Hadron Labs

Security Assessment

September 4th, 2024 — Prepared by OtterSec

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Table of Contents

Executive Summary 2

Overview 2

Key Findings 2

Scope 3

Findings 4

Vulnerabilities 5

OS-DRP-ADV-00 | Improper State Transition Handling 6

OS-DRP-ADV-01 | Inaccurate Exchange Rate on Burning of Assets 8

OS-DRP-ADV-02 | Incorrect Staking Amount in Callback 10

OS-DRP-ADV-03 | Unbonding Underflow Risk 11

General Findings 12

OS-DRP-SUG-00 | Missing Batch Status Check 13

OS-DRP-SUG-01 | Cross-Contract Configuration Inconsistency 14

Appendices

Vulnerability Rating Scale 15

Procedure 16

01 — Executive Summary

Overview

Hadron Labs engaged OtterSec to assess the `drop-contracts` program. This assessment was conducted between August 11th and August 22nd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a vulnerability in the Puppeteer hook execution functionality, where instead of maintaining the Claiming state to properly revert unbond statuses, the system improperly transitions to the Idle state if a specific error occurs (OS-DRP-ADV-00). Additionally, the exchange rate is inaccurately set to one when all derivative assets are burned, without accounting for unprocessed assets still involved in unbonding (OS-DRP-ADV-01). Furthermore, the bonded amount in the core contract records the native assets during bonding but incorrectly records the derivative assets during unbonding (OS-DRP-ADV-03).

We also recommended adding an explicit check for the unbond batch status (OS-DRP-SUG-00) and suggested including validation logic while updating the configuration to ensure that all configuration parameters match the corresponding settings in the related contracts (OS-DRP-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/hadronlabs-org/drop-contracts. This audit was performed against commit bdbb1a7.

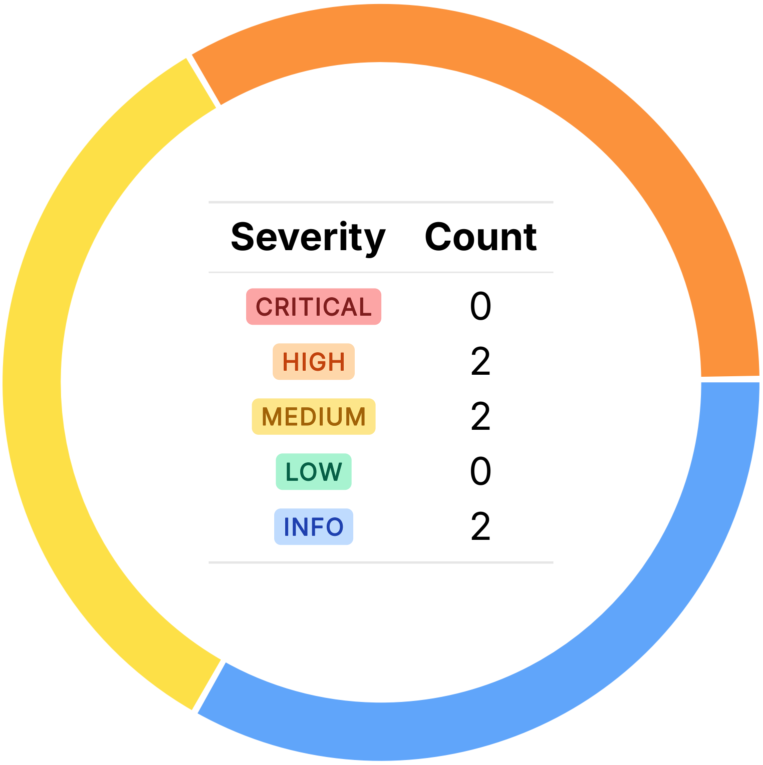
A brief description of the program is as follows:

Name	Description
drop-contracts	It contains the core smart contracts for the Drop Protocol, written in Rust utilizing the Cosmwasm framework.

03 — Findings

Overall, we reported 6 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	2
MEDIUM	2
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-DRP-ADV-00	HIGH	RESOLVED ✓	In `execute_puppeteer_hook`, instead of maintaining the `Claiming` state to properly revert `unbond` statuses, the system incorrectly transitions to the `Idle` state if an error occurs in the `ClaimRewardsAndOptionallyTransfer` transaction.
OS-DRP-ADV-01	HIGH	RESOLVED ✓	`query_exchange_rate` incorrectly sets the exchange rate to one when all `dAssets` are burned, without accounting for unprocessed assets still involved in unbonding.
OS-DRP-ADV-02	MEDIUM	RESOLVED ✓	`execute_stake` sends a callback with the full `non_staked_balance` instead of the actual `amount_to_stake`, resulting in faulty balance updates when the staked amount is less than the available non-staked balance.
OS-DRP-ADV-03	MEDIUM	RESOLVED ✓	`BONDED_AMOUNT` records the native assets or `LSM` shares during bonding but records the derivative assets (`dassets`) during unbonding.

Improper State Transition Handling HIGH

OS-DRP-ADV-00

Description

`execute_puppeteer_hook` fails to properly manage the state when handling errors from the `ClaimRewardsAndOptionallyTransfer` transaction. It handles responses from the puppeteer contract, processing both success and error responses, and updating the contract's state accordingly. When a transaction fails, the function determines the type of the failed transaction and transitions the contract's state to handle the error. The states are managed via a finite state machine (FSM), with the following relevant states: `Idle` (a default or inactive state where no specific action is performed) and `Claiming` (a state where the contract is in the process of claiming rewards and possibly transferring unbonded tokens).

>_ *drop-contracts/contracts/core/src/contract.rs* RUST

```
drop_puppeteer_base::msg::ResponseHookMsg::Error(err_msg) => {
            match err_msg.transaction {
                drop_puppeteer_base::msg::Transaction::Transfer { .. } // this one is for
                    ↪  transfering non-native rewards
                | drop_puppeteer_base::msg::Transaction::RedeemShares { .. }
                | drop_puppeteer_base::msg::Transaction::ClaimRewardsAndOptionalyTransfer { .. }
                    ↪  => { // this goes to idle and then ruled in tick_idle
                // IBC transfer for LSM shares and pending stake
                FSM.go_to(deps.storage, ContractState::Idle)?
            }
            drop_puppeteer_base::msg::Transaction::IBCTransfer { reason, .. } => {
                if reason == IBCTransferReason::LSMShare {
                    FSM.go_to(deps.storage, ContractState::Idle)?;
                }
            }
                _ => {}
            }
}
```

Currently, when the Puppeteer hook returns an error for a `ClaimRewardsAndOptionallyTransfer` transaction, the contract state is immediately set to `Idle`. This approach creates a vulnerability because the state transition to `Idle` is inappropriate for handling errors from `ClaimRewardsAndOptionallyTransfer`, and the contract does not correctly handle this error scenario associated with claiming rewards and unbonded tokens.

In the case of an error with `ClaimRewardsAndOptionallyTransfer`, the contract should remain in the `Claiming` state to revert the `UnbondBatchStatus` of any failed transfers back to `Unbonding`. This ensures that the system may properly retry or manage transfer operations, maintaining a correct state representation of pending unbonds. When the state transitions to `Idle` after a

`ClaimRewardsAndOptionallyTransfer` error, the contract may incorrectly leave the `UnbondBatchStatus` as it was, rather than reverting it to `Unbonding`. As a result, users cannot get their native tokens back from withdrawal vouchers.

Remediation

Modify `execute_puppeteer_hook` so that, on receiving an error for `ClaimRewardsAndOptionallyTransfer`, the contract remains in the `Claiming` state to handle the retry logic and ensure that any `unbond` statuses are correctly reverted to `Unbonding`.

Patch

Fixed in PR #173.

Inaccurate Exchange Rate on Burning of Assets HIGH OS-DRP-ADV-01

Description

The vulnerability in `query_exchange_rate` relates to an incorrect calculation of the exchange rate when all liquid staking tokens (`dAssets`) have been burned, but unprocessed `unbonding` requests (native assets that are in the process of being undelegated) still exist. The function queries the total supply of `dAssets` (`ld_total_supply`), which represents the total amount of `dAssets` that are currently in circulation. The problem occurs when all `dAssets` are burned and the total supply of `dAssets` (`ld_total_supply.amount.amount`) is zero. In this case, the function currently sets the exchange rate to one.

>_ *drop-contracts/contracts/core/src/contract.rs* RUST

```rust
fn query_exchange_rate(deps: Deps<NeutronQuery>, config: &Config) -> ContractResult<Decimal> {
    let fsm_state = FSM.get_current_state(deps.storage)?;
    if fsm_state != ContractState::Idle {
        return Ok(EXCHANGE_RATE
            .load(deps.storage)
            .unwrap_or((Decimal::one(), 0))
            .0);
    }
    let ld_total_supply: cosmwasm_std::SupplyResponse =
        deps.querier.query(&QueryRequest::Bank(BankQuery::Supply {
            denom: LD_DENOM.load(deps.storage)?,
        }))?;

    let mut exchange_rate_denominator = ld_total_supply.amount.amount;
    if exchange_rate_denominator.is_zero() {
        return Ok(Decimal::one());
    }
    [...]
}
```

This implies that each `dAsset` is worth exactly one native asset, which is not always correct. Even if the total `dAsset` supply is zero, there may still be unprocessed unbonding requests (`unprocessed_dasset_to_unbond`). These represent native assets that have not yet been undelegated or released from staking, which are effectively still tied to the `dAssets` that have been burned. Setting the exchange rate to one does not account for the value still locked in the unprocessed unbonding requests. The actual exchange rate should consider these unprocessed amounts.

This may result in financial losses for stakers if the exchange rate does not correctly reflect the value of their `dAssets` relative to the native assets (considering unprocessed unbonding amounts). Users might get less than their fair share of native assets when redeeming `dAssets`.

Remediation

Ensure the exchange rate is set to one only when the sum of `ld_total_supply` and `unprocessed_dasset_to_unbond` is zero.

Patch

Fixed in PR #164.

Incorrect Staking Amount in Callback MEDIUM OS-DRP-ADV-02

Description

`execute_stake` in `staker` utilizes an incorrect amount when sending a `sudo` callback after initiating a staking transaction. Specifically, it creates a `SubMsg` with a `sudo` callback that will be invoked once the staking transaction is processed. Here, the amount passed to the `Transaction::Stake` variant corresponds to the entire `non_staked_balance` instead of the actual `amount_to_stake`.

>_ *drop-contracts/contracts/staker/src/contract.rs* RUST

```rust
fn execute_stake(
    deps: DepsMut<NeutronQuery>,
    info: MessageInfo,
    items: Vec<(String, Uint128)>,
) -> ContractResult<Response<NeutronMsg>> {
    [...]
    let submsg: SubMsg<NeutronMsg> = msg_with_sudo_callback(
        deps,
        cosmos_msg,
        Transaction::Stake {
            amount: non_staked_balance,
        },
        ReplyMsg::SudoPayload.to_reply_id(),
        Some(info.sender.to_string()),
    )?;
    Ok
```

The purpose of the `Transaction::Stake amount: [...]` in the `sudo` callback is to keep track of how much has been successfully staked and to update the `NON_STAKED_BALANCE` accordingly. By sending `non_staked_balance`, the contract indicates that the entire `non_staked_balance` is staked, which is incorrect if `amount_to_stake < non_staked_balance`. This results in an overestimation of the amount that is staked and an underestimation of the remaining non-staked funds.

Remediation

Ensure the `sudo` callback is updated to utilize the correct `amount_to_stake` value instead of `non_staked_balance`, so that the amount recorded in the callback matches the actual staked amount.

Patch

Fixed in PR #171.

Unbonding Underflow Risk MEDIUM OS-DRP-ADV-03

Description

The BONDED_AMOUNT increases by the amount of native assets or LSM shares they bond. This addition reflects the assets locked in the contract. When users unbond assets, the process involves converting dAssets back to native assets. The BONDED_AMOUNT should then decrease by the amount of native assets being unbonded. However, the core contract currently records the dAssets when users unbond, which is incorrect. Consequently, if the exchange rate between native assets and dAssets is less than 1, then the total - dasset_amount is reduced more than expected.

Thus, if execute_unbond tries to subtract more from BONDED_AMOUNT than what is currently recorded (because the effective amount represented by dAssets after conversion is less than the bonded amount), this will result in an underflow.

Remediation

Remove the BONDED_AMOUNT.

Patch

Fixed in PR #167.

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-DRP-SUG-00	`get_unbonding_msg` does not verify if the `unbond` batch status is set to `New` before proceeding with the unbonding logic.
OS-DRP-SUG-01	`execute_update_config` lacks validation across multiple contracts, resulting in potential inconsistencies when updating configuration parameters.

Missing Batch Status Check

OS-DRP-SUG-00

Description

`get_unbonding_msg` is responsible for preparing and returning a message that initiates the unbonding process if certain conditions are met. It first checks if there is a `FAILED_BATCH_ID` in storage to see if a previously failed batch needs reprocessing. If not, it loads the current `UNBOND_BATCH_ID` batch data from storage. However, it does not explicitly check the status of the `unbond` batch after loading it from storage. It proceeds with the assumption that the processing batch is in the `New` state.

>_ *drop-contracts/contracts/core/src/contract.rs* RUST

```rust
fn get_unbonding_msg<T>(
    deps: DepsMut<NeutronQuery>,
    env: &Env,
    config: &Config,
    info: &MessageInfo,
    attrs: &mut Vec<cosmwasm_std::Attribute>,
) -> ContractResult<Option<CosmosMsg<T>>> {
    let funds = info.funds.clone();
    attrs.push(attr("knot", "024"));
    let (batch_id, processing_failed_batch) = match FAILED_BATCH_ID.may_load(deps.storage)? {
        Some(batch_id) => (batch_id, true),
        None => (UNBOND_BATCH_ID.load(deps.storage)?, false),
    };
    let mut unbond = unbond_batches_map().load(deps.storage, batch_id)?;
    [...]
}
```

Remediation

Add an explicit check for the `unbond` batch status in `get_unbonding_msg`.

Patch

Fixed in PR #172.

Cross-Contract Configuration Inconsistency OS-DRP-SUG-01

Description

There is a lack of validation for consistency among configuration parameters in `execute_update_config` across multiple contracts (`core` , `staker` , and `puppeteer`). In a system where multiple contracts interact, certain configuration parameters must remain consistent across these contracts to ensure proper functionality. Currently, `execute_update_config` allows for the independent updating of various configuration parameters. For example, the `base_denom` parameter (the base denomination utilized for calculations) may differ between `core` and `staker` , affecting user balances, resulting in the misallocation of funds or in unexpected behavior.

Remediation

Ensure to include validation logic in `execute_update_config` such that all configuration parameters match the corresponding settings in related contracts (`core` , `staker` , and `puppeteer`).

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 the 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.