1;
```

- If the value of one token (in USD) is higher, the contract will increase the deposit of the underweighted token by `imbalance%`.
  - The underweighted token gets an additional boost based on:

```
imbalance_value * imbalance_percent / token_price
```

- If `skew = true`, and there is a value imbalance between `token0` and `token1`, then the adjusted tick index **must** differ from the base tick index in a direction that incentivizes rebalancing the vault.
  - If tokens are balanced, tick is unchanged.
  - If token0 dominates,
    - tick increases (shifts right), placing liquidity to favor selling `token0`.
    - price of token0 becomes cheaper.
  - If token1 dominates,
    - tick decreases (shifts left), placing liquidity to favor selling `token1`.
    - Token1 becomes more expensive.
  - This is correct behavior since `tick_index = -log(price) / log(1.0001)` , meaning:
    - Increase tick index → lower price.
    - Decrease tick index → higher price.
  - Tick adjustment is capped by fee tier.

Withdraw

- Staleness is enforced - `check_staleness`.
- **Only** the LP token matching `config.lp_denom` is accepted.
- Withdrawals **must** burn LP tokens and return proportional token amounts.
- If no active DEX deposits exist, the vault directly handles the withdrawal by:
  - querying its own token balances,
  - computing the proportional share of tokens to return based on the LP token supply, and
  - burning the specified LP shares from the contract's own balance.

- Amount burned **cannot** be zero or greater that total LP shares.
- User receives the **correct** proportional amount of `token0` and/or `token1` during direct withdrawal.
  - the function `query_contract_balance` returns a `Vec<Coin>` of length 2, where `balances[0]` corresponds to `token_0` and `balances[1]` to `token_1`.
  - these balances are passed to `get_withdrawal_messages` as:
    - `total_amount_0 = balances[0].amount`
    - `total_amount_1 = balances[1].amount`
  - the user's withdrawal amounts are calculated as:
    - `withdraw_amount_0 = total_amount_0 * burn_amount / total_supply`
    - `withdraw_amount_1 = total_amount_1 * burn_amount / total_supply`
- A withdrawal **shouldn't** reduce the total number of LP shares (`config.total_shares`) without the user receiving a non-zero amount of tokens in return.
  - It is technically possible (though rare) for a user to burn LP shares and receive 0 amount of one or both tokens, especially when:
    - The vault holds very low balances of one token due to recent withdrawals or DEX delays.
    - The user is withdrawing a very small LP share amount.
    - Rounding in `multiply_ratio` leads to a floored zero result.
  - However, several factors significantly reduce the likelihood and systemic impact of this happening:
    - LP shares are issued with high resolution via `SHARES_MULTIPLIER = 10^9`, making precision loss minimal.
    - Vaults are typically highly liquid with many depositors, increasing total supply and granularity.
    - Most tokens used (e.g., USDC, NTRN, ATOM) have 6 or more decimals, which further improves withdraw precision.
    - Withdrawals always follow a full DEX withdrawal → balances should reflect full available funds.
  - To avoid even theoretical loss due to this edge case:
    - Client-side/UI check: Warn or block withdrawals that would result in zero output.
    - `MIN_WITHDRAW_SHARES` constant: Prevent burning trivial LP amounts that could result in rounding to 0.
- Re-deposit after a DEX withdrawal **must** always occur in `mmvault` since all active DEX deposits held by the vault are withdrawn.

Dex Withdraw

- All active positions owned by the contract (queried via `user_deposits_all`) must be included in the withdrawal batch.
- If the contract has active deposits, the function must create one or more valid `MsgWithdrawal` messages.

Create Token

- **Only** a whitelisted address can call `execute_create_token`.
- The function is callable **only once** — subsequent calls must fail if `lp_denom` is already set.
- The denom **must** follow the format `factory/{contract_address}/{subdenom}`.
- **Only** a successful `SubMsgResult::Ok` will write to state.
  - `lp_denom` is updated only inside the reply handler.
  - failed reply **does not** alter the config (no partial writes).

Findings

Finding	Type	Severity	Status
Missing Version Check in migrate() Allows Downgrade or Redundant Migrations	Implementation	Low	Acknowledged
Missing Check for Empty Whitelist During Config Update	Implementation	Low	Resolved
Vulnerability in protobuf Crate (Version 3.3.0)	Implementation	Informational	Acknowledged
Miscellaneous Code Quality Observations	Implementation	Informational	Acknowledged
Use config.total_shares Instead of Querying LP Supply from Chain	Implementation	Informational	Resolved
Redundant Configuration Save After check_staleness in deposit	Implementation	Informational	Resolved
Redundant Price Usage and Over-Parameterized get_mint_amount Interface	Implementation	Informational	Resolved

Missing Version Check in migrate() Allows Downgrade or Redundant Migrations

ID	IF-FINDING-001
Severity	Low
Impact	1 - Low
Exploitability	1 - Low
Type	Implementation
Status	Acknowledged

Involved artifacts

- src/contract.rs

Description

The `migrate()` function directly updates the contract configuration and version without verifying that the migration is actually an upgrade. CosmWasm provides the `get_contract_version()` helper for retrieving the current contract version stored in state, allowing for comparisons before applying migration logic. This pattern is essential to prevent unintentional downgrades or reapplying the same version.

Problem Scenarios

An admin could mistakenly or maliciously call `migrate()` using an older or identical contract version. Without a version check, the contract would proceed with the migration and overwrite the state, potentially introducing outdated logic or causing incompatibilities with newer state structures.

Recommendation

Use `cw2::get_contract_version()` to compare the currently stored version with the incoming version. Only proceed with migration if the incoming version is newer:

```
let current = get_contract_version(deps.storage)?.version.parse::<Version>()?;
let target = CONTRACT_VERSION.parse::<Version>()?;
if target <= current {
    return Err(ContractError::InvalidMigration {
        reason: "Cannot migrate to same or older version".to_string(),
    });
}
```

Missing Check for Empty Whitelist During Config Update

ID	IF-FINDING-002
Severity	Low
Impact	2 - Medium
Exploitability	1 - Low
Type	Implementation
Status	Resolved

Involved artifacts

- src/execute.rs

Description

The contract uses a `whitelist` to restrict access to privileged operations like `update_config`, `execute_create_token`, `dex_withdrawal`, and `dex_deposit`. While the `instantiate()` function correctly prevents deployment with an empty whitelist, the `update_config()` function allows the whitelist to be replaced with an empty vector. Once the whitelist is empty, all further admin actions become inaccessible.

Problem Scenarios

- Contract is instantiated with a valid whitelist (e.g., `["owner"]`).
- The owner calls `update_config()` and accidentally passes `whitelist: Some(vec![])`.
- The update is accepted, and the stored `whitelist` is now empty.
- All future calls to admin functions fail with `Unauthorized`, including any attempts to fix the issue.
- The only remaining recovery path is via `migrate()` .

While not exploitable by a malicious actor, this bug introduces the risk of **accidental self-bricking** of the contract. If an operator unintentionally clears the whitelist, no admin can perform critical updates or recover control.

Proof of Concept Test

```
#[test]
fn test_update_config_empty_whitelist_allowed() {
    // Setup
    let mut deps = mock_dependencies_with_custom_querier(setup_mock_querier());
    let env = mock_env();
    let initial_config = setup_test_config();

    // Store config with non-empty whitelist
    CONFIG.save(deps.as_mut().storage, &initial_config).unwrap();

    // Create update message with empty whitelist
    let update_msg = ConfigUpdateMsg {
        whitelist: Some(vec![]), // Empty vector
        max_blocks_old_token_a: None,
        max_blocks_old_token_b: None,
        deposit_cap: None,
        timestamp_stale: None,
        fee_tier_config: None,
        paused: None,
        imbalance: None,
        skew: None,
```

```
        oracle_contract: None,
    };

    // Execute update_config as owner
    let info = mock_info("owner", &[]);
    let res = execute(
        deps.as_mut(),
        env.clone(),
        info,
        ExecuteMsg::UpdateConfig { update: update_msg },
    )
    .unwrap();

    // Verify response attributes
    assert_eq!(res.attributes[0].key, "action");
    assert_eq!(res.attributes[0].value, "update_config");

    // Load updated config and verify whitelist was replaced with empty vec
    let updated_config = CONFIG.load(deps.as_ref().storage).unwrap();
    assert!(
        updated_config.whitelist.is_empty(),
        "Whitelist should be empty but isn't"
    );
}
```

Recommendation

Add an explicit check to `update_config()` to prevent setting the whitelist to an empty vector:

```
if let Some(whitelist) = update.whitelist {
    if whitelist.is_empty() {
        return Err(ContractError::EmptyValue {
            kind: "whitelist".to_string(),
        });
    }
    ...
}
```

Status

Resolved

Vulnerability in protobuf Crate (Version 3.3.0)

ID	IF-FINDING-003
Severity	Informational
Impact	3 - High
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Description

The `protobuf` crate version 3.3.0, used within the `mmvault` contract's dependencies, has a known vulnerability related to uncontrolled recursion. This issue arises due to inadequate safeguards when processing recursive data structures, leading to a stack overflow or crash. The vulnerability can be triggered when the crate processes user-supplied data, which might include nested fields that cause deep recursion.

Problem Scenarios

The `mmvault` contract depends on the `neutron-std` crate (version 5.1.2), which in turn relies on `protobuf` 3.3.0. This creates a potential attack vector where an attacker could craft malicious input that triggers the uncontrolled recursion in `protobuf`. Such an attack could lead to unexpected crashes in the contract's execution, disrupting liquidity operations, token transfers, or other vault interactions that rely on `protobuf` for data handling.

Recommendation

It is recommended to upgrade the `protobuf` crate to **version 3.7.2** or later to resolve the uncontrolled recursion issue. This update will eliminate the vulnerability and improve the stability of the entire `mmvault` contract's interactions with data encoded using `protobuf`. Ensure that all dependencies, including `neutron-std`, `slinky-oracle`, and `mmvault`, are updated to compatible versions to maintain system integrity and security.

References:

- RustSec Advisory: RUSTSEC-2024-0437
- GitHub Issue Discussion: stepancheg/rust-protobuf#749
- Debian Security Tracker: CVE-2024-7254

Miscellaneous Code Quality Observations

ID	IF-FINDING-004
Severity	Informational
Impact	0 - None
Exploitability	0 - None
Type	Implementation
Status	Acknowledged

Involved artifacts

- contracts/mmvault/src/contract.rs
- contracts/mmvault/src/execute.rs
- contracts/mmvault/src/utils.rs

Redundant `amount` Parameter in `ExecuteMsg::Withdraw`

Description:

In the `execute()` handler for `ExecuteMsg::Withdraw`, the `amount` parameter is explicitly passed:

```
ExecuteMsg::Withdraw { amount } => { ... withdraw(deps, _env, info, amount) }
```

However, the contract also retrieves the LP token amount directly from `info.funds`, validating it against `amount`. Since this must always match, the parameter is redundant and error-prone:

```
if lp_token.amount != amount {
    return Err(ContractError::LpTokenError);
}
```

Recommendation:

Eliminate the `amount` field from `ExecuteMsg::Withdraw`, and derive it only from `info.funds.first().amount`.

`WithdrawPayload` Should Be Defined Only Where It's Used

The `WithdrawPayload` is defined early in the `withdraw()` function:

```
let payload = WithdrawPayload {
    sender: info.sender.to_string(),
    amount: amount.to_string(),
};
```

However, it is only required if the `messages` list is **not empty** and the code proceeds to the `submessage` reply flow. Otherwise, it results in unnecessary allocation of the payload even in cases where it is never used (e.g. direct withdrawal with no active DEX deposits):

```
if messages.is_empty() {
    // early return - payload is unused
    ...
    return Ok(Response::new()...);
}
```

Recommendation:

Move the definition of `WithdrawPayload` into the conditional branch where `flatten_msgs_always_reply()` is called to ensure it's only created when actually needed. This improves readability and avoids unneeded computation.

Unreachable Error Arm in `handle_create_token_reply()`

Description:

This reply handler is triggered by a `reply_on_success(...)` message:

```
let msg = SubMsg::reply_on_success(MsgCreateDenom { ... }, CREATE_TOKEN_REPLY_ID);
```

However, the handler still includes logic to handle `SubMsgResult::Err`, which is unreachable under this flow:

```
SubMsgResult::Err(err) => Ok(Response::new()
    .add_attribute("action", "create_token_reply_error")
    .add_attribute("error", err)),
```

Recommendation:

Document this unreachable state explicitly or remove the error arm to avoid confusion.

Redundant Variable in `query_contract_balance()`

Description:

This line introduces a local variable unnecessarily used only once. In-place usage improves readability and reduces clutter:

```
let contract_address = env.contract.address;
```

Recommendation:

Use `env.contract.address.to_string()` directly when constructing the query.

Incorrect Tick Index Formula in Neutron DEX Documentation

Description:

The Neutron DEX documentation specifies that the tick index is calculated using the formula:

```
tick_index = log(price) / log(1.0001)
```

However, the actual formula used in the `mmvault` contract is:

```
tick_index = -log(price) / log(1.0001)
```

This reflects an inverted tick index convention where higher prices map to lower (negative) tick indices, and lower prices map to higher (positive) ticks. Tick 0 corresponds to price 1.0. This inversion is consistent across the `mmvault` implementation and confirmed as intentional.

Recommendation:

Update the Neutron DEX documentation to reflect the correct formula. Also clarify the tick index convention (i.e., tick decreases as price increases)

Incorrect Comments

Description:

The comment incorrectly describes the `pair_data` field. This likely results from a copy-paste mistake.

```
/// number of blocks until price is stale
pub pair_data: PairData,
```

Recommendation:

Correct the comment to accurately reflect the purpose of the field.

Use config.total_shares Instead of Querying LP Supply from Chain

ID	IF-FINDING-005
Severity	Informational
Impact	0 - None
Exploitability	2 - Medium
Type	Implementation
Status	Resolved

Involved artifacts

- contracts/mmvault/src/utils.rs

Description

In the `get_withdrawal_messages()` function, the contract determines the total LP token supply using:

```
let total_supply: Uint128 = deps.querier.query_supply(&config.lp_denom)?.amount;
```

However, the contract also maintains an internal record of issued LP shares via `config.total_shares`, which is updated consistently on every mint and burn:

```
// On mint
config.total_shares += amount_to_mint;

// On burn
config.total_shares = config.total_shares.checked_sub(amount)?;
```

Problem Scenarios

While `query_supply()` and `config.total_shares` are expected to match under normal operation, querying supply from the chain introduces unnecessary reliance on external state. If LP tokens are sent to or burned from the contract address outside of the expected `deposit()` / `withdraw()` flows, the on-chain total supply may become inconsistent with the internal accounting, even though users cannot redeem those tokens.

Additionally:

- Using `config.total_shares` is more consistent with how shares are managed throughout the contract.
- It is also more optimal, avoiding a bank module query.

Recommendation

Replace the use of `deps.querier.query_supply(...)` with the internal `config.total_shares` in:

```
let total_supply: Uint128 = config.total_shares;
```

This improves clarity, consistency, and aligns with the contract's own accounting model.

Status

Resolved

Redundant Configuration Save After check_staleness in deposit

ID	IF-FINDING-006
Severity	Informational
Impact	0 - None
Exploitability	3 - High
Type	Implementation
Status	Resolved

Involved artifacts

- src/execute.rs

Description

The `deposit` function calls `CONFIG.save(...)` immediately after invoking `check_staleness(...)`, which may mutate the `config`. However, `config` is guaranteed to be saved later in the function — after updating `config.total_shares` .

As a result, this early save introduces a redundant write to storage, as `config` is saved again just a few lines later.

Problem Scenarios

```
if let Some(response) = check_staleness(&env, &info, &mut config)? {
    CONFIG.save(deps.storage, &config)?;
    return Ok(response);
}

CONFIG.save(deps.storage, &config)?; // Redundant save
```

Recommendation

Remove the early `CONFIG.save(...)` after `check_staleness`. Instead, rely on the existing final `CONFIG.save(...)` near the end of the function, which will persist all mutations to `config`, including those made by `check_staleness` and the LP share accounting.

Status

Resolved

Redundant Price Usage and Over-Parameterized get_mint_amount Interface

ID	IF-FINDING-007
Severity	Informational
Impact	0 - None
Exploitability	3 - High
Type	Implementation
Status	Resolved

Involved artifacts

- mmvault/src/utils.rs

Description

The `get_mint_amount` function is responsible for calculating the number of LP shares to mint upon user deposit. However, the function redundantly recalculates the USD value of the vault's tokens (`total_value_token_0` and `total_value_token_1`) using raw token balances and prices, even though those values are already computed earlier in the deposit flow (`total_value_0` and `total_value_1`).

It also accepts overly complex input — including the full `Config` and per-token values — when it only needs:

- `total_shares`
- `deposit_value_incoming` (i.e., `deposited_value_token_0 + deposited_value_token_1`)
- `total_value` (i.e., total vault value before deposit)

Problem Scenarios

In the `deposit` function, both deposit and vault token values are already calculated:

```
let (deposit_value_0, deposit_value_1) = get_token_value(...); // already calculated
let (total_value_0, total_value_1) = get_token_value(...); // already calculated

let total_value = total_value_0.checked_add(total_value_1)?; // already calculated
let deposit_value = deposit_value_0.checked_add(deposit_value_1)?; // to add
```

However, the current implementation of `get_mint_amount` recomputes the same values using raw balances and prices:

```
let total_value_token_0 = total_amount_0 * prices.token_0_price;
let total_value_token_1 = total_amount_1 * prices.token_1_price;
let total_value_existing = total_value_token_0 + total_value_token_1 - deposit_value_incoming;
```

This duplicates work, adds room for inconsistencies, and bloats the function's parameter list.

Recommendation

Refactor the `get_mint_amount` function to use a minimal and self-contained interface. The function should receive only:

- `total_shares: Uint128`
- `deposit_value: PrecDec`
- `total_value: PrecDec`

```rust
pub fn get_mint_amount(
    total_shares: Uint128,
    deposit_value: PrecDec,
    total_value: PrecDec,
) -> Result<Uint128, ContractError> {
    let total_shares_to_mint = if total_shares.is_zero() {
        // Initial deposit: scale shares with precision multiplier
        deposit_value.checked_mul(PrecDec::from_ratio(SHARES_MULTIPLIER, 1u128))?
    } else {
        // Calculate value before this deposit
        let total_value_existing = total_value
            .checked_sub(deposit_value)
            .map_err(|_| ContractError::InvalidTokenAmount)?;

        // Proportional shares based on pre-deposit vault value
        deposit_value
            .checked_mul(PrecDec::from_ratio(total_shares, 1u128))?
            .checked_div(total_value_existing)?
    };

    if total_shares_to_mint.is_zero() {
        return Err(ContractError::InvalidTokenAmount);
    }

    precdec_to_uint128(total_shares_to_mint)
}
```

Then, in the `deposit` function:

```rust
let deposit_value = deposit_value_0.checked_add(deposit_value_1)?;
let total_value = total_value_0.checked_add(total_value_1)?;
let amount_to_mint = get_mint_amount(config.total_shares, deposit_value, total_value)?;
```

This keeps it deterministic, avoids redundant logic, and makes it easier to test.

Status

Resolved

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for $>33\%$. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

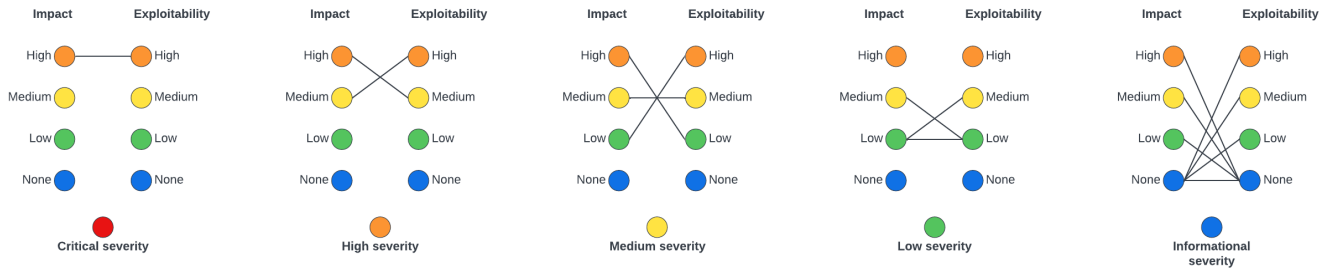

Figure 1: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Appendix: Diagrams

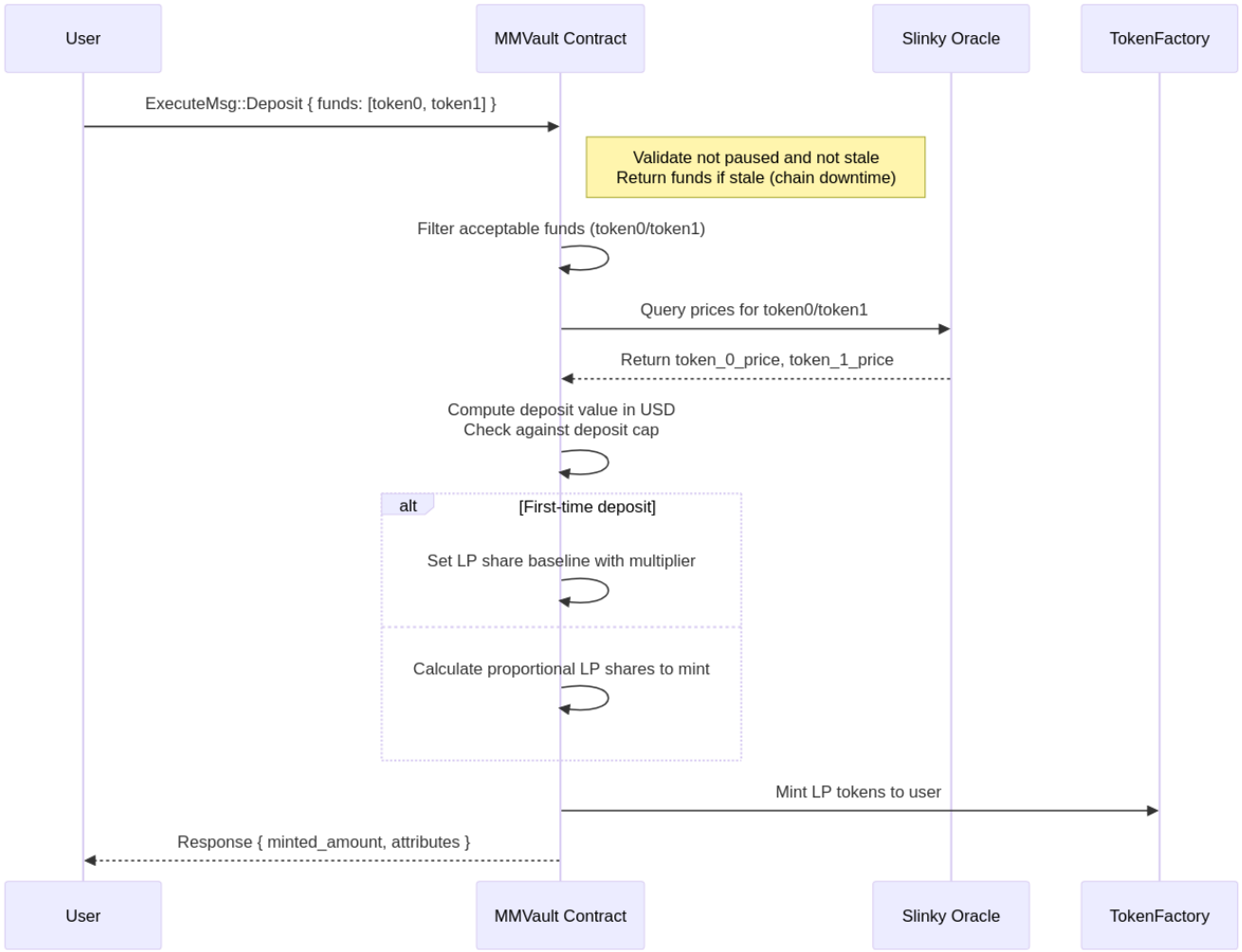

Figure 2: Deposit

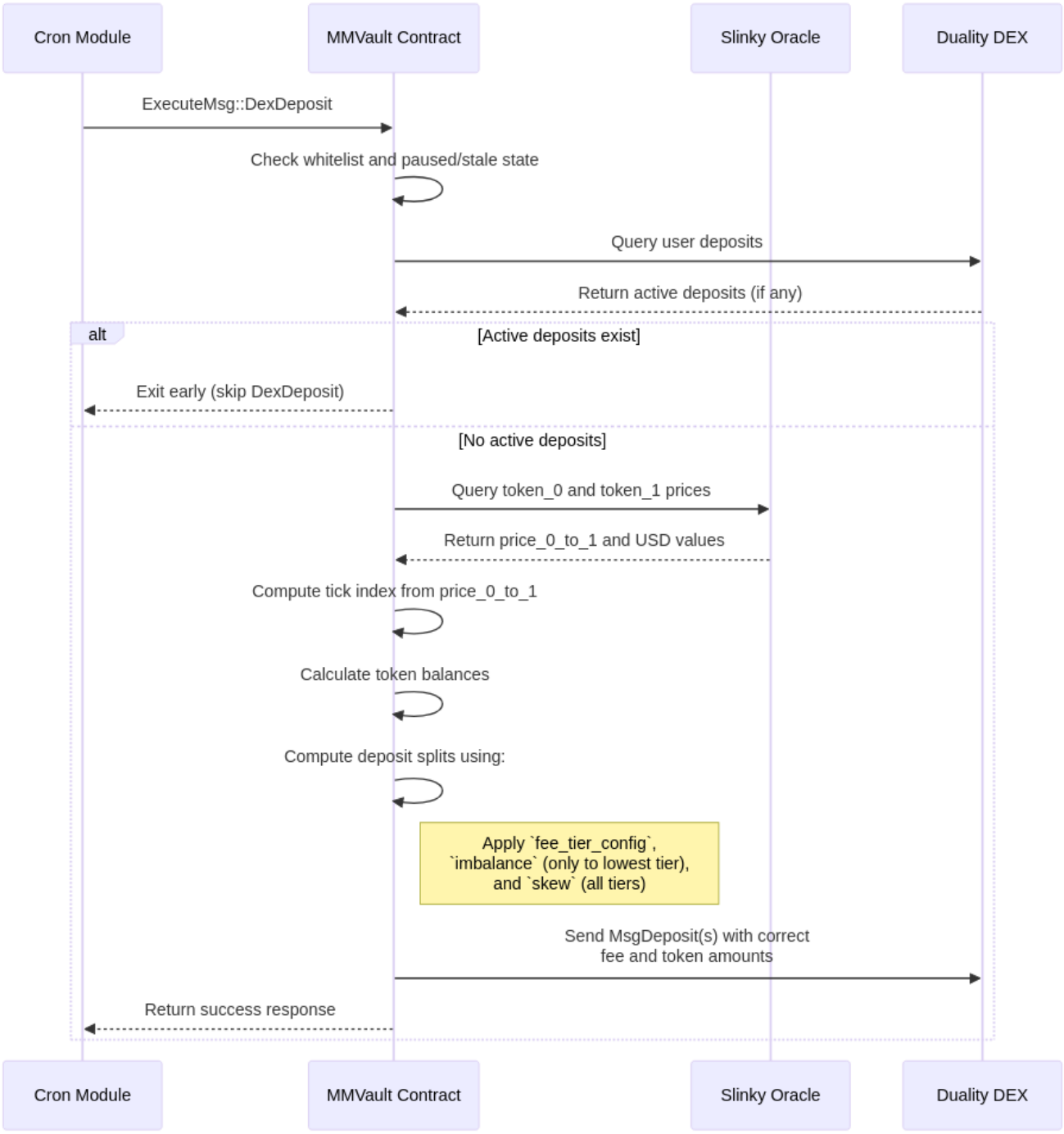

Figure 3: DEX Deposit

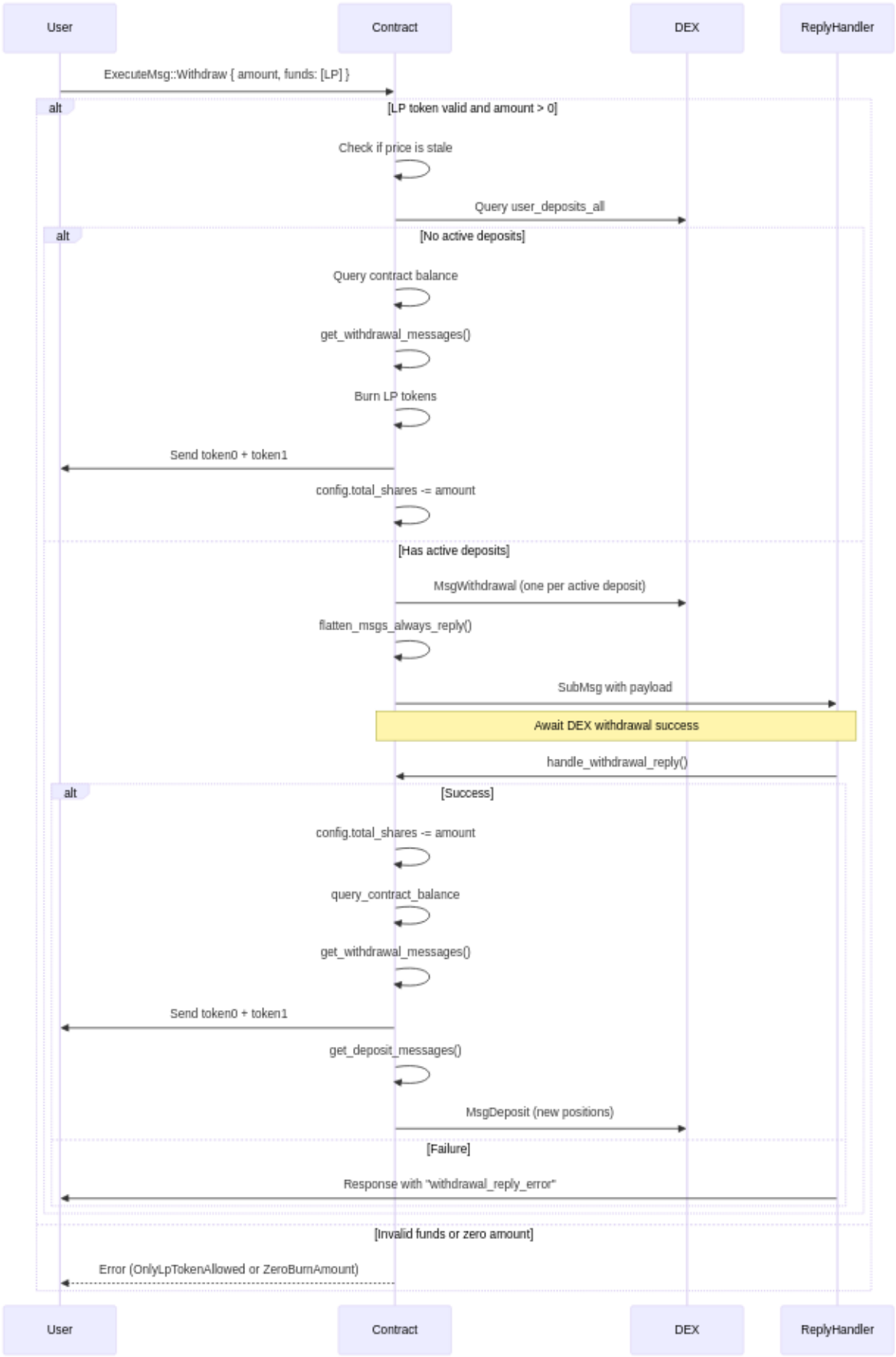

Figure 4: Withdraw

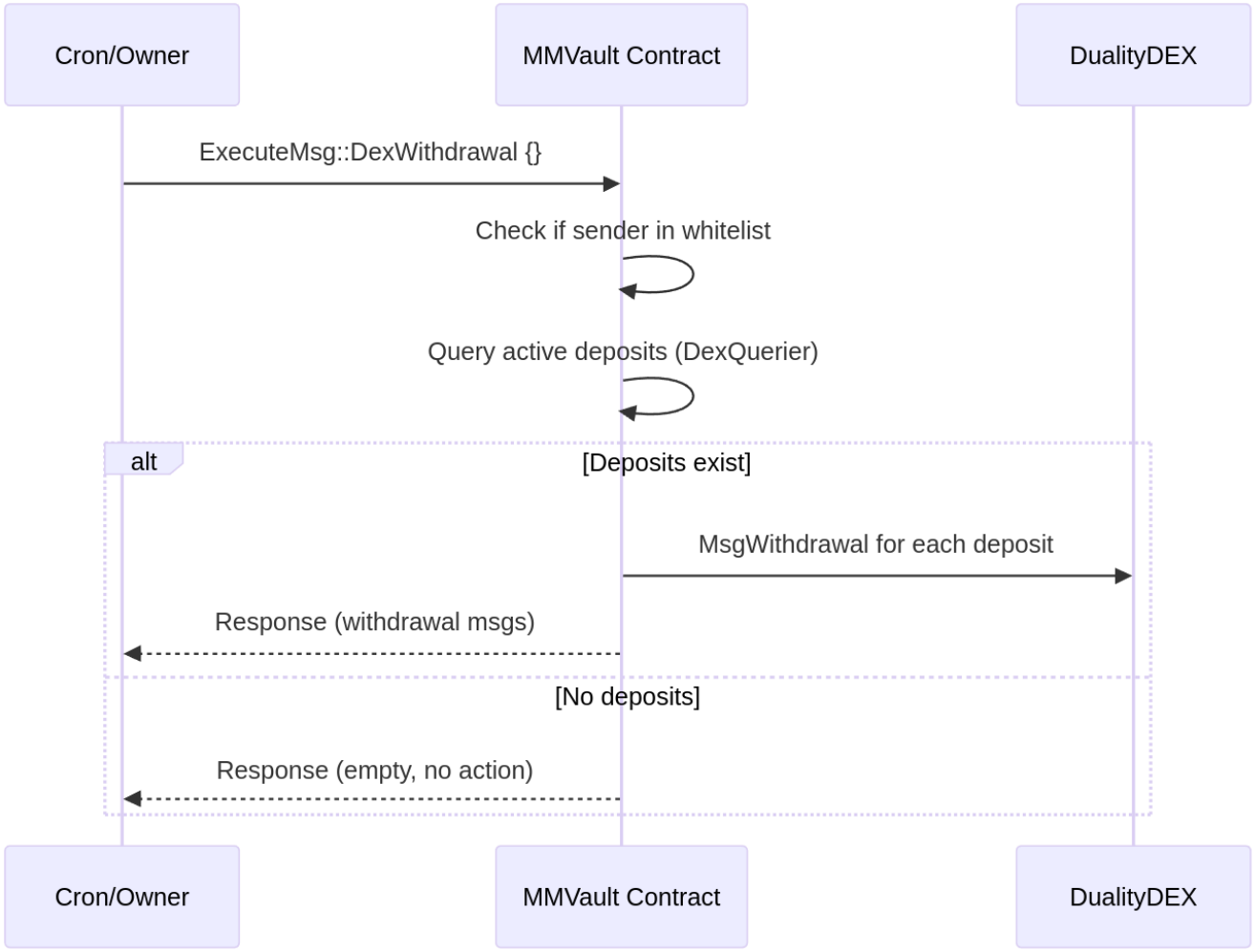

Figure 5: DEX Withdraw

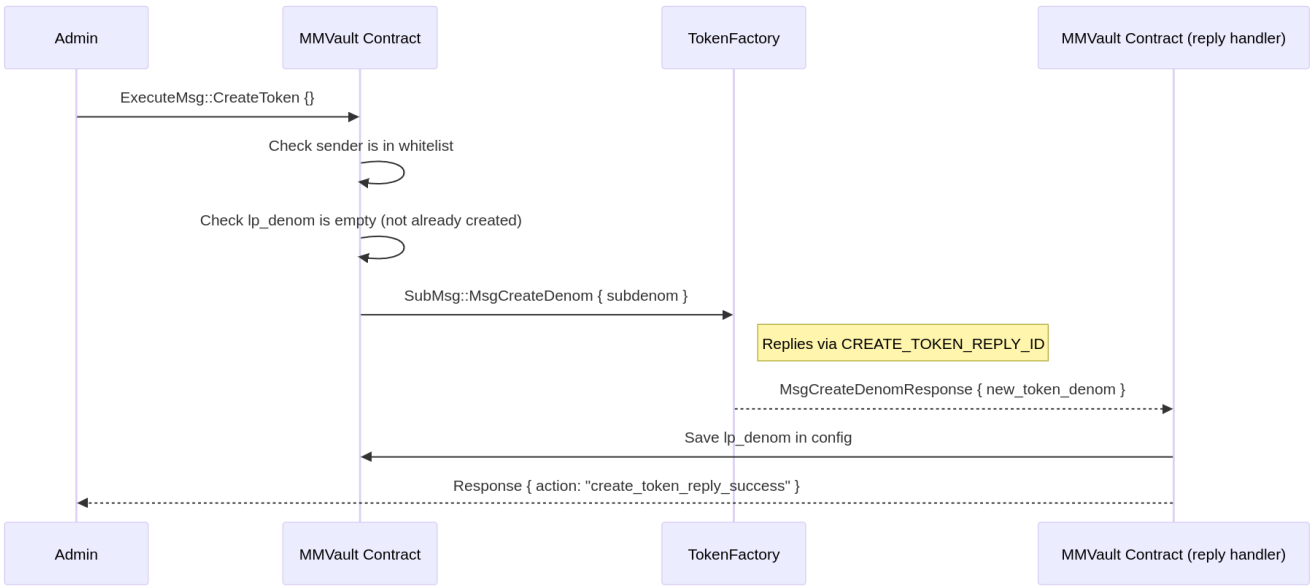

Figure 6: Create Token

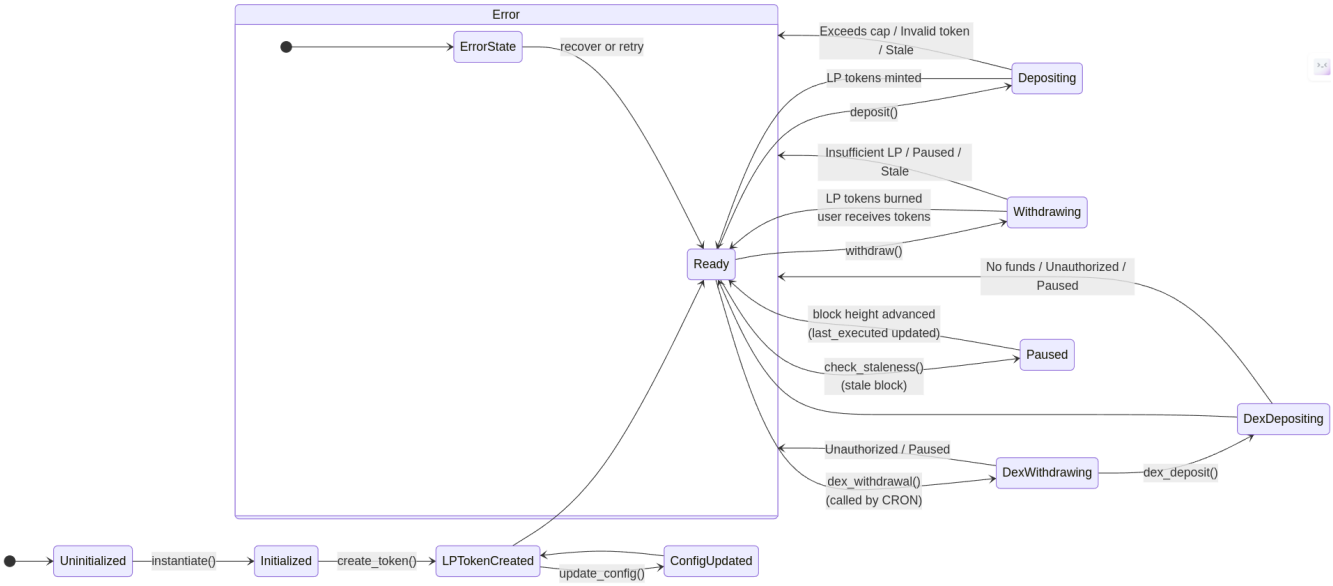

Figure 7: State Machine Diagram

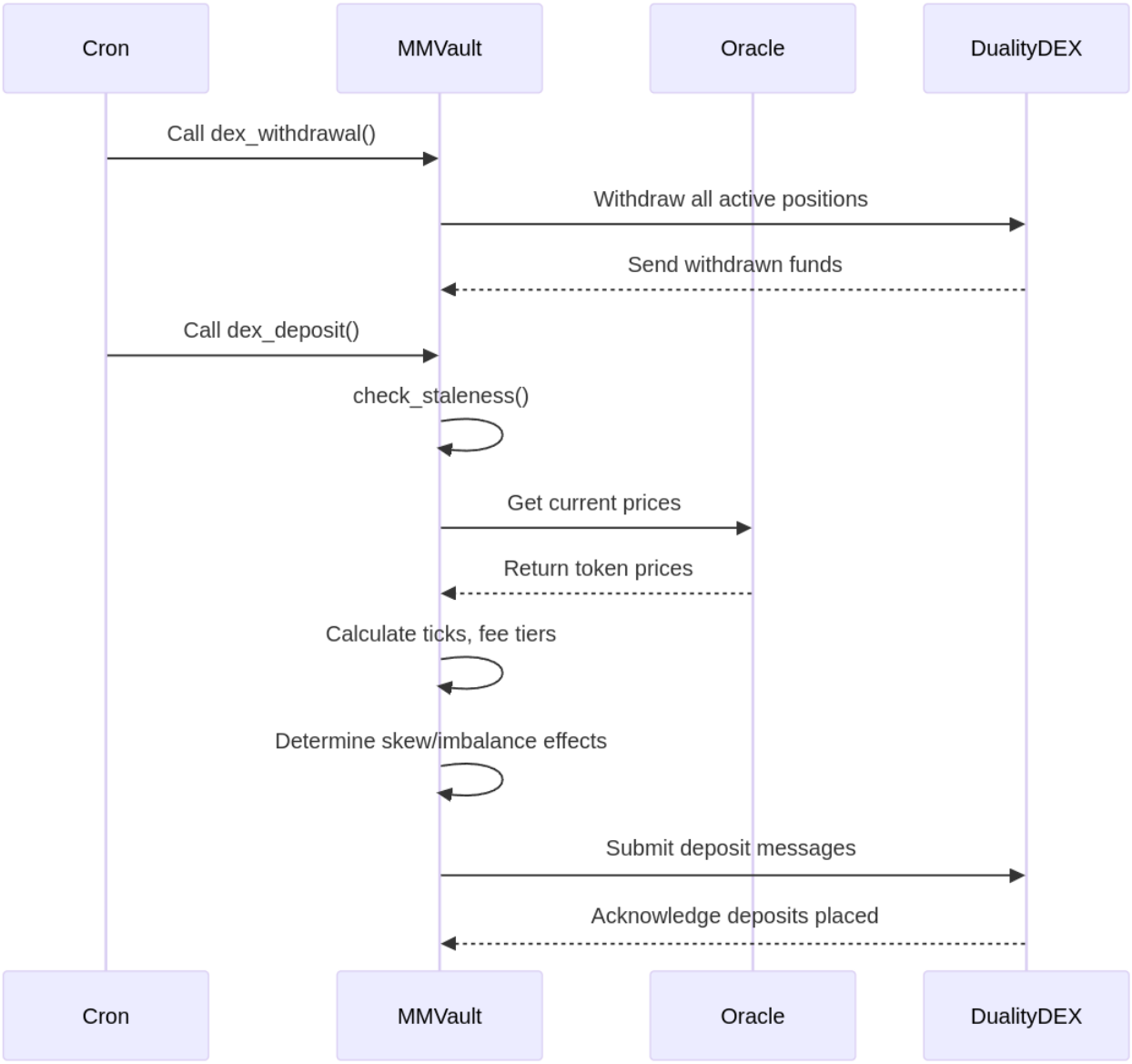

Figure 8: Interaction with DEX

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.