Cosmos SDK V0.53.0

Security Assessment

April 29th, 2025 — Prepared by OtterSec

Yordan Stoychev anatomic@osec.io

James Wang james.wang@osec.io

Table of Contents

01 — Executive Summary

Overview

Cosmos engaged OtterSec to assess the `sdk-v0.53.0` program. This assessment was conducted between April 8th and April 29th, 2025. We conducted a follow-up review of the `unordered-tx` and `epochs` programs from August 14th to October 5th. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a vulnerability where unordered transactions fail to check the uniqueness of the transactions nonce, breaking the assumption that each address and nonce pair maps to a single transaction, potentially resulting in query inconsistencies and adverse affects on systems relying on this invariant (OS-CDK-ADV-00). Additionally, we highlighted a mismatch in timestamp precision: the mempool utilizes Unix second as nonce while the antehandler only guarantees uniqueness of UnixNano timestamp, resulting in potential nonce collisions and unexpected transaction replacement (OS-CDK-ADV-01).

We also recommended documenting the current behavior where timestamps are utilized as nonces for unordered transactions, resulting in their delay until all ordered transactions from the same sender are processed (OS-CDK-SUG-00). We further advised against setting the current epoch start height without verifying that the start time is not in the future, as this may result in inaccurate block-based epoch calculations (OS-CDK-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/cosmos/cosmos-sdk. This audit was performed against PR#23708, PR#23815, PR#23933, PR#23974, PR#24010, PR#24354, PR#24573, PR#24581.

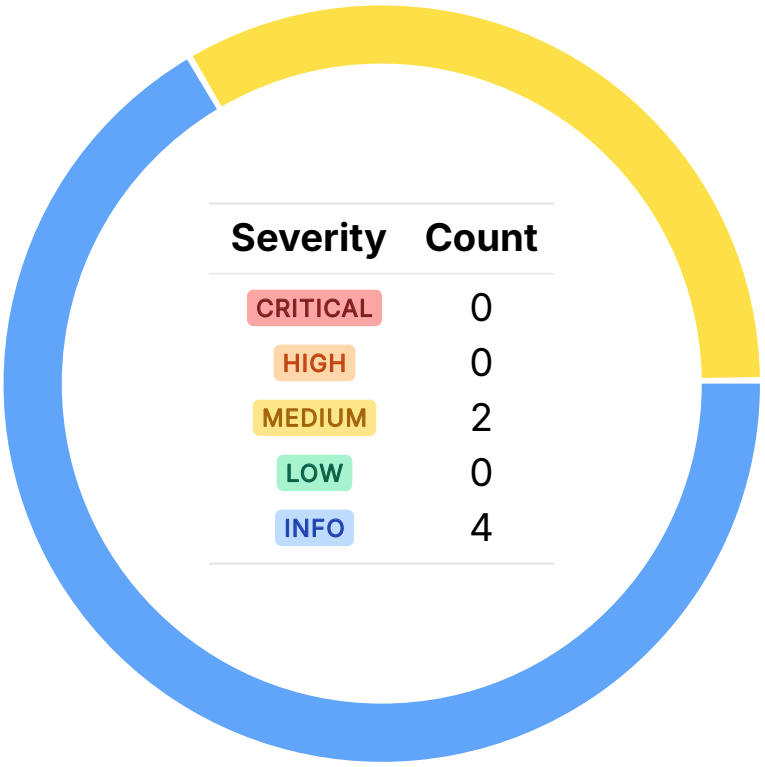
A brief description of the program is as follows:

Name	Description
ProtocolPool	This module handles the functionality around community pool funds, providing a separate module account for community pools to track their assets, and adds fund streaming options. It introduces the option to route community pool funds from `x/distribution`.
unordered-tx	Unordered transactions are designed to bypass the typical nonce management system to enable fire-and-forget transactions, allowing for more efficient parallel processing. They utilize an unordered-nonce which is the timeout-timestamp for a given address, which is now stored in state to ensure data persistence across node crashes or restarts.
epochs	It allows modules to register periodic tasks that are triggered at fixed time intervals, called "epochs." Each epoch has a start and end time, and when the block time exceeds the end time, the timer "ticks," executing the registered logic. It ensures that even after downtime, missed epochs are processed once the chain is back online, maintaining synchronization. It introduces a time-based hook module for adding callbacks to the applications.
PR#24354	This PR relates to an end block optimization for validators, reducing unnecessary `bech32` conversions.

03 — Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	0
INFO	4

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-CDK-ADV-00	MEDIUM	RESOLVED	The removal of nonce uniqueness in `unorderedtx` breaks the assumption that each `(address, nonce)` maps to a single transaction, potentially resulting in query inconsistencies and adverse effects on systems relying on this invariant.
OS-CDK-ADV-01	MEDIUM	RESOLVED	There is a mismatch in timestamp precision, where the `antehandler` only guarantees uniqueness of `UnixNano` timestamp while the `mempool` utilizes `Unix` seconds, resulting in potential nonce collisions and unexpected transaction replacement.

Nonce Uniqueness Violation in Unordered Transactions MEDIUM OS-CDK-ADV-00

Description

Transactions are typically validated and ordered based on a sequence number (nonce). This ensures that transactions are processed in a strict order for each account, preserving consistency and preventing double-spending. However, with the introduction of the `unorderedtx` module, this invariant is no longer enforced, creating a potential side effect where multiple transactions from an account account may now share the same nonce. Multiple transactions with the same nonce may undermine downstream usage of transaction sequence field.

>_ *x/auth/client/cli/query.go* GO

```
// QueryTxCmd implements the default command for a tx query.
func QueryTxCmd() *cobra.Command {
        [...]
            switch typ {
            case TypeAccSeq:
                        if args[0] == "" {
                                return fmt.Errorf("`acc_seq` type takes an argument
                                    ↪  '<addr>/<seq>'")
                        }
                        query := fmt.Sprintf("%s.%s='%s'", sdk.EventTypeTx,
                            ↪  sdk.AttributeKeyAccountSequence,         args[0])
                        [...]
                        return clientCtx.PrintProto(txs.Txs[0])
            default:
                        return fmt.Errorf("unknown --%s value %s", FlagType, typ)
                }
        [...]
}
```

One concrete manifestation is within `query::QueryTxCmd`, which allows users to query a transaction by `<address>/<nonce>` utilizing event attributes. This command assumes that such a pair will return at most one transaction, which will not be the case if multiple transactions exist with the same nonce. While this case is not critically impactful on its own, failure to communicate this change in behavior to developers may result in more serious issues in the future.

Remediation

Remove usage of `unorderedtx` sequence in cosmos-sdk, and document the change to notify developers to do the same for custom-developed code.

Patch

Resolved in f9f3bfb.

Timestamp Precision Mismatch MEDIUM OS-CDK-ADV-01

Description

Cosmos SDK utilizes unordered nonces based on timestamps to ensure replay protection for unordered transactions. However, there is an inconsistency in timestamp precision, where the `antehandler` utilizes `UnixNano` (nanosecond precision) in `keeper::ContainsUnorderedNonce` (shown below) and `keeper::TryAddUnorderedNonce`, while the `mempool` utilizes `Unix` seconds in `sender_nonce::Insert` (shown below) and `priority_nonce::Insert`.

>_ *types/mempool/sender_nonce.go* GO

```go
// Insert adds a tx to the mempool. It returns an error if the tx does not have
// at least one signer. Note, priority is ignored.
func (snm *SenderNonceMempool) Insert(_ context.Context, tx sdk.Tx) error {
    [...]
    // if it's an unordered tx, we use the timeout timestamp instead of the nonce
    if unordered, ok := tx.(sdk.TxWithUnordered); ok && unordered.GetUnordered() {
            timestamp := unordered.GetTimeoutTimeStamp().Unix()
            if timestamp < 0 {
                    return errors.New("invalid timestamp value")
            }
            nonce = uint64(timestamp)
    }
    [...]
}
```

This mismatch may result in valid transactions with distinct nanosecond-level timestamps to appear as duplicates in the `mempool`. Consequently, this may result in unexpected nonce collisions and transaction replacement in the `mempool`.

>_ *x/auth/keeper/keeper.go* GO

```go
// ContainsUnorderedNonce reports whether the sender has used this timeout already.
func (ak AccountKeeper) ContainsUnorderedNonce(ctx sdk.Context, sender []byte, timeout
    ↪  time.Time) (bool, error) {
        return ak.UnorderedNonces.Has(ctx, collections.Join(timeout.UnixNano(), sender))
}
```

Additionally, certain sanity checks in `antehandler` (within `basic::AnteHandle` and `unordered::ValidateTx`) concerning timeout timestamp checks also utilize `Unix` seconds. However, since it is not possible for timestamps to differ by less than one second since the `Unix` epoch, this does not have any impact.

Remediation

Ensure that the `mempool` also utilizes `UnixNano` consistently to match the `antehandler`'s replay protection logic and prevent transaction collisions created by timestamp truncation.

Patch

Resolved in f560014.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-CDK-SUG-00	Utilizing timestamps as nonces for `unorderedtx` results in them being delayed until all ordered transactions from the same sender are processed, which may hinder asynchronous behavior.
OS-CDK-SUG-01	Setting `CurrentEpochStartHeight` during `AddEpochInfo` without checking if `StartTime` is in the future may result in inaccurate block-based epoch calculations.
OS-CDK-SUG-02	Duplication of `ContinuousFund` entries is not checked in `ProtocolPool` `Genesis` validation, and may result in unintentional overwrite of entries by genesis provider.
OS-CDK-SUG-03	Removal of nonexistence `ContinuousFund` is not checked, and silently return success.

Scheduling Delay Between Ordered/Unordered Transactions OS-CDK-SUG-00

Description

In the current `mempool` implementation, `unorderedtx` utilize timeout timestamps as nonces. While it is possible for normal ordered nonces to collide with timestamp nonces, this possibility is extremely unlikely and unrealistic as timestamps are in a much higher range ($\approx 10^{10}$) than typical sequential nonces. Another concern is that `unorderedtx` are not processed by the `mempool` until all `orderedtx` from the same sender (up to the current nonce) are completed, effectively blocking `unorderedtx`. It is not explicitly documented whether this deferred execution is intentional, which raises concerns about correct utilization.

Remediation

Ensure this behavior is clearly documented if intentional.

Incorrect Setting of Epoch Start Height OS-CDK-SUG-01

Description

In `epoch::AddEpochInfo`, `CurrentEpochStartHeight` is set to the current block height if it is zero, without checking whether the epoch's `StartTime` is in the future. This creates a mismatch between when an epoch is configured to begin and the block height at which it is marked as having started. While this issue may not currently have visible impacts, it could have adverse effects.

>_ *x/epochs/keeper/epoch.go* GO

```go
// AddEpochInfo adds a new epoch info. Will return an error if the epoch fails validation,
// or re-uses an existing identifier.
// This method also sets the start time if left unset, and sets the epoch start height.
func (k *Keeper) AddEpochInfo(ctx sdk.Context, epoch types.EpochInfo) error {
    [...]
    if epoch.CurrentEpochStartHeight == 0 {
            epoch.CurrentEpochStartHeight = ctx.BlockHeight()
    }
    return k.EpochInfo.Set(ctx, epoch.Identifier, epoch)
}
```

Particularly, this premature setting may result in `NumBlocksSinceEpochStart` to return a non-zero value even before the epoch has actually started (based on its future `StartTime`). This undermines the semantic correctness of block-based epoch calculations.

>_ *x/epochs/keeper/epoch.go* GO

```go
// NumBlocksSinceEpochStart returns the number of blocks since the epoch started.
// if the epoch started on block N, then calling this during block N (after BeforeEpochStart)
// would return 0.
// Calling it any point in block N+1 (assuming the epoch doesn't increment) would return 1.
func (k *Keeper) NumBlocksSinceEpochStart(ctx sdk.Context, identifier string) (int64, error) {
    epoch, err := k.EpochInfo.Get(ctx, identifier)
    if err != nil {
            return 0, fmt.Errorf("epoch with identifier %s not found", identifier)
    }
    return ctx.BlockHeight() - epoch.CurrentEpochStartHeight, nil
}
```

This is not a serious issue since `AddEpochInfo` is currently only used in genesis, and not exposed to untrusted parties.

Remediation

Refactor the logic to only set `CurrentEpochStartHeight` if `StartTime` is not in the future, or clearly document `AddEpochInfo` argument requirements to prevent misuse.

Incomplete Check For ProtocolPool Genesis OS-CDK-SUG-02

Description

`ProtocolPool` keeps track of `ContinuousFund` by maintaining a mapping from `recipient` to `ContinuousFund` structures. However, `Genesis` validation doesn't check each `ContinuousFund` has a unique `recipient`, therefore, if multiple entries share the same `recipient`, the earlier entries will be overwritten by the latter entries.

This is not a serious issue since genesis can only be provided by trusted parties, but we still recommend adding checks to prevent mistakes from happening.

Remediation

Ensure there are no duplicate `ContinuousFund` entries in `ProtocolPool` `Genesis`.

Disallow Removal Of Nonexistence ContinuousFund OS-CDK-SUG-03

Description

Removal of nonexistence `ContinuousFund` is not checked, and silently succeeds. While this behavior does not corrupt any on chain states, it is semantically incorrect and could mislead users to believe that an `ContinuousFund` entry is actually removed when it is not.

Remediation

Check the existence of `ContinuousFund` entry before removal, and return an error if the entry does not exist.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.