Security Audit Report

Neutron Smart Contracts: Overrule + Liquidity Migration

Authors: Ivan Gavran, Andrey Kuprianov, Nikola Jovicevic

Last revised 5 November, 2023

Table of Contents

Audit overview

The Project

In September 2023, Informal Systems has conducted a security audit for Neutron. The audit covered two areas:

1. Governance smart contracts allowing the main Neutron DAO to overrule passed proposals of subDAOs. The audited repository was neutron-dao at the commit hash 1cd5d88, focusing on the following contracts:
    a. `neutron-dao/contracts/dao/pre-propose/cwd-pre-propose-single-overrule`
    b. `neutron-dao/contracts/subdaos/pre-propose/cwd-subdao-pre-propose-single`
    c. `neutron-dao/contracts/subdaos/cwd-subdao-timelock-single`
    d. `neutron-dao/contracts/subdaos/pre-propose/cwd-security-subdao-pre-propose`
2. The update of *lockdrop*, *reserve,* and *vesting-lp* smart contracts that enables transferring funds from XYK pools to concentrated liquidity (CL) pools. This implied auditing the neutron-dao repository at the commit hash 1cd5d88, and the neutron-tge-contracts repository at the commit hash 410d560. The contracts of particular interests for the audit were
    a. `neutron-dao/contracts/dao/voting/neutron-voting-registry`
    b. `neutron-dao/contracts/dao/voting/lockdrop-vault-for-cl-pools`
    c. `neutron-dao/contracts/dao/voting/vesting-lp-vault-for-cl-pools`
    d. `neutron-dao/contracts/tokenomics/reserve`
    e. `neutron-tge-contracts/contracts/lockdrop`
    f. `neutron-tge-contracts/contracts/vesting-lp`

The audit was performed from September 4, 2023 through September 29, 2023, by the following personnel:

- Ivan Gavran
- Andrey Kuprianov
- Nikola Jovicevic

Conclusions

We performed a thorough review of the contracts, as well as the deployment workflow, and found it to be implemented carefully. Nonetheless, some subtle mistakes were found. In our audit, we report the total of 11 findings: two of them of the `critical` severity, and one of the `high` severity.

Audit dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: CosmWasm
- **Artifacts**: neutron-dao, neutron-tge-contracts

Engagement Summary

- **Dates**: 04.09.2023 -- 29.09.2023
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	**#**
Critical	2
High	1
Medium	0
Low	2
Informational	6
Total	**11**

Identified Threats and the Audit Plan

When analyzing both parts of the audit, governance overrule contracts and liquidity migration contracts, we focused first on understanding the protocol, looking for potential errors in the protocol, and finally inspecting the code to find implementation errors.

During our analysis, we inspected in particular the following general threats:

1. Crossing Trust Boundaries: any input coming from an independent (and thus potentially malicious) source must not be accepted unconditionally.
2. Inter-Contract Interactions: whether the contracts employed during execution respect the expected contracts, and whether they can be influenced with malicious intents.
3. Error Handling: every potential failure in the sequence of the messages needs to be handled appropriately.
4. Mathematical Computations: whether all computations are performed correctly, and if there are problems due to e.g. incorrect datatype usage or rounding
5. Access Control: entry points of a contract should contain an access check (unless being permissionless is an intended setup).
6. Migration Setup: best practices when migrating contracts need to be followed.
7. State Variables: each state variable that gets changed needs to be eventually updated in the storage.
8. CosmWasm Message Transfer: whether the used patterns of messages and submessages correspond to the desired effects, e.g. whether message flows are uninterrupted, or all submessage results are processed correctly.

Specific to the analyzed contracts, we inspected the following questions:

1. Are the checks that are in place before creating an overrule proposal sufficient, and are they implemented correctly.
2. Are there any flaws resulting in the violation of the overrule contract invariant:
   If a subDAO (other than security subDAO) votes in a proposal, it will get timelocked and the main dao will get a chance to overrule it. If it votes to overrule, it will not be executed. Otherwise, it will be executed. A security subDAO won't be executed, but can only propose `Pause` and `RemoveSchedule` *messages.*
3. Is it possible for the Security subDAO to use its no-timelock power to propose and execute messages other than `Pause` and `RemoveSchedule`.
4. Is it possible that a migration status state gets stuck mid-migration and cannot be changed.
5. Inspect the querying of voting power and check numerical calculations.
6. Could the restriction on the maximum slippage block the migration.

In the audit process, we inspected the listed threats, resulting in the findings presented in the next section.

Findings

Title	Type	Severity	Status
LP Vesting contract may get permanently stuck in the migration process	IMPLEMENTATION	4 CRITICAL	RESOLVED
Reliance on contract token balance to provide liquidity may permanently disable Lockdrop contract migration	IMPLEMENTATION	4 CRITICAL	RESOLVED
Absence of safeguards in voting power calculations may lead to voting overturn	PROTOCOL IMPLEMENTATION	3 HIGH	RISK ACCEPTED
Risk of instantiating malicious contracts	IMPLEMENTATION	1 LOW	RISK ACCEPTED
Overly permissive handling of a pre-proposal error	PROTOCOL	1 LOW	RISK ACCEPTED
Duplicate contract names	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Non-intended behaviour upon failure to instantiate a pre-proposal module	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Documentation: quorum in overrule contracts	DOCUMENTATION	0 INFORMATIONAL	RESOLVED
Code Improvements	PRACTICE	0 INFORMATIONAL	RESOLVED
Premature state update in MigrateLiquidity message execution	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Explicit Check of the Assumptions	PRACTICE	0 INFORMATIONAL	RESOLVED

LP Vesting contract may get permanently stuck in the migration process

Title	LP Vesting contract may get permanently stuck in the migration process
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Threats	Broken state machine logic
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-tge-contracts/pull/60 https://github.com/neutron-org/neutron-integration-tests/pull/218

Involved artifacts

- neutron-tge-contracts at branch `neutron_audit_informal_04_09_2023`:
  - vesting-lp contract
  - vesting-lp-base handlers

Description

The `vesting-lp` contract performs its actions via delegating message execution, queries, and migration to the `vesting-lp-base` handlers.

Execution of all normal contract messages is preconditioned that the migration process is completed as follows:

```
pub fn execute(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, ContractError> {
    let migration_state = MIGRATION_STATUS.may_load(deps.storage)?;
    if migration_state.unwrap_or(MigrationState::Completed) !=
MigrationState::Completed {
        match msg {
            ExecuteMsg::MigrateLiquidity {
                slippage_tolerance: _,
            } => {}
            ExecuteMsg::Callback(..) => {}
            _ => return Err(ContractError::MigrationIncomplete {}),
        }
    }
    match msg {
    ...
```

This means that before the migration is completed, only the `MigrateLiquidity` message, as well as migration callbacks can execute.

`MigrateLiquidity` is processed by function execute_migrate_liquidity(), which looks like this:

```
fn execute_migrate_liquidity(
    deps: DepsMut,
    env: Env,
    slippage_tolerance: Option<Decimal>,
) -> Result<Response, ContractError> {
    let migration_state: MigrationState = MIGRATION_STATUS.load(deps.storage)?;
    if migration_state == MigrationState::Completed {
        return Err(ContractError::MigrationComplete {});
    }
    let migration_config: XykToClMigrationConfig =
XYK_TO_CL_MIGRATION_CONFIG.load(deps.storage)?;

    let vesting_infos = read_vesting_infos(
        deps.as_ref(),
        migration_config.last_processed_user,
        Some(migration_config.batch_size),
        None,
    )?;

    let vesting_accounts: Vec<_> = vesting_infos
        .into_iter()
        .map(|(address, info)| VestingAccountResponse { address, info })
        .collect();

    if vesting_accounts.is_empty() {
        MIGRATION_STATUS.save(deps.storage, &MigrationState::Completed)?;
    }
    let mut resp = Response::default();

    // get pairs LP token addresses
    let pair_info: PairInfo = deps
        .querier
        .query_wasm_smart(migration_config.xyk_pair.clone(), &PairQueryMsg::Pair
{})?;
```

```
    // query max available amounts to be withdrawn from pool
    let max_available_amount = {
        let resp: BalanceResponse = deps.querier.query_wasm_smart(
            pair_info.liquidity_token.clone(),
            &Cw20QueryMsg::Balance {
                address: env.contract.address.to_string(),
            },
        )?;
        resp.balance
    };

    if max_available_amount.is_zero() {
        return Ok(resp);
    }

    for user in vesting_accounts.into_iter() {
        ...
    }

    Ok(resp)
}
```

The migration state, which is initialized to `MigrationState::Started` in function migrate() is set to `MigrationState::Completed` in the code above only when no vesting accounts remain in the contract state to be processed, as determined by the variable `migration_config.last_processed_user`. This variable is, in turn, updated only in the last callback called in the chain of callbacks, namely post_migration_vesting_reschedule_callback().

In order to not exceed the gas limit, function `execute_migrate_liquidity` requests and processes vesting accounts in batches of size `migration_config.batch_size`; this means that this function may need to execute multiple times.

The problem lies in how the above two functionalities, namely batched processing and migration state machine, interact. In particular, the code above is executed at the beginning of each batch, and contains the following fragment:

```
    // query max available amounts to be withdrawn from pool
    let max_available_amount = {
        let resp: BalanceResponse = deps.querier.query_wasm_smart(
            pair_info.liquidity_token.clone(),
            &Cw20QueryMsg::Balance {
                address: env.contract.address.to_string(),
            },
        )?;
        resp.balance
    };

    if max_available_amount.is_zero() {
        return Ok(resp);
    }
```

The above fragment queries the current balance of liquidity tokens, and *terminates the function early if the balance reaches zero, assuming that nothing left to do*. The problem is that while the balance reaches zero, there might be still vesting accounts remaining, for which all tokens have been released. This means that the code that sets migration status to `MigrationState::Completed` never gets executed, and the contract gets stuck in the `MigrationState::Started` state.

Problem Scenarios

It may happen that at the moment of performing migration, the combination of the current set of vesting accounts, and the migration batch size will be such that the remainder of the set will have all funds released, but still some vesting accounts left; thus leading to the situation described above. **The result will be the contract being permanently locked in the** `MigrationState::Started` state, unable to execute any further messages.

Recommendation

Make the conditions of "no vesting accounts left" and "no available funds left" equal in power, and set the migration state to `MigrationState::Completed` in both cases.

Reliance on contract token balance to provide liquidity may permanently disable Lockdrop contract migration

Title	Reliance on contract token balance to provide liquidity may permanently disable Lockdrop contract migration
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Threats	Influence by dynamic environment
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-tge-contracts/pull/58 https://github.com/neutron-org/neutron-integration-tests/pull/210

Involved artifacts

- neutron-tge-contracts at branch `neutron_audit_informal_04_09_2023`:
  - lockdrop contract

Description

In the lockdrop contract, in function migrate_pair_step_2(), the following fragment is present:

```
    let token_denom = new_pool_info
         .assets
         .iter()
         .find_map(|x| match &x.info {
             AssetInfo::NativeToken { denom } if denom != UNTRN_DENOM =>
Some(denom.clone()),
             _ => None,
         })
         .ok_or_else(|| StdError::generic_err("No second leg of pair found"))?;
     attrs.push(attr("token_denom", token_denom.clone()));
     //calculate amount of token (ATOM|USDC) claimed from pool
     let token_balance = deps
         .querier
         .query_balance(&env.contract.address, token_denom.as_str())?
         .amount;
     attrs.push(attr("token_balance", token_balance.to_string()));

     //construct message to send NTRN and (ATOM|USDC) to new pool
     let base = Asset {
         amount: ntrn_to_new_pool,
         info: AssetInfo::NativeToken {
             denom: UNTRN_DENOM.to_string(),
         },
     };
     let other = Asset {
         amount: token_balance,
         info: AssetInfo::NativeToken {
             denom: token_denom.to_string(),
         },
     };
     let mut funds = vec![
         Coin {
             denom: UNTRN_DENOM.to_string(),
             amount: ntrn_to_new_pool,
         },
         Coin {
             denom: token_denom,
             amount: token_balance,
         },
     ];
     funds.sort_by(|a, b| a.denom.cmp(&b.denom));
     msgs.push(CosmosMsg::Wasm(WasmMsg::Execute {
         contract_addr: new_pool_addr,
         funds,
         msg: to_binary(&astroport::pair::ExecuteMsg::ProvideLiquidity {
             assets: vec![base, other],
             slippage_tolerance,
             auto_stake: None,
             receiver: None,
         })?,
     }));
```

In the above fragment, the current token balance of the contract is queried, with the intention to "calculate amount of token (ATOM|USDC) claimed from pool". The problem is, unlike NTRN amount, which is calculated as the difference between the previous and the current token balances, for the ATOM/USDC claimed amount the current token balance is used, and then passed directly as one of the asset amounts in the message `astroport::pair::ExecuteMsg::ProvideLiquidity`.

Function `migrate_pair_step_2()` is part of a sequence of calls, originating from processing of the message `ExecuteMsg::MigrateFromXykToCl`, which is permissionless. Anyone can call this function, providing additional funds with it, and this altering the contract token balance, which will lead to the wrong liquidity provided to the Astroport pair. As slippage tolerance is checked in this function, the additional provided liquidity may exceed the allowed slippage tolerance, thus making the migration to fail permanently.

Problem Scenarios

The following scenario is possible:

1. The new variant of the contract is deployed, and the migration starts
2. An attacker submits the message `ExecuteMsg::MigrateFromXykToCl` (which is permissionless), and sends additional funds with it.
3. The function fails, due to exceeding the slippage tolerance.
4. The contract is permanently locked, as migration will always fail from now on, and no other functions are allowed to execute (as migration is in progress).

Recommendation

We recommend to use the same scheme for getting the claimed amount of ATOM/USDC tokens as for NTRN tokens, namely as the difference in the contract token balance before and after withdrawing from the XYK pool.

Absence of safeguards in voting power calculations may lead to voting overturn

Title	Absence of safeguards in voting power calculations may lead to voting overturn
Project	Neutron Q3 2023
Type	PROTOCOL IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Threats	Crossing trust boundaries Influence by dynamic environment
Author(s)	Andrey Kuprianov
Status	RISK ACCEPTED
Link(s)	

Involved artifacts

- neutron-dao at branch `neutron_audit_informal_04_09_2023`:
  - neutron-voting-registry
  - Neutron vaults:
    - neutron-vault
    - credits-vault
    - investors-vesting-vault
    - lockdrop-vault
    - lockdrop-vault-for-cl-pools
    - vesting-lp-vault
    - vesting-lp-vault-for-cl-pools

Description

Neutron voting registry plays a central role in the Neutron system. Citing from Neutron documentation:

> *Instead of a single voting power module, Neutron DAO core contract interacts with the Voting Power Registry contract that keeps track of multiple Voting Vaults.*
> *A voting vault is a smart contract that implements the DAO DAO voting module interface, namely, it is capable of: Providing the total voting power at a given height,Providing the voting power of an address at a given height.*

> *The overall voting power of a given address is a sum of the voting powers that the address has in all of the registered voting vaults.*

More specifically, Neutron voting registry as well as all voting vaults it interacts with implement queries `VotingPowerAtHeight` and `TotalPowerAtHeight`, illustrated below on the example of voting registry:

```
QueryMsg::VotingPowerAtHeight { address, height } => {
    to_binary(&query_voting_power_at_height(deps, env, address, height)?)
}
QueryMsg::TotalPowerAtHeight { height } => {
    to_binary(&query_total_power_at_height(deps, env, height)?)
}
```

The implementation of query_voting_power_at_height iterates over all stored vaults, queries each of them, and sums up the results as follows:

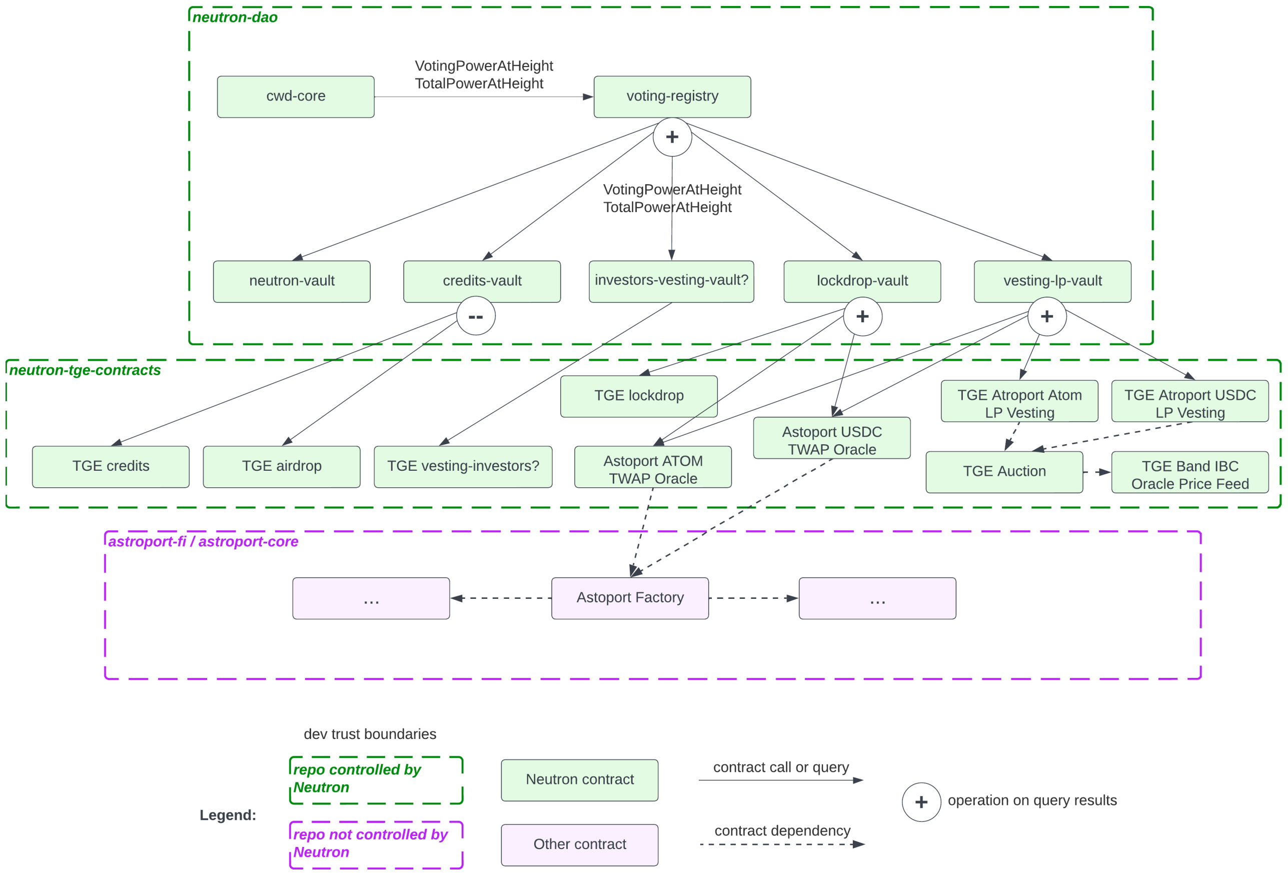
```
let mut resp = VotingPowerAtHeightResponse {
    power: Default::default(),
    height,
};
for vault in vaults {
    if let Some(vault_state) =
        VAULT_STATES.may_load_at_height(deps.storage, vault.clone(), height)?
    {
        if vault_state == VotingVaultState::Active {
            let vp_in_vault: VotingPowerAtHeightResponse =
deps.querier.query_wasm_smart(
                vault,
                &voting::Query::VotingPowerAtHeight {
                    height: Some(height),
                    address: address.clone(),
                },
            )?;
            resp.power = resp.power.checked_add(vp_in_vault.power)?;
        }
    }
}
Ok(resp)
```

At the time of writing, there are 5 vaults connected to the voting registry on Neutron mainnet; after the XYK to CL migration there are going to be 7 vaults in total: 5 active, and 2 inactive.

- Neutron vault is the simplest of all and doesn't do any further queries
- Credits vault to calculate total voting power calls into two further contracts: credits and airdrop
- Investors vesting vault both for total and individual voting power delegates the query to the HistoricalExtension of the vesting contract
- Lockdrop vault via a chain of subcalls, sums up results from delegating queries to three other contracts:
    - Queries the lockdrop contract with either `QueryUserLockupTotalAtHeight` or `QueryLockupTotalAtHeight`
    - Queries either ATOM or USDC oracle contract with the query TWAPAtHeight

- Lockdrop vault for CL pools behaves similar to the above, but delegates the calls to two other contracts, `atom_cl_pool_contract` and `usdc_cl_pool_contract`; the results are summed up.
- Vesting LP vault sums up the voting power obtained by queries to four other contracts: `atom_vesting_lp_contract`, `atom_oracle_contract`, `usdc_vesting_lp_contract`, `usdc_oracle_contract`.
- Vesting LP vault for CL pools behaves similar to the above, only delegates to other contracts.

Taking into account only the contracts deployed on Neutron mainnet now, the calculation of voting power can be illustrated as follows:



Problem Scenarios

The problem with the above approach is that Neutron voting registry delegates calculation of both total voting power, as well as individual, per-address voting power to multiple contracts, blindly trusting and summing up the returned results. **Notice the complete absence of safeguards: the returned results may exceed even the total supply, and the implementation will still sum everything up.** Moreover, the calculation is delegated to a large number of other contracts, some not under Neutron development authority; the exact number of dependencies is hard to assess. This represents already a very wide attack surface on the core Neutron DAO infrastructure, voting: **a single bug in any of the connected vaults or the other contracts they depend on may allow attackers to manipulate, or even completely overturn voting results in their favor.**

The problem is further exaggerated by the following:

- some of the vaults either contain non-trivial logic with complex mathematical calculations.
- some of the queried contracts, like oracles or price feeds collect dynamic data; it is possible that such data be manipulated by an attacker to skew the voting results.

We illustrate both of the above problems on the example of Vesting LP vault, which computes the voting power vy querying the vesting contract for the number of LP shares, and further delegating the calculation to the function `voting_power_from_lp_tokens()`:

```
fn get_voting_power(
    deps: Deps,
    config: &Config,
    height: u64,
    query_msg: &impl Serialize,
) -> ContractResult<Decimal256> {
    let mut voting_power = Decimal256::zero();
    for (vesting_lp, oracle) in [
        (
            &config.atom_vesting_lp_contract,
            &config.atom_oracle_contract,
        ),
        (
            &config.usdc_vesting_lp_contract,
            &config.usdc_oracle_contract,
        ),
    ] {
        voting_power += voting_power_from_lp_tokens(
            deps,
            deps.querier
                .query_wasm_smart::<Option<Uint128>>(vesting_lp, &query_msg)?
                .unwrap_or_default(),
            oracle,
            height,
        )?;
    }
    Ok(voting_power)
}
```

The function voting_power_from_lp_tokens() queries the oracle contract for a time-weighted average price (TWAP), and convert the LP shares into voting power expressed in NTRN:

```
pub fn voting_power_from_lp_tokens(
    deps: Deps,
    lp_tokens: Uint128,
    oracle_contract: impl Into<String>,
    height: u64,
) -> StdResult<Decimal256> {
    Ok(if lp_tokens.is_zero() {
        Decimal256::zero()
    } else {
        let twap: Decimal256 = deps
            .querier
            .query_wasm_smart::<Vec<(AssetInfo, Decimal256)>>(
                oracle_contract,
                &OracleQueryMsg::TWAPAtHeight {
                    token: AssetInfo::NativeToken {
                        denom: "untrn".to_string(),
                    },
                    height: Uint64::new(height),
                },
            )?
            .into_iter()
            .map(|x| x.1)
            .sum::<Decimal256>();

        Decimal256::new(Uint256::from(lp_tokens))
            .checked_div(twap.sqrt())
            .map_err(|err| StdError::generic_err(format!("{}", err)))?
    })
}
```

Finally, the oracle contract calculates TWAP as follows

```
fn twap_at_height(
    deps: Deps,
    token: AssetInfo,
    height: Uint64,
) -> Result<Vec<(AssetInfo, Decimal256)>, ContractError> {
    let config = CONFIG.load(deps.storage)?;
    let pair = config.pair.ok_or(ContractError::AssetInfosNotSet {})?;
    let price_last = PRICE_LAST
        .may_load_at_height(deps.storage, u64::from(height))?
        .ok_or(ContractError::PricesNotFound {})?;
    let mut average_prices = vec![];
    for (from, to, value) in price_last.average_prices {
        if from.equal(&token) {
            average_prices.push((to, value));
        }
    }

    if average_prices.is_empty() {
        return Err(ContractError::InvalidToken {});
    }

    // Get the token's precision
    let p = get_precision(deps.storage, &token)?;
    let one = Uint128::new(10_u128.pow(p.into()));

    average_prices
        .iter()
        .map(|(asset, price_average)| {
            if price_average.is_zero() {
                let price = query_prices(
                    deps.querier,
                    pair.contract_addr.clone(),
                    Asset {
                        info: token.clone(),
                        amount: one,
                    },
                    Some(asset.clone()),
                )?
                .return_amount;
                Ok((
                    asset.clone(),
                    Decimal256::from_integer(Uint256::from(price))
                        .div(Decimal256::from(one.to_decimal())),
                ))
            } else {
                let price_precision =
Uint256::from(10_u128.pow(TWAP_PRECISION.into()));
                Ok((asset.clone(), *price_average / price_precision))
            }
        })
        .collect::<Result<Vec<(AssetInfo, Decimal256)>, ContractError>>()
}
```

The above calculations contain non-trivial mathematical transformations, including both integer and decimal divisions and square roots; we have not fully explored the complete logic here, but there may be bugs both in calculations itself, as well as e.g. in the `Decimal256` library, with corner cases of it explored by an attacker to skew voting power results. Furthermore, the TWAP is calculated as an average price over a specific history of recent prices, and it is possible for an attacker to influence either the prices themselves, or the way they are obtained or transformed on the way from the source to the oracle, right before the required height, thus again skewing the voting power results.

We realize that the present scheme of calculation of voting power on Neutron stems from the desire of obtaining maximal decentralization and flexibility, maintaining at the central contracts only the bare minimum of functionality, while delegating as much power as possible to the community. Nevertheless, we believe that in case of matters of such vital importance as voting more precautions need to be taken; blindly trusting the results obtained from a multitude of other contracts is very dangerous.

Many things may go wrong with the present scheme:

- Some bug in one of the queried contract may allow an attacker to modify the internal contract storage in such a way that the returned voting power is disproportionally high.
- Some bug in the mathematical calculations of one of the contracts, or in one of the libraries on which these calculations depend (e.g. `Decimal256`) may affect the returned results. Such bug, if discovered by an attacker, may be also used to skew the voting results in their favor.
- An attacker may influence dynamic data that are fed into the system, creating favorable conditions to skew the voting power calculations.
- As the number of the contracts either directly implementing the voting vault API, or their subcontracts, grow, it is likely that some will receive less attention and scrutiny. That provides the ground for both inadvertent as well as malicious introduction of bugs that may eventually affect voting results. It is worth noting that some of the contracts which perform voting power results are not under Neutron development authority, i.e. the queries cross the trust boundaries. It is possible e.g. that Astroport contracts are upgraded, and the combination of upgraded contracts with Neutron contracts creates the possibility for an attack.

The main reason why all of this is possible is the absence of safeguards: there are no checks that validate that the returned results are sensible.

Recommendation

The bare minimum that should be checked is of course that the returned voting power result doesn't exceed the total NTRN supply, which is at the same time doesn't allow to mitigate the problem. What seems reasonable to us, is to modify the scheme of interactions of the voting registry with voting vaults as follows:

- Maintain, in the voting registry, a mapping from the voting vault address to the voting power allowance this vault can contribute in the voting; let's call it `voting_power_allowance`

    - Add an administrative call, say `ExecuteMsg::SetVotingPowerAllowance`, which the Neutron DAO can execute.
    - This will give the DAO the flexibility to control the amount of the voting power each vault may contribute to the overall results
- Maintain, in the voting registry, a mapping from the voting vault address to the maximal voting power this vault, *as it assesses internally*, may scan contribute in the voting; let's call it `max_voting_power`. The idea is that the vaults know better what maximal power they may contribute, and should update it regularly, to match the current cap of their voting power
    - Add a call, say `ExecuteMsg::SetMaxVotingPower` that only the respective vault may call to set/update its maximal voting power.
- Optionally, also maintain the equivalent of the above numbers, which would cap the allowance and maximum per single address (generically). Those could be called `voting_power_allowance_per_address` and `max_voting_power_per_address`.
    - The mechanics of calls would be the same as above: the allowance set by the Neutron DAO, and the maximim set by each vault.

The calculation of the voting power would be done as follows, when an individual vote is cast, for each casted vote, and each vault:

- check that the returned voting power for address doesn't exceed the `max_voting_power_per_address` or the `voting_power_allowance_per_address`; in case it does; cap it at the smaller of the two.
- additionally, check that the accumulated voting power casted by this vault doesn't exceed `max_voting_power` or `voting_power_allowance`; if it does, cap it at the smaller of the two.

The scheme outlined above is not the only possible one of course. It has been designed to provide the following advantages over the presently used one:

- Put safeguards in place that allow to restrict voting power casted by each vault.
- The safeguarding mechanism is double-sided: it allows both the vaults themselves to restrict their power based on the estimate of the current dynamic situation and its forecast, as well as give the Neutron DAO the means to put some hard limits on the amount of power each vault may have.
- The mechanism splits possible attacks on voting power into two necessary phases: if an attacker wants to exercise a disproportionally high voting power, they will have to first try to update the maximum voting power limit, and only then will they be able to exploit any possible misbehavior in the vault contract.
- The changes in maximum vault voting power may also be monitored, and any suspicious events acted upon in time.

Clarification from Neutron regarding the finding

That's a known issues and for now we accept the risk. But we have a fix for this issue in roadmap, which is quite complicated and requires a lot of refactoring and additional implementation for the contracts.

Risk of instantiating malicious contracts

Title	Risk of instantiating malicious contracts
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Threats	Risk of misconfigured smart contract instantiation
Author(s)	Ivan Gavran
Status	RISK ACCEPTED
Link(s)	

Involved artifacts

- cwd-subdao-core/src/contract.rs::instantiate
- cwd-subdao-proposal-single/src/contract.rs::instantiate
- cwd-subdao-pre-propose-single/src/contract.rs::instantiate
- cwd-subdao-timelock-single/src/contract.rs::instantiate

Description

In this finding we are exploring scenarios in which timelock and overrule modules are instantiated in such a way that they do not create overrule proposals.

In the first scenario, a malicious/erroneous sub-DAO's proposal module instantiates its pre-propose module with an incorrect content of `timelock_module_instantiate_info.msg.overrule_pre_propose` (e.g., pointing to a non-overrule pre-propose module, or a custom contract not connected to the main DAO). The `msg.overrule_pre_propose` is only checked for the validity of the address (deliberately so, as discussed in this comment).

In the second scenario, a malicious subDAO instantiates a custom timelock module by providing its own `code_id` under `msg.timelock_module_instantiate_info`.

Both of these scenarios pose a low risk because instantiations of new subDAOs is a closely controlled process.

Problem Scenarios

If instantiating a wrong overrule or timelock module were to succeed, it can override the intended functionality of the overrule mechanism by not creating overrule proposals or not actually overruling them when voted.

Recommendation

We suggest reconsidering the decision not to verify the connection between the `overrule_pre_propose` and the main DAO. Furthermore, we suggest querying the code info and comparing the checksum against the known checksum of the correct timelock module.

Clarification from Neutron regarding the finding

This is very small risk since we have quite controlled process of SubDAOs initialisation. So for now we leave it as it is.

Overly permissive handling of a pre-proposal error

Title	Overly permissive handling of a pre-proposal error
Project	Neutron Q3 2023
Type	PROTOCOL
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Threats	Minimal necessary authority principle
Author(s)	Ivan Gavran
Status	RISK ACCEPTED
Link(s)	

Involved artifacts

- subdaos/proposal/cwd-subdao-proposal-single/src/contract.rs

Description

1. If the pre-proposal hook errs on receiving the proposal-created hook, the CREATION_POLICY is changed to be `Anyone`. The consequences are
   a. For any future proposals coming from the pre-proposal module (if it is still active), we will raise the `InvalidProposer` error and won't be able to continue functioning.
   b. This makes it possible to go around the timelocking mechanism: if anybody can create a proposal (checked here), then the messages don't necessarily have to be timelock-wrapped.

Problem Scenarios

Inspecting the pre-proposal handling of the *proposal created* hook, it is not clear how an honest pre-proposal module could raise an error. One potential way in which this problem could occur would be through a dishonest pre-proposal module deployed, as suggested in the finding Risk of instantiating a malicious contract.

Recommendation

We suggest changing the permission for proposing to a reliable address (eg, to the security subdao address), instead of allowing everybody to propose. An analogous problem holds for the DAO proposals.

Remark & clarification from Neutron regarding the finding

This behavior is inherited from the original DAO-DAO code (link).

It's completely ok to allow everyone to create proposals if a Pre-Propose module raised an error.

Duplicate contract names

Title	Duplicate contract names
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	2 MEDIUM
Exploitability	0 NONE
Threats	Risk of misconfigured smart contract instantiation
Author(s)	Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-dao/commit/000a728f8e48d4ef5efb13f550f5c53820c418ca https://github.com/neutron-org/neutron-dao/commit/8bdad6fb3904b5ac390b276a466aaacaba27bb3e

Involved artifacts

- subdaos/pre-propose/cwd-subdao-pre-propose-single/src/contract.rs
- subdaos/pre-propose/cwd-security-subdao-pre-propose/src/contract.rs
- dao/pre-propose/cwd-pre-propose-single/src/contract.rs

Description

Two contracts that are meant to be simultaneously active, `cwd-subdao-pre-propose-single` and `cwd-security-subdao-pre-propose` have the same value for `CONTRACT_NAME` , namely " `crates.io:cwd-subdao-pre-propose-single` " (check here and here). Furthermore, the same name is used for `dao/pre-propose/cwd-pre-propose-single` (here).

Problem Scenarios

This can create confusion when instantiating contracts.

Recommendation

Provide unique names to contracts with different purpose.

Non-intended behaviour upon failure to instantiate a pre-proposal module

Title	Non-intended behaviour upon failure to instantiate a pre-proposal module
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	CosmWasm submessages semantics
Author(s)	Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-dao/pull/88/files

Involved artifacts

- packages/cwd-voting/src/pre_propose.rs

Description

In the function `into_initial_policy_and_messages`, setting the policy to `ModuleMayPropose` is postponed until the pre-proposal instantiation is finished (link).

```
                // Anyone can propose will be set until instantiation succeeds, then
                // `ModuleMayPropose` will be set. This ensures that we fail open
                // upon instantiation failure.
                ProposalCreationPolicy::Anyone {},
                vec![SubMsg::reply_on_success(
                    info.into_wasm_msg(dao_address),
                    pre_propose_module_instantiation_id(),
                )],
```

Upon failure, however, the whole transaction will be reverted, because there is no handler for failure. This contradicts the intention to *fail open upon instantiation failure* expressed in the comment above.

Problem Scenarios

We do not see any safety issues due to this problem because the whole transaction will fail upon failure to instantiate a pre-proposal module.

Recommendation

If the intention is to fail-open, then that must be handled in the `ReplyOn::Error` handler. Alternatively, we recommend removing the confusing comment.

Remark

This behavior is inherited from the original DAO-DAO contract (link).

Documentation: quorum in overrule contracts

Title	Documentation: quorum in overrule contracts
Project	Neutron Q3 2023
Type	DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Code-documentation alignment
Author(s)	Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-docs/pull/110

Involved artifacts

- neutron-docs/docs/neutron/dao/overrules.md

Description

Among the caveats for overrule contracts, there is one that says: "*Quorum should be set to the **absolute count type** so that even if significant voting power is against overruling, it would happen anyway.*" However, this is wrong: it should be set to **absolute percentage type**. (And it indeed is set to absolute percentage type in the code, check here.)

Problem Scenarios

A mistake in the documentation could create confusion.

Recommendation

Change `absolute count type` to `absolute percentage type`.

Code Improvements

Title	Code Improvements
Project	Neutron Q3 2023
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Threats	Best Coding Practices
Author(s)	Nikola Jovicevic Andrey Kuprianov Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-dao/pull/94 https://github.com/neutron-org/neutron-tge-contracts/pull/67/files

Description

In this issue we describe a number of improvements to the code that do note affect the functionality, improve readability or are a good practice.

1. In the `migrate` functions of the *reserve* and *distribution* contracts, a call to function `set_contact_version` is missing.
2. The variable total_lokups should be renamed to `total_lockups`, to improve readability.
3. At a couple of places in the vesting-related packages (here, here, here, here), a submessage is added using the following idiom: `response = response.add_sumbessage(SubMsg::new(...))`. But this is equivalent to writing `response = response.add_message(...)`.
4. The `MigrationState::Started` is not used in the Lockdrop. (The logic immediately starts from the `MigrationState::MigrateLiquidity` state, so that element of the enum is superfluous.)
5. The function is_subdao_legit is implemented in two steps:
    a. for a given address `subdao_core`, we query the main dao's list of subdaos for one with the address `subdao_core` and save the result to `subdao`
    b. if we get back `Ok(…)`, we return `Ok(subdao.addr == *subdao_core)`

However, if we got back `Ok`, there is no need to compare the addresses and we could return `Ok(true)`.

6. The function comment above lockdrop execute should include the description of the Migrate message. (Or, alternatively, it should be deleted completely.)
7. This assignment could be done before checking for max values (ie, here) and the checks would still be meaningful. That way, the double work of checking for nil would be avoided and the code would become clearer.

Premature state update in MigrateLiquidity message execution

Title	Premature state update in MigrateLiquidity message execution
Project	Neutron Q3 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Code Clarity
Author(s)	Nikola Jovicevic
Status	ACKNOWLEDGED
Link(s)	

Involved artifacts

- neutron-tge-contracts at branch `neutron_audit_informal_04_09_2023`:
  - lockdrop contract

Description

In the lockdrop contract, the migration process for each pool involves three sequential steps, managed by the MigrateLiquidity message. These steps are executed consecutively, one after another. At the conclusion of the migration process for the first pool, specifically during the third step, the `MIGRATION_STATUS` is updated to `MigrationState::MigrateUsers`, which is basically on the half of the liquidity migration process (since the second pool is still not migrated). The update to `MgrationState::MigrateUsers` is repeated once more after the second pool is migrated.

Problem Scenarios

The premature state update in this case does not directly impact the core functionality of the system. However, in the event of future upgrades or modifications to the contract (e.g., if during the migration there would be a check of the status), this inconsistency may pose a problem.

Recommendation

To address this issue, we advise to implement a mechanism for tracking the progress of the migration. There are several viable approaches to achieve this:

1. **Boolean or Integer Variable:** Introducing a boolean or integer variable in the storage, dedicated to tracking the migration progress on the pool level. This variable can be updated each time a pool's liquidity is migrated, ensuring that the `MIGRATION_STATUS` is only transitioned when both pools have completed their respective migrations.
2. **Additional Queries for Pool Balances:** Implementing additional queries to check the balances of the pools involved in the migration process. This method allows for a more dynamic assessment of the progress and ensures that the state update occurs only when both pools have successfully migrated their funds.

Clarification from Neutron regarding the finding

The premature state update in this case does not directly impact the core functionality of the system: all the liquidity migration steps dispatched as simple messages, and the `MigrateExecuteMsg::MigrateLiquidity` *handler initialises migration of both pairs as simple messages. Therefore currently there is no such a case that migration can be half-complete at the point of* `MigrationState::MigrateUsers` *committed to the contract's state because it's only committed at the end of the* `MigrateExecuteMsg::MigrateLiquidity` *which only ends when migration of both pairs is completed successfully. Whereas the approach and code are not clean, we decided to go like that for the sake of time.*

Explicit Check of the Assumptions

Title	Explicit Check of the Assumptions
Project	Neutron Q3 2023
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Code Clarity
Author(s)	Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron-dao/pull/96/files#diff-aab427e50f62f5c437595ce83da1bcd7c6bbbf3c41ef963b6d807a4682fd268b

Involved artifacts

- neutron-dao/contracts/subdaos/cwd-subdao-timelock-single/src/contract.rs

Description

Inside the `execute_execute_proposal` function of the `timelock` contract, all messages from `proposal.msgs` are added to the response as submessages, with a `reply_on_error` handle.

```
let msgs: Vec<SubMsg<NeutronMsg>> = proposal
        .msgs
        .iter()
        .map(|msg| SubMsg::reply_on_error(msg.clone(), proposal_id))
        .collect();
```

However, the variable `proposal.msgs` should only ever contain a single element, of the type `ExecuteMsg::ExecuteTimelockedMsgs`.

Problem Scenarios

Iterating over `proposal.msgs` is creating an unnecessary confusion about the cardinality of `proposal.msgs`.

Recommendation

We suggest being explicit about the fact that there should only be a single message in `proposal.msgs`: we suggest first checking if that is really so, and afterwards adding as a submessage `msgs[0]`

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:

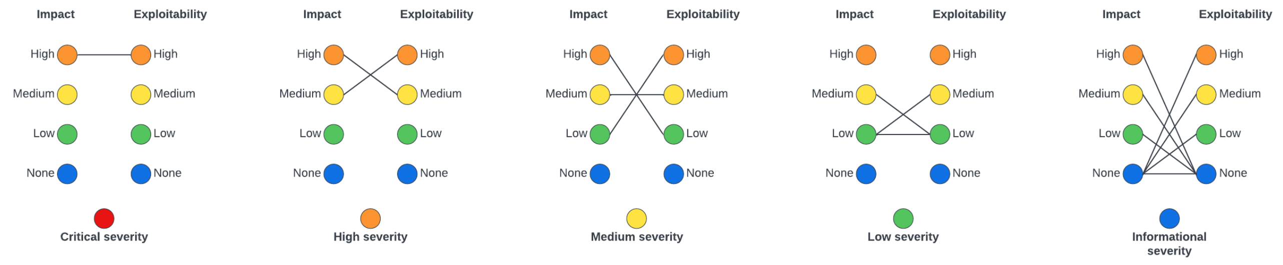
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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

Severity Score	Examples
🟠 **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

Disclaimer