



Prepared for  
Alex Johnson  
Kevin Kz Chen  
Barry Plunkett  
Cosmos Labs

Prepared by  
Jisub Kim  
Jaeu Kim  
Zellic

February 11, 2026

# Cosmos Group Module

## Blockchain Security Assessment

Placeholder text for the main body of the report.

## Contents

|                                                                               |           |
|-------------------------------------------------------------------------------|-----------|
| <b>About Zellic</b>                                                           | <b>4</b>  |
| <hr/>                                                                         |           |
| <b>1. Overview</b>                                                            | <b>4</b>  |
| 1.1. Executive Summary                                                        | 5         |
| 1.2. Goals of the Assessment                                                  | 5         |
| 1.3. Non-goals and Limitations                                                | 5         |
| 1.4. Results                                                                  | 5         |
| <hr/>                                                                         |           |
| <b>2. Introduction</b>                                                        | <b>6</b>  |
| 2.1. About Cosmos Group Module                                                | 7         |
| 2.2. Methodology                                                              | 7         |
| 2.3. Scope                                                                    | 9         |
| 2.4. Project Overview                                                         | 9         |
| 2.5. Project Timeline                                                         | 10        |
| <hr/>                                                                         |           |
| <b>3. Detailed Findings</b>                                                   | <b>10</b> |
| 3.1. Group admin can manipulate vote outcomes by removing members             | 11        |
| 3.2. Member departure through LeaveGroup can create zero-weight groups        | 13        |
| 3.3. Decision policy validation uses unguarded type assertion                 | 14        |
| 3.4. Negative voting-period values are accepted by decision policy validation | 15        |
| 3.5. Proposer autovote behavior contradicts protobuf documentation            | 16        |
| 3.6. Group and group-policy version fields are stored but never validated     | 18        |
| 3.7. Threshold decision policy is missing zero-total-weight check             | 19        |
| 3.8. Insufficient test coverage                                               | 20        |

---

|           |                            |           |
|-----------|----------------------------|-----------|
| <b>4.</b> | <b>Threat Model</b>        | <b>21</b> |
| 4.1.      | Overview                   | 22        |
| 4.2.      | Group management functions | 22        |
| 4.3.      | Group policy functions     | 26        |
| 4.4.      | Proposal functions         | 30        |
| 4.5.      | Decision policy evaluation | 34        |
| 4.6.      | Background tasks           | 37        |

---

|           |                           |           |
|-----------|---------------------------|-----------|
| <b>5.</b> | <b>Assessment Results</b> | <b>38</b> |
| 5.1.      | Disclaimer                | 39        |

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website [zellic.io](https://zellic.io) and follow [@zellic\\_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at [hello@zellic.io](mailto:hello@zellic.io).



## 1. Overview

### 1.1. Executive Summary

Zellic conducted a security assessment for Cosmos Labs from January 30th to February 5th, 2026. During this engagement, Zellic reviewed Cosmos Group module's code for security vulnerabilities, design issues, and general weaknesses in security posture.

---

### 1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can errors or panics in the EndBlocker path cause the chain to halt?
  - Does the internal math and ORM package handle edge cases correctly, including zero and negative values?
  - Can group admins or members manipulate proposal outcomes by exploiting governance state transitions?
  - Are input validation and state consistency maintained across all code paths that modify group membership and decision policies?
- 

### 1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Modules other than the Group module
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

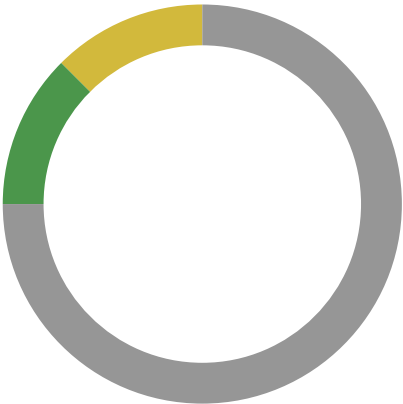
---

### 1.4. Results

During our assessment on the scoped Cosmos Group module, we discovered eight findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

Breakdown of Finding Impacts

| Impact Level             | Count |
|--------------------------|-------|
| <div>Critical</div>      | 0     |
| <div>High</div>          | 0     |
| <div>Medium</div>        | 1     |
| <div>Low</div>           | 1     |
| <div>Informational</div> | 6     |



## 2. Introduction

### 2.1. About Cosmos Group Module

Cosmos Labs contributed the following description of the Cosmos Group module:

The Group module provides on-chain multisig and governance capabilities through groups, group policies, and proposals.

---

### 2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

**Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

**Nondeterminism.** Nondeterminism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes but is not limited to vectors like wall-clock times, map iteration, and other sources of undefined behavior (UB) in Go.

**Arithmetic issues.** This includes but is not limited to integer overflows and underflows, floating-point associativity issues, loss of precision, and unfavorable integer rounding.

**Complex integration risks.** Several high-profile exploits have been the result of unintended consequences when interacting with the broader ecosystem, such as via IBC (Inter-Blockchain Communication Protocol). Zellic will review the project's potential external interactions and summarize the associated risks. If applicable, we will also examine any IBC interactions against the ICS Specification Standard to look for inconsistencies, flaws, and vulnerabilities.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational"

finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

## 2.3. Scope

The engagement involved a review of the following targets:

### Cosmos Group Module

|            |                                                                                           |
|------------|-------------------------------------------------------------------------------------------|
| Type       | Go                                                                                        |
| Platform   | Cosmos                                                                                    |
| Target     | cosmos-sdk                                                                                |
| Repository | <a href="https://github.com/cosmos/cosmos-sdk">https://github.com/cosmos/cosmos-sdk</a> ↗ |
| Version    | 9b6f685ebddccb89747d6f125aa0f53f988db61d                                                  |
| Programs   | types.go<br>keeper/*.go<br>internal/*.go                                                  |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.4 person-weeks. The assessment was conducted by two consultants over the course of one calendar weeks.

## Contact Information

---

The following project manager was associated with the engagement:

**Jacob Goreski**  
✈ Engagement Manager  
[jacob@zellic.io](mailto:jacob@zellic.io) ↗

---

The following consultants were engaged to conduct the assessment:

**Jisub Kim**  
✈ Engineer  
[jisub@zellic.io](mailto:jisub@zellic.io) ↗

**Jaeeu Kim**  
✈ Engineer  
[jaeeu@zellic.io](mailto:jaeeu@zellic.io) ↗

---

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

**January 30, 2026** Start of primary review period

---

**February 2, 2026** Kick-off call

---

**February 5, 2026** End of primary review period

## 3. Detailed Findings

### 3.1. Group admin can manipulate vote outcomes by removing members

|                   |                |                 |        |
|-------------------|----------------|-----------------|--------|
| <b>Target</b>     | Group module   |                 |        |
| <b>Category</b>   | Business Logic | <b>Severity</b> | Medium |
| <b>Likelihood</b> | Low            | <b>Impact</b>   | Medium |

#### Description

A group admin can manipulate the outcome of an active proposal by removing members who have already voted. When `UpdateGroupMembers` removes a member, their existing votes are not deleted but are ignored during tally because the member no longer exists in the group.

```
// x/group/keeper/tally.go
switch {
case sdkerrors.ErrNotFound.Is(err):
    // If the member left the group after voting, then we simply skip the
    // vote.
    continue
```

The root cause is that `doUpdateGroup` does not call `abortProposals` when group membership changes, unlike `doUpdateGroupPolicy`, which does. Additionally, `TallyProposalsAtVPEnd` uses the group's current state rather than a snapshot, so membership changes are reflected in the tally.

The attack scenario is as follows:

1. A proposal is submitted and members vote.
2. The admin observes that the proposal will pass or fail based on current votes.
3. The admin removes opposing voters via `UpdateGroupMembers`.
4. At the voting period end, the removed members' votes are skipped and `TotalWeight` is reduced.
5. The effective threshold drops, and the proposal outcome is reversed.

#### Impact

Group admins have unilateral control over proposal outcomes by selectively removing members during active voting periods. This undermines the governance mechanism that is designed to constrain admin power through collective decision making.

## Recommendations

Consider one of the following mitigations:

1. Calling `abortProposals` in `doUpdateGroup` to match `doUpdateGroupPolicy` behavior
2. Validating `GroupVersion` during tally, rejecting proposals with stale versions
3. Snapshotting member weights at vote time and using those weights during tally

## Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in [PR #25915](#). They intend to merge these changes into the production branch.

### 3.2. Member departure through LeaveGroup can create zero-weight groups

|                   |                 |                 |     |
|-------------------|-----------------|-----------------|-----|
| <b>Target</b>     | Group module    |                 |     |
| <b>Category</b>   | Coding Mistakes | <b>Severity</b> | Low |
| <b>Likelihood</b> | Medium          | <b>Impact</b>   | Low |

#### Description

The UpdateGroupMembers function validates that a group cannot have zero total weight, but LeaveGroup does not perform this check. When the last member calls MsgLeaveGroup, the group's TotalWeight becomes "0".

```
// x/group/keeper/msg_server.go
// UpdateGroupMembers - checked
if totalWeight.IsZero() {
    return errorsmod.Wrap(errors.ErrInvalid, "group must not be empty")
}

// LeaveGroup - unchecked
updatedWeight, err := math.SubNonNegative(groupWeight, memberWeight)
if err != nil {
    return nil, err
}
```

#### Impact

The LeaveGroup function allows groups to enter a zero-weight state that UpdateGroupMembers explicitly prevents, creating inconsistent validation across group membership operations.

#### Recommendations

Consider adding a zero-weight check to LeaveGroup to match the validation in UpdateGroupMembers.

#### Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in commit [d725084c](#).

### 3.3. Decision policy validation uses unguarded type assertion

| Target     | Group module  |          |               |
|------------|---------------|----------|---------------|
| Category   | Code Maturity | Severity | Informational |
| Likelihood | N/A           | Impact   | Informational |

#### Description

The `validateDecisionPolicies` function performs a type assertion without the `, ok` safety pattern. The same assertion in `GetDecisionPolicy` uses the safe pattern.

```
// x/group/keeper/msg_server.go
err = groupPolicy.DecisionPolicy.GetCachedValue().(group.DecisionPolicy)
    .Validate(g, k.config)

// x/group/types.go
decisionPolicy, ok := g.DecisionPolicy.GetCachedValue().(DecisionPolicy)
```

This function is called from `LeaveGroup` and `UpdateGroupMembers`.

#### Impact

It is practically harmless under normal conditions, but this may cause inconsistent defensive coding compared to the guarded assertion in `GetDecisionPolicy`.

#### Recommendations

Consider using the safe `, ok` assertion pattern consistent with the rest of the module.

#### Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in commit [c13236e2](#).

### 3.4. Negative voting-period values are accepted by decision policy validation

| Target     | Group module    |          |               |
|------------|-----------------|----------|---------------|
| Category   | Coding Mistakes | Severity | Informational |
| Likelihood | N/A             | Impact   | Informational |

#### Description

The `ValidateBasic` methods for both `ThresholdDecisionPolicy` and `PercentageDecisionPolicy` check that `VotingPeriod` is not zero but do not reject negative values. Since `VotingPeriod` is a `time.Duration` (signed `int64`), negative values are representable.

```
// x/group/types.go
if p.Windows == nil || p.Windows.VotingPeriod == 0 {
    return errorsmod.Wrap(errors.ErrInvalid, "voting period cannot be zero")
}
```

#### Impact

A negative voting period sets `VotingPeriodEnd` before the block time, causing every proposal under such a policy to expire immediately. There is no security impact beyond wasted gas fees. Nevertheless, other policy fields such as `Threshold` already enforce strict positivity, so accepting negative durations is an unnecessary inconsistency that could mask configuration errors.

#### Recommendations

Consider updating the validation to reject nonpositive values for consistency.

#### Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in commit [771dda7a](#).

### 3.5. Proposer autovote behavior contradicts protobuf documentation

|                   |               |                 |               |
|-------------------|---------------|-----------------|---------------|
| <b>Target</b>     | Group module  |                 |               |
| <b>Category</b>   | Code Maturity | <b>Severity</b> | Informational |
| <b>Likelihood</b> | N/A           | <b>Impact</b>   | Informational |

#### Description

The Proposers field documentation states that proposer signatures are unconditionally counted as yes votes, but the implementation only does so when Exec is set to EXEC\_TRY.

```
// x/group/keeper/msg_server.go
// Proposers field comment: "Proposers signatures will be counted as yes
// votes."

// Actual behavior - conditional:
if msg.Exec == group.Exec_EXEC_TRY {
    for _, proposer := range msg.Proposers {
        _, err = k.Vote(ctx, &group.MsgVote{
            ProposalId: id,
            Voter:      proposer,
            Option:     group.VOTE_OPTION_YES,
        })
    }
}
```

#### Impact

The Exec field's own documentation correctly describes this as conditional, but the two field comments within the same message contradict each other. Developers may incorrectly assume proposers always receive automatic yes votes.

#### Recommendations

Consider updating the Proposers field documentation to clarify that autovoting only applies when Exec is EXEC\_TRY.

## Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in commit [76d074c5](#) ↗.

### 3.6. Group and group-policy version fields are stored but never validated

|                   |               |                 |               |
|-------------------|---------------|-----------------|---------------|
| <b>Target</b>     | Group module  |                 |               |
| <b>Category</b>   | Code Maturity | <b>Severity</b> | Informational |
| <b>Likelihood</b> | N/A           | <b>Impact</b>   | Informational |

#### Description

When a proposal is created, GroupVersion and GroupPolicyVersion are stored in the proposal. However, these fields are never checked during voting, tallying, or execution.

```
// x/group/keeper/msg_server.go
m := &group.Proposal{
    // [...]
    GroupVersion:      groupInfo.Version,
    GroupPolicyVersion: policyAcc.Version,
}
```

The protobuf documentation states that version increments "will cause proposals based on older versions of this group to fail", but no such mechanism exists.

#### Impact

The unused version fields create a false sense of security. Developers may incorrectly assume membership changes automatically invalidate existing proposals.

#### Recommendations

Consider implementing version validation logic to match the documented behavior or updating the documentation to accurately describe the current behavior.

#### Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in [PR #25915](#). They intend to merge these changes into the production branch.

This issue will be documented by the Cosmos Labs.

### 3.7. Threshold decision policy is missing zero-total-weight check

| Target     | Group module  |          |               |
|------------|---------------|----------|---------------|
| Category   | Code Maturity | Severity | Informational |
| Likelihood | N/A           | Impact   | Informational |

#### Description

The `ThresholdDecisionPolicy.Allow()` method does not check for zero `TotalWeight`. When a group's total weight is zero, the effective threshold is clamped via `min(threshold, totalPower)`, causing the comparison `0 >= 0` to evaluate to true and accept proposals without any votes.

```
// x/group/types.go
realThreshold := min(threshold, totalPowerDec) // min(1, 0) = 0

if yesCount.Cmp(realThreshold) >= 0 {           // 0 >= 0 -> true
    return DecisionPolicyResult{Allow: true, Final: true}, nil
}
```

The `PercentageDecisionPolicy.Allow()` method has an explicit zero-weight guard, but `ThresholdDecisionPolicy` does not.

#### Impact

This may cause inconsistent defensive coding between the two decision policy implementations. In practice, exploitation is constrained because proposers must be group members, a newly created group's policy account holds no funds, and in an existing group, the last remaining member already has full control over proposal outcomes.

#### Recommendations

Consider adding a zero total weight check to `ThresholdDecisionPolicy.Allow()` matching the existing guard in `PercentageDecisionPolicy`.

#### Remediation

This issue has been acknowledged by Cosmos Labs, and a fix was implemented in commit [625e3917](#).

### 3.8. Insufficient test coverage

|                   |                |                 |               |
|-------------------|----------------|-----------------|---------------|
| <b>Target</b>     | Project-wide   |                 |               |
| <b>Category</b>   | Protocol Risks | <b>Severity</b> | Informational |
| <b>Likelihood</b> | N/A            | <b>Impact</b>   | Informational |

#### Description

The Group module has a solid test suite covering happy-path scenarios across most components, including multistep governance workflows such as group creation, policy creation, proposal submission, voting, and execution. However, negative test coverage for edge cases and error conditions could be improved.

In particular, boundary value testing for decision policy inputs and error-path testing for state mutation operations are underrepresented.

#### Impact

Expanding the test suite with negative tests would help catch edge cases where validation is inconsistent or incomplete across similar operations, reducing the likelihood of bugs going unnoticed.

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. However, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

## Recommendations

Consider expanding the test suite to include negative tests and boundary conditions for decision policy validation and group membership operations.

## Remediation

This issue has been acknowledged by Cosmos Labs.

## 4. Threat Model

This section provides a comprehensive threat model for the Group module, detailing the intended behavior, preconditions, inputs, and potential risks for the module's core functions.

### 4.1. Overview

The Group module allows the creation and management of on-chain multi-sig accounts through groups of members with weighted voting power. Groups can have associated group policies that define decision-making rules for executing messages. Members submit proposals containing arbitrary messages, vote on them according to their weights, and execute approved proposals through the group policy account.

The module maintains several key data structures through ORM-based tables, including groups, group members, group policies, proposals, and votes. Each entity has a version number that increments on updates, allowing the system to detect stale state and prevent certain race conditions.

### 4.2. Group management functions

Here is an overview of the group management functions.

#### **Function: CreateGroup(msg \*MsgCreateGroup)**

**Module:** contrib/x/group/keeper/msg\_server.go:30

##### **Intended behavior**

This function creates a new group with the specified admin, members, and metadata. The function validates all member addresses and weights, ensuring that all initial members have positive weights. It computes the total weight by summing all member weights and stores the group with an autoincremented group ID. Each member is stored in the group member table with the current block time as the AddedAt timestamp. The function emits an EventCreateGroup event upon success.

##### **Preconditions**

- The admin address must be valid.
- All member addresses must be valid.
- All member weights must be positive decimal values.
- Group metadata length must not exceed the configured maximum.
- Member metadata length must not exceed the configured maximum.

##### **Inputs**

- Admin: The address of the group administrator.
- Members: A list of member requests containing address, weight, and metadata.

- Metadata: Arbitrary metadata string for the group.

**Branching behavior**

- If the admin address is invalid, the function returns an error.
- If any member address is invalid, the function returns an error.
- If any member weight is not a positive decimal, the function returns an error.
- If metadata length exceeds the limit, the function returns an error.
- If the group creation in the table fails, the function returns an error.
- If any member creation in the table fails, the function returns an error.

**External call analysis**

The function does not make external calls to other modules beyond basic address validation through the account keeper and event emission.

**Function: UpdateGroupMembers (msg \*MsgUpdateGroupMembers)**

**Module:** contrib/x/group/keeper/msg\_server.go:102

**Intended behavior**

This function updates the members of an existing group. It can add new members, update existing member weights, or remove members by setting their weight to zero. The function recalculates the total group weight by subtracting old weights and adding new weights. For members being removed, the function deletes them from the group member table. For new members, the function sets the AddedAt field to the current block time, while for updated members, it preserves the original AddedAt timestamp. After updating members, the function increments the group version and validates that all associated decision policies remain valid with the new group weight. The function ensures that the group is not left empty after the update.

**Preconditions**

- The caller must be the group admin.
- The group must exist.
- The member updates list must not be empty.
- All member addresses in the updates must be valid.
- The group must have at least one member with nonzero weight after the update.
- All decision policies associated with the group must remain valid after the weight change.

**Inputs**

- GroupId: The ID of the group to update.
- Admin: The address of the group administrator.
- MemberUpdates: A list of member updates containing address, weight, and metadata.

**Branching behavior**

- If the group ID is zero, the function returns an error.
- If the member updates list is empty, the function returns an error.
- If the admin is not authorized, the function returns an error.
- If any member address is invalid, the function returns an error.
- If a member with zero weight does not exist in the group, the function returns an error.
- If the resulting group weight is zero, the function returns an error.
- If any decision policy becomes invalid after the update, the function returns an error.

**External call analysis**

The function does not make external calls beyond validation and state updates.

**Function: UpdateGroupAdmin(msg \*MsgUpdateGroupAdmin)**

**Module:** contrib/x/group/keeper/msg\_server.go:226

**Intended behavior**

This function transfers admin rights of a group to a new address. It validates that the new admin address differs from the current admin, updates the group's admin field, increments the group version, and persists the changes. The function does not affect group members or group policies associated with the group.

**Preconditions**

- The caller must be the current group admin.
- The group must exist.
- The new admin address must be valid.
- The new admin address must differ from the current admin address.

**Inputs**

- GroupId: The ID of the group to update.
- Admin: The address of the current group administrator.
- NewAdmin: The address of the new group administrator.

**Branching behavior**

- If the group ID is zero, the function returns an error.
- If the new admin equals the old admin, the function returns an error.
- If either admin address is invalid, the function returns an error.
- If the caller is not the group admin, the function returns an error.

**External call analysis**

The function does not make external calls beyond address validation.

**Function: UpdateGroupMetadata(msg \*MsgUpdateGroupMetadata)****Module:** contrib/x/group/keeper/msg\_server.go:258**Intended behavior**

This function updates the metadata of an existing group. It validates the metadata length, updates the group's metadata field, increments the group version, and persists the changes. The function does not affect group members, total weight, or group policies.

**Preconditions**

- The caller must be the group admin.
- The group must exist.
- The metadata length must not exceed the configured maximum.
- The admin address must be valid.

**Inputs**

- GroupId: The ID of the group to update.
- Admin: The address of the group administrator.
- Metadata: The new metadata string for the group.

**Branching behavior**

- If the group ID is zero, the function returns an error.
- If the metadata length exceeds the limit, the function returns an error.
- If the admin address is invalid, the function returns an error.
- If the caller is not the group admin, the function returns an error.

**External call analysis**

The function does not make external calls beyond address validation.

**Function: LeaveGroup(msg \*MsgLeaveGroup)****Module:** contrib/x/group/keeper/msg\_server.go:924**Intended behavior**

This function allows a group member to voluntarily leave a group. It retrieves the member's weight, subtracts it from the group's total weight, deletes the member from the group member table, increments the group version, validates that all associated decision policies remain valid with the new total weight, and persists the updated group. The function does not prevent the last member from leaving if this would result in a group with zero total weight.

**Preconditions**

- The group must exist.
- The caller must be a member of the group.

- The address must be valid.
- All decision policies associated with the group must remain valid after the member leaves.

#### Inputs

- `GroupId`: The ID of the group to leave.
- `Address`: The address of the member leaving the group.

#### Branching behavior

- If the group ID is zero, the function returns an error.
- If the address is invalid, the function returns an error.
- If the caller is not a group member, the function returns an error.
- If any decision policy becomes invalid after the member leaves, the function returns an error.

#### External call analysis

The function does not make external calls beyond address validation and state updates.

### 4.3. Group policy functions

Here is an overview of the group policy functions.

#### Function: `CreateGroupPolicy(msg *MsgCreateGroupPolicy)`

**Module:** `contrib/x/group/keeper/msg_server.go:333`

#### Intended behavior

This function creates a new group policy account for an existing group. It validates the decision policy, ensures the caller is the group admin, generates a unique account address using ADR-028 module credentials, creates an unclaimable base account at that address, and stores the group policy with version 1. The function increments a sequence counter to derive unique addresses and handles rare address collisions by retrying with the next sequence value. The decision policy is validated against the current group state and configuration limits.

#### Preconditions

- The group must exist.
- The caller must be the group admin.
- The group ID must be nonzero.
- The decision policy must be valid.
- The decision policy must be compatible with the group's total weight and the module configuration.
- The metadata length must not exceed the configured maximum.

### Inputs

- Admin: The address of the group administrator.
- GroupId: The ID of the group for which to create the policy.
- Metadata: Arbitrary metadata for the group policy.
- DecisionPolicy: The decision policy defining voting rules.

### Branching behavior

- If the group ID is zero, the function returns an error.
- If the metadata length exceeds the limit, the function returns an error.
- If the decision policy cannot be unpacked, the function returns an error.
- If the decision policy fails basic validation, the function returns an error.
- If the caller is not the group admin, the function returns an error.
- If the decision policy is incompatible with the group, the function returns an error.
- If address generation fails, the function returns an error.
- If the generated address already exists, the function retries with the next sequence value.

### External call analysis

The function interacts with the account keeper to check for existing accounts and to create the new group policy account. It also validates the decision policy against group state.

### Function: `CreateGroupWithPolicy(msg *MsgCreateGroupWithPolicy)`

**Module:** `contrib/x/group/keeper/msg_server.go:285`

### Intended behavior

This function creates a new group and its associated group policy in a single atomic operation. It first calls `CreateGroup` to create the group, then calls `CreateGroupPolicy` to create the policy. If the `GroupPolicyAsAdmin` flag is set, the function additionally updates both the group admin and group policy admin to be the group policy address itself, creating a self-administered group. This allows the group to manage itself through proposals without relying on an external admin.

### Preconditions

- All preconditions for `CreateGroup` must be satisfied.
- All preconditions for `CreateGroupPolicy` must be satisfied.
- If `GroupPolicyAsAdmin` is true, the admin field must be valid for both the group and policy updates.

### Inputs

- Admin: The initial admin address.
- Members: The list of initial group members.

- GroupMetadata: Metadata for the group.
- GroupPolicyMetadata: Metadata for the group policy.
- DecisionPolicy: The decision policy for the group policy.
- GroupPolicyAsAdmin: A flag indicating whether to set the group policy as admin.

**Branching behavior**

- If group creation fails, the function returns an error and does not create the policy.
- If policy creation fails, the function returns an error.
- If GroupPolicyAsAdmin is true and either admin update fails, the function returns an error.

**External call analysis**

The function calls CreateGroup, CreateGroupPolicy, and potentially UpdateGroupAdmin and UpdateGroupPolicyAdmin internally.

**Function: UpdateGroupPolicyAdmin(msg \*MsgUpdateGroupPolicyAdmin)**

**Module:** contrib/x/group/keeper/msg\_server.go:432

**Intended behavior**

This function transfers admin rights of a group policy to a new address. It validates that the new admin differs from the current admin, updates the group policy's admin field, increments the group policy version, and persists the changes. The function does not affect the associated group or its members.

**Preconditions**

- The caller must be the current group policy admin.
- The group policy must exist.
- The new admin address must be valid.
- The new admin address must differ from the current admin address.

**Inputs**

- Admin: The address of the current group policy administrator.
- GroupPolicyAddress: The address of the group policy to update.
- NewAdmin: The address of the new group policy administrator.

**Branching behavior**

- If the new admin equals the old admin, the function returns an error.
- If the new admin address is invalid, the function returns an error.
- If the caller is not the group policy admin, the function returns an error.

**External call analysis**

The function does not make external calls beyond address validation.

**Function: UpdateGroupPolicyDecisionPolicy(msg \*MsgUpdateGroupPolicyDecisionPolicy)**

**Module:** contrib/x/group/keeper/msg\_server.go:455

**Intended behavior**

This function updates the decision policy of a group policy. It validates the new decision policy, ensures it is compatible with the current group state and module configuration, updates the group policy's decision policy field, increments the group policy version, and persists the changes. The function does not affect existing proposals.

**Preconditions**

- The caller must be the group policy admin.
- The group policy must exist.
- The new decision policy must be valid.
- The new decision policy must be compatible with the group's total weight and the module configuration.

**Inputs**

- Admin: The address of the group policy administrator.
- GroupPolicyAddress: The address of the group policy to update.
- DecisionPolicy: The new decision policy.

**Branching behavior**

- If the decision policy cannot be unpacked, the function returns an error.
- If the decision policy fails basic validation, the function returns an error.
- If the caller is not the group policy admin, the function returns an error.
- If the decision policy is incompatible with the group, the function returns an error.

**External call analysis**

The function validates the decision policy against the current group state retrieved from storage.

**Function: UpdateGroupPolicyMetadata(msg \*MsgUpdateGroupPolicyMetadata)**

**Module:** contrib/x/group/keeper/msg\_server.go:493

**Intended behavior**

This function updates the metadata of a group policy. It validates the metadata length, updates the group policy's metadata field, increments the group policy version, and persists the changes. The function does not affect the decision policy or associated proposals.

**Preconditions**

- The caller must be the group policy admin.
- The group policy must exist.
- The metadata length must not exceed the configured maximum.

**Inputs**

- Admin: The address of the group policy administrator.
- GroupPolicyAddress: The address of the group policy to update.
- Metadata: The new metadata string.

**Branching behavior**

- If the metadata length exceeds the limit, the function returns an error.
- If the caller is not the group policy admin, the function returns an error.

**External call analysis**

The function does not make external calls beyond validation and state updates.

## 4.4. Proposal functions

Here is an overview of the proposal functions.

**Function: SubmitProposal(msg \*MsgSubmitProposal)**

**Module:** contrib/x/group/keeper/msg\_server.go:515

**Intended behavior**

This function allows group members to submit a new proposal containing arbitrary messages for execution. It validates that all proposers are members of the group, ensures all message signers equal the group policy address, validates the decision policy against the current group state, creates the proposal with status PROPOSAL\_STATUS\_SUBMITTED, sets the voting period end time, and stores the proposal. If the metadata contains a JSON-encoded ProposalMetadata object, the function validates that the metadata title and summary match the proposal title and summary. If the Exec field is set to EXEC\_TRY, the function automatically casts yes votes for all proposers and attempts immediate execution.

**Preconditions**

- The proposers list must not be empty.
- All proposer addresses must be valid.
- All proposers must be members of the group.
- The group policy must exist.
- All message signers must equal the group policy address.

- The decision policy must be valid for the current group state.
- The title, summary, and metadata length must not exceed configured limits.
- If metadata contains a JSON ProposalMetadata object, the metadata title must match the proposal title and the metadata summary must match the proposal summary.

### Inputs

- GroupPolicyAddress: The address of the group policy.
- Proposers: The list of addresses submitting the proposal.
- Messages: The list of messages to execute if the proposal passes.
- Metadata: Arbitrary metadata for the proposal.
- Title: The proposal title.
- Summary: The proposal summary.
- Exec: An enum indicating whether to attempt immediate execution.

### Branching behavior

- If the proposers list is empty, the function returns an error.
- If any proposer address is invalid, the function returns an error.
- If any proposer is not a group member, the function returns an error.
- If the title, summary, or metadata length exceeds limits, the function returns an error.
- If metadata contains ProposalMetadata with mismatched title or summary, the function returns an error.
- If any message signer is not the group policy address, the function returns an error.
- If the decision policy is invalid, the function returns an error.
- If Exec is set to EXEC\_TRY and voting or execution fails, the function returns an error but the proposal is still created.

### External call analysis

The function validates message signers through the codec, checks group membership through the ORM, and potentially calls Vote and Exec if Exec is set to EXEC\_TRY.

### Function: Vote(msg \*MsgVote)

**Module:** contrib/x/group/keeper/msg\_server.go:701

### Intended behavior

This function allows a group member to cast a vote on a submitted proposal. It validates the vote option, ensures the proposal is in the submitted status, checks that the voting period has not ended, verifies the voter is a group member, creates a new vote record, and stores it. The ORM ensures each voter can only vote once per proposal by returning an error if a vote already exists. If the Exec field is set to EXEC\_TRY, the function attempts immediate execution after the vote is cast.

### Preconditions

- The proposal must exist.
- The proposal status must be PROPOSAL\_STATUS\_SUBMITTED.
- The voting period must not have ended.
- The voter must be a member of the group.
- The voter must not have already voted on this proposal.
- The vote option must be valid.
- The metadata length must not exceed the configured maximum.
- The voter address must be valid.

#### Inputs

- ProposalId: The ID of the proposal to vote on.
- Voter: The address of the voter.
- Option: The vote option (yes, no, abstain, no with veto).
- Metadata: Arbitrary metadata for the vote.
- Exec: An enum indicating whether to attempt immediate execution.

#### Branching behavior

- If the proposal ID is zero, the function returns an error.
- If the vote option is unspecified or invalid, the function returns an error.
- If the metadata length exceeds the limit, the function returns an error.
- If the voter address is invalid, the function returns an error.
- If the proposal does not exist, the function returns an error.
- If the proposal status is not submitted, the function returns an error.
- If the voting period has ended, the function returns an error.
- If the voter is not a group member, the function returns an error.
- If the voter has already voted, the function returns an error.
- If Exec is set to EXEC\_TRY and execution fails, the function returns an error.

#### External call analysis

The function checks group membership through the ORM and potentially calls Exec if Exec is set to EXEC\_TRY.

#### Function: WithdrawProposal(msg \*MsgWithdrawProposal)

**Module:** contrib/x/group/keeper/msg\_server.go:659

#### Intended behavior

This function allows a proposal to be withdrawn before execution. It can be called by either the group policy admin or any of the proposal's proposers. The function validates that the proposal is in the submitted status, updates the proposal status to PROPOSAL\_STATUS\_WITHDRAWN, and persists the change. The function does not prune votes or delete the proposal from state.

**Preconditions**

- The proposal must exist.
- The proposal status must be PROPOSAL\_STATUS\_SUBMITTED.
- The caller must be either the group policy admin or one of the proposers.
- The address must be valid.

**Inputs**

- ProposalId: The ID of the proposal to withdraw.
- Address: The address of the caller (admin or proposer).

**Branching behavior**

- If the proposal ID is zero, the function returns an error.
- If the address is invalid, the function returns an error.
- If the proposal does not exist, the function returns an error.
- If the proposal status is not submitted, the function returns an error.
- If the caller is neither the group policy admin nor a proposer, the function returns an error.

**External call analysis**

The function does not make external calls beyond address validation and state retrieval.

**Function: Exec(msg \*MsgExec)**

**Module:** contrib/x/group/keeper/msg\_server.go:826

**Intended behavior**

This function executes the messages from a proposal that has been accepted by the decision policy. If the proposal is still in submitted status, the function first performs tallying to determine the result. If the proposal is accepted and has not already been executed successfully, the function executes the messages in a cached context. If execution succeeds, the cached context is flushed and the proposal is pruned from state. If execution fails, the proposal remains in state with the executor result set to failure. The function enforces the minimum execution period by rejecting execution attempts before the minimum time has elapsed since proposal submission. The function also enforces the maximum execution period by rejecting execution attempts after the voting period end plus the maximum execution period.

**Preconditions**

- The proposal must exist.
- The proposal status must be either PROPOSAL\_STATUS\_SUBMITTED or PROPOSAL\_STATUS\_ACCEPTED.
- If the proposal status is submitted, the voting period may still be active.
- The current block time must be at or after the proposal submission time plus the

minimum execution period.

- The current block time must be before the proposal voting period end plus the maximum execution period.

### Inputs

- `ProposalId`: The ID of the proposal to execute.
- `Executor`: The address of the executor (not validated, used only for context).

### Branching behavior

- If the proposal ID is zero, the function returns an error.
- If the proposal does not exist, the function returns an error.
- If the proposal status is not submitted or accepted, the function returns an error.
- If the current time is before the minimum execution date, the function returns an error.
- If the current time is after the expiry date, the function returns an error.
- If the proposal status is submitted, the function performs tallying.
- If the proposal is not accepted after tallying, the function returns without executing.
- If the proposal has already been executed successfully, the function returns without executing.
- If message execution fails, the proposal executor result is set to failure and the proposal remains in state.
- If message execution succeeds, the proposal is pruned from state.

### External call analysis

The function calls message handlers through the router, which may invoke other modules. The execution occurs in a cached context to ensure atomicity. The function also calls the tally logic if the proposal is still in submitted status.

## 4.5. Decision policy evaluation

Here is an overview of the decision policy evaluation functions.

### Function: `Tally(ctx sdk.Context, p Proposal, groupID uint64)`

**Module:** `contrib/x/group/keeper/tally.go:15`

### Intended behavior

This function computes the tally result for a proposal by iterating through all votes and accumulating vote counts weighted by the voter's current group membership weight. If a voter has left the group since voting, their vote is skipped. If the proposal has already been finalized with status accepted or rejected, the function returns the previously stored final tally result without recomputing. The function does not modify the proposal or any other state.

### Preconditions

- The proposal must exist.
- The group must exist.
- All votes reference valid group members at the time they were cast.

### Inputs

- `ctx`: The SDK context.
- `p`: The proposal to tally.
- `groupID`: The ID of the group associated with the proposal.

### Branching behavior

- If the proposal status is accepted or rejected, the function returns the final tally result without recomputing.
- For each vote, if the voter is not a current group member, the vote is skipped.
- For each vote, if retrieving the group member fails for reasons other than being not found, the function returns an error.

### External call analysis

The function retrieves votes from the vote table and group members from the group member table. It does not make external module calls.

## Function: `ThresholdDecisionPolicy.Allow(tallyResult TallyResult, totalPower string)`

**Module:** `contrib/x/group/types.go:82`

### Intended behavior

This function determines whether a proposal passes based on a threshold decision policy. The real threshold is the minimum of the configured threshold and the group's total weight, which accounts for cases where the total weight has decreased below the threshold. If the yes count is greater than or equal to the real threshold, the proposal is allowed and the result is final. If the yes count plus all undecided votes is less than the real threshold, the proposal is rejected and the result is final. Otherwise, the result is not final and voting continues.

### Preconditions

- The threshold must be a positive decimal.
- The tally result counts must be nonnegative decimals.
- The total power must be a nonnegative decimal.

### Inputs

- `tallyResult`: The current tally result containing yes, no, abstain, and veto counts.
- `totalPower`: The total voting weight of the group.

**Branching behavior**

- If the yes count is greater than or equal to the real threshold, the function returns allow true and final true.
- If the yes count plus undecided votes is less than the real threshold, the function returns allow false and final true.
- Otherwise, the function returns allow false and final false.

**External call analysis**

The function does not make external calls. It performs decimal arithmetic on the input values.

**Function: PercentageDecisionPolicy.Allow(tallyResult TallyResult, totalPower string)**

**Module:** contrib/x/group/types.go:199

**Intended behavior**

This function determines whether a proposal passes based on a percentage decision policy. If the total power is zero, the proposal is rejected and the result is final. The function computes the yes percentage by dividing the yes count by the total power. If the yes percentage is greater than or equal to the configured percentage, the proposal is allowed and the result is final. If the yes count plus all undecided votes divided by the total power is less than the configured percentage, the proposal is rejected and the result is final. Otherwise, the result is not final and voting continues.

**Preconditions**

- The percentage must be a positive decimal between zero and one.
- The tally result counts must be nonnegative decimals.
- The total power must be a nonnegative decimal.

**Inputs**

- `tallyResult`: The current tally result containing yes, no, abstain, and veto counts.
- `totalPower`: The total voting weight of the group.

**Branching behavior**

- If the total power is zero, the function returns allow false and final true.
- If the yes percentage is greater than or equal to the configured percentage, the function returns allow true and final true.
- If the yes count plus undecided votes divided by total power is less than the configured percentage, the function returns allow false and final true.
- Otherwise, the function returns allow false and final false.

**External call analysis**

The function does not make external calls. It performs decimal arithmetic on the input values.

## 4.6. Background tasks

Here is an overview of the background tasks.

### Function: `PruneProposals(ctx sdk.Context)`

**Module:** `contrib/x/group/keeper/keeper.go:378`

#### Intended behavior

This function is called during the end block phase to prune proposals that have expired. A proposal is considered expired if its voting period end plus the maximum execution period is before the current block time. The function retrieves all proposals with voting period end before the expiry cutoff, deletes them from the proposal table, and emits a `EventProposalPruned` event for each pruned proposal. The function does not prune votes associated with the proposals, as they are expected to have been pruned already during finalization.

#### Preconditions

- The function is called during the end block phase.
- The maximum execution period is configured.

#### Inputs

- `ctx`: The SDK context containing the current block time.

#### Branching behavior

- If retrieving proposals by the voting period end fails, the function returns nil (not an error).
- For each expired proposal, if deletion fails, the function returns an error.
- For each expired proposal, if event emission fails, the function returns an error.

#### External call analysis

The function retrieves proposals from the proposal table and emits events. It does not call other modules.

### Function: `TallyProposalsAtVPEnd(ctx sdk.Context)`

**Module:** `contrib/x/group/keeper/keeper.go:405`

#### Intended behavior

This function is called during the end block phase to tally and finalize proposals whose voting period has ended. The function retrieves all proposals with a voting period end at or before the current block time. For proposals with status aborted or withdrawn, the function prunes the proposal and its votes. For proposals with status submitted, the function performs tallying, determines the result based on the decision policy, updates the proposal status to accepted or

rejected, stores the final tally result, prunes the votes, and emits a tally event. For proposals with status accepted or rejected, the function does nothing.

**Preconditions**

- The function is called during the end block phase.
- All proposals have valid associated group policies and groups.
- All decision policies are valid.

**Inputs**

- `ctx`: The SDK context containing the current block time.

**Branching behavior**

- If retrieving proposals by the voting period end fails, the function returns nil (not an error).
- For each proposal, if retrieving the group policy fails, the function returns an error.
- For each proposal, if retrieving the group fails, the function returns an error.
- If the proposal status is aborted or withdrawn, the function prunes the proposal and votes.
- If the proposal status is submitted, the function performs tallying and updates the proposal.
- If tallying fails, the function prunes votes, sets the status to rejected, and emits a tally error event.
- If the result is final or the voting period has ended, the function prunes votes and updates the final tally result.

**External call analysis**

The function calls the tally logic, retrieves group and policy information, and emits events. It does not call external modules.

## 5. Assessment Results

During our assessment on the scoped Cosmos Group module, we discovered eight findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

---

### 5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.