Zellic

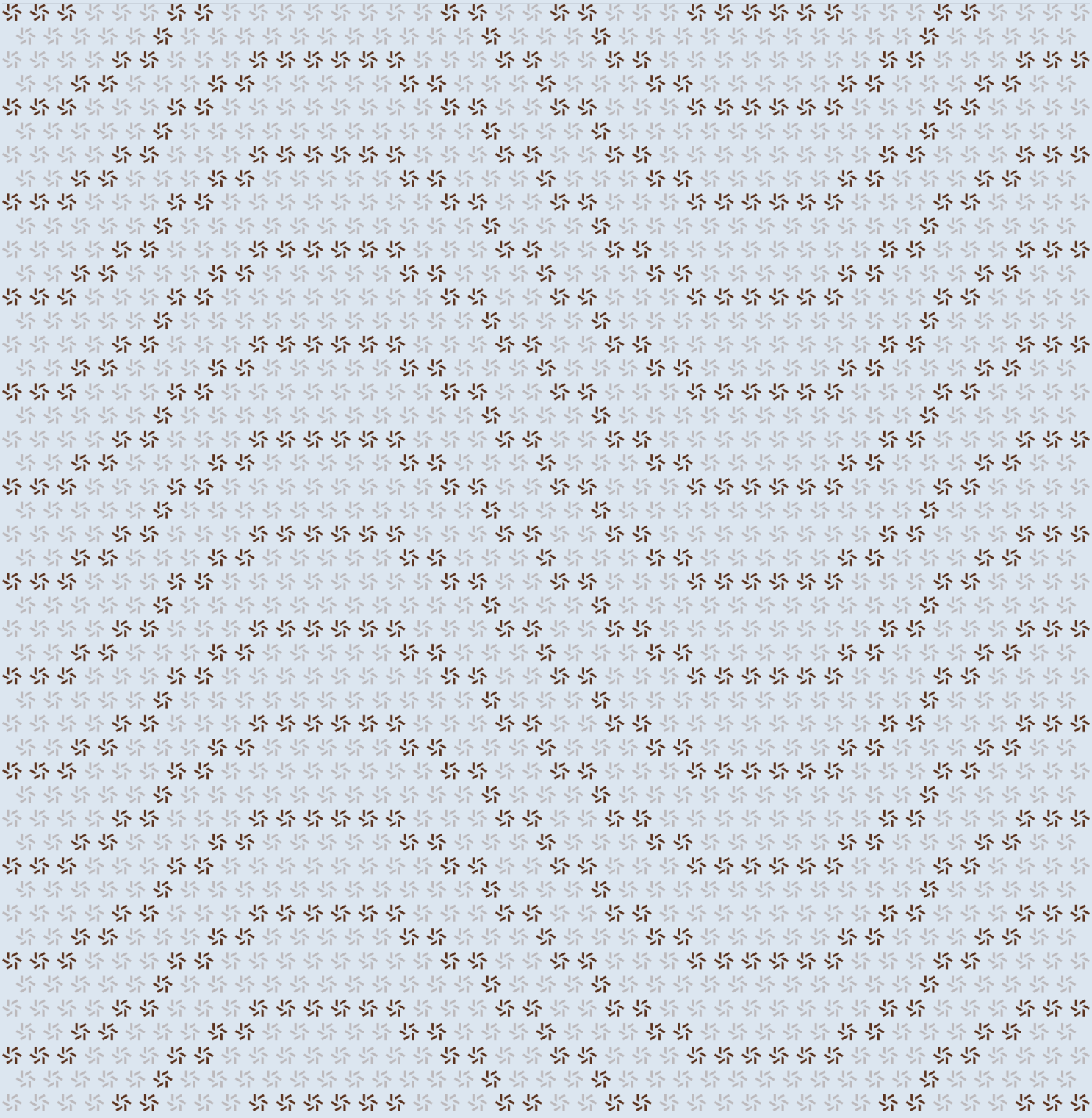
Prepared for
0xripleys
Solend

Prepared by
Filippo Cremonese
Varun Verma
Zellic

November 14, 2024

SpringSui

Sui Move 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 Solend from November 11th to November 14th, 2024. During this engagement, Zellic reviewed SpringSui'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:

- Do the staking and redemption processes work correctly, so users' funds are safe and properly allocated?
- Does the system stay accurate and up-to-date during key moments, like when epochs change, to prevent any operational issues?
- Are rewards collected and distributed fairly, without any errors or unexpected behaviors?
- Are there risks from mismanaging validators, like too much stake in one validator or the incorrect handling of inactive ones?

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. During this assessment, time constraints limited the ability to validate that the accounting aligns perfectly with all validator states under every possible condition.

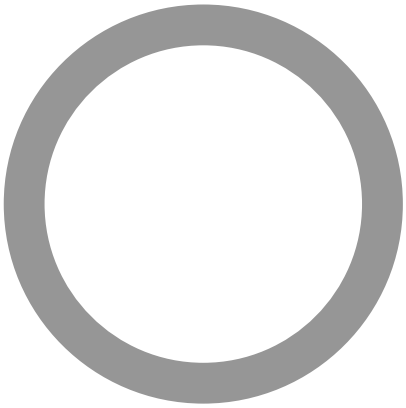
1.4. Results

During our assessment on the scoped SpringSui contracts, we discovered three findings, all of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Solend in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	3



2. Introduction

2.1. About SpringSui

Solend contributed the following description of SpringSui:

> SpringSui is a liquid staking implementation on Sui which allows instant unstaking at any time.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts 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:

SpringSui Contracts

Type	Move
Platform	Sui
Target	SpringSui
Repository	https://github.com/solendprotocol/liquid-staking ↗
Version	`a229e156cf6a7dbd2c221dd2475fb1f66bff231f`
Programs	`Liquid staking`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

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:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Varun Verma
Engineer
varun@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

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

3. Detailed Findings

3.1. Refreshing validator updates

Target	storage.move		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `refresh` function within `storage.move` calls `self.refresh_validator_info(i)` up to four times for the same validator index. Specifically, redundant calls occur in `take_from_inactive_stake`, `join_fungible_staked_sui_to_validator`, and `unstake_approx_n_sui_from_validator`, along with one explicit call within the `refresh` loop.

Impact

These redundant calls increase the gas cost of operations and impact code readability. Nevertheless, the gas overhead is not substantial since it is called once per epoch.

Recommendations

We recommend refactoring the `refresh` function to ensure `self.refresh_validator_info(i)` is only called once per validator index within the loop.

Remediation

The recommended fix requires a design refactor, and the team plans to address these changes at a later date.

3.2. Active stake withdraw invariant

Target	storage.move		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `unstake_approx_n_sui_from_active_stake` function tries to unstake close to `target_unstake_sui_amount` SUI from a validator's active stake. While the `unstaked_sui_amount` should match `target_unstake_sui_amount` in the else case, tracking any differences helps catch calculation errors early and ensures accurate accounting.

Impact

In the else case, we currently calculate what portion of the validator's fungible stake tokens to unstake based on the target SUI amount, using ceiling division to ensure we meet the withdrawal request:

```
// ceil(target_unstake_sui_amount * fungible_staked_sui_amount /
    total_sui_amount)
let split_amount = (
    ((target_unstake_sui_amount as u128)
        * (fungible_staked_sui_amount as u128)
        + (total_sui_amount as u128)
        - 1)
    / (total_sui_amount as u128)
) as u64;

self.split_from_active_stake(system_state, validator_index,
    split_amount as u64, ctx);

let unstaked_sui_amount = unstaked_sui.value();
self.jo
in_to_sui_pool(unstaked_sui);
```

Currently there is no check to verify if `unstaked_sui_amount` matches `target_unstake_sui_amount`, which would help track any mismatches.

Recommendations

When `unstaked_sui_amount` differs from `target_unstake_sui_amount`, emit an event to track these discrepancies without halting operations. This provides visibility into potential mismatches while keeping the system running.

Remediation

The team plans to address these changes at a later date.

3.3. Overpermissive safety check

Target	liquid_staking.move		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `liquid_staking::redeem` function performs a sanity-check assertion to ensure the amount of unstaked SUI is congruent with the amount of assets under management. However, this check is performed after fees are deducted from the unstaked amount.

```
public fun redeem<P: drop>(
    self: &mut LiquidStakingInfo<P>,
    lst: Coin<P>,
    system_state: &mut SuiSystemState,
    ctx: &mut TxContext
): Coin<SUI> {
    self.refresh(system_state, ctx);

    let old_sui_supply = (self.total_sui_supply() as u128);
    let old_lst_supply = (self.total_lst_supply() as u128);

    let sui_amount_out = self.lst_amount_to_sui_amount(lst.value());
    let mut sui = self.storage.split_n_sui(system_state, sui_amount_out, ctx);
    // AUDIT - note this function fails if not enough SUI can be unstaked

    // deduct fee
    let redeem_fee_amount =
    self.fee_config.get().calculate_redeem_fee(sui.value());
    self.fees.join(sui.split(redeem_fee_amount as u64));

    // [...]

    // invariant: sui_out / lst_in <= old_sui_supply / old_lst_supply
    // -> sui_out * old_lst_supply <= lst_in * old_sui_supply
    assert!(
        (sui.value() as u128) * old_lst_supply <= (lst.value() as u128) *
    old_sui_supply,
        ERedeemInvariantViolated
    );
```

```
    // [...]
}
```

Impact

The check is slightly ineffective, as it does not account for fees. If too many SUI were unstaked due to a bug in a configuration with high fees, the invariant may not detect the anomalous condition.

Essentially, high fees could mask excessive unstaking since the deducted fees could offset the excess `sui.value()` in the final calculation.

Recommendations

Enforce the invariant on the amount of SUI unstaked, inclusive of fees.

Remediation

The team plans to address these changes at a later date.

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. Suggestion to use `Option` types

The `find_validator_index_by_address` function returns the index of a given `ValidatorInfo` struct associated with the given validator address:

```
public(package) fun find_validator_index_by_address(self: &Storage,
    validator_address: address): u64 {
    let mut i = 0;
    while (i < self.validator_infos.length()) {
        if (self.validator_infos[i].validator_address == validator_address) {
            return i
        };

        i = i + 1;
    };

    i
}
```

If the validator is not found, the function returns the length of the list of validators (which is an invalid index since it corresponds to a position beyond the end of the list).

The codebase currently uses `find_validator_index_by_address` in two occasions, and both usages correctly check the return value. However, in-bound error reporting is more error prone, and we discourage this practice. Move has native support for nullable types via its `Option` type, which we recommend adopting.

5. 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.

5.1. Module: liquid_staking.move

Function: `create_lst_with_stake`

This function creates a liquid-staking pool that accepts staked SUI positions and direct SUI deposits. The function validates that the total SUI supply and liquid staking token (LST) supply ratios are within acceptable bounds before initializing the liquid-staking system.

Inputs

- `system_state`
  - **Validation**: Originates from `use sui_system::sui_system::{SuiSystemState}` and cannot be arbitrarily created.
  - **Impact**: Guarantees that `system_state` represents the actual on-chain system state.
- `fee_config`
  - **Validation**: Must be a valid `FeeConfig` object.
  - **Impact**: Ensures that appropriate fee structures are applied during operations.
- `lst_treasury_cap`
  - **Validation**: Validates that the total supply of LSTs is greater than zero.
  - **Impact**: Ensures the proper initialization and validation of the LST supply.
- `fungible_staked_suis`
  - **Validation**: Nonempty, valid staked SUI tokens.
  - **Impact**: Staked SUI tokens are processed and contribute to storage.
- `sui`
  - **Validation**: Must be a valid SUI coin.
  - **Impact**: Joins to the SUI pool to initialize the liquid-staking storage.

Branches and code coverage (including function calls)

Intended branches

- Handles staked SUI conversion properly.
  - ☑ Test coverage

- Total SUI supply matches LST supply.
  - ☑ Test coverage
- LST supply matches treasury cap supply.
  - ☑ Test coverage
- Some edge cases such as staking to an invalid validator.
  - ☐ Test coverage

Negative behavior

- Empty treasury cap supply.
  - ☑ Negative test
- Invalid LST/SUI supply ratio (both too high and too low).
  - ☑ Negative test
- Empty staked positions vector.
  - ☑ Negative test
- SUI amount too high > 2x treasury supply.
  - ☑ Negative test
- Invalid fee configurations.
  - ☐ Negative test

Function call analysis

- `create_lst_with_stake -> storage::new(ctx)`
  - **External/Internal?** External.
  - **Argument control?** `ctx` is somewhat controllable by the caller.
  - **Impact**: Initializes storage for liquid staking.
- `create_lst_with_stake -> storage.join_fungible_stake(system_state, fungible_staked_sui, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled by `fungible_staked_suis` provided by the caller.
  - **Impact**: Joins staked SUI to storage.
- `create_lst_with_stake -> create_lst_with_storage(fee_config, lst_treasury_cap, storage, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** `fee_config`, `lst_treasury_cap`, `storage`, and `ctx` partially controlled by the caller.
  - **Impact**: Completes the initialization of `LiquidStakingInfo` with the respective information.

Function: `decrease_validator_stake()`

The `decrease_validator_stake` function allows the admin of an LST to deallocate SUI staked with a validator. It validates SUI amounts, interacts with the system state to unstake the requested amount,

and updates the staking pool while emitting an event to record the unstaking action.

The function does not guarantee that the requested amount of SUI is unstaked. It first unstakes inactive stake, and then if needed, it unstakes from the active stake. If not enough SUI can be unstaked from both sources, the function unstakes the maximum amount possible.

If the requested amount of SUI is lower than `MIN_STAKE_THRESHOLD` (1 SUI), the function may unstake more than the requested amount, due to how the internal function `unstake_approx_n_sui_from_inactive_stake` works.

The amount of unstaked SUI is joined to the pool of unstaked SUI managed by the LST.

Inputs

- `self`
  - **Validation**: Must be a mutable reference to `LiquidStakingInfo<P>`.
  - **Impact**: Represents the protocol's internal staking state.
- `_`
  - **Validation**: Requires a valid `AdminCap<P>`.
  - **Impact**: Ensures that only authorized administrators can invoke this function.
- `system_state`
  - **Validation**: Originates from `use sui_system::sui_system::{SuiSystemState}` and cannot be arbitrarily created.
  - **Impact**: Ensures the operation is performed on the actual system state.
- `validator_address`
  - **Validation**: Must be a valid validator address associated with this LST.
  - **Impact**: SUI is unstaked from this validator.
- `target_unstake_sui_amount`
  - **Validation**: No validation performed directly.
  - **Impact**: Target amount of SUI to be unstaked. May be greater or lower than the actual unstaked amount.

Branches and code coverage (including function calls)

Intended branches

- Basic functionality is tested via a test that increases and decreases the stake for one validator, while simulating rewards for one epoch.
  - ☑ Test coverage
- Test with randomized staked/unstaked amount.
  - ☑ Test coverage
- More complex scenarios — multiple validators, multiple epochs, unstaking small amounts (partially covered by the randomized test).
  - ☐ Test coverage

Negative behavior

- Verifies zero stake is added for dust amounts.
  - ☑ Negative test
- Tries to unstake from an invalid validator.
  - ☐ Negative test

Function call analysis

- `decrease_validator_stake -> self.refresh(system_state, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** Caller has no control.
  - **Impact**: Refreshes internal state before the unstaking operation.
- `decrease_validator_stake -> self.storage.find_validator_index_by_address(validat`
  - **External/Internal?** Internal (friend module).
  - **Argument control?** `validator_address`.
  - **Impact**: Retrieves the index of the validator info from the address of the validator from which to unstake SUI.
- `decrease_validator_stake -> self.storage.unstake_approx_n_sui_from_validator(...`
  - **External/Internal?** Internal (friend module).
  - **Argument control?** `target_unstake_sui_amount`.
  - **Impact**: Performs the unstaking operation, returning the actual amount of unstaked SUI.
- `decrease_validator_stake -> emit_event(IncreaseValidatorStakeEvent { ... })`
  - **External/Internal?** External (standard library).
  - **Argument control?** No direct control.
  - **Impact**: Emits event for logging and tracking unstaking activity.

Function: `increase_validator_stake()`

The `increase_validator_stake` function allows the protocol to allocate additional SUI to a validator, ensuring minimum staking requirements are met. It validates SUI amounts, interacts with the system state to add the stake, and updates the staking pool while emitting an event to record the staking action.

Inputs

- `self`
  - **Validation**: Must be a mutable reference to `LiquidStakingInfo<P>`.
  - **Impact**: Represents the protocol's internal staking state.
- `_`

    - **Validation**: Requires a valid `AdminCap<P>`.
    - **Impact**: Ensures that only authorized administrators can invoke this function.
- `system_state`
    - **Validation**: Originates from `use sui_system::sui_system::{SuiSystemState}` and cannot be arbitrarily created.
    - **Impact**: Ensures the operation is performed on the actual system state.
- `validator_address`
    - **Validation**: Must be a valid on-chain validator address.
    - **Impact**: Ensures the SUI is staked to an existing and functional validator.
- `sui_amount`
    - **Validation**: Must be greater than or equal to `MIN_STAKE_AMOUNT`.
    - **Impact**: Prevents staking of insufficient SUI amounts.

Branches and code coverage (including function calls)

Intended branches

- Verifies stake is added correctly to a validator's inactive stake.
    - ☑ Test coverage
- Validates staking works across different validators.
    - ☑ Test coverage
- Tests handling of amounts below the minimum stake threshold.
    - ☑ Test coverage
- Tests staking after epoch advancement.
    - ☑ Test coverage
- Verifies that staking works correctly immediately after reward collection and distribution.
    - ☐ Test coverage

Negative behavior

- Verifies zero stake is added for dust amounts.
    - ☑ Negative test
- Tries to stake with a validator that does not exist.
    - ☐ Negative test

Function call analysis

- `increase_validator_stake -> self.refresh(system_state, ctx)`
    - **External/Internal?** Internal.
    - **Argument control?** Caller has no control.
    - **Impact**: Refreshes internal state before staking operation.
- `increase_validator_stake -> self.storage.split_up_to_n_sui_from_sui_pool()`
    - **External/Internal?** Internal.
    - **Argument control?** Indirectly controlled by `sui_amount`.

    - **Impact**: Withdraws SUI from staking pool for staking.
- `increase_validator_stake -> system_state.request_add_stake_non_entry(...)`
    - **External/Internal?** External.
    - **Argument control?** Partially controlled by `validator_address` and `sui_amount`.
    - **Impact**: Stakes SUI to the specified validator and updates the system.
- `increase_validator_stake -> emit_event(IncreaseValidatorStakeEvent { ... })`
    - **External/Internal?** Internal.
    - **Argument control?** Arguments derived from function inputs and outputs.
    - **Impact**: Emits event for logging and tracking staking activity.

Function: `mint()`

The mint function allows a user to deposit their SUI in exchange for LSTs.

The amount of LSTs minted is derived from the current SUI-to-LST exchange rate, giving the user an amount of LSTs proportional to the total supply of LST and of SUI managed by the LST.

A mint fee is taken from the deposited SUI amount before the minted LST amount is computed.

Inputs

- `system_state`
    - **Validation**: Originates from `use sui_system::sui_system::{SuiSystemState}` and cannot be arbitrarily created.
    - **Impact**: Guarantees that `system_state` represents the actual on-chain system state.
- `sui`
    - **Validation**: Implicitly, must be an amount greater than zero.
    - **Impact**: The SUI coins used to mint LST.

Branches and code coverage (including function calls)

Intended branches

- Handles SUI-to-LST conversion properly, ensuring LSTs minted and total SUI and LST supplies' accounting matches expectations.
    - ☑ Test coverage
- Edge cases like very small mints.
    - ☐ Test coverage

Negative behavior

- Minting resulting in zero LSTs.
  - ☐ Negative test

Function call analysis

- `mint -> emit_event(RedeemEvent { typename, sui_amount_in, lst_amount_out, fee_amount })`
  - **External/Internal?** External (standard library).
  - **Argument control?** Partially controlled by the caller through `sui`.
  - **Impact**: Emits a mint event and records metadata about the mint.
- `mint -> self.lst_treasury_cap.mint(lst_mint_amount, ctx)`
  - **External/Internal?** External (standard library).
  - **Argument control?** No direct control.
  - **Impact**: Mints the specified amount of LST.
- `mint -> self.storage.join_to_sui_pool(sui_balance)`
  - **External/Internal?** Internal (friend module).
  - **Argument control?** The user provides the `sui_balance`.
  - **Impact**: Joins the SUI to the unstaked SUI pool.

Function: `redeem()`

The redeem function allows a user to exchange their LSTs for an amount of SUI. It computes the SUI output based on the current LST-to-SUI exchange rate, deducts a redeem fee, and ensures the protocol's accounting invariants hold. After burning the LST, it returns the redeemed SUI to the user.

Inputs

- `lst`
  - **Validation**: Ensure `lst.value()` is greater than zero.
  - **Impact**: Prevents invalid or zero-value redemptions.
- `system_state`
  - **Validation**: Originates from `use sui_system::sui_system::{SuiSystemState}` and cannot be arbitrarily created.
  - **Impact**: Guarantees that `system_state` represents the actual on-chain system state.

Branches and code coverage (including function calls)

Intended branches

- Handles staked SUI conversion properly. Total SUI supply matches the LST supply, and the LST supply matches the treasury cap supply (200 MIST).

- ☑ Test coverage
- Some edge cases like very large redemptions.
  - ☐ Test coverage

Negative behavior

- Redeeming zero LST tokens, tests redemption when validators do not have enough stake.
  - ☐ Negative test

Function call analysis

- `redeem -> emit_event(RedeemEvent { typename, lst_amount_in, sui_amount_out, fee_amount })`
  - **External/Internal?** Internal.
  - **Argument control?** Partially controlled by the caller through `lst`.
  - **Impact**: Emits a redemption event and records key data points about the redemption process.
- `redeem -> self.lst_treasury_cap.burn(lst)`
  - **External/Internal?** Internal.
  - **Argument control?** Caller controls `lst`.
  - **Impact**: Burns the specified amount of LST, reducing the total supply.
- `redeem -> self.storage.split_n_sui(system_state, sui_amount_out, ctx)`
  - **External/Internal?** External.
  - **Argument control?** Indirectly controlled by `lst` input.
  - **Impact**: Splits the specified amount of SUI from storage, impacting the SUI pool.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Sui.

During our assessment on the scoped SpringSui contracts, we discovered three findings, all of which were informational in nature.

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.