Security Audit Report

Timewave: Stride Covenant

Authors: Nikola Jovicevic, Ivan Golubovic

Last revised 17 August, 2023

Table of Contents

Audit overview

The Project

In collaboration with Stride Chain, Timewave has developed a system encompassing six smart contracts slated for deployment on the Neutron Chain. These contracts facilitate interactions with Interchain Accounts (ICA) on both Gaia and Stride chains, as well as with the Astroport exchange on Neutron. This system represents Timewave's strategic solution for future agreements within the Cosmos ecosystem, with a particular focus on Stride's liquid staking mechanism. The process can be summarized as follows:

- Tokens are allocated to the Gaia ICA, initiated by one of the designated smart contracts.
- These funds are then directed to two destinations: the ICA on Stride and a smart contract linked to Astroport.
- Upon reaching Stride, the Atoms are subjected to liquid staking, gaining stAtoms in return.
- These stAtoms are subsequently relayed to the smart contract associated with the Astroport exchange.
- From this contract, tokens are transferred to the Astroport exchange, resulting in the acquisition of LP tokens.
- An authorized entity can retrieve these LP tokens using one of the system's smart contracts.

The whole mechanism is intended to be permissionless, but currently it requires some manual intervention due to technology being available at the moment (will be later discussed in the report).

Scope of this audit

The audit was scheduled from July 24, 2023, to August 18, 2023. The audit team consisted of the following personnel:

- Nikola Jovicevic
- Ivan Golubovic

During the audit, our focus was on analyzing the complete “Stride LP Covenant” system comprised of 6 different CosmWasm smart contracts.

Conducted work

The audit project encompassed the following activities:

- Manual code inspection of the Stride LP Covenant. We conducted a thorough examination of the codebase, documenting our insights in the "System Overview" section. The manual code inspection revealed the majority of the findings presented in this report.
- Thread modeling and inspection is documented in “Threat Inspection” chapter, giving the detailed insight in conducted work, identified threats and code snippets of interest, executed tests etc.
- Where possible, we attempted to reproduce the findings using Timewave’s unit/end-to-end test suite, as well as try to use the environment to test edge cases.

Conclusions

Overall, we found the codebase to exhibit exceptional quality, characterized by well-structured and comprehensible code. The comprehensive test suite includes unit, and valuable end-to-end testing environment. During the audit, we identified 2 low and 3 informational severity findings, confirming our previous impression of well designed system. Our collaboration with the Timewave team was exemplary, significantly enhancing the quality of work executed within the audit's timeframe.

Audit dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Rust
- **Artifacts**:
    - Commit hash: 52029460638cdf8c7be5a3844a7b929094b48808

Engagement Summary

- **Dates**: 26.07.2023. to 18.08.2023
- **Method**: Manual code review, protocol analysis, design analysis, testing
- **Employees Engaged**: 2

Severity Summary

Finding Severity	#
Critical	0
High	0
Medium	0
Low	2
Informational	3
Total	**5**

System Overview

Cosmos Hub is the central blockchain in the Cosmos Network, a decentralized network of independent, scalable, and interoperable blockchains. Through the use of the Inter-Blockchain Communication (IBC) protocol, Cosmos Hub facilitates the transfer of tokens and data between different chains, positioning itself as the "Internet of Blockchains." Its modular framework allows developers to create custom blockchains tailored to specific applications while maintaining connectivity with the broader Cosmos ecosystem.

Stride blockchain is an innovative platform that integrates advanced blockchain technology with a focus on enhancing liquidity and ensuring secure transactions. One of its standout features is liquidity staking, a mechanism that allows users to lock up their assets in return for rewards, thereby promoting stability and trust within the ecosystem. This process not only incentivizes participation but also ensures a seamless and efficient trading experience for all users. As Stride continues to evolve, its commitment to providing a robust and decentralized financial infrastructure remains evident, making it a preferred choice for many in the crypto community.

Neutron is a blockchain platform that emphasizes its ability to support smart contracts. It is designed to provide a decentralized platform where developers can create and deploy smart contracts for various applications. Neutron's infrastructure is built to ensure that these contracts run efficiently, securely, and without any interference. The platform aims to offer a robust environment for the development and execution of smart contracts, making it a potential choice for businesses and developers looking to leverage blockchain technology for their specific needs.

Stride LP Covenant

Timewave in alliance with Stride, has conceived a system that encompasses the aforementioned blockchains. This system underpins the forthcoming phase of the Cosmos network evolution - interchain collaboration. Timewave is actually developing the Interchain Allocator, bifurcated into two distinct tools: Balancer and Covenant. Specifically for Stride, Timewave has devised the Covenant system.

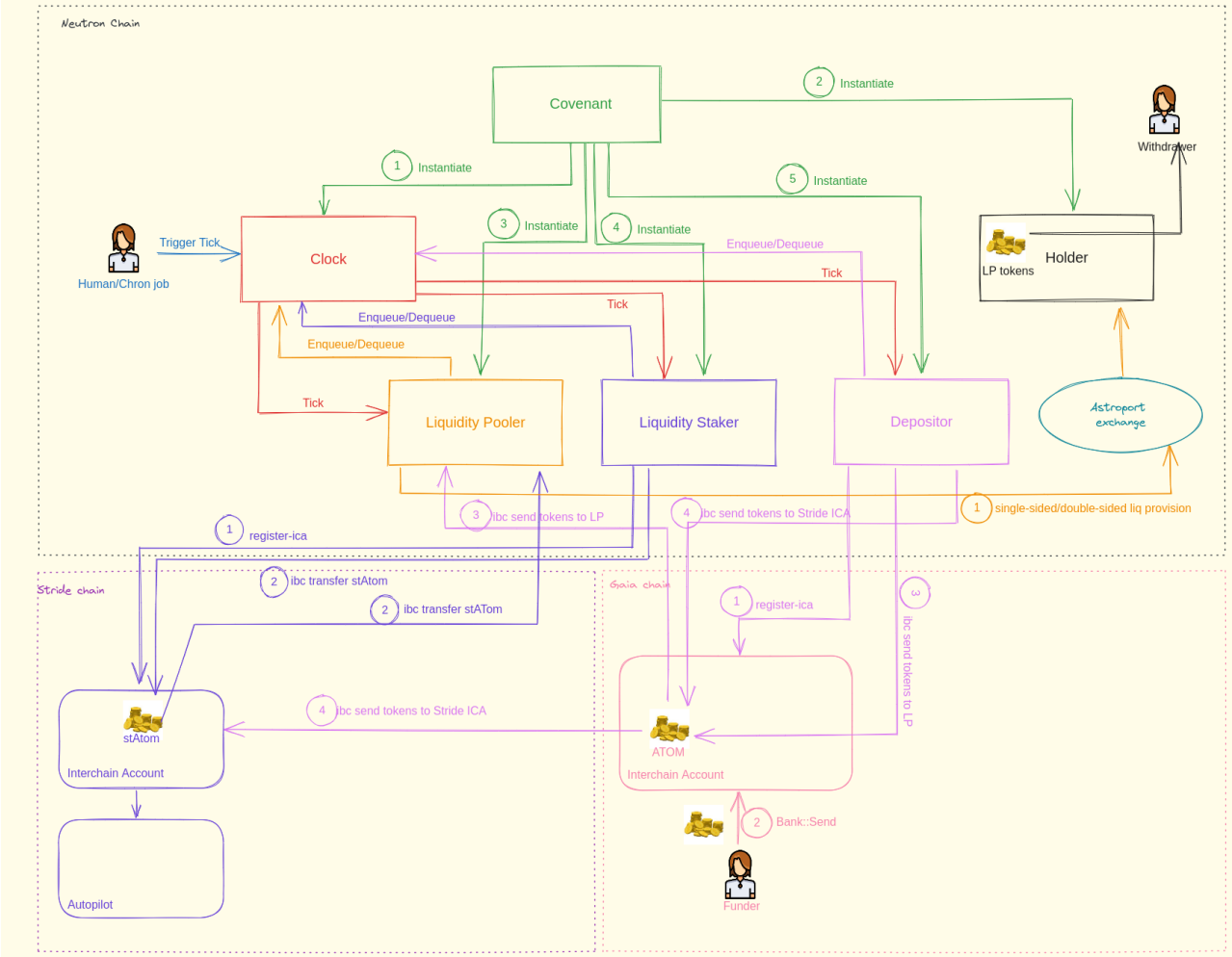
The Covenant is essentially a mechanism engineered to support interchain agreements. The stipulated agreement is delineated as:

1. Receive Atoms from the Cosmos Hub community pool.
2. Partition these into bifurcated segments, with one segment marekd for liquid staking on the Stride blockchain.
3. Utilize Atoms and stAtoms procured from Stride to integrate with the Astroport pool on the Neutron blockchain.

To realize this, Timewave has orchestrated a suite of six interconnected smart contracts that liaise among themselves and with ICA accounts on both Stride and Cosmos Hub, as well as the Astroport pool on Neutron. These contracts create a distinct state machine, wherein each state transition corresponds to a step towards fulfilling the agreement's objectives. The contracts are:

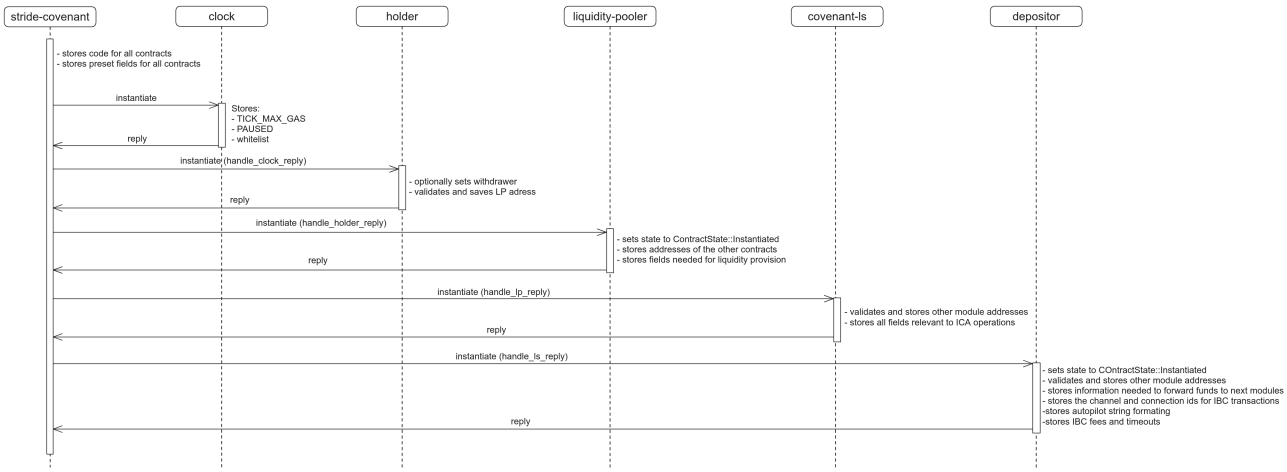
1. Covenant
2. Clock
3. Depositor
4. Liquidity Staker
5. Liquidity Pooler
6. Holder

The subsequent diagram delineates the system's core logic (excluding migration):



Instantiation

Instantiation process for the contracts is actually led by the Covenant smart contract. Once it receives the Instantiate message, it instantiates other contracts. The flow of instantiation is shown in the following picture:



The funds flow

The entire procedure is initiated on the Gaia (Cosmos Hub) chain, which is the genesis of the funds. A quantum of 450,000 Atoms is sent to an Interchain Account, instantiated by the Depositor contract. These Atoms undergo partitioning (currently designed at a 50-50 ratio) and are then dispatched to two distinct destinations via IBC: the Liquidity Pooler contract and the Interchain Account on the Stride chain. The Atoms relayed to Stride undergo liquid staking, recieving stAtoms in return. These stAtoms are subsequently transferred to the Liquidity Pooler contract through IBC. The Liquidity Pooler contract's mandate is to integrate with the Astroport exchange on the Neutron chain using Atom/stAtom liquidity. In reciprocation, LP tokens are procured and relayed to the Holder contract, which functions as a repository accessible only to authorized users.

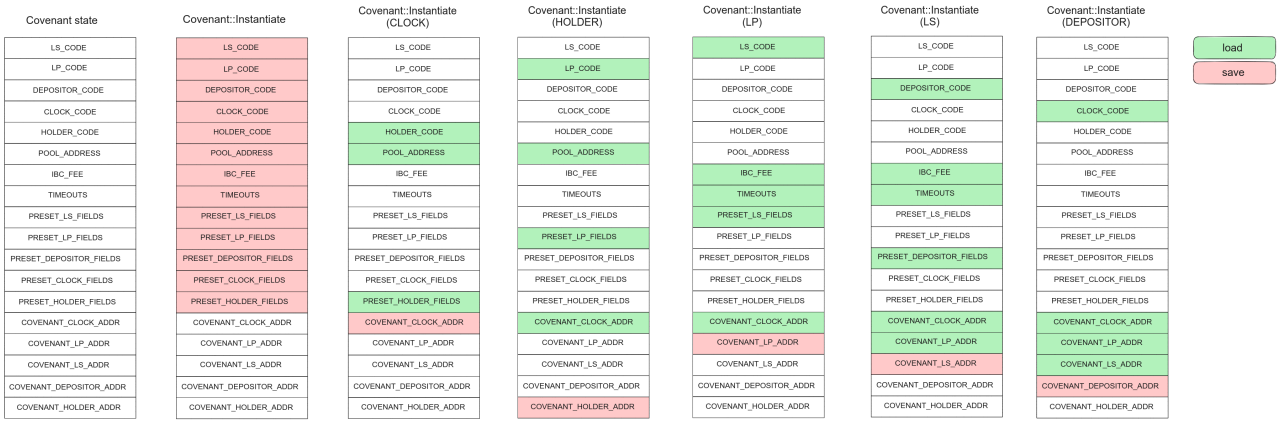
Contracts flow

Covenant

The Covenant contract serves as the system's configurator, responsible for instantiating all ancillary contracts. It sequentially dispatches instantiation messages to specific contracts, awaiting a response before proceeding to the subsequent contract. The instantiation sequence is as follows:

1. Clock
2. Holder
3. Liquidity Pooler
4. Liquidity Staker
5. Depositor

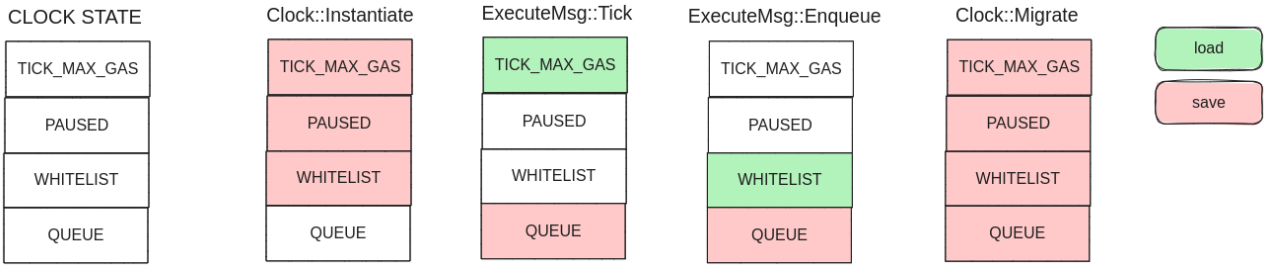
In the following picture state structure and its loading/saving depending on the message being called are shown:



Clock

The Clock contract functions as a state machine advancement apparatus. Its pivotal operation, the `Tick` message, can be dispatched to any of the three contracts: Liquidity Pooler, Liquidity Staker, or Depositor. However, prior to dispatch, these contracts must be whitelisted within the Clock contract and queued for the Tick message. The queuing or dequing is performed using `Enqueue` or `Dequeue` message from the Clock.

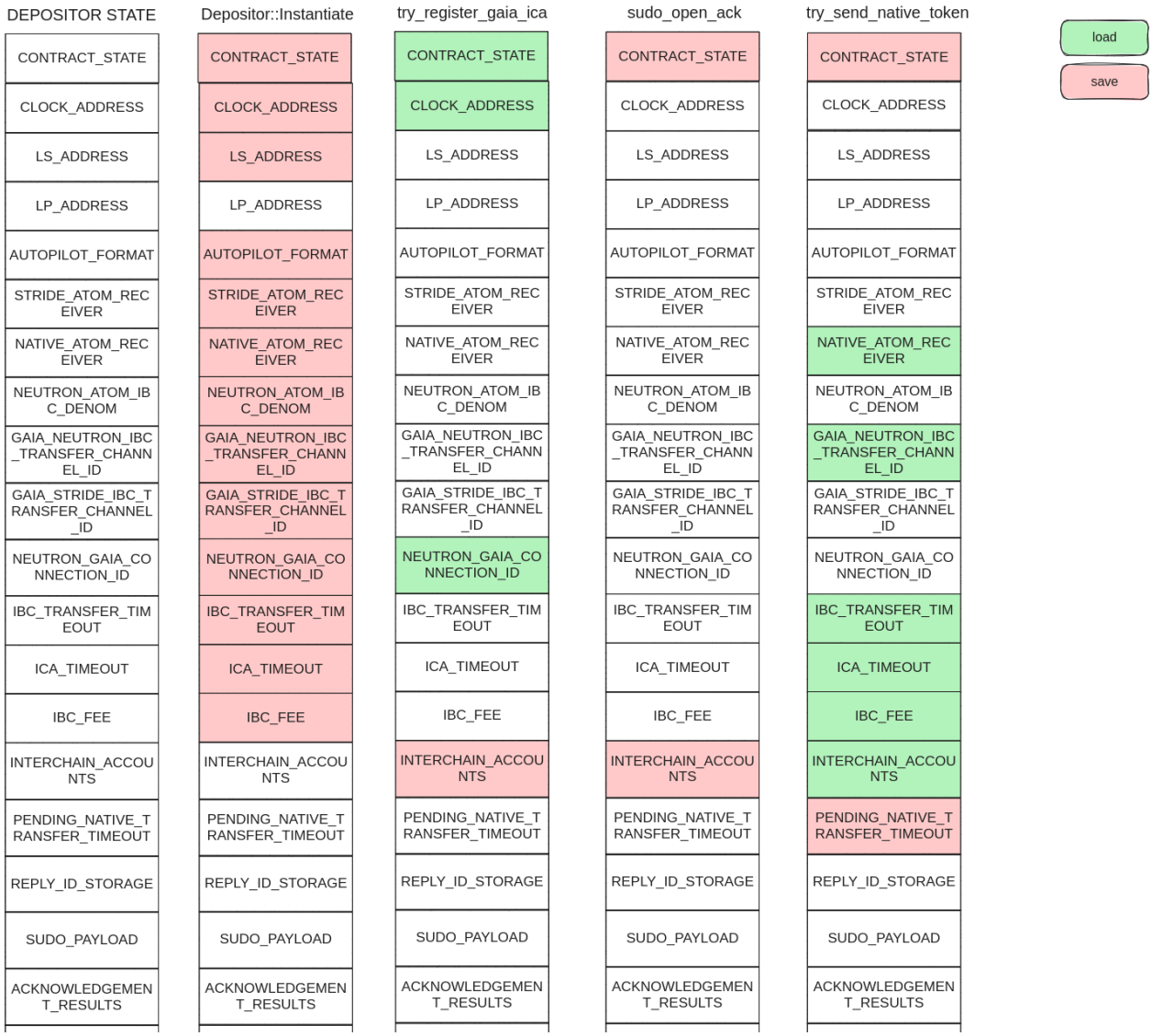
The state changes depending on the contract message are shown in the following picture:

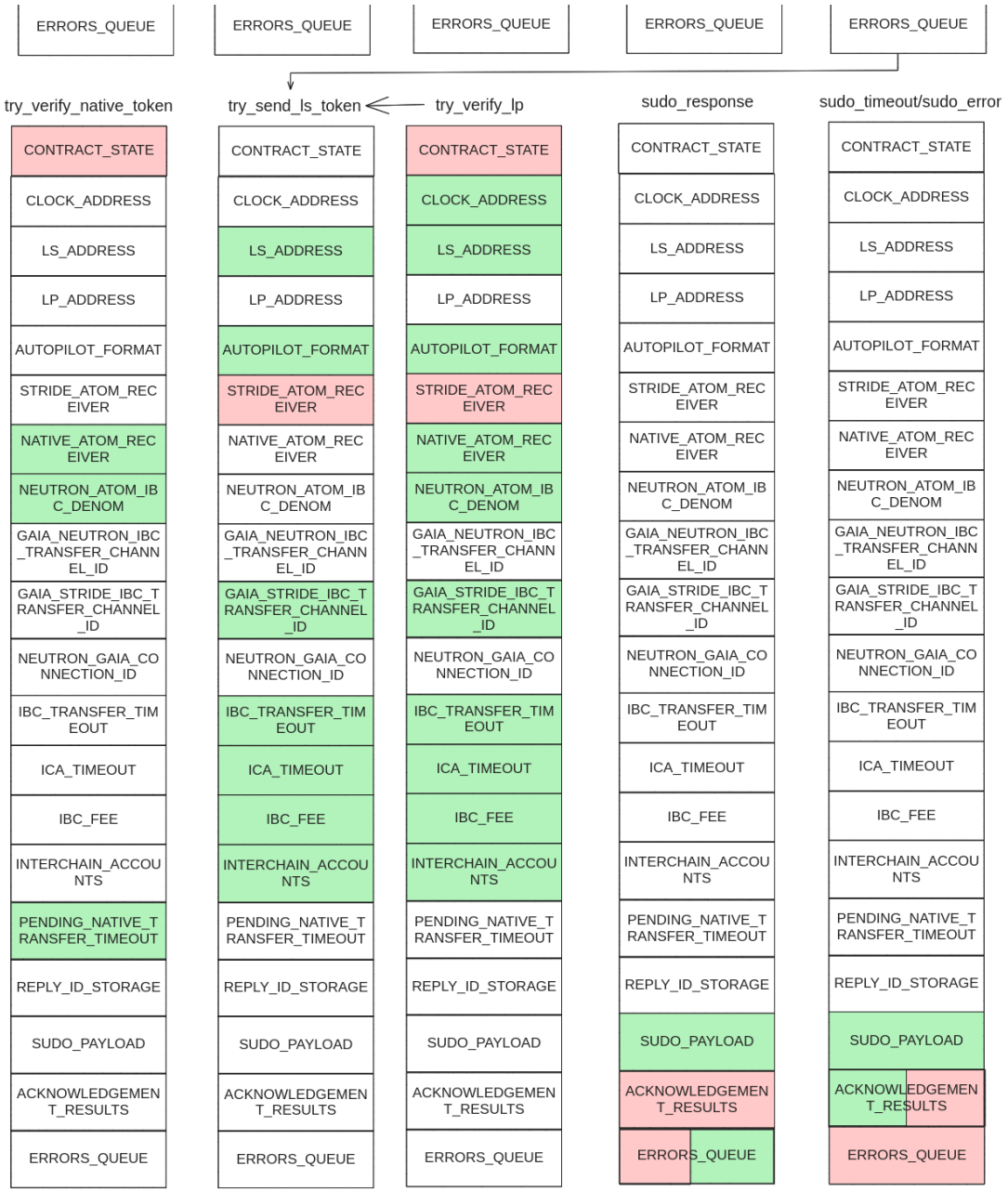


Depositor

The Depositor contract is the first of 4 contracts that interacts with actual funds of this whole agreement. It first creates an Interchain Account on Gaia chain in order to receive funds on it. These are received via `Bank::Send` message from the community pool on Gaia. Now, the Depositor orchestrates the transfer of these funds. However, not all the Atoms are sent to one destination, they are first split in two portions and then:
1. First portion of Atoms is directly sent from Gaia Interchain Account to Liquidity Pooler contract using IBC,
2. Second portion of atoms is sent from Gaia Interchain Account to Stride Interchain Account using IBC where this funds are automatically liquid staked using Stride's Autopilot module.

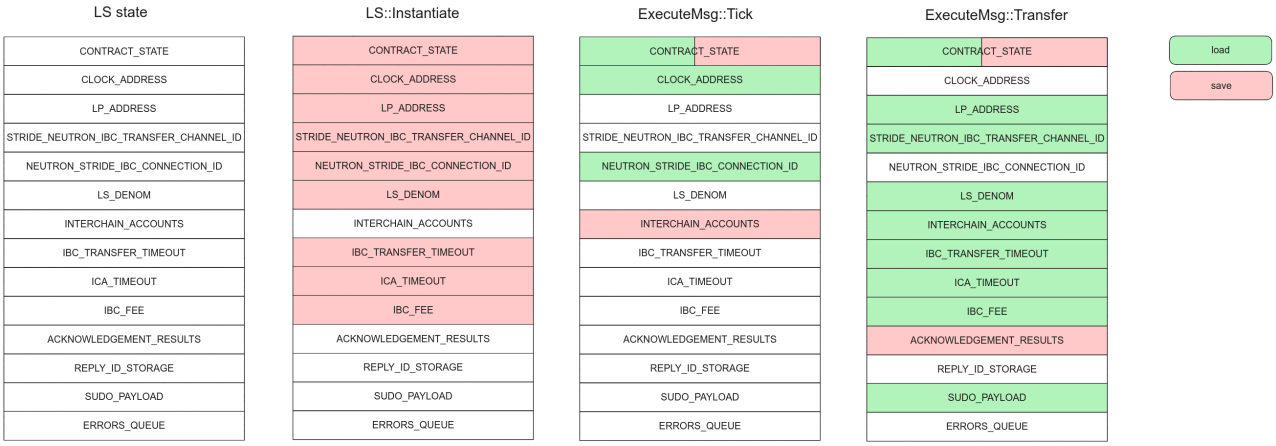
The state changes depending on the message being called from Depositor are shown in the picture:





Liquid Staker

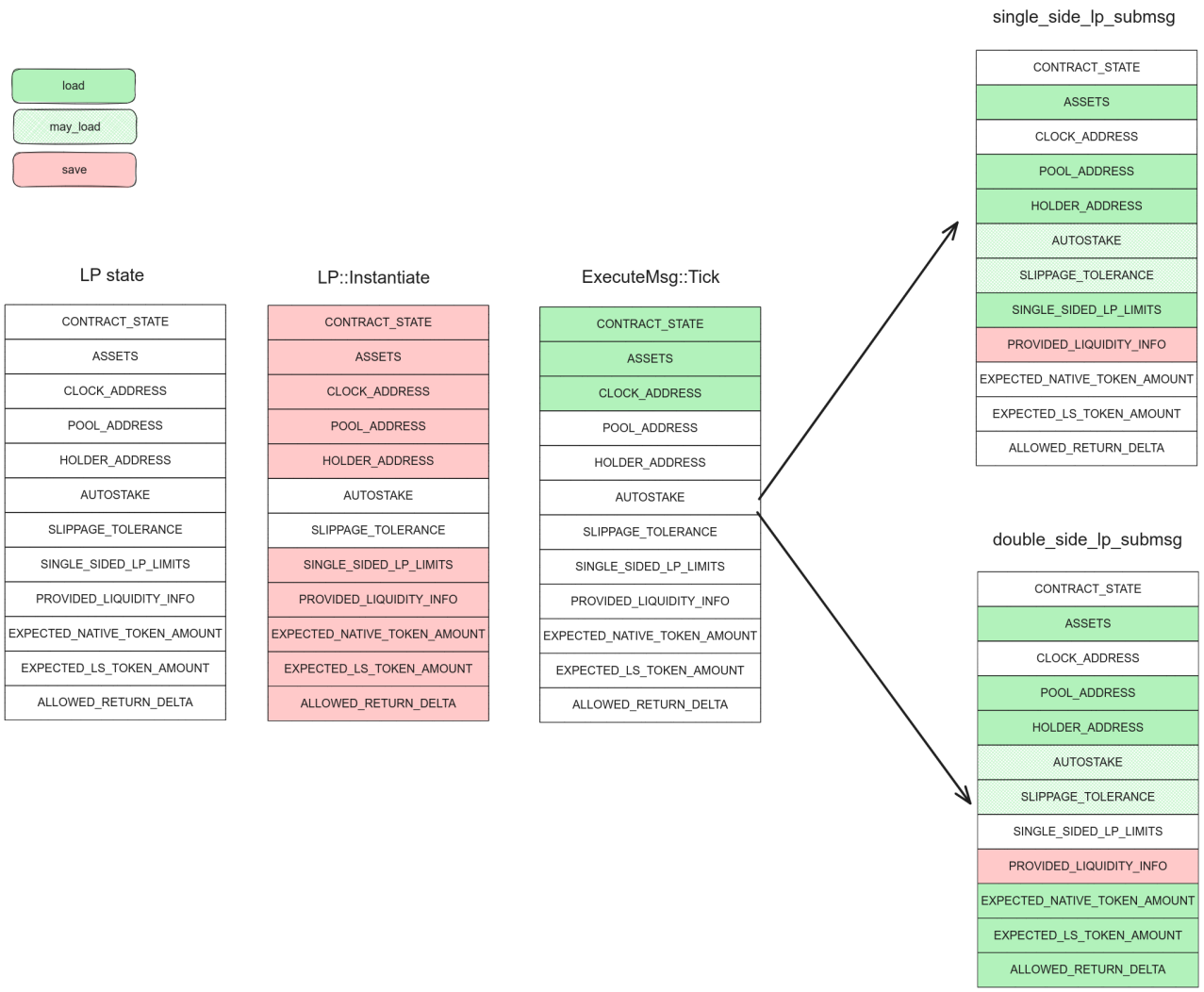
The Liquid Staker contract predominantly liaises with the Stride chain, facilitating the overarching flow. This contract comprises an Interchain Account on the Stride chain, designated for Atom receipt. Post the Atoms' conversion to stAtoms via Autopilot and liquid staking, the Liquid Staker's `Transfer` message facilitates the stAtoms' transfer to the Liquidity Pooler contract using IBC.



Liquidity Pooler

This contract receives Atoms and stAtoms as described in previous phases. These tokens are then used to join Astroport pool on Neutron chain. However, the pool is joined in double sided or single sided manner. The deciding factor is the pool ratio and the amount of each of the tokens in the contract. After the Atoms/stAtoms are liquid provided to pool, the LP tokens are received in exchange and then sent to Holder contract.

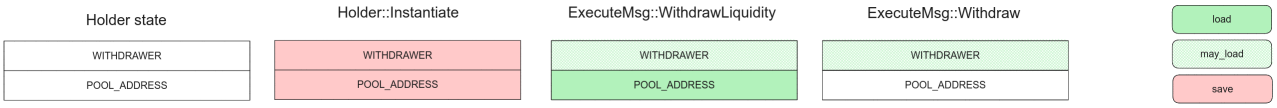
Liquidity Pooler state changes and its loading are shown in the following figure:



Holder

The holder contract acts as an account for the received LP tokens from Astroport. From this account only authorized accounts can withdraw using its `Withdraw` and `WithdrawLiquidity` messages.

Holder state changes and loading from it are shown in the following picture:



Threat Inspection

Threat Model

Threat Model Entry: Unauthorized Fund Redirection

System Component: Stride LP Covenant System

Description: The Stride LP Covenant system operates across multiple environments, including six smart contracts on the Neutron chain, two Interchain Accounts (one on the Cosmos Hub (Gaia chain) and another on the Stride chain), and the Astroport exchange on the Neutron blockchain. The flow of tokens involves multiple transfers, starting from an account on the Cosmos Hub, progressing through various interchain accounts and contracts, and culminating in the Holder account after being invested in the Astroport exchange pool.

Given the intricate nature of these transfers, which involve both IBC and interchain accounts, there's a significant amount of state management and updates required. The system's future goal is to operate autonomously and in a permissionless manner, further complicating the potential attack surface.

Potential Threat: An attacker could exploit vulnerabilities in the transfer logic, state management, or error handling to redirect funds to a malicious account.

Attack Vector:

- Exploiting contract logic during token transfers.
- Manipulating state variables or storage to misroute funds.
- Taking advantage of inadequate error handling or recovery mechanisms.

Threat analysis results

The diagram provided in the System Overview chapter could be used to trace the primary fund flow withing the audited system. Based on the connections shown in the diagram, and the understandings after code inspection performed, the first and most obvious point of attack would be the LS smart contract's exposed transfer message. The message performs the transfer of funds from Interchain Account (ICA) on Stride to LPer smart contract:

```
#[clocked]
#[cw_serde]
pub enum ExecuteMsg {
    /// The transfer message allows anybody to permissionlessly
    /// transfer a specified amount of tokens of the preset ls_denom
    /// from the ICA of the host chain to the preset lp_address
    Transfer { amount: Uint128 },
}
```

However, the message is designed in such a way that the sender is able to provide only the amount of tokens to be sent. The address of the receiver and the denom are fixed. To ensure the safety of the funds being transferred this way, we tried to craft a transaction that would change the contract state, or message parameters in such a way to change the receiver of the funds, but it was not feasible.

Another critical area of focus was the IBC transfers originating from the ICA on Gaia, directed towards Stride and the LPer smart contract. These transfers are triggered by the Depositor contract and are based on messages dispatched from Neutron to the respective blockchains, subsequently initiating IBC token transfers. Direct configuration of these transfers by any Depositor contract message sender is restricted, hence the only way to interfere with these would be changing the state of the contract in a malicious way and this was our goal when trying to exploit the threat here. Specifically, we aimed to modify the stored addresses used for transfers:

```
/// liquid staker module address to query the stride ICA address to autopilot to
```

```
pub const LS_ADDRESS: Item<Addr> = Item::new("ls_address");
/// liquid pooler module address to forward the native tokens to
pub const LP_ADDRESS: Item<Addr> = Item::new("lp_address");
// formatting of stride autopilot message.
/// we use string match & replace with relevant fields to obtain the valid message.
pub const AUTOPILOT_FORMAT: Item<String> = Item::new("autopilot_format");
/// addr and amount of atom to liquid stake on stride
pub const STRIDE_ATOM_RECEIVER: Item<WeightedReceiver> = Item::new("stride_atom_recei
ver");
/// addr and amount of atom
pub const NATIVE_ATOM_RECEIVER: Item<WeightedReceiver> = Item::new("native_atom_recei
ver");
```

The code inspected in the Depositor contract does not leave any room to craft such a state change transaction that would change any of the addresses. Unit tests in the Depositor contract offer a good overview of the functionality and a good place to start with changes to act as a malicious user, but we weren't able to create any harmful scenario there either. While the contract does undergo numerous state changes, our analysis did not identify any exploitable vulnerabilities for fund misappropriation.

The Holder contract and the withdrawal functionality were another point of focus for this particular threat. The state variable of interest in this case was:

```
/// address authorized to withdraw liquidity and the underlying assets
pub const WITHDRAWER: Item<Addr> = Item::new("withdrawer");
```

This privileged entity is authorized to call messages from this contract with the intent to access funds. The address is initialized during the Covenant contract's instantiation, and our analysis did not identify any viable methods to manipulate this address or deceive the contract into unauthorized withdrawals.

It is also a good place to highlight migration function of the contracts. While some could potentially offer an attacker a method to modify the contract's state (e.g., migrate message in Holder contract), CosmWasm's process of migration incorporates mechanisms to ensure security against such threats.

Threat Model Entry: Exploitation of Unprotected Contract Entry Points

System Component: Smart Contracts

Description: The audited system encompasses six distinct smart contracts. Each of these contracts presents various exposed messages, which, by design, can be invoked by any entity in a permissionless fashion. These messages play critical roles, ranging from state modifications, managing funds, to altering contract configurations.

Potential Threat: An attacker could exploit these unprotected entry points to manipulate contract behavior, potentially leading to unauthorized state changes, misappropriation of funds, or altering the contract's operational parameters.

Attack Vector:

- Invoking exposed messages with malicious intent.
- Manipulating state variables to achieve unintended outcomes.
- Altering contract configurations to disrupt normal operations or gain undue advantages.

Threat Analysis Results

For a comprehensive threat assessment, it's imperative to identify the primary accessible entry points within the system, which comprises six distinct smart contracts. The key entry points include:

- `Instantiate` messages
- `Query` messages

- `Tick`, `Queue`, and `Enqueue` messages associated with the Clock smart contract
- `Tick` message from the Depositor smart contract
- `Withdraw` and `WithdrawLiquidity` messages from the Holder smart contract
- `Tick` message from the LPer smart contract
- `Tick` and `Transfer` messages from the LS smart contract

In a hypothetical scenario, `instantiate` messages could be employed to establish the entire system of six contracts. Subsequently, this system could be linked to an arbitrary pool with the intention of triggering fund movements. However, the initiation of this flow is carefully controlled by design. Transfers from the Cosmos Hub to the Interchain Account are contingent upon proposals from participating entities. Given this safeguard, the likelihood of tokens being redirected to an account established by an unverified entity is minimal.

Upon examination of `query` messages, it's evident that they are designed to prohibit any internal state modifications within the contracts. Their functionality appears to be limited to fetching data from the store, making any malicious state changes impossible.

Both `enqueue` and `dequeue` messages, which facilitate interactions with the Clock contract, are fortified through a "whitelisting" mechanism. Within the contract, there exists a predefined list of contracts exclusively permitted to access the ticking queue (i.e., the queue set to trigger state machine advancements).

`Tick` messages embedded within contracts incorporate stringent access control measures. The sole authorized sender for this message type is the Clock contract. This validation process is executed through the `verify_clock` function:

```
/// attempts to advance the state machine. performs `info.sender` validation
fn try_tick(deps: DepsMut, env: Env, info: MessageInfo) ->
NeutronResult<Response<NeutronMsg>> {
    // Verify caller is the clock
    verify_clock(&info.sender, &CLOCK_ADDRESS.load(deps.storage)?)?;

    let current_state = CONTRACT_STATE.load(deps.storage)?;
    match current_state {
        ContractState::Instantiated => try_register_stride_ica(deps, env),
        ContractState::ICACreated => Ok(Response::default()),
    }
}
```

The `withdraw` message associated with the Holder contract, as discussed in the prior threat analysis, also employs access control mechanisms to ensure only authorized entities can initiate withdrawals.

Threat Model Entry: Variable redemption rate

System Component: Liquid Staker Smart Contract and Interchain Account (ICA) on Stride

Description: The Liquid Staker smart contract facilitates the creation of an interchain account on the Stride blockchain. This account acts as a conduit for liquid staking of atoms, resulting in the acquisition of stAtoms. While the quantity of native atoms within the system is known, the exact amount to be obtained from Stride remains uncertain due to the variable redemption rate. To address this unpredictability, the contract includes an exposed message that enables the transfer of available funds from the interchain account on Stride to the subsequent smart contract in the sequence.

Potential Threat: The combination of an indeterminate amount due to the redemption rate and the exposed transfer message poses a significant risk. Malicious actors might exploit this uncertainty and the exposed message to their advantage.

Attack Vector:

- Exploiting the exposed transfer message to redirect funds to unauthorized addresses.
- Triggering the transfer message prematurely, potentially disrupting the intended flow of funds or operations.
- Leveraging the unpredictability of stAtoms quantities to deplete the token pool.

Threat Analysis Results

The exposed transfer message aspect and utilizing it is discussed in the first threat at the beginning of the chapter.

Due to the variable redemption rate, the exact quantity of stAtoms acquired post-liquid staking remains indeterminate. These stAtoms are designated for transfer to the LPer smart contract. To mitigate potential exploitation from this ambiguity, the LPer smart contract's state incorporates the following variable:

```
/// stride redemption rate is variable so we set the expected ls token amount
pub const EXPECTED_LS_TOKEN_AMOUNT: Item<Uint128> = Item::new("expected_ls_token_amou
nt");
```

This amount is later used in the calculations regarding the price range validation:

```
  /// validates the existing pool balances to match our initial expectations.
/// if `PriceRangeError` is returned, it most likely means that the pool had a
/// significant shift in its balance ratio.
fn validate_price_range(
    pool_native_amount: Uint128,
    pool_ls_amount: Uint128,
    expected_native_token_amount: Uint128,
    expected_ls_token_amount: Uint128,
    allowed_return_delta: Uint128,
) -> Result<(), ContractError> {
    // find the min and max return amounts allowed by deviating away from expected
return amount
    // by allowed delta
    let min_return_amount =
expected_ls_token_amount.checked_sub(allowed_return_delta)?;
    let max_return_amount =
expected_ls_token_amount.checked_add(allowed_return_delta)?;

    // derive allowed proportions
    let min_accepted_ratio: Decimal = Decimal::from_ratio(min_return_amount,
expected_native_token_amount);
    let max_accepted_ratio = Decimal::from_ratio(max_return_amount,
expected_native_token_amount);

    // we find the proportion of the price range being validated
    let validation_ratio = Decimal::from_ratio(pool_ls_amount, pool_native_amount);

    // if current return to offer amount ratio falls out of [min_accepted_ratio,
max_return_amount],
    // return price range error
```

```
    if validation_ratio < min_accepted_ratio || validation_ratio > max_accepted_ratio
{
        return Err(ContractError::PriceRangeError {});
    }

    Ok(())
}
```

By adopting this methodology, the LPer is safeguarded against an influx of either numerous minor transactions or excessively large stAtoms quantities. Such transactions, given the LPer's logic for pool participation, could adversely impact the pool's balance ratio and associated prices. Nonetheless, the computed interval, within which the stAtoms quantity is anticipated to reside, effectively prevents these scenarios.

During the threat exploit analysis, a low-severity finding was identified. This has been documented in the "Findings" section of this report.

The end-to-end testing environment developed in the solution was also utilized in this threat assessment. We tried to change the amounts (as a consequence of redemption rate variation) that would be sent to LPer contract (e.g., setting amounts to interval extremes, out-of-bounds values, etc.). In all instances, the system exhibited consistent behavior, ensuring the overall system's integrity remained uncompromised.

Threat Model Entry: Risk of Unfavorable Liquid Provisioning Due to Variable Pool Balances

System Component: LPer Smart Contract and Astroport DEX Pool

Description: The LPer smart contract is responsible for forwarding funds to the Astroport DEX pool. Given the pool's fluctuating prices, there exists a possibility that liquid provisioning might occur at a disadvantageous price. Additionally, the pool incorporates a mechanism that allows for a single sided provision. This mechanism, if exploited, could adversely affect the pool's price by allowing the provision of only one type of token to the pool.

Potential Threat: The dynamic nature of pool balances combined with the single sided provision mechanism presents a dual threat. On one hand, users might end up liquid provisioning at suboptimal prices, and on the other, malicious actors could manipulate the pool's price by leveraging the single sided provision.

Attack Vector:

- Exploiting the variable pool balances to liquid provide at unfavorable prices, leading to potential losses for users.
- Utilizing the single token provision mechanism to inject only one type of token, thereby manipulating the pool's price to the detriment of other participants.
- Intentionally changing the pool ratio to force liquid provisioning outside the desired range, potentially causing disruptions in the pool's operations.

Threat analysis results

The LPer smart contract operates as a continuous monitor of accessible balances and serves as a conduit to the Astroport pool. A thorough code review was conducted, and the components regarding this threat have been delineated.

For ensuring liquidity provisioning occurs at an optimal price, the pool ratio is mandated to remain within specified parameters. The LPer smart contract's state encompasses:

```
/// the native token amount we expect to receive from depositor
pub const EXPECTED_NATIVE_TOKEN_AMOUNT: Item<Uint128> = Item::new("expected_native_to
ken_amount");
```

The aforementioned state amount is employed in the previously discussed `validate_price_range` function. The pool's ratio undergoes validation against this expected ratio. Should the validation be unsuccessful, provisioning is halted. If both the ratio and available balances allow a dual-sided liquidity provision, the LPer contract joins the pool. An additional state variable

```
/// slippage tolerance parameter for liquidity provisioning
pub const SLIPPAGE_TOLERANCE: Item<Decimal> = Item::new("slippage_tolerance");
```

plays role in ensuring pool prices remain within acceptable deviations.

An alternative provisioning approach is termed as "single-sided provision." Here, only a singular token is available within the LPer contract, resulting in the pool acquiring tokens from a singular source. To maintain pool stability, two variables are stored:

```
/// amounts of native and ls tokens we consider ok to single-side lp
pub const SINGLE_SIDED_LP_LIMITS: Item<SingleSideLpLimits> = Item::new("single_side_l
p_limit");
/// keeps track of ls and native token amounts we provided to the pool
pub const PROVIDED_LIQUIDITY_INFO: Item<ProvidedLiquidityInfo> =
    Item::new("provided_liquidity_info");
```

The key information used to prevent price deviations is the slippage tolerance.

Unit tests were utilized to analyze if different values could affect LPer behaviour in a way that would damage the pool, but the contract remains to behave in the expected manner.

Threat Model Entry: Risk of ICA/IBC Packet Timeouts Due to Unreliable Relayers

System Component: Interchain Account (ICA) and Relayers

Description: The ICA/IBC relies on relayers for the transmission of packets. However, the reliability of these relayers can be inconsistent. If an ICA packet times out, it results in the closure of the ICA channel, given that it operates on an ordered channel system. This can disrupt the intended flow of operations and potentially lead to funds not reaching their intended destinations. IBC can sometimes fail to complete. This can be attributed to various reasons, including relayer failures. Incomplete transfers can result in funds not reaching their intended destinations, leading to potential losses or misallocations.

Potential Threat: Unreliable relayers can cause ICA/IBC packet timeouts, leading to the unintended closure of the channel and potential misdirection or loss of funds.

Attack Vector:

- Malicious or faulty relayers causing intentional ICA packet delays or timeouts.
- External disruptions or network issues leading to unintentional ICA packet timeouts.

Threat analysis results

The code was thoroughly analyzed to determine that the process of transferring via Interchain Accounts (ICA) occurs in the following places:

- Transfer of Atom tokens from ICA on Gaia to LPer smart contract
- Transfer of Atom tokens from ICA on Gaia to ICA on Stride
- Tranfer of stAtom tokens from ICA on Stride to LPer smart contract

The code of interest for this threat is pretty similar in all three cases:

```
let msg_transfer_timeout = env
```

```
                .block
                .time
                // we take the wrapping ICA tx timeout into account and assume the
worst
                .plus_seconds(ica_timeout.u64())
                // and then add the preset ibc transfer timeout
                .plus_seconds(ibc_transfer_timeout.u64());

            // we store that timeout for later validation of pending transfers
            PENDING_NATIVE_TRANSFER_TIMEOUT.save(deps.storage,
&msg_transfer_timeout)?;

            // transfer message that will send funds from the ICA on gaia to our LP
module
            let lper_msg = MsgTransfer {
                source_port: "transfer".to_string(),
                source_channel,
                token: Some(coin),
                sender: address,
                receiver: receiver.address,
                timeout_height: None,
                timeout_timestamp: msg_transfer_timeout.nanos(),
            };

            let lp_protobuf = to_proto_msg_transfer(lper_msg)?;

            // tx to our ICA that wraps the transfer message defined above
            let submit_msg = NeutronMsg::submit_tx(
                controller_conn_id,
                INTERCHAIN_ACCOUNT_ID.to_string(),
                vec![lp_protobuf],
                "".to_string(),
                ica_timeout.u64(),
                fee,
            );
```

The timeout component of the transfer is comprised of added ICA timeout, IBC timeout and block timestamp. This way, the code ensures that the packets are definitely not reaching their destination if the timeout expired. The recovery path is the subject to a latter threat, but the timeout set this way is shown to be correct after our analysis and testing.

The end-to-end environment utilized for this threat analysis showed us that if the timeout expires, not additional changes to the balances of accounts are made.

The recovery paths for expired timeouts are also examined. They are constructed on the basis of contract state changes. Namely, it means that if a transfer fails, a state in which the contract is should ensure a proper retry. Timeout checks and balance queries are also implemented to secure these paths.

While reviewing this threat, a low-severity finding was identified regarding the ICA registration process. It is more connected to the possible errors in contracts data store, but could lead to a contract locked state and delayed transfers.

Threat Model Entry: Risk of Double Sending Funds Due to Depositor Retry Logic

System Component: Depositor Smart Contract and IBC Transfers

Description: The depositor smart contract design presents a potential vulnerability where funds could be sent twice to the LPer. This risk arises from the inherent uncertainties in IBC transfers, which can either fail or experience significant delays. Before initiating a retry to send tokens, it's crucial not only to verify the LPer's balance increment but also to ascertain that no pending IBC packets might still be received.

Potential Threat: The possibility of inadvertently sending funds twice to the LPer due to retry logic, especially in scenarios where IBC transfers are delayed or fail.

Attack Vector:

- Unintentional double sends due to IBC transfer delays or failures.
- Malicious actors exploiting the retry logic to trigger double sends.
- External disruptions causing IBC packets to be delayed beyond expected timeframes.

Threat analysis results

This threat is linked to the one previously discussed. Timeouts are strategically employed to ensure that a transfer isn't redundantly initiated while another transfer is still in progress. The focal code segments for this threat evaluation include balance verifications and timeout validations:

```
if lper_native_token_balance.amount >= receiver.amount {
        // if funds have arrived on LP module, we advance the state and attempt to
        // send the remaining funds to ICA on stride
        CONTRACT_STATE.save(deps.storage, &ContractState::VerifyLp)?;
        let ls_token_msg = try_send_ls_token(env, deps)?;

        return Ok(Response::default()
            .add_submessage(ls_token_msg)
            .add_attribute("method", "try_verify_native_token")
            .add_attribute("receiver_balance", lper_native_token_balance.amount));
    } else if env.block.time.nanos() >= pending_transfer_timeout.plus_minutes(5).nano
s() {
        // funds are still not on the LP module and the msgTransfer timeout is due
        // we can safely retry sending the funds again by reverting the state
        // to ICACreated
        CONTRACT_STATE.save(deps.storage, &ContractState::ICACreated)?;
        return Ok(Response::default()
            .add_attribute("method", "try_verify_native_token")
            .add_attribute("status", "pending_transfer_timeout_due")
            .add_attribute("contract_state", "ica_created")
        );
    }
```

Our objective was to potentially bypass the aforementioned logic, either by artificially augmenting the LP balance from an external source or by manipulating timeout values. However, given the absence of a tangible incentive for an attacker, artificially inflating the LP balance was ultimately deemed non-threatening. This is because the Stride funds would merely be expedited to the LP, and the Atoms would still be destined for the pool. This scenario was discussed with the Timewave team during a sync meeting, where it was recognized as possible but concurrently acknowledged as lacking sufficient motivation for a potential adversary.

Threat Model Entry: Risk of Misconfigured Smart Contract Instantiation

System Component: Smart Contract Configuration

Description: Smart contracts, when not correctly instantiated, can lead to a number of issues including state locks, logic discrepancies, and trapped funds. Such misconfigurations can disrupt the intended flow of operations, potentially resulting in financial losses or system malfunctions.

Potential Threat: Improperly configured smart contract instantiation poses a risk of operational disruptions, logic errors, and funds becoming inaccessible.

Attack Vector:

- Inadvertent errors during the instantiation process leading to faulty contract configurations.
- Malicious actors intentionally misconfiguring contracts to exploit vulnerabilities or trap funds.
- External factors or dependencies causing unforeseen configuration issues.

Threat analysis results

In one of the previously described threats *"Threat Model Entry: Exploitation of Unprotected Contract Entry Points"* a highly unlikely scenario of someone instantiating all 6 contracts to create this system is presented, but this was not identified as a exploitable threat due to lack of motivation for potential attackers to create such a scenario. This comes from the fact that a funds flow should start after kind of consensus between different entities to send funds to ICA on Gaia.

To further analyze the potential risks regarding this threat, our main focus were the contract states, their instantiation and variable definitions, and their possible malicious use.

Unit tests and end-to-end testing environment were used to try and misconfigure a contract after its instantiation, but no possible paths to this were found. The migration logic as the only available way to directly change the state does not fall into the consideration since it is guarded by CosmWasm's special treatment for contract migration.

While assessing the threat, we identified two findings connected to the contract states:
State loading optimization
Memory reduction

These findings are actually possible optimizations in current codebase.

It should also be noted that the contract state changes are implemented with minimalist approach and we were not able to abuse any of those. This was also mentioned in previous threats, with one finding regarding state transitions, but the overall impression is that the transitions are kept at a reasonable level.

Other Aspects to Emphasize and Exercise Caution With

One finding is documented where all the typing mistakes, recommendations and small errors are detected during our code inspection.

Findings

Title	Type	Severity
Prevent division by zero	IMPLEMENTATION	1 LOW
Error handling should include state transition	IMPLEMENTATION	1 LOW
Minor code changes and recommendations	IMPLEMENTATION	0 INFORMATIONAL
State loading optimization	IMPLEMENTATION	0 INFORMATIONAL
Memory reduction	IMPLEMENTATION	0 INFORMATIONAL

Prevent division by zero

Title	Prevent division by zero
Project	Timewave: Stride Covenant
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Issue	

Involved artifacts

- covenants/contracts/lper/src/contract.rs

Description

In the LPer contract, the `validate_price_range` function is created to validate amounts received from the LS and Depositor contracts against expected amounts. Subsequently, these amounts are compared to pool balances. If the provided amounts do not adhere to the required ratio, a `PriceRangeError` is returned.

A potential division by zero vulnerability has been identified within the calculations of the aforementioned function. Specifically, the denominator in the calculation is `expected_native_token_amount`, sourced from the contract's state. This state variable is initialized during contract instantiation via message parameters.

Problem Scenarios

While a panic due to division by zero is an expected behavior, the root cause here could be inadvertent misconfigurations. If `expected_native_token_amount` is inadvertently (or maliciously) initialized to zero, it would trigger a panic. Although panicking on division by zero is a standard response, preemptive input validation can mitigate unnecessary computations, tick executions, and gas consumption. Given that this value is initialized, the division by zero would occur post several other operations on different contracts, such as interchain account creations and IBC transfers. Early detection and prevention of this scenario are feasible and recommended.

Using one of the unit tests the panic is confirmed:

```
test suite_test::tests::test_validate_price_range_out_of_bounds - should panic ... FAILED

failures:

---- suite_test::tests::test_validate_price_range_out_of_bounds stdout ----
note: panic did not contain expected string
      panic message: `"Denominator must not be zero"`,
 expected substring: `"Price range error"`

failures:
    suite_test::tests::test_validate_price_range_out_of_bounds

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 3 filtered out; finished in 9.71s
```

Recommendation

It is advised to introduce input validation during the instantiation process of the LPer contract. If `expected_native_token_amount` is initialized to zero, the contract execution should be halted immediately. Furthermore, the individual or entity configuring the contract should be promptly notified about the erroneous values to ensure corrective action.

Error handling should include state transition

Title	Error handling should include state transition
Project	Timewave: Stride Covenant
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Issue	

Involved artifacts

- covenants/contracts/depositor/src/contract.rs

Description

Upon the creation of an interchain account on the Cosmos Hub, the state of the `Depositor` contract transitions to `ContractState::ICACreated`. The subsequent execution logic post this state transition is defined as:

```
ContractState::ICACreated => {
    let ica_address = get_ica(deps.as_ref(), &env, INTERCHAIN_ACCOUNT_ID);
    match ica_address {
      Ok((_, _)) => {
        try_send_native_token(env, deps)
      },
      Err(_) => Ok(Response::default()
        .add_attribute("method", "try_tick")
        .add_attribute("ica_status", "not_created")
    ),
  }
}
```

The failure in this execution flow can be attributed to the error branch in the `get_ica(...)` function. This function's primary role is to formulate a key and retrieve the interchain account from the contract's store.

Problem Scenarios

Although the likelihood of encountering this error is relatively low, potential scenarios such as I/O interruptions or data corruption within the datastore are possible. In the event of such an error, the contract remains stagnated in

its preceding state, specifically `ContractState::ICACreated`. There is no evident mechanism to reinitiate the state transition or advance the state machine.

If the Depositor contract is locked in this state, it is trying to load interchain account on every tick. If the error is of such nature that it cannot be cleared, the loading is not going to provide any results, the whole system of 6 contracts is stuck here.

Recommendation

Given that the codebase acknowledges the possibility of this error and has made provisions for its occurrence, it is advisable to ensure robust error handling. This is to prevent the contract from becoming trapped in an immutable state. Proper error recovery or state rollback mechanisms should be integrated to circumvent scenarios where the contract state becomes irrevocably locked.

State loading optimization

Title	State loading optimization
Project	Timewave: Stride Covenant
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- contracts/covenant/src/contract.rs
- contracts/depositor/src/contract.rs
- contracts/lper/src/contract.rs
- contracts/ls/src/contract.rs

Description

Upon a detailed examination of the solution, it's evident that there are multiple instances where state variables are individually loaded from storage. This approach, while functional, can lead to inefficiencies, especially when dealing with a large number of state variables. Specific instances include:

- Covenant contract's execute message:
  `let ibc_transfer_timeout = IBC_TRANSFER_TIMEOUT.load(deps.storage)?;`
  `let ica_timeout = ICA_TIMEOUT.load(deps.storage)?;`
  `let source_channel = GAIA_NEUTRON_IBC_TRANSFER_CHANNEL_ID.load(deps.storage)?;`
  `et receiver = NATIVE_ATOM_RECEIVER.load(deps.storage)?;`
  `let fee = IBC_FEE.load(deps.storage)?;`
- In the Depositor contract :
  `let gaia_stride_channel = GAIA_STRIDE_IBC_TRANSFER_CHANNEL_ID.load(deps.storage)?;`
  `let ibc_transfer_timeout = IBC_TRANSFER_TIMEOUT.load(deps.storage)?;`
  `let ica_timeout = ICA_TIMEOUT.load(deps.storage)?;`
  `let fee = IBC_FEE.load(deps.storage)?;`
  and
  `let ibc_transfer_timeout = IBC_TRANSFER_TIMEOUT.load(deps.storage)?;`

```
let ica_timeout = ICA_TIMEOUT.load(deps.storage)?;
let source_channel =
GAIA_NEUTRON_IBC_TRANSFER_CHANNEL_ID.load(deps.storage)?;
let receiver = NATIVE_ATOM_RECEIVER.load(deps.storage)?;
let fee = IBC_FEE.load(deps.storage)?;
```

- In the LPer contract:

```
let pool_address = POOL_ADDRESS.load(deps.storage)?;
let slippage_tolerance = SLIPPAGE_TOLERANCE.may_load(deps.storage)?;
let auto_stake = AUTOSTAKE.may_load(deps.storage)?;
let asset_data = ASSETS.load(deps.storage)?;
let holder_address = HOLDER_ADDRESS.load(deps.storage)?;
let expected_ls_token_amount =
EXPECTED_LS_TOKEN_AMOUNT.load(deps.storage)?;
let expected_native_token_amount =
EXPECTED_NATIVE_TOKEN_AMOUNT.load(deps.storage)?;
let allowed_return_delta = ALLOWED_RETURN_DELTA.load(deps.storage)?;
```

and

```
let pool_address = POOL_ADDRESS.load(deps.storage)?;
let slippage_tolerance = SLIPPAGE_TOLERANCE.may_load(deps.storage)?;
let auto_stake = AUTOSTAKE.may_load(deps.storage)?;
let asset_data = ASSETS.load(deps.storage)?;
let single_side_lp_limits = SINGLE_SIDED_LP_LIMITS.load(deps.storage)?;
let holder_address = HOLDER_ADDRESS.load(deps.storage)?;
```

- In the LS contract:

```
let fee = IBC_FEE.load(deps.storage)?;
let source_channel =
STRIDE_NEUTRON_IBC_TRANSFER_CHANNEL_ID.load(deps.storage)?;
let lp_receiver = LP_ADDRESS.load(deps.storage)?;
let denom = LS_DENOM.load(deps.storage)?;
let ibc_transfer_timeout = IBC_TRANSFER_TIMEOUT.load(deps.storage)?;
let ica_timeout = ICA_TIMEOUT.load(deps.storage)?;
```

Recommendation

To optimize the efficiency of state variable loading, it's advisable to consolidate related state variables into structured data types. This would allow for a single storage load operation to retrieve multiple related state variables, thereby reducing the number of individual storage accesses.

For instance, in the Depositor contract, there's a definition:

```
/// addr and amount of atom to liquid stake on stride
pub const STRIDE_ATOM_RECEIVER: Item<WeightedReceiver> =
Item::new("stride_atom_receiver");
```

Here, the `WeightedReceiver` struct is utilized. A similar approach can be adopted by defining a struct, say

`IBCParameters`, which could encapsulate all related IBC data such as channels, connections, and fees. This would allow for a single `Item` to store all the IBC-related state variables.

Memory reduction

Title	Memory reduction
Project	Timewave: Stride Covenant
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- covenants/contracts/covenant/src/state.rs
- covenants/contracts/covenant/src/error.rs
- covenants/contracts/depositor/src/error.rs

Description

Optimization of State Variables in the Depositor Contract

Upon review of the `Depositor` contract, it has been observed that the state variable `LP_ADDRESS`, defined as:

```
// liquid pooler module address to forward the native tokens to
pub const LP_ADDRESS: Item<Addr> = Item::new("lp_address");
```

is not utilized within the contract's operations. To enhance the contract's efficiency and reduce unnecessary state storage, it is recommended to remove this redundant state variable.

Benefits:

- Reduction in the contract's storage footprint.
- Streamlined state management leading to easier code maintainability.

Refinement of Error Definitions Across Contracts

A comprehensive analysis of the contract reveals that several error definitions (depositor, covenant) are not invoked or referenced within the contract's logic. To optimize memory usage and improve contract clarity, it is advisable to eliminate these error definitions.

Benefits:

- Enhanced memory efficiency by reducing the contract's overhead.
- Improved code readability and maintainability by eliminating unused components.

Minor code changes and recommendations

Title	Minor code changes and recommendations
Project	Timewave: Stride Covenant
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	

Involved artifacts

- contracts/clock/state.rs
- contracts/clock/contract.rs
- contracts/covenant/contract.rs
- contracts/lper/msg.rs

Description

1. The state variable TICK_MAX_GAS could be renamed to TICK_GAS_LIMIT in order to make it more readable. This is especially useful in cases where new constants regarding this limit are defined, and later used: MIN_TICK_MAX_GAS, MAX_TICK_MAX_GAS would be easier to read if they were renamed to MIN_TICK_GAS_LIMIT, MAX_TICK_GAS_LIMIT
2. In documentation and code comments, be consistent when referring to one of the 6 contracts in terms of whether you call it contract or module. It'c causing confusion with this mixed terms. Since modules are present in CosmosSDK for example or generally speaking more suitable for some features of a chain, these are proper smart contracts, let's call them that way.
3. Wrong comment here. It should say "Load the fields relevant to LP instantiation"
4. Wrong comment here. It should say "only whitelisted contracts can be enqueued"
5. Wrong comment here. It should note that channel id is wrong, not sequence id.
6. Wrong attribute value here. It should be "handle_ls_reply".
7. Wrong comment here. It should be "…[min_accepted_ratio, max_accepted_ratio]".
8. Large size difference between variants here. The size of an enum in memory is determined by the size of its largest variant plus the size of the tag (which indicates which variant it is). In the provided code, the `MigrateMsg` enum has two variants: `UpdateConfig` and `UpdateCodeId`, and there's a significant size difference between these two. When you create an instance of an enum, Rust allocates enough memory to store the largest variant. This means that even if you're using the smaller `UpdateCodeId` variant, you're still using up the memory required for the larger `UpdateConfig` variant. This can lead to inefficient memory usage, especially if the smaller variants are used more frequently than the larger ones. This enum should be refactored. Consider breaking the enum into smaller enums or structs if the variants serve very different purposes, or you can use Box for larger variants, which will store the data on the heap and the enum will only contain a pointer to the heap, making the enum's size consistent across variants.

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:

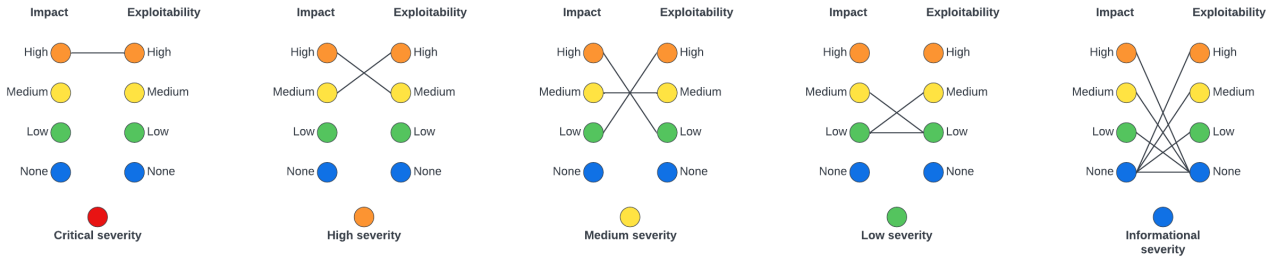
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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

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

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

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

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

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