Security Audit Report

Neutron 2024 Q1 - Liquidity Migration

Authors: Ivan Gavran, Aleksandar Ignjatijevic

Last revised 3 April, 2024

Table of Contents

Audit Overview

Scope

In March 2024, Informal Systems conducted a security audit for Neutron. The audit focused on liquidity migration of `lockdrop` and `vesting` contracts from XYK pools to PCL pools. (The reserve contract's migration was audited in a previous audit.) The audit consisted of code review and designing an extended set of end-to-end test cases, built on top of the existing testing infrastructure.

The audit was performed from February 28, 2024 to March 21, 2024 by the following personnel:

- Ivan Gavran
- Aleksandar Ignjatijevic

Relevant Code Commits

The scope of the audit were the following repositories:

- neutron-tge-contracts at e86b087
- neutron-dao at ef1b14f

The provided set of end-to-end tests was neutron-integration-tests, branch `feat/migrate-to-pcl`.

Conclusion

We performed a thorough code review of the liquidity migration and found it to be elegantly designed, relying on existing code primitives of withdrawing and depositing funds. We also augmented the existing end-to-end testsuite with model-based test scenarios, which varied different interleavings of steps within migration. (The testing methodology is described in Appendix: Model-based Testing .)

Overall, we found two high-impact issues in the code, and a number of lower risk ones. The dev team promptly addressed the findings. We reviewed the fixes and found them to be properly address the reported issues.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: CosmWasm

Engagement Summary

- **Dates**: 28.2.2024. - 21.3.2024.
- **Method**: Manual code review, protocol analysis, model-based fuzzing

Severity Summary

Finding Severity	**#**
Critical	0
High	2
Medium	0
Low	2
Informational	2
Total	**6**

The table above contains only the findings for which we provided both the problematic scenario and the root cause analysis. On top of that, we provided 5 more problematic scenarios obtained by model-based testing, for which the dev team performed the root cause analysis independently. (For details, see Testing Issues.)

Findings

Title	Type	Severity	Status
An attacker can write to the lockdrop's contract memory at will	Implementation	3 High	Resolved
Reward amounts in lockdrop-pcl contract are miscalculated in case of time-extended migration	Implementation	3 High	Resolved
The migration may temporarily fail for some users due to manipulation of pool ratios	Protocol	1 Low	Risk Accepted
Lockdrop-PCL is relying on an unspecified behavior of the Incentive's PendingRewards query	Implementation	1 Low	Resolved
Miscellaneous code concerns	Practice	0 Informational	Risk Accepted
Miscellaneous security concerns	Practice	0 Informational	Resolved

An attacker can write to the lockdrop's contract memory at will

Type	Implementation
Severity	3 High
Impact	3 High
Exploitability	2 Medium
Status	Resolved

Involved artifacts

- neutron-tge-contracts/contracts/lockdrop/src/contract.rs::handle_migrate_liquidity_to_pcl_pools
- neutron-tge-contracts/contracts/lockdrop/src/contract.rs::handle_claim_rewards_and_unlock_for_lockup
- neutron-tge-contracts/contracts/lockdrop-pcl/src/contract.rs::handle_claim_rewards_and_unlock_for_lockup
- https://github.com/informalsystems/neutron-tge-contracts-audit-internal/blob/feat/pcl-lp-migration/contracts/lockdrop/src/testing.rs#L10 .

Description

The `ExecuteMsg::MigrateLiquidityToPCLPools` takes a `user_address_raw` optional string parameter. This string denotes the address of the user whose liquidity is supposed to be migrated.

The address is validated first, turning `user_address_raw` into `user_address` of type `Addr`, and then the corresponding user is loaded with the following code-snippet:

```
let mut user_info = USER_INFO
        .may_load(deps.storage, &user_address)?
        .unwrap_or_default();
```

The problem with this snippet is that if there is no user under `user_address`, it will return a default `UserInfo` structure. The default structure is:

```
UserInfo {
    pub total_ntrn_rewards: 0
    pub ntrn_transferred: false,
    pub lockup_positions_index: 0,
}
```

Because `total_ntrn_rewards` equals 0, the check that follows will pass:

```
if user_info.total_ntrn_rewards == Uint128::zero() {
        user_info.total_ntrn_rewards = update_user_lockup_positions_and_calc_rewards(
            deps.branch(),
            &config,
            &state,
            &user_address,
        )?;
        USER_INFO.save(deps.storage, &user_address, &user_info)?;
    }
```

The non-existent user will first enter the `update_user_lockup_positions_and_calc_rewards` function, not causing much trouble there because there are no lockups associated with that user, and most of the function body will turn into no-ops.

After returning from the function, the user will be saved into the `USER_INFO` map.

After that point, the function iterates over lockups (no-op in this case) to create messages for the migration to continue. Since no messages will be generated, this message handler will simply return `Ok(Response::default()).`

Problem Scenarios

This behavior enables an attacker to run a script sending batches of `ExecuteMsg::MigrateLiquidityToPCLPools` messages with arbitrary `user_address_raw` fields (up to the validation).

This will result in validators' memory filling up and bloating the state of the contract, as a consequence making the sync/state export/state import extremely slow, and potentially resulting in out-of-memory restarts of individual validators.

- Note A: This attack is expensive due to gas cost, and can only be carried out over a number of blocks (or otherwise gas limit would be reached).
- Note B: This attack can be performed at any point, given that the migration entrypoints remain active after the migration is performed.
- Note C: The lockdrop::handle_claim_rewards_and_unlock_for_lockup and lockdrop-pcl::handle_claim_rewards_and_unlock_for_lockup contain similar code snippets. There, though, the attack will not go through because of the call to compatible_load, which would result in an error and revert the whole transaction. We recommend nonetheless to add an explicit check there and return an error early in the function.

The test illustrating the attack can be found at https://github.com/informalsystems/neutron-tge-contracts-audit-internal/blob/feat/pcl-lp-migration/contracts/lockdrop/src/testing.rs#L10 .

We attach a (failing) test illustrating the problem:

```
fn handle_migrate_liquidity_to_pcl_pools_test(){

    let env = mock_env();
    let user_addr = Some("newaddr".to_string());
    let info = mock_info("addr0000", &[]);
    let mut deps = mock_dependencies();
    let owner = Addr::unchecked("owner");
    let token_info_manager = Addr::unchecked("token_info_manager");


    let msg = InstantiateMsg {
        owner: Some(owner.to_string()),
        token_info_manager: token_info_manager.to_string(),
        init_timestamp: env.block.time.seconds(),
        lock_window: 10_000_000,
        withdrawal_window: 500_000,
        min_lock_duration: 1u64,
        max_lock_duration: 52u64,
        max_positions_per_user: 14,
        credits_contract: "credit_contract".to_string(),
        lockup_rewards_info: vec![LockupRewardsInfo {
            duration: 1,
            coefficient: Decimal256::zero(),
        }],
        auction_contract: "auction_contract".to_string(),
    };

    instantiate(deps.as_mut(), env.clone(), info.clone(), msg).unwrap();
    let user_info_keys = USER_INFO
        .keys(deps.as_mut().storage, None, None, cosmwasm_std::Order::Ascending)
        .collect::<Result<Vec<Addr>, StdError>>()
        .expect("error");
    dbg!(user_info_keys.len());
    assert!(user_info_keys.len() == 0);

    let _r = handle_migrate_liquidity_to_pcl_pools(deps.as_mut(), info, env,
user_addr).unwrap();

    let user_info_keys = USER_INFO
        .keys(deps.as_mut().storage, None, None, cosmwasm_std::Order::Ascending)
        .collect::<Result<Vec<Addr>, StdError>>()
        .expect("error");
    dbg!(user_info_keys.len());
    assert!(user_info_keys.len() == 0);

}
```

Recommendation

We recommend checking for the address's existence as early as possible, ideally at the very beginning of the `handle_migrate_liquidity_to_pcl_pools`.

Status

Successfully resolved in PR#84.

Reward amounts in lockdrop-pcl contract are miscalculated in case of time-extended migration

Type	Implementation
Severity	3 High
Impact	3 High
Exploitability	2 Medium
Status	Resolved

Involved artifacts

- neutron-tge-contracts/lockdrop-pcl/src/contract.rs
- astroport-core/contracts/tokenomics/incentives/src/utils.rs

Description

The basic idea in calculating the amount of rewards belonging to each lockup position is the following: every time there is a claim from the pool, the `pool_info`'s field `incentives_rewards_per_share` is updated by adding to it `received_amount / total_lp_balance_deposited`. (More details on how the calculation is performed can be found in Appendix: Calculation of lockdrop pool rewards.) This ensures that each lockup that existed in between two claims gets "assigned" its share of reward.

There is one exception, though: each time new liquidity is provided to the pool, the pool also sends pending rewards to the `lockdrop-pcl` contract, but `pool_info.incentives_rewards_per_share` does not get updated.

Problem Scenarios

Assume two contracts, `A` and `B`, being migrated from the XYK to the PCL pool. Assume also that `A` is migrated first and then there is a long break before `B` is migrated. Furthermore, there are no claims from the `lockdrop-pcl` contract before `B` gets migrated.

For simplicity (but without loss of generality), assume that at the moment that `A` is being migrated, there is 0 ASTRO in lockdrop-pcl and 0 pending rewards. Let's consider the following time points:

1. `A` gets migrated: lockdrop-pcl ASTRO amount: 0; pending rewards: 0
2. Some time passes in which `A` is the only lockup in the contract. lockdrop-pcl ASTRO amount: 0; pending rewards: 100
3. `B` gets migrated. Now the pool sends all the pending rewards to `lockdrop-pcl`. lockdrop-pcl ASTRO amount: 100; pending rewards: 0
4. Some time passes with both `A` and `B` in the pool: lockdrop-pcl ASTRO amount: 100; pending rewards: 20

5. `A` claims its rewards. According to the formula, `incentives_rewards_per_share` will get calculated using the `received_amount = current_balance - pervious_balance`. The problem is that `previous_balance` is the balance just before the claim, thus: 100. Therefore, `received_amount = 20`. Now, `A` will receive its rewards using `incentives_per_share` that completely disregards the rewards accrued while it was alone in the pool. This amount will stay in the `lockdrop-pcl` contract.

NOTE: The consequences of the problem presented above are going to be minimized if all the contracts are executed in a quick succession one after another, followed by a claim. Alternatively, they will be maximized for longer periods between the first migration and the first claim from `lockdrop-pcl`.

Recommendation

Update the state, similarly to how it is done in update_pool_on_dual_rewards_claim, after each deposit. Make sure to use the amount of LP tokens before the deposit happened in the calculation.

Status

Successfully resolved in PR#85.

The migration may temporarily fail for some users due to manipulation of pool ratios

Type	Protocol
Severity	1 Low
Impact	1 Low
Exploitability	1 Low
Status	Risk Accepted

Description

In this finding, we are describing a scenario in the migration of certain users' funds will not succeed due to imbalanced ratios of liquidity pools. The failure will have no material consequences other than slowing down the migration process and requiring a retry mechanism for the users whose liquidity was not successfully migrated.

Assume an attacker whose intention is to disrupt the ratio within the `xyk` pool before or during the migration. Symmetrically, the attacker may consider disrupting the ratio of the `PCL` pool. Such an attacker should be prepared to lose some funds while performing the attack.

Let us focus our attention on disrupting the `xyk` pool. The attacker may do so with the least amount of liquidity around the end of the migration process (for the `PCL` , the attacker would likely act at the very beginning of the process), when the pool is shallow. Principally, the attack is possible because of the high percentage of liquidity in the pools coming from Neutron contracts, thus making sure that the pools will eventually be fairly shallow.

Disruption of the `xyk` pool ratio will result in trying to `provide_liquidity` with assets in the disrupted ratio to the PCL pool. This will result in the transaction failing, because of the slippage tolerance on the PCL pool side. Depending on the attacker's next actions, other users may be migrated successfully or the fail scenario will be the same for them.

This will keep happening until either the malicious user decides to sell `NTRN` back to the pool, or until the market makers have balanced the pool.

Problem Scenarios

The attacker buys significant amount of `NTRN` from the pool, using its pair token, in the same block when the migration of the user `u` is happening. That would disrupt the pool ratio between those two tokens and now in the pool will be very low amount of NTRN and a very large amount of its pair token.

In the migration process, user `u` will get `NTRN`s and paired tokens in the (disbalanced) proportion of the pool.

From here the problem branches into two subscenarios:

A. new PCL pool is fresh and has no funds in it

B. PCL pool has already correct ratio in itself

Subscenario A: There is a fresh PCL pool with no funds in it.

This scenario is possible only if the pool ratio has been disrupted right before the migration of the first user `u`, and the pool was empty before that.

User `u` will be migrated to the empty PCL pool, establishing the *wrong* ratio as the pool ratio.
All subsequent users' migration will fail, due to slippage tolerance on the PCL pool side. This will keep happening until the market makers correct the PCL pool ratio.
If the attacker is able to beat the market makers and sell whole amount of `NTRN` back to the `xyk` pool, that would make his losses only the fee price.

Subscenario B: PCL pool already has the correct ratio in itself

User `u` will not be migrated, due to slippage tolerance on the PCL pool side. All subsequent users will meet the same faith until the market makers have fixed the `xyk` pool ratio.
Same as in Subscenario A, if the attacker is able to beat the market makers and sell whole amount of `NTRN` back to the `xyk` pool, that would make his losses only the fee price.

Furthermore, a symmetrical attack may be performed on the PCL pool when it is in its shallow state.

Recommendation

Having insight into users whose migration has failed and having in mind a retry mechanism for said users, renders the attack harmless in the long run. Furthermore, once the xyk pool becomes very shallow, there may be little incentive there for market makers: thus, it would be a good idea to have an own market maker constantly working between the two pools.

Lockdrop-PCL is relying on an unspecified behavior of the Incentive's PendingRewards query

Type	Implementation
Severity	1 Low
Impact	1 Low
Exploitability	1 Low
Status	Resolved

Involved artifacts

- neutron-tge-contracts/contracts/lockdrop-pcl/src/contract.rs

Description

When handling the `ExecuteMsg::ClaimRewardsAndOptionallyUnlock` message, `lockdrop-pcl` first claims all the pending rewards, then updates the relevant state variables, and finally sends the rewards to the user in `callback_withdraw_user_rewards_for_lockup_optional_withdraw`.

In that function, it is invoking the `IncentivesQueryMsg::PendingRewards` query: since all the rewards have been recently claimed, there will be no pending rewards. The logic that follows does not need pending amounts, but only the corresponding `AssetInfo` (it relies on the updated state variable `pool_info.incentives_rewards_per_share`). Conveniently, the invoked query will return a vector of `Asset`s that will contain all the assets with `0` amounts.

This behavior is indeed exhibited by the query, but it is not the only possible behavior given the specification of the query:

> *"PendingToken returns the amount of rewards that can be claimed by an account that deposited a specific LP token in a generator"*

Another possible return value in the situation with no pending rewards would be an empty vector.

Problem Scenarios

If Astroport were to change the return format of the query, there would likely be no warning of it since it is not a breaking change. Then, the logic of the withdraw callback would break, since the loop going over all pending rewards (in reality: over all assets that may qualify for rewards) would turn into a no-op.

Recommendation

We recommend sending a list of assets to check as a parameter to the callback function.

Status

Successfully resolved in PR#87.

Miscellaneous code concerns

Type	Practice
Severity	0 Informational
Impact	1 Low
Exploitability	0 None
Status	Risk Accepted

We list here various code-related concerns that do not pose security threats, but addressing them may make the codebase more robust, readable, and maintainable.

- README.md from lockdrop-vault-pcl-pools has some misleading information. It is supposed to collect voting power from Lockdrop-pcl contract (but instead it claims to be collecting from the `Lockdrop` contract).
- There are a couple of misleading comments in the `callback_transfer_all_rewards_before_migration` function (here, here, and here). They suggest that rewards are being claimed in the function, when in reality they are only distributed.
- `handle_migrate_liquidity_to_pcl_pools` function has `USER_INFO.save()` (here) placed inside `if` statement, while in `handle_claim_rewards_and_unlock_for_lockup` has it outside the same `if` statement (here). Same `USER_INFO.save()` could be found outside `if` statement in `lockdrop-pcl` contract (here). If called outside of the `if` statement, that makes (if it was intentionally placed there) unnecessary call to memory. Optimize it.
- There is some confusing naming throughout the code:
    - during migration, a field of the `UpdatePoolOnDualRewardsClaim` is called `prev_ntrn_balance`, when in reality it refers to the ASTRO balance (here). The same problem exists in the comment description and the signature of the `update_pool_on_dual_rewards_claim` (link)
    - `PoolInfo` has a field called `generator_ntrn_per_share`, which refers to ASTRO per share. The same naming confusion happens in the structure `LockupInfoV2` here.
- The `Config` structure in the `neutron-lockdrop-vault-for-cl-pools` looks like this

```
pub struct Config {
    pub name: String,
    pub description: String,
    pub lockdrop_contract: Addr,
    pub usdc_cl_pool_contract: Addr,
    pub atom_cl_pool_contract: Addr,
    pub owner: Addr,
}
```

It would be clearer if the field `lockdrop_contract` would be named `lockdrop_cl_contract`, similarly to other fields. Equivalently, the config structure in `neutron-vesting-lp-vault-for-cl-pools` should rename `atom_vesting_lp_contract` → `atom_vesting_lp_cl_contract` and `usdc_vesting_lp_contract` → `usdc_vesting_lp_cl_contract.`

Vesting-lp-pcl

In this section we give several code improvement suggestions regarding *vesting-lp-pcl* contract grouped in few groups.

Unnecessary creation of new variable

Following variables are created and used either only once, or they were created by assigning values from function parameters.

- Unnecessary creation of new variable `owner` in function `instantiate`. Problematic code fragment: source.
    - Proposed solution: Validate address when creating `Config` struct.

```
    CONFIG.save(
        deps.storage,
        &Config {
            deps.api.addr_validate(&msg.owner)?,
            vesting_token: Option::from(msg.vesting_token),
            token_info_manager: deps.api.addr_validate(&msg.token_info_manager)?,

            extensions: Extensions {
                historical: true,
                managed: false,
                with_managers: true,
            },
        },
    )?;
```

- Unnecessary creation of new variable `ma` in function `instantiate`. Problematic code fragment: source.
    - Proposed solution: Validate address when saving `VESTING_MANAGERS`.

```
        VESTING_MANAGERS.save(deps.storage, deps.api.addr_validate(m)?, &())?;
```

- Unnecessary creation of new variable `sender` in function `receive_cw20`. Problematic code fragment: source.
    - Proposed solution: `sender` variable is used only once (link) and it was cloned. Avoid doing that, just pass `info.sender`.
- Unnecessary creation of new variable `xyk_vesting_lp_contract` in function `is_sender_xyk_vesting_lp`. Problematic code fragment: source.
    - Proposed solution: Don't create new variable, just load it from storage and unwrap it.

```
    if *sender == XYK_VESTING_LP_CONTRACT.load(store).unwrap() {
        return true;
    }
```

Unwrapping loaded `XYK_VESTING_LP_CONTRACT` can cause `panic`, but that will never happen because in order for that to happen, saving `XYK_VESTING_LP_CONTRACT` within `instantiate` function has to fail, but that would result in contract not being instantiated, etc.

- Unnecessary creation of new variable `height` in function `handle_migrate_xyk_liquidity`. Problematic code fragment: source.
    - Proposed solution: `env` is passed into the function as parameter and accessing its fields is faster and better practice than creating new variable to store one of the fields. Use `env.block.height` instead of `height` here and here.

DRY

- In *vesting-lp-pcl* contract, instantiation has been done by using exactly the same code as in vesting-base/src/builder.rs. Code fragments in question: here and here.
- `VestingBaseBuilder` does not support `msg.vesting_token` as part of build. Thus, there is a part of code that does the building from scratch, augmented with `msg.vesting_token` here, duplication the builder code. We recommend refactoring `VestingBaseBuilder` to support `vesting_token`. (Alternatively, calling the build like it was called in *vesting* contract, and then updating `Config` with `msg.vesting_token` value, would still be an improvement.) An example of good usage of `VestingBaseBuilder` : source.

Code Practice

- It is a better practice to use built in functions such as `as_str()` instead of doing something like this: `.addr_validate(&msg.xyk_vesting_lp_contract.clone())?`. Problematic code fragment : source.
    - Proposed solution:

```
XYK_VESTING_LP_CONTRACT.save(
        deps.storage,
        &deps
            .api
            .addr_validate(msg.xyk_vesting_lp_contract.as_str())?,
    )?;
```

- Suboptimal calls for address validation. Calling address validation two times on the same variable is generally not a good practice. Problematic code fragments: source1, source2.
    - Proposed solution: Create single variable as validated address and pass reference to that variable to functions.

```
    let validated_sender = deps.api.addr_validate(&cw20_msg.sender)?;
```

Miscellaneous security concerns

Type	Practice
Severity	0 Informational
Impact	1 Low
Exploitability	0 None
Status	Resolved

We list here various concerns that, to the best of our knowledge, don't pose immediate security threats, but may lead to security vulnerabilities or problems if not taken care of properly.

- Usage of `+` instead of `checked_add` when calculating VP from lockdrop-pcl contract (vulnerable part). This could lead to potential overflow when calculating VP. This problem also occurs when calculating VP from lockdrop contract (vulnerable part). Interesting thing to note is that it was handled well when calculating VP from vesting-lp and vesting-lp-pcl contracts. Realistically, this is not going to happen with expected VP amounts, but still it is a good practice to use `checked_add` consistently.
- In the `migrate` functions of the *lockdrop* and *vesting-lp* contracts, a call to function `set_contact_version` is missing.
- In *vesting-lp contract,* `vesting_info.save()` is placed inside of the `dust_threshold` check, but outside `if !user_share.is_zero()` check. If user has been migrated and his vesting has been updated to `0`, then there will be unnecessary call to memory to update the state from `0` → `0`. (code in question here). Because `VESTING_INFO_HISTORICAL` is a `SnapshotMap` this could give false data, that the `SnapshotMap` has been changed in that block, even though it shouldn't be changed. That can lead malicious user to spam the chain with message to migrate user (who was already migrated) and access the memory to save new data to the `SnapshotMap`. This is unnecessary and should be changed so that `vesting_info` is updated with said values only if `user_share!=0`.
- There is no validation of `msg.dust_threshold` and `msg.max_slippage` implemented within the `migrate` endpoint. This means that a non-careful invocation of `migrate` may set non-reasonable parameters. We suggest adding to the code constant values limiting what those two parameters may be.

Status

Successfully resolved in PR#86 and commit cc3da98.

Testing Issues

Here we list the issues that occurred during model-based testing (for details of the testing approach, see Appendix: Model-based Testing). The detailed testing report is out of scope of this document and will be provided separately to the dev team, but here we group together the issues that occurred while testing.

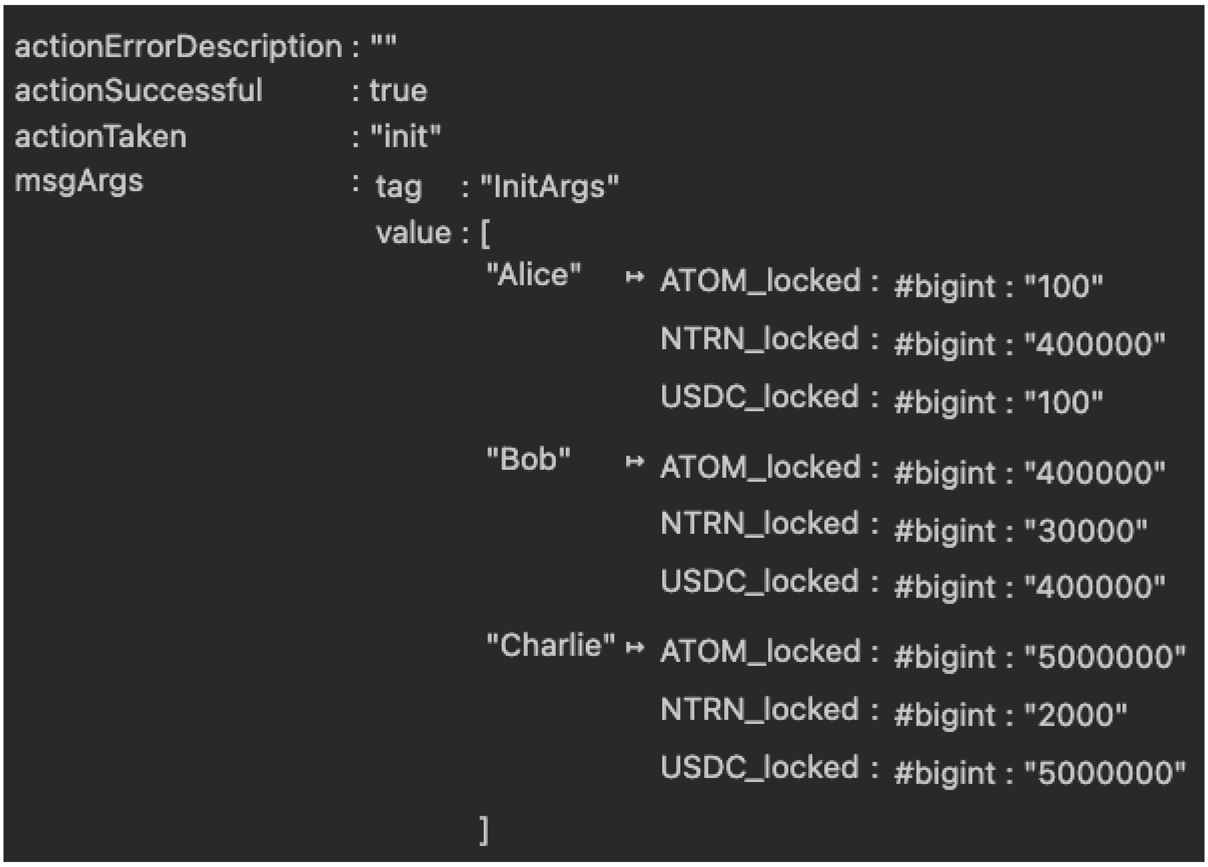
At the time of finalizing the audit, we had clear root-cause understanding for some of the testing issues, while for the others we identified that there was a problem, but without understanding its root cause. In the meantime, the dev team performed the analysis of reported issues, identified the root causes, and is working on addressing them.

Small locked amounts cause claims to fail

We have run a test with Alice having locked **100 uATOM** and **100 uUSDC**.

When trying to claim rewards on XYK pool, this scenario caused claim to crash and further more some other things fail. Full stack trace can be found in tests repository.

Test has initial states:



```
actionErrorDescription : ""
actionSuccessful       : true
actionTaken            : "init"
msgArgs                : tag    : "InitArgs"
                         value : [
                              "Alice"   ↦ ATOM_locked : #bigint : "100"
                                          NTRN_locked : #bigint : "400000"
                                          USDC_locked : #bigint : "100"
                              "Bob"     ↦ ATOM_locked : #bigint : "400000"
                                          NTRN_locked : #bigint : "30000"
                                          USDC_locked : #bigint : "400000"
                              "Charlie" ↦ ATOM_locked : #bigint : "5000000"
                                          NTRN_locked : #bigint : "2000"
                                          USDC_locked : #bigint : "5000000"
                         ]
```

Bellow can be found summary of tests:

Full summary

TGE / Migration / PCL contracts

  Pre-Migration setup

   Deploy✅

   Airdrop✅

   Auction✅

   Advance few blocks just in case ✅

   Migrate to new contracts ✅

   Advance few blocks just in case 2 ✅

  Quint generated steps

   Quint generated step ADVANCE BLOCK from step 1

    ✓ advancing 10 blocks (10296 ms)

   Quint generated step MIGRATE from step 2

```
  ✓ fill liquidity migration contracts
  migrate Bob participant
    ✓ gather state before migration (114 ms)
    ✓ migrate the user (970 ms)
    ✓ gather state after migration (101 ms)
    check user liquidity migration
      XYK user lockups
        ✓ XYK lockup lp token addresses (7 ms)
        generator rewards
          ✓ claimable generator ntrn debt (2 ms)
          ✓ generator rewards are transferred to the user (1 ms)
        astroport lp
          ✓ lp tokens marked as transferred (1 ms)
          ✓ staked lp amount decreases
      lockdrop participation rewards
        ✓ no balance change for PCL lockdrop contract (1 ms)
        ✓ no paired assets and lp received by user (1 ms)
      PCL user lockups
        ✓ no user lockup info before migration
        ✓ PCL lockup lp token addresses
        ✓ lockup positions consistency (1 ms)
        astroport lp
          ✓ lp tokens are locked (1 ms)
          ✓ lockup shares are roughly equal
          ✓ lp tokens not marked as transferred (1 ms)
          ✓ staked lp amount increases
Quint generated step CLAIM_REWARDS_XYK from step 3
  ❌ for Alice without withdraw (1900 ms)
Quint generated step CLAIM_REWARDS_XYK from step 4
  ✓ for Charlie without withdraw (2071 ms)
Quint generated step ADVANCE BLOCK from step 5
  ✓ advancing 10 blocks (10235 ms)
Quint generated step MIGRATE from step 6
  ✓ fill liquidity migration contracts
  migrate Charlie participant
    ✓ gather state before migration (119 ms)
    ✓ migrate the user (958 ms)
    ✓ gather state after migration (103 ms)
    check user liquidity migration
      XYK user lockups
        ✓ XYK lockup lp token addresses
        generator rewards
          ✓ claimable generator ntrn debt (2 ms)
          ✓ generator rewards are transferred to the user (2 ms)
        astroport lp
          ✓ lp tokens marked as transferred (2 ms)
          ✓ staked lp amount decreases
      lockdrop participation rewards
        ✓ no balance change for PCL lockdrop contract
        ✓ no paired assets and lp received by user
      PCL user lockups
        ✓ no user lockup info before migration
        ✓ PCL lockup lp token addresses
        ✓ lockup positions consistency (1 ms)
        astroport lp
          ✓ lp tokens are locked (1 ms)
          ✓ lockup shares are roughly equal (1 ms)
          ✓ lp tokens not marked as transferred
          ✓ staked lp amount increases
Quint generated step ADVANCE BLOCK from step 7
  ✓ advancing 10 blocks (10097 ms)
Quint generated step CLAIM_REWARDS_XYK from step 8
  ✓ for Alice without withdraw (2065 ms)
Quint generated step ADVANCE BLOCK from step 9
  ✓ advancing 10 blocks (10286 ms)
Quint generated step CLAIM_REWARDS_PCL from step 10
  check generator state
    ✓ check generator stake presence (7 ms)
    ✓ check generator rewards presence (8 ms)
  user Bob testing post migration state
    ✓ no withdrawal available from XYK (2039 ms)
    Bob claim with withdrawal
      ✓ gather state before withdrawal on PCL (133 ms)
      ✓ claim & withdraw USDC lockup from PCL (895 ms)
      ✓ claim & withdraw ATOM lockup from PCL (1036 ms)
      ✓ gather state after withdrawal (165 ms)
      funds flow
        ✓ lp tokens staked in generator
        ✓ lp tokens received by the user
        ✓ no ntrn received by the user
        generator rewards
          ✓ astro (4 ms)
          ✓ external rewards (2 ms)
Quint generated step ADVANCE BLOCK from step 11
  ✓ advancing 10 blocks (10084 ms)
Quint generated step CLAIM_REWARDS_XYK from step 12
  ✓ for Alice without withdraw (2068 ms)
Quint generated step ADVANCE BLOCK from step 13
  ✓ advancing 10 blocks (10267 ms)
Quint generated step CLAIM_REWARDS_PCL from step 14
  user Bob testing post migration state
    ✓ no withdrawal available from XYK (2053 ms)
    Bob claim with withdrawal
      ✓ claim & withdraw USDC lockup from PCL fails (1032 ms)
      ✓ claim & withdraw ATOM lockup from PCL fails (1029 ms)
Quint generated step ADVANCE BLOCK from step 15
  ✓ advancing 10 blocks (10232 ms)
Quint generated step MIGRATE from step 16
  ✓ fill liquidity migration contracts
  migrate Alice participant
    ✓ gather state before migration (87 ms)
    ❌ migrate the user (976 ms)
    ✓ gather state after migration (108 ms)
    check user liquidity migration
      XYK user lockups
        ✓ XYK lockup lp token addresses
        generator rewards
          ✓ claimable generator ntrn debt (3 ms)
          ✓ generator rewards are transferred to the user (1 ms)
        astroport lp
          ❌ lp tokens marked as transferred (4 ms)
```

❌ **staked lp amount decreases**

lockdrop participation rewards

✓ no balance change for PCL lockdrop contract (2 ms)

✓ no paired assets and lp received by user (1 ms)

PCL user lockups

✓ no user lockup info before migration

❌ **PCL lockup lp token addresses**

❌ **lockup positions consistency**

astroport lp

❌ **lp tokens are locked**

❌ **lockup shares are roughly equal**

❌ **lp tokens not marked as transferred**

❌ **staked lp amount increases**

confirm lockdrop withdrawal completeness

❌ **no XYK lp tokens kept by XYK lockdrop (4 ms)**

no generator rewards left to be paid

✓ for XYK pairs (7 ms)

Error from step 3:

- TGE / Migration / PCL contracts › Quint generated steps › Quint generated step CLAIM_REWARDS_XYK from step 3 › for **Alice** without withdraw

```
failed to execute message; message index: 0: dispatch: submessages: Generic error: No
rewards available to claim!: execute wasm contract failed
Failed tx hash: 5E76C5CD2A78A79D3F5F872005B22E2380BC869B08FF16FCF2718EBBF4BA7D0A
```

Error from step 16:

- TGE / Migration / PCL contracts › Quint generated steps › Quint generated step MIGRATE from step 16 › migrate **Alice** participant › migrate the user

```
failed to execute message; message index: 0: dispatch: submessages: dispatch:
submessages: dispatch: submessages: dispatch: submessages: dispatch: submessages:
dispatch: submessages: : invalid coins
Failed tx hash: 40AD2A2C41F592A858EC2F977B5A8183D628D956E7F456EB7CA55DB8BF485B60
```

Error from this step leads to all other parts of it to fail.

When running the same test, but with larger initial amount of ATOMs/USDCs locked, there is no error.

Failure to account for rewards sent to the lockdrop contract when depositing makes expected test properties failed

This problem is explained in the finding Reward amounts in lockdrop-pcl contract are miscalculated in case of time-extended migration . It occurred in many test cases. Here, we present the simplest one:

1. Bob migrates Alice
2. Charlie migrates Bob
3. Alice attempts to withdraw from PCL, but gets no rewards, even though she should.

Trying to send more rewards than available

Problem is present when executing trace: link.

Brief summary of actions taken in the trace:

- Alice is migrated
- Charlie claims rewards XYK ✅
- Alice claims rewards PCL ✅
- Charlie migrated to PCL ✅
- Bob migrated to PCL ✅
- Bob withdraws rewards PCL ❌
    - claiming of ATOM fails with message supplied in the code snippet bellow.
    - because of this, some further checks fail as well.

```
failed to execute message; message index: 0: dispatch: submessages: dispatch:
submessages: spendable balance 27325factory/
neutron1tmxxpvd6hnr6vw9vr6cs9m0qu7um3dmwyltknu/urwrd is smaller than 39214factory/
neutron1tmxxpvd6hnr6vw9vr6cs9m0qu7um3dmwyltknu/urwrd: insufficient funds
Failed tx hash: 2B33A3DCAE3F1E09A47AA0CDE26943FE08702B26FDC767051B46D086BE8F18D4
```

- Charlie withdraws rewards PCL ❌
    - claiming of ATOM fails with the same message as Bob's withdraw..
    - because of this, some further checks fail as well.
- Charlie claims rewards PCL ❌
    - this should fail-safe because there is no position open.
    - that is not the case, because there is still open position for ATOM that failed before
- Bob claims rewards PCL ❌
    - this should fail-safe because there is no position open.
    - that is not the case, because there is still open position for ATOM that failed before
- Alice claims rewards PCL ✅

Vesting claim after migration fails

Problem is present when executing trace: link.

Brief summary of actions taken in the trace:

- Alice claims rewards XYK ✅
- Alice migrated to PCL ✅
- Alice claims PCL ✅
- Charlie withdraws XYK ✅
- Charlie migrated to PCL (no-op) ✅
- Bob migrated to PCL ❌
  - Bobs has locked: 5000000 uATOM and 2000 uUSDC.
  - this operation fails in the last part when should claim vesting.

```
TGE / Migration / PCL contracts › Quint generated steps › Quint generated step
MIGRATE from step 10 › execute migration of vesting for user Bob › should claim

    expect(received).toBeGreaterThanOrEqual(expected)

    Expected: >= 38.95
    Received:    32

      3630 | // e.g. 10% is tolerance = 0.1
      3631 | const isWithinRangeRel = (value: number, target: number, tolerance:
number) => {
    > 3632 |   expect(value).toBeGreaterThanOrEqual(target - target * tolerance);
           |                 ^
      3633 |   expect(value).toBeLessThanOrEqual(target + target * tolerance);
      3634 | };

      at toBeGreaterThanOrEqual (src/testcases/run_in_band/tge.quint.test.ts:3632:17)
```

Lockup shares are not roughly equal and external rewards are not in range

Problem is present when executing trace: link.

Brief summary of actions taken in the trace:

- Bob migrated to PCL ✅
- Charlie claims rewards XYK ✅
- Alice migrated to PCL ✅
- Charlie migrated to PCL ❌
    - `lockup shares are roughly equal` fail!
    - error message:

```
  ● TGE / Migration / PCL contracts › Quint generated steps › Quint generated step
MIGRATE from step 7 › migrate Charlie participant › check user liquidity migration ›
PCL user lockups › astroport lp › lockup shares are roughly equal

    expect(received).toBeGreaterThanOrEqual(expected)

    Expected: >= 14.25
    Received:    14

      3630 | // e.g. 10% is tolerance = 0.1
      3631 | const isWithinRangeRel = (value: number, target: number, tolerance:
number) => {
    > 3632 |   expect(value).toBeGreaterThanOrEqual(target - target * tolerance);
           |                 ^
      3633 |   expect(value).toBeLessThanOrEqual(target + target * tolerance);
      3634 | };

      at toBeGreaterThanOrEqual (src/testcases/run_in_band/tge.quint.test.ts:3632:17)
      at Object.isWithinRangeRel (src/testcases/run_in_band/tge.quint.test.ts:2100:27)
```

- Charlie withdraws rewards PCL ✅
- Alice claims rewards PCL ✅
- Bob withdraws rewards PCL ✅
- Alice withdraws rewards PCL ✅
- Charlie withdraws rewards PCL ✅
    - this fails-safe because there is open position
- Bob claims rewards PCL ✅
    - this fails-safe because there is open position
- Alice withdraws rewards PCL ❌
    - `external rewards` fail!
    - error message:

```
 ● TGE / Migration / PCL contracts › Quint generated steps › Quint generated step
CLAIM_REWARDS_PCL from step 12 › user Alice testing post migration state › Alice
claim with withdrawal › funds flow › generator rewards › external rewards

    expect(received).toBeLessThanOrEqual(expected)

    Expected: <= 9226.5
    Received:    9237
```

```
      3631 | const isWithinRangeRel = (value: number, target: number, tolerance:
number) => {
      3632 |   expect(value).toBeGreaterThanOrEqual(target - target * tolerance);
    > 3633 |   expect(value).toBeLessThanOrEqual(target + target * tolerance);
           |                 ^
      3634 | };

      at toBeLessThanOrEqual (src/testcases/run_in_band/tge.quint.test.ts:3633:17)
      at Object.isWithinRangeRel (src/testcases/run_in_band/tge.quint.test.ts:2961:27)
```

Appendix: Model-based Testing

In the course of the audit, we generated 20 tests that interleaved the steps of migrations and changed the parameters of it. (In principle, we could have generated any number of tests, the upper bound comes from the time needed to run and analyze test results.)

Our test generation consists of several steps. First, we developed a Quint model of the migration. Then we used the model to create a number of test traces (in the form of JSON). We adapted the existing tge.auction.test.ts to be able to read the test traces from JSON and execute them. After each step was executed in the engine, we checked a number properties, following the properties checked in the `tge.auction.test.ts`. The scope of the test were direct properties of migration and we did not check for the voting power (this may be a useful future addition).

We omit examples of JSON traces in this report, but we illustrate the Quint model with its key parts:

Model state

The model state consists of the state variable `users`, which is a mapping from `Addr` to `UserData`, defined as following:

```
type UserData = {
        migrated: bool,
        xyk_liquidity_withdrawn: bool,
        pcl_liquidity_withdrawn: bool,
        has_xyk_rewards: bool,
        has_pcl_rewards: bool,
        only_vesting: bool
    }
```

In `UserData`, we follow the logical state of each user (user=lockup in our model).

Initialization action: users are given different number of tokens initially

```
action init: bool =
        val emptyMsgInfo = {sender: ""}
        all {
            val usersNTRNLocked = USER_NAMES.mapBy(
                name => AMOUNTS.oneOf()
            )
            val usersATOMLocked = USER_NAMES.mapBy(
                name => AMOUNTS.oneOf()
            )
            val usersUSDCLocked = USER_NAMES.mapBy(
                name => AMOUNTS.oneOf()
            )
            stepInfo' = {
                actionTaken: "init",
                msgInfo: emptyMsgInfo,
                msgArgs: InitArgs(USER_NAMES.mapBy(
                    name => {
                        NTRN_locked: usersNTRNLocked.get(name),
                        ATOM_locked: usersATOMLocked.get(name),
                        USDC_locked: usersUSDCLocked.get(name)
                    })
                ),
                actionSuccessful: true,
                actionErrorDescription: ""
            },

            val onlyVestingUsers = USER_NAMES.mapBy(
                name => Set(true, false).oneOf()
            )
            users' = USER_NAMES.mapBy(
                name => {
                    migrated: false,
                    xyk_liquidity_withdrawn: false,
                    pcl_liquidity_withdrawn: false,
                    has_xyk_rewards: true,
                    has_pcl_rewards: false,
                    only_vesting: onlyVestingUsers.get(name)
                }
            ),
            numSteps' = 0

        }
```

Step Action

The step action defines what can happen in each next step. In the model, the step action is defined as follows

```
action step =
        all{
            numSteps' = numSteps + 1,
            val msg_info = {sender: USER_NAMES.oneOf()}
            any {
                val withdraw = Set(true, false).oneOf()
                claim_rewards_xyk(msg_info, withdraw),

                advance_block,

                val withdraw = Set(true, false).oneOf()
                claim_rewards_pcl(msg_info, withdraw),

                val user_address = oneOf(USER_NAMES.union(Set("")))
                migrate(msg_info, user_address)
            }
        }
```

In natural language, it says that:
a) *either* a randomly generated user withdraws from the xyk pool [6-7], *or*
b) the block is advanced [9], *or*
c) a randomly generated user withdraws from from the pcl pool [11-12], *or*
d) a randomly generated user migrates a (potentially different) randomly generated address [14-15].

Test property definition

With test properties, we can describe what kind of test scenarios we want to generate. Take the following example:

```
val DESIRED_NUMBER_OF_STEPS = 15
val fullMigrationHappened_inv = all {
        numSteps > DESIRED_NUMBER_OF_STEPS,
        users.keys().forall(
            addr =>
            (users.get(addr).only_vesting == false
            and
            users.get(addr).xyk_liquidity_withdrawn_before_migration == false)
            implies
            users.get(addr).migrated == true
            )
    }
```

In this property, we require that the trace is longer than 15 [line 3], and that at the end of the trace, all users for which migration was possible have indeed been migrated [line 10].

Having this property defined, we can generate any number of traces satisfying it. For each step of the trace, we check its expected precondition and postcondition. If cases when the conditions are not satisfied, there is a problem in the codebase.

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.

Impact Score	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:

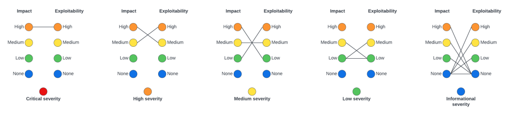
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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)

Exploitability Score	**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.



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.

Severity Score	**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

Severity Score	Examples
🟢 **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: Calculation of lockdrop pool rewards

The transfer of rewards happens in following steps (for simplification, we avoid mentioning 3rd party generator rewards since they do not exist at the moment):

1. user's total one-time NTRN rewards are calculated and saved under `USER_INFO.total_ntrn_rewards`.
2. In case there are some pending rewards, we save how many ASTROs we currently have, claim rewards on behalf of the whole lockdrop contract, and then update the state:
   a. we increase the info on `pool_info.generator_ntrn_per_share` (NOTE: should be ***astro*** per share) by `(astro_received - astro_previous)/lp_balance`. By doing this, we make sure that in every claim we assign a fair share of rewards to each user (more precisely: to each lockup) who is still part of the lockdrop contract.
3. Now we want to send lockup rewards to users who earned them. We calculate this in the following way:
   a. first, the amount of LP tokens corresponding to the lockup is calculated in the following way: `lp_amount = (lockup.lp_units_locked / pool.amount_in_lockups) * balance_of_lockdrop_contract`. The first ratio corresponds to the "fair share of this particular lockup", which is then multiplied with the total amount of LP tokens available.
   b. then, total claimable rewards are calculated in the following way: `total_lockup_astro_rewards = floor(pool.astro_per_share * lp_amount)`. This is saying: I know how many ASTROs each share earned thus far (was updated at every claim), so I multiply this with the number of shares that this lockup has. This now includes all rewards ever earned (including those that are already paid out).
   c. In order to get `pending_astro_rewards`, the amount that needs to be sent to the user, we need to understand what was already sent to them (eg, at previous claims, before the migration started). Thus, `pending_astro_rewards = total_lockup_astro_rewards - lockup.generator_astro_debt` and afterwards we update `lockup.generato_astro_debt = total_lockup_astro_rewards`. Finally, we send all those rewards to the user.
4. If the user did not claim their one-time NTRN rewards for participating in the lockdrop (as calculated in the step 1), they too are sent, and the airdrop reward is claimed from the credits contract (with corresponding credit tokens burnt).

Disclaimer