Zellic

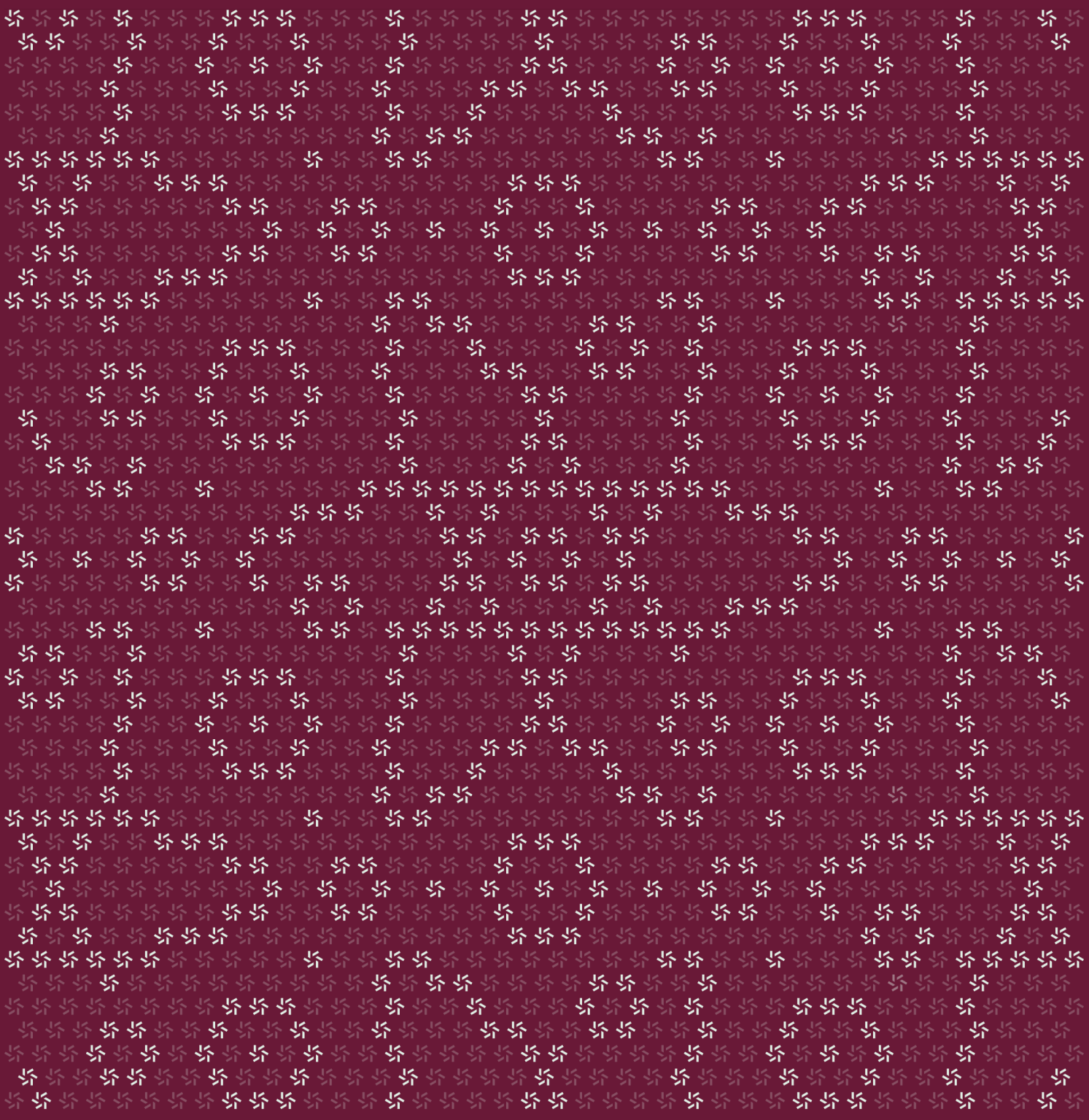
Prepared for

Prepared by
Hojung Han
Jaeeu Kim
Zellic

November 27, 2024

Cosmos SDK Liquid Stake Module

Cosmos Application Security Assessment



Contents

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 ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for The Hub team at Informal Systems from November 12th to November 20th, 2024. During this engagement, Zellic reviewed Cosmos SDK Liquid Stake 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:

- Are users able to get more token or share than they are entitled?
- Can deposits or withdrawals be manipulated to unfairly reward or penalize users?
- Is it possible for any role to lock user funds?

1.3. Non-goals and Limitations

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

- Other modules that are not within this audit's scope
- Front-end components
- Infrastructure relating to the project
- Key custody

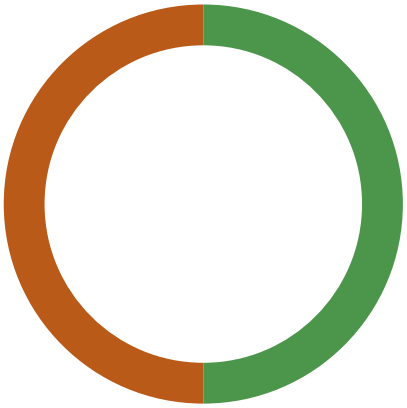
1.4. Results

During our assessment on the scoped Cosmos SDK Liquid Stake Module modules, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of The Hub team at Informal Systems in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	0
Low	1
Informational	0



2. Introduction

2.1. About Cosmos SDK Liquid Stake Module

The Hub team at Informal Systems contributed the following description of Cosmos SDK Liquid Stake Module:

> The Cosmos SDK branch used by the Cosmos Hub includes extensions that enable liquid staking. While implemented as changes to existing modules, these additional features are hereinafter referred to as the liquid staking module (LSM). In a nutshell, LSM enables the creation of a semi-fungible liquid staking primitive, i.e., LSM shares. LSM shares are derivatives of the delegation shares. They are tied to a delegator and a validator pair and they represent the underlying delegation shares. By issuing LSM shares, the underlying staked ATOM can become "liquid" while still being slashable. The LSM shares are tokens that can be used in various DeFi protocols and transferred between users or between chains via IBC. LSM shares are not fungible (as they are tied to a delegator/validator pair) and are issued by the Hub directly and thus don't depend on the security of any entity other than the Cosmos Hub itself.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Cosmos SDK Liquid Stake Module Modules

Type	Cosmos
Platform	Cosmos
Target	Cosmos SDK
Repository	https://github.com/cosmos/cosmos-sdk ↗
Version	`ec58adf8e2fa4149c093d871d2fd398ca508525a`

Programs	`x/staking/*`
	`x/distribution/*`
	`x/slashing/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.1 person-weeks. The assessment was conducted by two consultants over the course of 1.4 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Hojung Han
Engineer
hojung@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 12, 2024	Start of primary review period
November 13, 2024	Kick-off call
November 20, 2024	End of primary review period

3. Detailed Findings

3.1. Accounting validator bonds share could be broken

Target	x/staking/keeper/msg_server.go		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `ValidatorBond` function sets the `delegation.ValidatorBond` to `true` and increases the `validator.ValidatorBondShares` by the delegation shares. If once the delegation is marked as `ValidatorBond`, `ValidatorBondShares` will be increased or decreased based on the delegation shares. The `ValidatorBondShares` is used to account for the validator bond shares that affect the liquid shares of the validator.

```
func (k msgServer) ValidatorBond(goCtx context.Context,
    msg *types.MsgValidatorBond) (*types.MsgValidatorBondResponse, error) {
    // ...
    if !delegation.ValidatorBond {
        delegation.ValidatorBond = true
        k.SetDelegation(ctx, delegation)
        validator.ValidatorBondShares =
    validator.ValidatorBondShares.Add(delegation.Shares)
    // ...

func (k msgServer) Delegate(ctx context.Context, msg *types.MsgDelegate)
    (*types.MsgDelegateResponse, error) {
    // ...
        if delegation.ValidatorBond {
        if err := k.IncreaseValidatorBondShares(ctx, valAddr, newShares);
    err != nil {
            return nil, err
        }
    }
    // ...

func (k msgServer) Undelegate(ctx context.Context, msg *types.MsgUndelegate)
    (*types.MsgUndelegateResponse, error) {
    // ...
    if delegation.ValidatorBond {
        if err := k.SafelyDecreaseValidatorBond(ctx, addr, shares);
    err != nil {
            return nil, err
```

```
        }
    }
    // ...
```

In current implementation, the `TokenizeShares` function will revert if the delegation is for a validator bond. However, the `ValidatorBond` function does not check if the part of delegation shares are already tokenized or if other users could send tokens to this delegator. This tokenized share is not accounted for in the `ValidatorBondShares`.

In this case, if the delegator redeems the tokenized shares, its unbonded and undelegatable amount will be increased by the tokenized shares not accounted for in the `ValidatorBondShares`. This will result in a balance mismatch between the `ValidatorBondShares` and the actual unbonded and undelegatable amount.

```
func (k msgServer) TokenizeShares(goCtx context.Context,
    msg *types.MsgTokenizeShares) (*types.MsgTokenizeSharesResponse, error) {
    // ...
    if delegation.ValidatorBond {
        return nil, types.ErrValidatorBondNotAllowedForTokenizeShare
    }
    // ...
```

Impact

Using the following scenario, a malicious user can manipulate `ValidatorBondShares` to become a negative value.

1. The malicious user calls `Delegate` with 2,000 stake.
2. The malicious user calls `TokenizeShares` with 1,000 stake.
3. The malicious user calls `ValidatorBond` with 1,000 stake.
4. The malicious user calls `RedeemTokensForShares` with 1,000 stake.
5. The malicious user calls `Undelegate` with 2,000 stake.

As a result, the `ValidatorBondShares` will be -1,000 compared to the initial `ValidatorBondShares`.

This results in a permanent balance mismatch and can even lead to negative values. Once the `ValidatorBondShares` becomes negative, the function `SafelyIncreaseValidatorLiquidShares` and `SafelyDecreaseValidatorBond`, which are used in delegate or undelegate, will always fail. Ultimately, this undermines the stability of the protocol.

Recommendations

In a situation where a delegator holds validator bond shares, the validator bond shares should increase when redeeming tokenized shares from the delegator.

Remediation

The issue has been patched in commit 2436420 ↗.

3.2. Function `BeforeTokenizeShareRecordRemoved` does not work as expected

Target	x/staking/keeper/msg_server.go		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

In the `RedeemTokensForShares` function, when a user redeems all tokens, the delegation is deleted through unbonding, and the tokenized record is removed. Before the record is removed, the hook function `BeforeTokenizeShareRecordRemoved` is called, which triggers `WithdrawSingleShareRecordReward` to claim the reward.

```
// x/staking/keeper/msg_server.go
func (k msgServer) RedeemTokensForShares(goCtx context.Context,
    msg *types.MsgRedeemTokensForShares)
    (*types.MsgRedeemTokensForSharesResponse, error) {
    // ...
    returnAmount, err := k.Unbond(ctx, record.GetModuleAddress(), valAddr,
    shares)
    // ...

    // Note: since delegation object has been changed from unbond call, it
    gets latest delegation
    _, err = k.GetDelegation(ctx, record.GetModuleAddress(), valAddr)
    if err != nil && !errors.Is(err, types.ErrNoDelegation) {
        return nil, err
    }

    // this err will be ErrNoDelegation
    if err != nil {
        if k.hooks != nil {
            if err := k.hooks.BeforeTokenizeShareRecordRemoved(ctx,
    record.Id); err != nil {
                return nil, err
            }
        }
        err = k.DeleteTokenizeShareRecord(ctx, record.Id)
        if err != nil {
            return nil, err
        }
    }
```

```
    // ...

// x/distribution/keeper/hooks.go
func (h Hooks) BeforeTokenizeShareRecordRemoved(ctx context.Context, recordID
    uint64) error {
    err := h.k.WithdrawSingleShareRecordReward(ctx, recordID)
    if err != nil {
        h.k.Logger(ctx).Error(err.Error())
    }
    return err
}
```

However, in the `WithdrawSingleShareRecordReward` function, it checks `delegationFound`, which is always `false` because the hook is called when `ErrNoDelegation` is encountered. As a result, the reward-claiming process is never executed through this path.

```
// x/distribution/keeper/keeper.go
func (k Keeper) WithdrawSingleShareRecordReward(ctx context.Context, recordID
    uint64) error {
    // ...
    delegationFound := true
    _, err = k.stakingKeeper.Delegation(ctx, record.GetModuleAddress(),
    valAddr)
    if err != nil {
        if !goerrors.Is(err, stakingtypes.ErrNoDelegation) {
            return err
        }

        delegationFound = false
    }

    sdkCtx := sdk.UnwrapSDKContext(ctx)
    if validatorFound && delegationFound {
        // withdraw rewards into reward module account and send it to reward
    owner
        cacheCtx, write := sdkCtx.CacheContext()
        _, err = k.WithdrawDelegationRewards(cacheCtx,
    record.GetModuleAddress(), valAddr)
        // ...
```

Impact

In current codebase, the `Unbond` function calls the `BeforeDelegationSharesModified` hook, and this function will claim rewards before removing the delegation. Nevertheless, the `BeforeTok-`

`enizeShareRecordRemoved` hook does not work as expected, and the `WithdrawDelegationRewards` is not reachable using this path.

Recommendations

Remove duplicate and unreachable code. If there is a possibility that the unbond hook does not exist, the call position should be adjusted to claim the rewards before the delegation is removed while redeeming all tokens.

Remediation

This issue has been acknowledged by The Hub team at Informal Systems.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Tokenized share rewards and token denomination in staking module

This discussion revolves around two key questions raised about the design and implementation of tokenized shares in the staking module:

1. **Reward distribution for tokenized shares.** It was confirmed that rewards are sent to the `record owner` of tokenized shares, irrespective of the wallet holding the tokens. This behavior is intentional and allows tokenized shares to be locked in the module (e.g., Hydro) while still receiving staking rewards. However, tokenized shares are not fungible and are unlikely to be swapped on a DEX due to differing values based on slashing risks.

2. **Token denomination and liquidity fragmentation.** Tokenized shares generate different denominations for each tokenization instance, which might seem to fragment liquidity. However, this design reflects the nonfungibility of tokenized shares. It was noted that users could consolidate tokenizations into a single transaction to avoid fragmentation, and this does not pose significant challenges for Liquid Staking Module use cases.

The Hub team at Informal Systems clarified these behaviors as intentional and aligned with the system's goals.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Staking

The staking module processes various messages for validator and delegation management. Each message type has specific security considerations and validation requirements.

Additionally, within the scope of this audit, unlike the existing Cosmos SDK staking module, the system enhances the utility of locked-up staked assets by tokenizing them, even while they remain staked.

The following section outlines the recent updates to the staking module.

States

Added

- `TotalLiquidStakedTokens`
  - Tracks the total amount of liquid staked tokens to monitor progress against the `GlobalLiquidStakingCap`.
- `PendingTokenizeShareAuthorizations`
  - Stores a queue of addresses undergoing reactivation/unlocking for tokenized shares.

Modified

Two items have been added to `Validator`.

- `ValidatorBondShares`
  - Represents the number of shares self-bonded by the validator.
- `LiquidShares`
  - Represents the number of shares either tokenized or owned by a liquid-staking provider.

BeginBlock

Added

- `RemoveExpiredTokenizeShareLocks`
  - Finds and releases all expired `TokenizeShareLocks` up to the current block time.

Chain parameters

Added

- `ValidatorBondFactor`
  - Defines the factor for validator bonds.
- `GlobalLiquidStakingCap`
  - Sets the global cap for liquid staking.
- `ValidatorLiquidStakingCap`
  - Sets the liquid-staking cap for individual validators.

Messages

Added

- `MsgTokenizeShares`
  - Tokenizes a specified amount from an existing delegation.
  - Creates a new `TokenizeShareRecord`.
  - Unbonds the specified amount from the existing delegation.
  - Transfers the unbonded tokens to the `ModuleAddress` specified in the `TokenizeShareRecord`.
  - Assigns tokenized shares to the delegator corresponding to the unbonded tokens.
  - Creates a new delegation between the `ModuleAddress` in the `TokenizeShareRecord` and the validator.
- `MsgRedeemTokensForShares`
  - Redeems tokenized shares to reclaim the originally delegated assets.
  - Unbonds the specified amount from the delegation between the `ModuleAddress` and validator. Deletes the delegation data if the entire amount is unbonded.
  - Transfers the share tokens to the `NotBondedPool` and burns them.
  - Transfers the corresponding original tokens to the delegator and updates delegation information between the validator and delegator.
- `MsgTransferTokenizeShareRecord`
  - Transfers ownership of tokenized delegation to another user.
  - Removes the relationship between the existing `TokenizeShareRecord` and the current owner.
  - Creates a new relationship between the `TokenizeShareRecord` and the new owner.
- `MsgEnableTokenizeShares`
  - Initiates reauthorization for tokenization for a delegator's address.
  - Adds a lock to `PendingTokenizeShareAuthorizations` that will be released after the unbonding period to allow tokenization.
- `MsgDisableTokenizeShares`

    - Disables tokenization for a delegator's address.
    - Removes expired locks from `PendingTokenizeShareAuthorizations` if they exist.
    - Creates a new permanent lock and adds it to `PendingTokenizeShareAuthorizations`.
- `MsgUnbondValidator`
    - Transitions the validator from the `Bonded` state to `Unbonding` state.
    - Sets the validator to a jail state and prepares it for unbonding.
- `MsgValidatorBond`
    - Designates a specific delegation as a validator bond, allowing the validator to receive additional liquid-staking delegations.
    - Marks a specific delegation as the validator's `ValidatorBond`.

Modified

- `MsgDelegate`
    - Increases `TotalLiquidStakedTokens` by the amount of tokens delegated, if the delegator is a liquid staker.
    - Increases the validator's `LiquidShares` by the number of shares delegated, if the delegator is a liquid staker.
    - Increases the validator's `ValidatorShares` by the newly delegated shares, if the existing delegation's `ValidatorBond` is true.
- `MsgBeginRedelegate`
    - Deducts the redelegated amount converted to validator shares from `ValidatorBondShares`, if the source delegation's `ValidatorBond` is true.
    - Adds the shares calculated by the destination validator for the redelegated amount to the destination validator's `LiquidShares` and removes it from the source validator's `LiquidShares`, if the delegator is a liquid staker.
    - Increases the destination validator's `ValidatorBondShares` by the redelegated amount converted to shares, if the destination delegation's `ValidatorBond` is true.
- `MsgUndelegate`
    - Deducts the undelegated amount converted to validator shares from `ValidatorBondShares`, if the specified delegation's `ValidatorBond` is true.
    - Decreases `TotalLiquidStakedTokens` by the amount of tokens undelegated, if the delegator is a liquid staker.
    - Decreases the validator's `LiquidShares` by the number of shares undelegated, if the delegator is a liquid staker.
- `MsgCancelUnbondingDelegation`
    - Increases `TotalLiquidStakedTokens` by the amount of tokens for which unbonding is canceled, if the delegator is a liquid staker.
    - Increases the validator's `LiquidShares` by the number of shares for which unbonding is canceled, if the delegator is a liquid staker.
    - Increases the validator's `ValidatorShares` by the number of shares for which unbonding is canceled, if the existing delegation's `ValidatorBond` is true.

Invariants

This section describes new/removed invariants compared to the existing staking module.

Validator-management messages

- `MsgEditValidator`
  - Validation for `msg.MinSelfDelegation` has been removed (as it was not actually used).
- `MsgUnbondValidator`
  - The `msg.ValidatorAddress` must have been registered through a prior `MsgCreateValidator`.
  - The `msg.ValidatorAddress` must not already be in jail.
- `MsgValidatorBond`
  - The `msg.ValidatorAddress` must have been registered through a prior `MsgCreateValidator`.
  - The `msg.DelegatorAddress` must have delegated self-staked tokens to the validator in advance.
  - The `msg.DelegatorAddress` must not be a liquid staker.
  - The delegation between `msg.ValidatorAddress` and `msg.DelegatorAddress` must not already be set as a validator bond.

Share-tokenization messages

- `MsgTokenizeShares`
  - The `msg.Amount.Amount` must be greater than zero and not nil.
  - The `msg.Amount.Denom` must match the regular expression `[a-zA-Z][a-zA-Z0-9/:._-]{2,127}`.
  - The `msg.Amount.Amount` cannot exceed the number of tokens owned by the delegator for `msg.Amount.Denom`.
  - The tokenize-share lock for `msg.DelegatorAddress` must not be in `TOKENIZE_SHARE_LOCK_STATUS_LOCKED` or `TOKENIZE_SHARE_LOCK_STATUS_LOCK_EXPIRING`.
  - A delegation must exist between the validator specified by `msg.ValidatorAddress` and the delegator specified by `msg.DelegatorAddress`.
  - The delegation's `ValidatorBond` must not be true.
  - The `msg.Amount.Denom` must match the denom specified in the chain parameters.
  - If the delegator specified by `msg.DelegatorAddress` is a vesting account, the tokenized shares must not exceed the amount available for use after the vesting period.

    - The delegator specified by `msg.DelegatorAddress` must not be in the middle of a redelegation process.
    - The shares to be unbonded by the delegator specified by `msg.DelegatorAddress` must not exceed the delegation's total shares.
- `MsgRedeemTokensForShares`
    - The `msg.Amount.Amount` must be greater than zero and not nil.
    - The `msg.Amount.Denom` must match the regular expression `[a-zA-Z][a-zA-Z0-9/:._-]{2,127}`.
    - The `msg.Amount.Amount` cannot exceed the number of tokens owned by the delegator for `msg.Amount.Denom`.
    - A `TokenizeShareRecord` related to `msg.Amount.Denom` must exist.
    - The validator in the `TokenizeShareRecord` must have been registered through a prior `MsgCreateValidator`.
    - A delegation relationship must exist between the `Module Address` in the `TokenizeShareRecord` and the validator.
    - The shares calculated for the `msg.Amount.Amount` based on the validator in the `TokenizeShareRecord` must not be zero.
    - The shares to be unbonded by the delegator specified by `msg.DelegatorAddress` must not exceed the delegation's total shares.
- `MsgTransferTokenizeShareRecord`
    - The `msg.Sender` must be the `record.Owner`.
- `MsgEnableTokenizeShares`
    - The tokenize-share lock related to `msg.DelegatorAddress` must not be in `TOKENIZE_SHARE_LOCK_STATUS_UNLOCKED`.
    - The tokenize-share lock related to `msg.DelegatorAddress` must not be in `TOKENIZE_SHARE_LOCK_STATUS_LOCK_EXPIRING`.
- `MsgDisableTokenizeShares`
    - The tokenize-share lock related to `msg.DelegatorAddress` must not be in `TOKENIZE_SHARE_LOCK_STATUS_LOCKED`.

5.2. Distribution

The distribution module is responsible for distributing rewards (block rewards, fees, etc.) between validators and delegators. It handles validator commissions, distributes rewards to delegators, and manages community-pool funds, implementing the network's reward-distribution mechanism.

The following section outlines the recent updates about distribution module.

Messages

Added

- `MsgWithdrawTokenizeShareRecordReward`
    - `TokenizeShareRecord` owners can withdraw rewards from the

`TokenizeShareRecord`.

- `MsgWithdrawAllTokenizeShareRecordReward`
  - `TokenizeShareRecord` owners can withdraw rewards from every `TokenizeShareRecord`.

Invariants

Added

- `MsgWithdrawTokenizeShareRecordReward`
  - The validator of the specified `TokenizeShareRecord` must be a currently registered validator.
  - A delegation must exist between the `Module Address` of the specified `TokenizeShareRecord` and the validator.
  - The `Owner` of the specified `TokenizeShareRecord` must match the owner (signer) included in the transaction.
- `MsgWithdrawAllTokenizeShareRecordReward`
  - The validator of the specified `TokenizeShareRecord` must be a currently registered validator.
  - A delegation must exist between the `Module Address` of the specified `TokenizeShareRecord` and the validator.

5.3. Slashing

The slashing module enables Cosmos SDK–based blockchains to disincentivize any attributable action by a protocol-recognized actor with value at stake by penalizing them (slashing). The following section outlines the recent updates about slashing module .

Messages

Modified

- `MsgUnjail`
  - Removed checking for the minimum `SelfDelegation` when validator is unjailed.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed in git repository (https://github.com/cosmos/cosmos-sdk/tree/v0.50.10-lsm).

During our assessment of the scoped Cosmos SDK Liquid Stake Module, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

6.1. 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.