Security Audit Report

Archway 2024 Q2

Authors: Darko Deuric, Josef Widder

Last revised 16 May, 2024

Table of Contents

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang, Cosmos SDK
- **Artifacts:**
  - x/cwerrors
  - x/cwica

Engagement Summary

- **Dates**: 15 Apr 2024 to 10 May 2024
- **Method**: Manual code review, protocol analysis
- **Employees Engaged**: 2

Severity Summary

Finding Severity	**#**
Critical	0
High	1
Medium	1
Low	1
Informational	4
Total	**7**

System overview

This system integrates two audited modules, `x/cwica` and `x/cwerrors`, designed to facilitate efficient interchain account (ICA) transactions and error management within CosmWasm smart contracts.

`x/cwica` Module

The `x/cwica` module simplifies the process of managing ICA functionality for contracts, acting as a wrapper around the built-in Interchain Accounts (ICA) protocol (ICS-27). Key features include:

1. **RegisterInterchainAccount:** A contract address can register an ICA, with each contract limited to one ICA per IBC connection.
2. **SendTx:** Facilitates submitting transactions for execution on a counterparty chain, with configurable message limitations.
3. **UpdateParams:** Allows fine-tuning of parameters, such as setting maximum messages allowed in `SendTx`.

The module also provides three IBC handlers:

1. **HandleChanOpenAck:** Handles the acknowledgment of a successfully opened IBC channel.
2. **HandleAcknowledgement:** Communicates transaction outcomes to the originating contract or stores errors in `x/cwerrors`.
3. **HandleTimeout:** Manages transaction timeouts.

This module empowers contracts to directly control interchain accounts, simplifying cross-chain management via IBC packets.

`x/cwerrors` Module

The `x/cwerrors` module manages error handling, providing a consistent mechanism for contracts to identify and respond to protocol errors. It offers two handling modes:

1. **Pull Mechanism:** Errors are stored for a configurable period and can be queried during this period.
2. **Subscription Mode:** Contracts can subscribe to immediate error callbacks for a fee, preventing errors from being stored. Subscriptions can be extended or pruned as needed.

The module's subscription system ensures real-time alerts. Errors that occur in callbacks are captured to avoid propagation, while queries are accessible through whitelisted endpoints.

By combining these modules, contracts can manage ICAs efficiently while maintaining robust error detection and handling.

Threat model

The threat report consists of invariants and liveness properties, and it also examines potential threats and edge cases for x/cwerrors and x/cwica modules.

x/cwerrors

The x/cwerrors module is designed as a central point for managing error-related data flow from protocol modules (such as x/cwica or x/callback) to smart contracts.

Invariants

1. **In the x/cwerrors module, a callback to a contract only occurs if an error is present.** ✅
   - The `SetError` function is triggered only under three specific conditions:
     - Failure during callback execution,
     - Packet timeout,
     - Acknowledgment failure of a packet.
   - Access to `SetError` is restricted to only two modules, cwica and callback, via `expected_keepers.go`.
   - Additionally, if there's a callback failure when executing a `sudoError` message, `StoreErrorInState` is called within the x/cwerrors module.
2. **For every error reported to the cwerror module, there will be at most one callback.** ✅
   - This is ensured by assigning a unique error ID to each error, applicable for both storage and transient storage.
3. **Each error handled by the cwerror module will either trigger a callback or be stored, depending on the situation**: ✅
   - If the contract is currently subscribed to errors and an error is detected in the same block, the contract will be immediately notified through the transient store.
   - If the contract is not subscribed to errors, the error will be stored in the regular key-value (kv) store. The contract can then perform a Stargate query to check for errors at its convenience.
   - If the execution of an error callback fails within the `x/cwerrors/abci.go` file, the error will be stored in the kv store. It's important to note that it cannot be stored in the transient store, as this store is cleared at the end of the block when `Store.Commit()` is called, which results in a new `dbadapter.Store` being assigned and the previous one being discarded and garbage collected.
4. **When a contract invokes the `Errors` query function, only the errors that have been stored will be returned** ✅
   - Regarding the storage of errors, three collections are typically updated, although only two are particularly relevant here:
     - **ContractErrors**: This collection maps a contract address to an error ID, helping to track which errors are associated with which contracts.
     - **Errors**: This collection maps each error ID to the specific details of the error, referred to as `SudoError`.

     To retrieve errors for a specific contract, the process involves using the `ContractErrors` collection to iterate through all the error IDs associated with a contract address. Following this, each corresponding `SudoError` is fetched from the `Errors` collection for every error ID.
     This segment of the unit tests illustrates correctness by ensuring that if the same error is reported multiple times, it is returned multiple times.
5. **If a contract never subscribes, it will never appear on the subscription list, and conversely, if the contract's subscription fee is transferred away, it will be added to the subscription list.** ✅

- Subscription is handled through a specific message: `MsgSubscribeToError`. The contract will only be subscribed if the provided `fee` equals or exceeds the `SubscriptionFee`.

6. **At block height x, there should be no subscriptions stored that expire before x.** ✅
   - This essentially means that the pruning mechanism must function properly. It has been confirmed that the `PruneSubscriptionsEndBlock` function is invoked within the cwerrors' endBlocker. This function removes all contract addresses that match the current block height, based on the specified range, ensuring that subscriptions are pruned accordingly.

Liveness properties

The liveness properties ensure:

1. If a protocol module reports an error within a subscription period, the contract will receive an error callback, meaning no error goes unnoticed. ✅
2. If a contract is not subscribed when an error is reported, the error is stored in the permanent store. ✅
3. No error is stored for more blocks than specified by `ErrorStoredTime`. ✅

Threats and Corner Cases

1. **Threat: Errors may inadvertently be overlooked if the sequence of `EndBlocker` executions is incorrect.** ✅

   Conclusion: It is crucial to understand that errors are logged during the `EndBlocker` phase within x/callback/abci.go, and actual callbacks are processed during the `EndBlocker` within x/cwerrors/abci.go. Therefore, it is essential that the `EndBlocker` execution for cwerrors occurs after that of the callbacks. Generally, the `EndBlocker` for cwerrors should be the final one in the sequence of executions. At present, cwfees and cwica follow cwerrors' `EndBlocker`, which currently does not cause issues, but it could pose problems in the future.

   Recommendation: Document the concern regarding the sequence of EndBlocker executions, particularly noting that cwfees and cwica currently follow cwerrors' EndBlocker without issues but may pose problems in the future.

2. **Corner Case: Consider a scenario where user subscribes from 1-2pm and then resubscribes at 2:01pm. Could there be a race condition with errors that occur between 2-2:01? What if the user resubscribe in the same block that an error occurs** ❓

   Conclusion: During the 2-2:01 period, any errors that occur will be stored in the kv storage. If two messages occur in the same block—1. MsgTimeoutPacket and 2. MsgSubscribeToError—the first message (via `OnTimeoutPacket`) will cause the error to be added to the kv storage (since the contract is not yet subscribed). Immediately after, MsgSubscribeToError subscribes the contract. This raises a question: Can something unexpected happen during the cwerrors' `EndBlocker` execution?

   A timeout error will not be part of the transient store, meaning the contract won't be notified at that moment even though it's subscribed. The error is not lost; it remains in the kv storage. If MsgSubscribeToErrors is executed before MsgTimeout, the contract would be automatically notified of the error.

   This behavior should be documented clearly to help users understand that the order of messages within a block probably matters. Specifically, if a user is subscribed, they should be notified of errors from the current block until their subscription expires, without depending on the sequence of message execution.

3. **Corner Case: Consider the scenario where a contract performs a (Stargate) query in the same block where an error is scheduled to be pruned.** ✅

Conclusion: Pruning of errors within the cwerror module is handled during the `EndBlocker` phase. Therefore, any Stargate query intended to fetch errors must be executed before the `EndBlocker` is called within the same block.

4. **Corner Case: Unconstrained error addition to the cwerror module storage in the event of a malicious contract.** ❓

   Conclusion: Consider a scenario where a contract triggers 100 actions in a protocol module (e.g., a transfer or a packet send). In this case, potentially, 100 errors could be reported. The number of errors stored in the error module depends on the contract's code. The question arises whether such behavior should be defended against in the protocol.

   In essence, a contract could invoke a protocol module frequently, leading to the storage of an error for each call. This could pose a troublesome scenario.

   Recommendation: Implementing something like a rate limiter (limiting errors per block and per contract) might be necessary to address this situation.

5. **Threat: Denial of Service (DoS) via Excessive Error Logging and Querying** ❓
   Conclusion: When contracts or their administrators initiate callbacks through `MsgRequestCallback`, these are scheduled and later executed during the `EndBlocker` phase within the callback module. If the contract code invoked during these callbacks fails, the errors are captured by the `SetError` function and logged within the cwerrors module.
   A potential vulnerability arises if a malicious actor repeatedly requests callbacks with the knowledge that the executions will fail, deliberately causing a high volume of errors to be logged. Although measures such as block reservation fees, future reservation fees, and transaction fees exist, they might not fully deter such behavior if the primary intent is to disrupt rather than avoid costs.
   The accumulation of errors over multiple blocks, despite the `MaxBlockReservationLimit` preventing overload within a single block, can significantly bloat the cwerrors storage. The real threat surfaces when these stored errors become the target of repeated, intensive querying operations, particularly through gRPC queries like `errors, err := qs.keeper.GetErrorsByContractAddress(sdk.UnwrapSDKContext(c), contractAddr)`.
   This could potentially lead to a DoS attacks.
   Given this, it may be prudent to implement more robust defensive strategies such as rate limiting for error queries, enhanced scrutiny of callback requests, or dynamic adjustment of fees based on observed system load and error generation rates.
6. **Threat: Mismanagement of Subscription Extensions** ❌
   Conclusion: This threat arises from the current implementation of the `SetSubscription` function, which does not account for the additive nature of subscription renewals expected by users. Instead of extending an existing subscription, a new subscription resets the period, leading to potential loss of the remaining time from the prior subscription. This issue has been detailed further in the report to highlight the need for adjusting the subscription logic to align with user expectations and ensure fair handling of paid subscription periods.
7. **Threat: The risk of iterating over all stored errors in every block to identify candidates for pruning.** ✅
   Conclusion: The Walk method in the `PruneErrorsCurrentBlock` function operates within the bounds of the prefix range, ensuring it only iterates over entries scheduled for action at the current block height. This eliminates the necessity to inspect every entry in the `DeletionBlocks` collection, leading to significant performance optimization. This optimization is particularly beneficial in scenarios where the dataset is extensive but only a fraction of it is pertinent at any given block height.

x/cwica

The purpose of this module is to enable contracts to handle ICA tasks easily, without having to manage all IBC details themselves. It acts as a wrapper around built-in ICA functionality and represents the controller chain side of ICA. This module provides two key functionalities:

1. **Registering an Account:**
   Contracts can register an ICA account.
2. **Sending Transactions:**
   Contracts can create IBC packets and send them.

To register an account, the `HandleChanOpenAck` function is required. For sending a packet, the `HandleAcknowledgement` and `HandleTimeout` functions are needed. For the purposes of the audit, we assume that the core IBC logic underlying these functions is functioning correctly.

Invariants

1. **Only contracts already recognized by the system will be allowed to register an interchain account** ✅
   - This is guaranteed by checking `HasContractInfo` function. This also means that sender address has to be contract address
2. **It's not feasible for another contract to register ICA on behalf of this contract.** ✅
   - This is done with usual authorization check via `GetSigners` function which returns `msg.ContractAddress` meaning that the only entity allowed to register ICA is a contract which actually sent the register message.
   - Contract cannot use some other's entity ICA, which is guaranteed by obtaining unique portID / channel ID.
   - It's designed to be one contract address per ICA
3. **No packet for an unregistered account should ever be sent** ✅
   - It's not possible to skip registration step and somehow communicate with ICA on the host side. During `HandleChanOpenAck`, which happens after ICA is created on the host, calling contract will save the ICA address to be able to send packets.
4. **If the acknowledgment contains no errors, cwerror should not receive any error reports.** ✅
5. **If the acknowledgment contains an error, no success message should be reported.** ✅
6. **If a packet times out, no success message should be reported.** ✅

Threats

1. **Threat: An error occurring during the `HandleAcknowledgement` message can close the channel and result in network spam.** ✅
   Conclusion:
   In the core IBC logic of the `Acknowledgement` function, the entire transaction is reverted if the `OnAcknowledgementPacket` callback fails. Although it's uncommon for an acknowledgment to fail, the `HandleAcknowledgement` function invokes custom contract logic via `Sudo`, which introduces potential threats.
   A contract may not define the `Sudo` entry point, potentially causing `HandleAcknowledgement` to return an error. Alternatively, the `Sudo` message itself could fail due to various reasons like invalid input data, bugs, or intentional errors.
   Since the channel is *ORDERED*, it becomes unusable if the `Sudo` message fails, and the relayer will continue to send the same faulty acknowledgment message. An attacker could exploit this vulnerability by deploying

numerous smart contracts and transactions with faulty `Sudo` handlers to spam the network with acknowledgment messages.
Despite these threats, the error returned when calling the `Sudo` message **is not directly passed back to the core `Acknowledgement` function**. The error is only visible in the logs. Other situations that could cause `HandleAcknowledgement` to return an error are much less likely (such as un/marshalling failures, `SetError` function failures, or `AccAddressFromBech32` function failures).

Findings

Finding	Severity	Status
Potential Overpayment of Subscription Fees	HIGH	RESOLVED
Mismanagement of Subscription Extensions	MEDIUM	RESOLVED
Potential Misuse of Sudo Calls Without Gas Limit in HandleAcknowledgement	LOW	RISK ACCEPTED
Duplication of Entries in Key and Value of Different Maps in x/cwerrors/keeper/keeper.go	INFORMATIONAL	RESOLVED
Unused Enums, Parameter, and Inadequate Comments in x/cwica Module	INFORMATIONAL	RESOLVED
Redundant Error Checks for AccAddressFromBech32 in SendTx and RegisterInterchainAccount Functions	INFORMATIONAL	ACKNOWLEDGED
Documentation Enhancements Needed for CWICA and CWERRORS Specifications	INFORMATIONAL	RESOLVED

Potential Overpayment of Subscription Fees

Project	Archway 2024 Q2
Type	IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

x/cwerrors/keeper/subscriptions.go

Description

The `SetSubscription` function in the system checks if the provided fee is less than the required `SubscriptionFee`, returning an error if this condition is met. However, there is no check to prevent the contract from overpaying the subscription fee. The function transfers the fee from the contract's account to the `FeeCollector` module regardless of the amount over the required fee.

Problem Scenarios

Code Block Analysis:

```
if fee.IsLT(params.SubscriptionFee) {
    return -1, types.ErrInsufficientSubscriptionFee
}
err = k.bankKeeper.SendCoinsFromAccountToModule(ctx, contractAddress,
authtypes.FeeCollectorName, sdk.NewCoins(fee))
if err != nil {
    return -1, err
}
```

The above code ensures that the fee provided must at least meet the minimum required subscription fee but does not cap the fee at this minimum. As a result, if a contract sends more than the required fee, the excess amount is transferred without a way to revert or adjust this overpayment.

Risk Implication:

This oversight could lead to financial losses for contracts that mistakenly overpay their subscription fees.

Recommendation

It is advisable to modify the subscription function to either:

- Refund any excess amount over the required subscription fee.
- Implement a check to ensure that the exact subscription fee is paid, rejecting any amounts that exceed this fee.

In either scenario, clear documentation should be provided to users outlining how fees are handled, what amounts should be paid, and the consequences of not adhering to these guidelines.

Status update

This issue has been addressed with this commit.

Mismanagement of Subscription Extensions

Project	Archway 2024 Q2
Type	IMPLEMENTATION PROTOCOL
Severity	MEDIUM
Impact	LOW
Exploitability	HIGH
Status	RESOLVED

Involved artifacts

x/cwerrors/keeper/subscriptions.go

Description

When a contract pays a fee at block x to subscribe for a certain number of blocks (`SubscriptionPeriod`), their subscription is expected to be active until block `SubscriptionPeriod + x` . If a subscriber chooses to renew or extend their subscription before the initial period ends, the expected behavior might be that the subscription period would cumulatively extend beyond the original end date.

Problem Scenarios

In the current `SetSubscription` function implementation, if a subscription is already active and the subscriber adds a new subscription before the old one expires, the new subscription period does not sum up with the remaining period of the current subscription. Instead, the subscription is reset to start from the current block height for another full subscription period.

This behavior might lead subscribers to believe they can extend their existing subscription by subscribing again during the active period, expecting the two periods to be additive. However, the function resets the subscription period to start from the current block height and runs for the full `SubscriptionPeriod` , disregarding any remaining time from the previous subscription.

Recommendation

To mitigate this issue, it would be beneficial to adjust the subscription logic to accumulate any remaining time from an existing subscription when a new payment is made. For example, the end height for a renewed subscription could be calculated as:

```
subscriptionEndHeight := max(ctx.BlockHeight(), existingEndHeight) +
params.SubscriptionPeriod
```

Status update

This issue has been addressed with this commit.

Potential Misuse of Sudo Calls Without Gas Limit in HandleAcknowledgement

Project	Archway 2024 Q2
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Exploitability	LOW
Status	RISK ACCEPTED

Involved artifacts

x/cwica/keeper/ibc_handlers.go

Description

In the `HandleAcknowledgement` function, there is a `Sudo` call that isn't wrapped with the `ExecuteWithGasLimit` function, which can lead to potential misuse. This differs from other `Sudo` calls, which are typically wrapped with `ExecuteWithGasLimit` when called from within `endBlocker`. The absence of a gas limit wrapper potentially allows malicious or poorly implemented contracts to attempt draining the relayer's funds by implementing an infinite loop or otherwise exploiting gas consumption.

Problem Scenarios

1. **Infinite Loop Concerns:**
   Communication with the Archway team clarified that infinite loops cannot cause an uncontrolled draining of the relayer's funds because a `max_gas` parameter serves as a hard cap on the gas usage per transaction leading to gas error that aborts the transaction.
2. **EndBlocker Execution:**
   For other `Sudo` calls, the Archway team uses `ExecuteWithGasLimit` because these calls tend to execute in `endBlocker` with an `InfiniteGasMeter`, where no entity is directly responsible for paying the gas fees.

Recommendation

Despite the cap on gas usage (`max_gas`), a malicious contract could still drain up to the product of `gas_price` and `max_gas`, which warrants marking this finding as Low severity. Proper safeguards could still be in place to minimize potential exploitation.

Duplication of Entries in Key and Value of Different Maps in x/cwerrors/keeper/keeper.go

Project	Archway 2024 Q2
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	LOW
Exploitability	NONE
Status	RESOLVED

Involved artifacts

x/cwerrors/keeper/keeper.go

Description

The `Keeper` struct within the `x/cwerrors` module contains multiple instances where key-value pairs in various maps include duplicated data. Specifically, the `ContractErrors`, `DeletionBlocks`, and `SubscriptionEndBlock` maps store parts of their keys redundantly as values.

Problem Scenarios

1. **ContractErrors**: This map's key consists of `contractAddress` and `ErrorId`, and it redundantly stores `ErrorId` as the value.
2. **DeletionBlocks**: In this map, a composite key including `BlockHeight` and `ErrorId` is used, with `ErrorId` again stored as the value.
3. **SubscriptionEndBlock**: The key here includes `BlockHeight` and `contractAddress`, with `contractAddress` redundantly repeated as the value.

This redundancy can lead to increased storage requirements and might complicate state management within the blockchain, affecting both performance and gas costs.

Recommendation

To optimize the storage and improve efficiency, the following changes are recommended:

- **Remove Redundant Values**: The duplicated data in the values should be removed as it is already available in the keys. For instance, `ContractErrors` and `DeletionBlocks` can simply store a placeholder or minimal value since `ErrorId` is part of the key. Similarly, `SubscriptionEndBlock` can store a minimal value instead of repeating `contractAddress`.

By adopting these changes, the storage footprint could be reduced, and the system's performance could be enhanced by decreasing the load on state management operations. This approach aligns with optimizations seen in other blockchain projects, such as the reduction of position ID storage sizes in the Osmosis project, thereby promoting more efficient data usage and potentially lowering operational costs.

Status update

This issue has been addressed with this commit.

Unused Enums, Parameter, and Inadequate Comments in x/cwica Module

Project	Archway 2024 Q2
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	RESOLVED

Involved artifacts

x/cwica

test-contracts

Description

During inspection of the `x/cwica` module, the following issues were discovered:

1. **Unused Enums:**
   The enums `ErrInvalidAccountAddress` and `ErrInterchainAccountNotFound` defined in the `x/cwica/types/errors.go` file are not being utilized anywhere in the module's codebase.
2. **Unused Parameter:**
   The `relayer` parameter in both the `HandleTimeout` and `HandleAcknowledgement` functions is currently unused.
3. **Inadequate Comments:**
   The comment for the `HandleTimeout` function mentions that it "passes the timeout data to the appropriate contract via a sudo call," but no such call exists in the function.
4. **Typographical Error:**
   A typo has been identified in the test contract file `x/cwica/src/contract.rs`, where "regsiter_msg" should be corrected to "register_msg."

Recommendation

- **Remove Unused Variables:**
  Remove the unused variables `ErrInvalidAccountAddress` and `ErrInterchainAccountNotFound` from the `x/cwica/types/errors.go` file to clean up the codebase.

- **Remove Unused Parameter:**
  Consider removing the `relayer` parameter from the `HandleTimeout` and `HandleAcknowledgement` functions if it is not intended for future use.
- **Update Comments:**
  Revise the comment for the `HandleTimeout` function to accurately reflect its functionality.
- **Correct Typographical Error:**
  Update the typo "regsiter_msg" to "register_msg" in the `x/cwica/src/contract.rs` file for clarity and accuracy.

Status update

This issue has been addressed with this PR.

Redundant Error Checks for AccAddressFromBech32 in SendTx and RegisterInterchainAccount Functions

Project	Archway 2024 Q2
Type	**IMPLEMENTATION**
Severity	**INFORMATIONAL**
Impact	**NONE**
Exploitability	**NONE**
Status	**ACKNOWLEDGED**

Involved artifacts

x/cwica/keeper/msg_server.go

Description

The `SendTx` and `RegisterInterchainAccount` functions currently validate addresses using `sdk.AccAddressFromBech32` after the `ValidateBasic` function has already performed this validation. This leads to redundant error checking and can be optimized. Specifically, both functions contain the following validation:

```
senderAddr, err := sdk.AccAddressFromBech32(msg.ContractAddress)
if err != nil {
    return nil, errors.Wrapf(sdkerrors.ErrInvalidAddress, "failed to parse address:
%s", msg.ContractAddress)
}
```

Recommendation

Replace the error-prone `sdk.AccAddressFromBech32` with the more efficient `sdk.MustAccAddressFromBech32`, which directly converts the address, assuming validation was handled earlier.

With the assumption that `ValidateBasic` will always run before `SendTx` or `RegisterInterchainAccount`, the functions can be refactored to:

```
senderAddr := sdk.MustAccAddressFromBech32(msg.ContractAddress)

if !k.sudoKeeper.HasContractInfo(ctx, senderAddr) {
    return nil, errors.Wrapf(types.ErrNotContract, "%s is not a contract address",
msg.ContractAddress)
}
```

Documentation Enhancements Needed for CWICA and CWERRORS Specifications

Project	Archway 2024 Q2
Type	DOCUMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	RESOLVED

Involved artifacts

x/cwerrors/spec

x/cwica/spec

Description

Both the CWICA and CWERRORS modules' documentation currently lack certain clarifications and detailed descriptions necessary for effective integration and error handling. Improvements to these documents will enhance understanding and implementation by developers.

Problem Scenarios

1. **Error Handling Clarification (CWICA):**
   The "How to use in CW contract" section suggests automatic Sudo callbacks, which can be misleading. The sentence "*Once the transactions have been submitted, the contract will receive a callback at the Sudo entrypoints*" may be partly incorrect. It's important to clarify that in case of a timeout or error, callbacks are received only if you've explicitly subscribed to error handling and this subscription is handled through an external process.
2. **Misleading Link and Lack of How-To (CWICA):**
   The link "Here is how to integrate them" only points to a proto file and lacks practical ("how to") guidance . This should be replaced or supplemented with a detailed integration guide or walkthrough.
3. **Incomplete Transaction Lifecycle Description (CWICA):**
   The `MsgSendTx` operation facilitates the submission of a collection of transactions from an Interchain Account (ICA) on a counterparty chain. The documentation currently suggests that successful submission ("On Success" part) of `MsgSendTx` guarantees execution on the counterparty chain, which is a misleading implication. In reality, a successful submission only ensures that the transaction packet is recorded as sent on the originating chain. Several conditions must be met for execution on the counterparty chain, including:
   - **Relayer Activity:** The packet needs to be picked up and relayed by a network participant.
   - **Timeliness:** The transaction must be relayed within the timeout specified, otherwise, it will expire.
   - **Counterparty Chain Processing:** The counterparty chain must successfully process the transaction without internal errors.

The specification should therefore clarify that successful submission of `MsgSendTx` does not guarantee execution but only the creation and sending of an IBC packet. Additionally, it would be beneficial to link to sections detailing error handling and what happens if transactions fail after being sent, to give developers a comprehensive understanding of potential failure points and how they are handled.

4. **Lack of IBC Handler Descriptions (CWICA):**
   The documentation omits descriptions of the IBC handlers. This section should include detailed explanations of these handlers, their error handling processes and "On Success" / "This message is expected to fail if" parts.
5. **Explicit Connection to Protocol Modules (CWERRORS):**
   The CWERRORS spec needs to clearly describe how errors from protocol modules are communicated to the CWERRORS module. Adding a direct statement on this mechanism will clarify the error reporting process. Something like: if a protocol module detects an error condition, it should report the error to cwerror.
6. **Error Deletion Process (CWERRORS):**
   The current CWERRORS spec implies that errors are explicitly marked for deletion and then removed. However, this process might not be accurately described. Instead, errors are associated with an expiration date when stored, and are automatically pruned after reaching the designated expiration block.
   Clarifications:
   - **Error Retention:** The spec clarifies here that errors are retained for a configurable number of blocks (defined by a module parameter) and are then automatically pruned once this limit is reached. So, rather than actively *marking* errors for deletion, the pruning occurs naturally as the expiration date is reached.
   - **Transient Store Pruning:** Errors stored in the transient store are pruned automatically after each block, ensuring that this temporary storage remains efficient and does not accumulate stale errors. It might be beneficial to include this clarification in the specifications as well.

   Ultimately, the focus here lies on the phrase "marking errors." It could imply going through each one and flagging them for deletion which is not the case. It seems like this is just a matter of semantics.

Recommendation

All above points should be addressed in their respective documentation sections to prevent misunderstandings and ensure developers can effectively integrate and utilize these features. Each specification should include examples, practical guides, and explicit descriptions of processes to aid in understanding and implementation.

Status update

This issue has been addressed with this PR.

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:

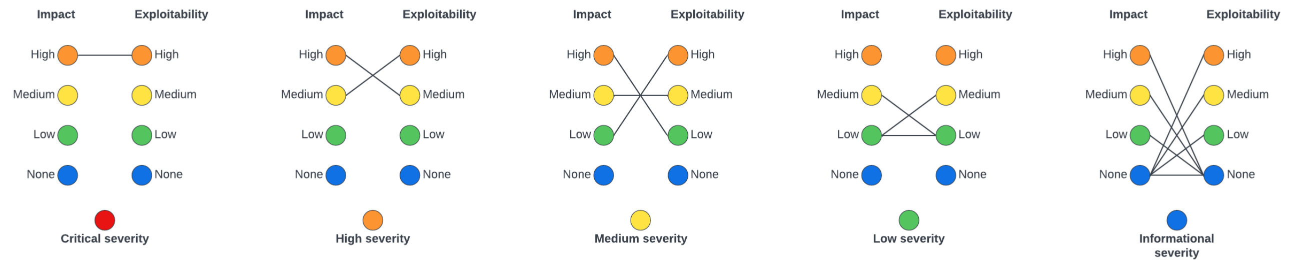
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

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

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

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

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