Zellic

Prepared for
Bob DeFarmer & WhammerGuy
Garen
Silo

Prepared by
Nipun Gupta
Ayaz Mammadov
Zellic

March 28, 2024

Silo Staking

Smart Contract 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 Silo from March 11th to March 27th, 2024. During this engagement, Zellic reviewed Silo Staking'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:

- Is it possible to manipulate the exchange rate of utoken and ustake token in a manner that could benefit an attacker?
- Can an attacker potentially steal rewards or delegated staked tokens from other users?
- Is it feasible for users to withdraw their tokens before the unbonding period of their batch has elapsed?
- Could an attacker render it impossible for other users to withdraw their delegated staked tokens?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

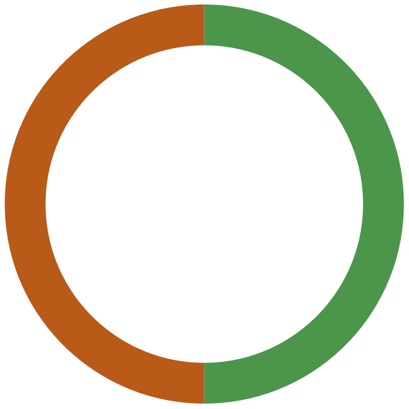
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Silo Staking contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	0
🟩 Low	1
⬜ Informational	0



2. Introduction

2.1. About Silo Staking

Silo contributed the following description of Silo Staking:

> Silo is building MEV & liquid staking architectures on Sei. Silo's flagship application — iSEI — offers users the ability to deposit their Sei and stake it to earn rewards, while maintaining liquidity across the Defi ecosystem.

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 contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

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:

Silo Staking Contracts

Repository	https://github.com/silostaking/silo-contracts ↗
Version	silo-contracts: `65494fa49e482bbc4f1b9bca3b93565db3d4a248`
Programs	• hub/src/claim.rs • hub/src/constants.rs • hub/src/contract.rs • hub/src/error.rs • hub/src/execute.rs • hub/src/gov.rs • hub/src/helpers.rs • hub/src/lib.rs • hub/src/math.rs • hub/src/queries.rs • hub/src/state.rs • hub/src/types/coins.rs • hub/src/types/gauges.rs • hub/src/types/keys.rs • hub/src/types/mod.rs • hub/src/types/staking.rs • token/src/lib.rs
Type	CosmWasm
Platform	CosmWasm contracts

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-weeks. The assessment was conducted over the course of 2.5 calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Ayaz Mammadov
Engineer
ayaz@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 11, 2024	Kick-off call
March 14, 2024	Start of primary review period
March 27, 2024	End of primary review period

3. Detailed Findings

3.1. Calling reconcile before token distribution from unbonding leads to funds stuck in the contract

Target	execute.rs		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Sei chains, by default, have a limit of seven undelegations at a time per validator-delegator pair. In order to support unbonding requests from many users, the contract bundles unbonding requests together and submits them in batches. The contract submits a batch every three days, such that there are at most seven undelegations at a time with each validator. This three-day interval is defined by the `epoch_period` parameter.

During the three-day period, the contract accepts unbonding requests from users and stores them in an `IndexedMap` data structure under the `unbond_requests` key and the aggregated properties of the pending batch under the `pending_batch` key. Each user's share in the batch is proportional to the amount of iSEI tokens the user requests to burn.

At the end of the three-day period, anyone can invoke the `ExecuteMsg::SubmitUnbond` function to submit the pending batch to be unbonded. The contract calculates the amount of Sei to unbond based on the Sei/iSEI exchange rate at the time, burns the iSEI tokens, and initiates undelegations with the validators.

At the end of the following 21-day unbonding period, anyone can call `ExecuteMsg::Reconcile`, which, depending on the current balance of the contract, directly marks the batches as reconciled if the current balance of Sei is greater than what is expected or deducts the difference between the actual balance of Sei and expected Sei from the batches.

The tokens are also distributed in the first block after the end of the unbonding period (21 days). These tokens are distributed in the endblocker. Here is the relevant code:

```
func EndBlocker(ctx sdk.Context, k keeper.Keeper) []abci.ValidatorUpdate {
    defer telemetry.ModuleMeasureSince(types.ModuleName, time.Now(),
    telemetry.MetricKeyEndBlocker)

    return k.BlockValidatorUpdates(ctx)
}
```

```
func (k Keeper) BlockValidatorUpdates(ctx sdk.Context) []abci.ValidatorUpdate {
        //...
        balances, err := k.CompleteUnbonding(ctx, delegatorAddress, addr)
```

```
func (k Keeper) CompleteUnbonding(ctx sdk.Context, delAddr sdk.AccAddress,
    valAddr sdk.ValAddress) (sdk.Coins, error) {

    //...
    // loop through all the entries and complete unbonding mature entries
    for i := 0; i < len(ubd.Entries); i++ {
        entry := ubd.Entries[i]
        if entry.IsMature(ctxTime) {
            ubd.RemoveEntry(int64(i))
            i--

            // track undelegation only when remaining or truncated shares are
    non-zero
            if !entry.Balance.IsZero() {
                amt := sdk.NewCoin(bondDenom, entry.Balance)
                if err := k.bankKeeper.UndelegateCoinsFromModuleToAccount(
```

```
func (e UnbondingDelegationEntry) IsMature(currentTime time.Time) bool {
    return !e.CompletionTime.After(currentTime)
}
```

As shown above, the tokens are distributed to the contract using the call `k.bankKeeper.UndelegateCoinsFromModuleToAccount` in the endblocker. If the `reconcile` function is called in the same block as the token distribution, it would be executed before the token distribution and `utoken_actual` would be less than `utoken_expected`, and thus the `utoken_to_deduct` amount would be deducted from the batches to be reconciled.

```
    let unlocked_coins = state.unlocked_coins.load(deps.storage)?;
    let utoken_expected_unlocked =
     Coins(unlocked_coins).find(&stake.utoken).amount;

    let utoken_expected = utoken_expected_received + utoken_expected_unlocked;
    let utoken_actual = deps.querier.query_balance(&env.contract.address,
     stake.utoken)?.amount;

    if utoken_actual >= utoken_expected {
        mark_reconciled_batches(&mut batches);
        // println!("here");
        for batch in &batches {
            state.previous_batches.save(deps.storage, batch.id, batch)?;
        }
        let ids = batches.iter().map(|b|
    b.id.to_string()).collect::<Vec<_>>().join(",");
        let event = Event::new("silohub/reconciled")
            .add_attribute("ids", ids)
            .add_attribute("utoken_deducted", "0");
        return Ok(Response::new().add_event(event).add_attribute("action",
    "silohub/reconcile"));
    }

    let utoken_to_deduct = utoken_expected - utoken_actual;
let reconcile_info = reconcile_batches(&mut batches, utoken_to_deduct);
```

However, there is a time check in the `reconcile` function, which filters out the batches with `current_time > b.est_unbond_end_time` to be reconciled:

```
let mut batches = all_batches
    .into_iter()
    .filter(|b| current_time > b.est_unbond_end_time)
    .collect::<Vec<_>>();
```

This means that if `est_unbond_end_time` is t1, the reconcile could only be called at time t1+1, but token distribution would happen at block with timestamp t1. However, there might be a few cases where reconcile and token distribution happen in the same block. In the CosmWasm code, the time is measured in seconds, while in the Sei blockchain, the time is measured in nanoseconds. The time stored in the Rust code truncates the nanoseconds part as shown in the below code:

```
/// Returns seconds since epoch (truncate nanoseconds)
#[inline]
pub fn seconds(&self) -> u64 {
    self.0.u64() / 1_000_000_000
}
```

Assuming that a batch is submitted at epoch timestamp 1000.8, the value of `est_unbond_end_time` stored in the CosmWasm contract would be 1000 + 21days(1814400) = 1815400, and reconcile could be called a second after that, which is 1815401. But as per the Go code, the exact unbonding time to distribute the tokens will be 1815400.8.

Now, let us assume two blocks, B1 at time 1815400.7 and the next block B2 at time 1815401.1 (as the approximate block time of the Sei blockchain is 0.4 seconds). At block B1, neither reconcile could be called as the value of `current_time` will be 1815400 and `est_unbond_end_time` is 1815400, nor will the token distribution happen as 1815400.7 is less than 1815400.8.

At the next block B2, the time will be 1815401.1; at this time, reconcile could be called as `current_time > b.est_unbond_end_time` would be true and token distribution will happen in the endblocker. In this reconcile call, as the tokens have not been distributed yet, the contract would assume that this is due to slashing and thus deduct those Sei from the batches.

Impact

The tokens would be stuck in the contract, and users would not be able to withdraw them unless the contract is migrated and the admin rescues these tokens and distributes them to the users.

Recommendations

We recommend adding a few seconds of delay in the reconcile call so it could not be called in the same block as the token distribution.

Remediation

This issue has been acknowledged by Silo, and a fix was implemented in commit c86bb206 ↗.

3.2. First depositor Issue

Target	execute.rs		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The first depositor issue exists in the context of `bond`, as the amount of tokens received is determined by the exchange rate of the supply and the amount to be bonded. This is compounded by the existence of the `donate` functionality, which allows for the inflation of share prices.

```
pub(crate) fn compute_mint_amount(
    ustake_supply: Uint128,
    utoken_to_bond: Uint128,
    current_delegations: &[Delegation],
) -> Uint128 {
    let utoken_bonded:
    u128 = current_delegations.iter().map(|d| d.amount).sum();
    if utoken_bonded == 0 {
        utoken_to_bond
    } else {
        ustake_supply.multiply_ratio(utoken_to_bond, utoken_bonded)
    }
}
```

Impact

When a new platform is launched, a malicious attacker could mint one share, and when another user attempts to bond, the transaction can be front-run with the `donation` functionality to inflate the share price, leading to a truncation in the calculations. Consequently, the user attempting to bond will not receive their bonded tokens as expected.

Recommendations

Add a check to ensure that the user gets more than zero shares or add an option that allows a user to ensure they get a minimum amount of shares (slippage check).

Remediation

While this issue does not affect the currently deployed contracts, as it exists only during the initial state of the pool, the Silo team acknowledged the finding and implemented a fix in the commit `f74faf45` ↗.

We understand that the fix does not fully mitigate the issue and have recommended to the team to use a small amount of seed liquidity for future deployments of similar contracts. The Silo team has agreed to use seed liquidity for upcoming deployments, effectively mitigating potential similar issues in future deployments.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1. Module: hub/src/execute.rs

Message: `ExecuteMsg::Bond`

This message could be used by anyone to provide utokens and start staking.

The parameters that a user can control are the following:

`receiver` — This is the address to which ustake tokens are minted to. If this address is not provided, the address of `info.sender` is used. `info.funds` — The list of tokens sent along with the message.

The message executes the `bond` function with the `donate` parameter set to false. The function first validates the funds sent to it and then finds the validator with the smallest amount of delegation to delegate tokens to it. Next, it calculates the amount of ustake to be minted using the formula shown below, where `utoken_to_bond` is the amount of utokens supplied by the user:

$$ustake\_to\_be\_minted = \frac{(ustake\_supply)*(utoken\_to\_bond)}{total\_amount\_of\_delegations}$$

Next, it increases the total supply of the ustake tokens and creates an appropriate message to mint these tokens to the `receiver`.

The function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

Message: `execute::check_received_coin_msg`

This message is used to check the difference in coins between a snapshot before a withdraw delegator rewards message and after. There are two snapshots supplied: one for `stake.denom` and one for `stake.utoken`.

The function `add_to_received_coins` is called and supplied with a vector of coins `received_coins`, where the current balance and the snapshot balance are compared and the difference is added to the `received_coins`. The difference is then used to update the `unlocked_coins` state. This callback is relevant for functions that call `burn/delegate` and other messages that affect the token delegations.

The arguments supplied to `check_received_coin_msg` are the snapshots that are taken at the time of the invocation of the contract before delegation-related messages are sent.

Message: `ExecuteMsg::Donate`

This message could be used by anyone to provide utokens to the contract without receiving any ustake in return.

The parameter that a user can control is `info.funds` — the list of tokens sent along with the message.

The message executes the `bond` function with the `donate` parameter set to true. The function first validates the funds sent to it and then finds the validator with the smallest amount of delegation to delegate tokens to it. As the `info.sender` does not receive any ustake in return, neither the function does not calculate the mint amount nor the total supply of ustake is increased.

The function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

Message: `ExecuteMsg::Harvest`

The message could be used by anyone. It calls the `harvest` function, which withdraws the delegation rewards and restakes these rewards.

The parameters that a user can control are the following:

`validators` — The list of validators from which delegation rewards are to be withdrawn for redelegation. `withdrawals` — A vector of a tuple containing `WithdrawType` and `DenomType`. A custom withdrawals vector could only be sent by the operator; therefore, addresses without access control could not provide a custom withdrawals vector. `stages` — A vector of a tuple containing `StageType`, `DenomType`, and `Option<Decimal>` for price and `Option<Uint128>` for max amount. A custom stages vector could only be sent by the operator; therefore, addresses without access control could not provide a custom stages vector.

The message executes the `harvest` function, which first withdraws the delegation rewards from the provided validators. If validators are not provided, all the validators are used. Next, there is an optional callback to withdraw LPs and then to swap these tokens to the required utokens. The function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

After updating the `unlocked_coins`, the function sends a callback message `CallbackMsg::Reinvest` to this contract, which will reinvest the received tokens stored in `unlocked_coins`.

Message: `ExecuteMsg::Rebalance`

The message could only be used by the address with `owner` privileges and is used to rebalance the delegations based on the `delegation_goal`.

The parameter that a user can control is `min_redelegation` — the minimum value that the delegations should contain after rebalancing.

The message executes the `rebalance` function. It calls the `compute_redelegations_for_rebalancing` function, which computes the redelegation moves that will make each validator's delegation the targeted amount as per the `delegation_goal`. The `new_redelegations` returned from the `compute_redelegations_for_rebalancing` call are filtered such that any redelegation amount less than `min_redelegation` are skipped. Next, the function executes the redelegation message.

The `rebalance` function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

Message: `ExecuteMsg::Reconcile`

The message could be used by anyone and is used to reconcile the batches that have not been reconciled yet.

This message does not expect any user parameters.

The message executes the `reconcile` function. This function filters all the batches that have not been reconciled yet and for which the unbonding period has passed. It then checks if the utoken balance of the contract is greater than or equal to the utoken expected. If it is, the filtered batches are marked reconciled. If not, the contract assumes that the validators are slashed, due to which the received utoken amount after the unbonding period is less than expected and deducts the difference in amount evenly from each unreconciled batch.

Message: `execute::reinvest`

This function checks the number of unlocked coins in the same atomic transaction that was harvested from delegator rewards.

It goes over the unlocked coins, checking if they are the relevant stake/bond tokens. In the case of `stake.utoken`, these coins are redelegated through a Cosmos message. In the case of `stake.denom`, these coins are burned. Otherwise, the coin is ignored.

A fee is sent to the protocol contract. It then removes the coins from the `unlocked_coins` that have been redelegated/burned.

Then the exchange rate is updated to be up to date with the newly received rewards.

There are no arguments supplied in this callback.

Message: `ExecuteMsg::SubmitBatch`

The message could be used by anyone and is used to submit a batch for unbonding.

This message does not expect any user parameters.

The message executes the `submit_batch` function, which first verifies that the current time is greater than the unbond start time. Then, it calculates the utoken to unbond based on the total supply of ustake, the amount of ustake to burn during unbonding and the current delegations. The formula used to calculate the utoken to unbond is shown below:

$$\frac{(total\_amount\_of\_delegations)*(ustake\_to\_burn)}{total\_supply\_of\_ustake}$$

Next, it calls the `compute_undelegations` function, which, given the current delegations made to validators and a specific amount of utoken to unstake, computes the undelegations to make such that the delegated amount to each validator is as even as possible.

Then, the function stores the `pending_batch` as the `previous_batches`, initiates a new `pending_batch`, and burns the ustake amount of the batch. The function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

Message: `ExecuteMsg::TuneDelegations`

The message could only be used by the address with `owner` privileges and is used to update the `delegation_goal` depending on the type of the delegation strategy.

This message does not expect any user parameters.

The message executes the `tune_delegations` function. The function would first call `get_wanted_delegations`, which calculates the wanted delegations based on the delegation strategy (the amp + emp gauges) and returns a struct `WantedDelegationsShare` and a boolean. The value of the boolean is false if the delegation strategy is `Uniform`, and true otherwise. If this boolean is true, the returned struct is saved in the `delegation_goal` storage.

Message: `ExecuteMsg::AcceptOwnership`

This message can only be invoked by the newly proposed owner and is used to accept the ownership proposal.

This message does not accept any user parameters; `state.new_owner` is cleared and `state.owner` is updated to the new owner.

Message: `ExecuteMsg::AddValidator`

This message can only be invoked by the owner and is used to add a new validator.

The only controllable parameter is `validator` — the address of the validator to update the list with.

A query is made to the staking module to ensure that the validator exists. The new validator is added if it is not already present in the list.

Message: `ExecuteMsg::Callback`

The callback message dispatches Silo callbacks from previous messages. Only the Silo contract can invoke the callback functionality; these callbacks are results of operations from earlier calls into the Silo contract.

The different subfuctionalities of the callback message are the following:

`CallbackMsg::Reinvest` — This subfunctionality is responsible for reinvesting accrued rewards. It is called from the `harvest` message and ensures that bonded tokens compound. No user parameters are supplied.

`CallbackMsg::WithdrawLps` — This subfunctionality is responsible for removing LP positions if there are any. It is only accessible to `operator` through the `harvest` function, which invokes the callback. The user parameters are the withdrawal steps supplied during `harvest`.

`CallbackMsg::SingleStageSwap` — This subfunctionality is responsible for swapping withdrawal rewards into other tokens. It is only accessible to `operators` through the `harvest` function, which invokes the callback. The user parameters are the swap stages supplied during `harvest`.

`CallbackMsg::CheckReceivedCoin`— This subfunctionality is used by the contract to check the amount of delegator rewards actually withdrawn. It is used as a callback after the withdraw reward message is executed, also called by `harvest`.

Message: `ExecuteMsg::DropOwnershipProposal`

This message can only be invoked by the owner and is used to drop the ownership proposal.

This message does not accept any user parameters; `state.new_owner` is cleared to drop the ownership proposal.

Message: `ExecuteMsg::RemoveValidator`

This message can only be invoked by the owner and is used to remove an existing validator.

The only controllable parameter is `validator` — the address of the validator to remove.

The removal fails if this is the only validator present. Otherwise, redelegations are performed on the remaining validators based on the delegation strategy.

If redelegations are made, then the function also stores the current snapshot of the balance, which is passed to the `CallbackMsg::CheckReceivedCoin` message, and the newly received coins are added to the `unlocked_coins` during the callback.

Message: `ExecuteMsg::TransferOwnership`

This message can only be invoked by the owner and is used to transfer contract ownership.

The only controllable parameter is `new_owner` — the address of the newly proposed owner.

The address of the new owner is updated in `state.new_owner`. However, the ownership does not get changed in this function itself.

Message: `ExecuteMsg::QueueUnbond`

This message can be invoked by anyone and is used to queue staked tokens for unbonding.

The parameters that a user can control are the following:

`receiver` — This is the address for which the shares are added to the unbonding batch. If this address is not provided, the address of `info.sender` is used. `info.funds` — Staked tokens sent along with the message. The denom is validated to be that of the staked token.

The functionality for the message is implemented with the `queue_unbond` handler. The staked tokens sent by the user are added to the current pending batch. The user's shares are then updated in an indexed map, which stores the shares for all receivers.

Moreover, if the current block time exceeds the unbonding start time for the current batch, `ExecuteMsg::SubmitBatch` is invoked. This submits the current pending batch to be unbonded.

Message: `ExecuteMsg::UpdateConfig`

This message can only be invoked by the admin and allows updating the contract configuration.

All the controlled parameters are optional and update the current config if set.

`protocol_fee_contract` — This represents the address of the contract responsible for collecting protocol fees.

`protocol_reward_fee` — This is fee that is applied when staking rewards are reinvested. This needs to be lower than the reward fee cap.

`operator` — Updates the current operator for the contract. The operator can perform certain privileged operations.

`stages_preset` — Represents preset swap configs for multiple tokens.

`withdrawals_preset` — Represents preset config for LP withdrawals in the harvest message.

`allow_donations` — Allows/disallows donations through `ExecuteMsg::Donate`.

`delegation_strategy` — This represents the delegation strategy, defining distribution between validators.

`vote_operator` — Sets an operator for the voting operations.

`chain_config` — Sets config for the contract.

`default_max_spread` — The default max spread for single-stage swaps.

`epoch_period` — This represents how often the unbonding queue is executed.

`unbond_period` — The staking module's unbonding time in seconds.

Message: `ExecuteMsg::Vote`

This message can only be invoked by the vote operator.

The following user parameters are expected:

`proposal_id` — The proposal ID to vote on. `vote` — The chosen vote option.

The function sends a message to the Cosmos governance module, with the chosen proposal ID and vote.

Message: `ExecuteMsg::VoteWeighted`

This message can only be invoked by the vote operator.

The following user parameters are expected:

`proposal_id` — The proposal ID to vote on. `votes` — The list of chosen vote options and their respective weights.

The function sends a message to the Cosmos governance module, with the chosen proposal ID and votes along with their weights.

Message: `ExecuteMsg::WithdrawUnbonded`

This message can only be invoked by users with unclaimed unbond requests.

The parameter that a user can control is `receiver` — this is the address to which the underlying tokens are transferred. If this address is not provided, the address of `info.sender` is used.

The function first fetches all unclaimed unbond requests belonging to the user's address. It then loads the previous batches for each request ID. If the batch has been reconciled and the current block time has exceeded the unbond end time, then the refund amount is computed as follows:

$$utoken\_to\_refund = \frac{(batch\_utoken\_unclaimed)*(request\_shares)}{batch\_total\_shares} “‘$$

This amount is added to the total refund amount. Next, the requested shares are subtracted from the total shares and the utoken to refund are subtracted from the utoken unclaimed. The batch is removed if no remaining shares are left. The unbonding request is removed.

The total refund amount for the underlying token is transferred back to the receiver.

4.2. Module: token/src/execute.rs

Message: `ExecuteMsg::BurnFrom`

The message is disabled and simply reverts the transaction with the following error: "`burn_from` command is disabled".

Message: `ExecuteMsg::Burn`

The message asserts that the `info.sender` of the message is the minter and calls `cw20_execute` with the provided user input. The transaction will revert if the `info.sender` does not have the mint capabilities.

5. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Sei Mainnet.

During our assessment on the scoped Silo Staking contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

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.