Solend Liquidstaking

Security Assessment

October 25th, 2024 — Prepared by OtterSec

Michał Bochnak embe221ed@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Solend Protocol engaged OtterSec to assess the `liquid-staking` program. This assessment was conducted between October 16th and October 18th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified multiple vulnerabilities concerning the minting and creation of Liquid Staking Token, including the lack of validation to ensure a proper ratio, resulting in an imbalanced Liquid Staking Token supply and incorrect calculations (OS-SLS-ADV-00), and another issue where an invalid amount of Liquid Staking Token is minted (OS-SLS-ADV-03). Additionally, we highlighted the failure to remove validators, resulting in potential inconsistencies when a validator is marked inactive and later reactivated with a different ID (OS-SLS-ADV-01).

We also made recommendations for refactoring the code to mitigate possible security issues (OS-SLS-SUG-01) and emphasized the lack of proper validations in multiple areas of the codebase (OS-SLS-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/solendprotocol/liquid-staking. This audit was performed against commit 84cce71.

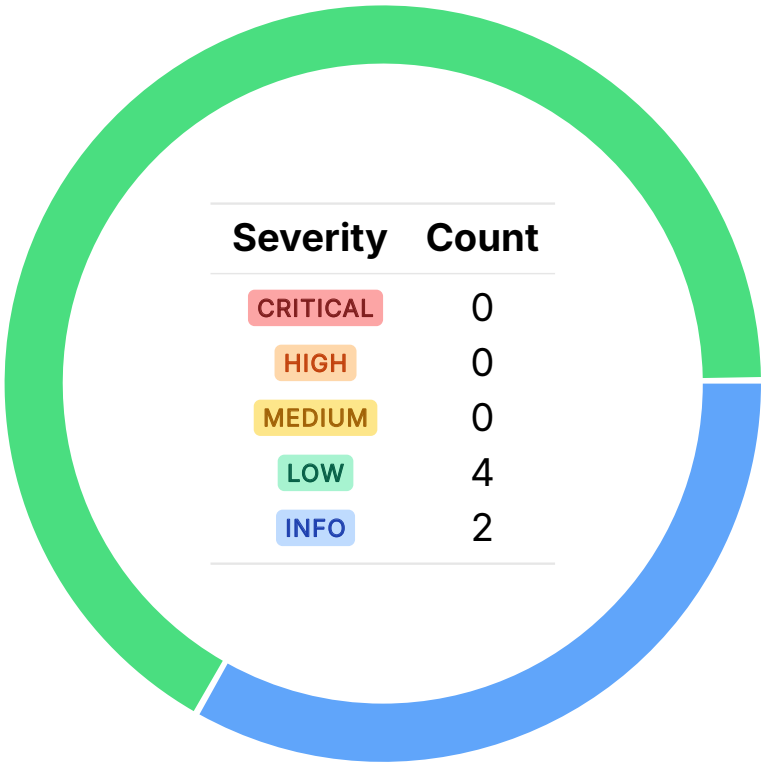
A brief description of the program is as follows:

Name	Description
liquid-staking	It manages the minting and redemption of liquid staking tokens (LSTs) in exchange for SUI, while handling validator stakes and fees. The protocol includes various operations for managing validator priority, increasing or decreasing stakes, and collecting fees.

03 — Findings

Overall, we reported 6 findings.

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



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

04 — Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SLS-ADV-00	LOW	RESOLVED	`create_lst_with_stake` lacks validation to ensure a proper ratio among `SUI`, `fungible_staked_sui`, and `lst_treasury_cap`, resulting in an imbalanced Liquid Staking Token (LST) supply and incorrect calculations.
OS-SLS-ADV-01	LOW	RESOLVED	The protocol lacks the functionality to remove validators, resulting in potential inconsistencies when a validator is marked as inactive and later reactivated with a different `staking_pool_id`.
OS-SLS-ADV-02	LOW	RESOLVED	The assertion check in `mint` and `redeem` may violate expected constraints, which will result in the function aborting execution.
OS-SLS-ADV-03	LOW	RESOLVED	`sui_amount_to_lst_amount` may return zero under certain conditions, resulting in potential inaccuracies in `LST` minting.

Imbalance in LST Supply LOW OS-SLS-ADV-00

Description

Currently, `create_lst_with_stake` does not validate the relationship between the `SUI`, `fungible_staked_sui`, and `lst_treasury_cap.total_supply` values. The function only ensures that `lst_treasury_cap.total_supply` and the total staked `SUI` in the system are greater than zero.

>_ contracts/sources/liquid_staking.move RUST

```
public fun create_lst_with_stake<P: drop>(
    system_state: &mut SuiSystemState,
    fee_config: FeeConfig,
    lst_treasury_cap: TreasuryCap<P>,
    mut fungible_staked_suis: vector<FungibleStakedSui>,
    sui: Coin<SUI>,
    ctx: &mut TxContext
): (AdminCap<P>, LiquidStakingInfo<P>) {
    [...]
    vector::destroy_empty(fungible_staked_suis);
    storage.join_to_sui_pool(sui.into_balance());
    assert!(lst_treasury_cap.total_supply() > 0 && storage.total_sui_supply() > 0,
        ↪  EInvalidLstCreation);
    create_lst_with_storage(
        fee_config,
        lst_treasury_cap,
        storage,
        ctx
    )
}
```

The `LST` created should ideally represent a proportional claim on the underlying staked `SUI` assets. If there is no relationship between the total `LST` supply and the staked `SUI`, users may receive `LST` tokens that over- or under-represent the actual value of the staked assets. The misalignment between `LST` and staked `SUI` may result in incorrect pricing when users interact with the liquid staking system.

Remediation

Implement a check that will validate whether the amounts are reasonable.

Patch

Resolved in 288ce2a.

Improper Validator Management LOW OS-SLS-ADV-01

Description

In the current implementation of the protocol, there is no logic to remove validators from the list of `validator_infos`. It is possible to remove an existing validator, which will just mark it as inactive. The list of active validators may be retrieved utilizing `active_validator_addresses`.

>_ *contracts/sources/storage.move* RUST

```
fun get_or_add_validator_index_by_staking_pool_id_mut(
    self: &mut Storage,
    system_state: &mut SuiSystemState,
    staking_pool_id: ID,
    ctx: &mut TxContext
): u64 {
    [...]

    let validator_address = system_state.validator_address_by_pool_id(&staking_pool_id);
    let exchange_rates = system_state.pool_exchange_rates(&staking_pool_id);
    let latest_exchange_rate = exchange_rates.borrow(ctx.epoch());
    [...]
}
```

Furthermore, the ability to reactivate a validator with a different `staking_pool_id` may introduce complexities. When a validator is reactivated, associating it with a different staking pool may not be straightforward. The existing codebase utilizes `staking_pool_id` as a unique identifier for each validator. Thus, allowing a validator to have multiple associations will result in ambiguity about which `staking_pool_id` should be referenced in various contexts.

Remediation

Add logic that will keep the validators list updated and ensure that this logic is run at the beginning of each epoch.

Patch

Resolved in f8b8b00.

Abortion Due to Failure of Assertion Check LOW OS-SLS-ADV-02

Description

The assertion in `mint` within `liquid_staking`, while intended to maintain the balance between Liquid Staking Tokens (`LST`) and `SUI`, may abort under certain conditions.

>_ *contracts/sources/liquid_staking.move* RUST

```
public fun mint<P: drop>(
    self: &mut LiquidStakingInfo<P>,
    system_state: &mut SuiSystemState,
    sui: Coin<SUI>,
    ctx: &mut TxContext
): Coin<P> {
    [...]
    // invariant: lst_out / sui_in <= old_lst_supply / old_sui_supply
    // -> lst_out * old_sui_supply <= sui_in * old_lst_supply
    assert!(
        (lst.value() as u128) * old_sui_supply <= (sui_balance.value() as u128) *
            ↪  old_lst_supply, EMintInvariantViolated
    );
    [...]
}
```

If `old_lst_supply == 0` and `old_sui_supply > 0`, the assertion will always fail. In this case, the conversion of the provided `SUI` amount to `LST` utilizing `sui_amount_to_lst_amount` will return the `sui_amount` itself.

Remediation

Ensure to handle the above-specified case to prevent the failure of the assertion checks.

Patch

Resolved in f8b8b00.

Minting of Zero LST LOW OS-SLS-ADV-03

Description

It is theoretically possible for `sui_amount_to_lst_amount` to return zero when the calculated `lst_amount` is a very small fraction due to the supply ratio. The function computes the `LST` amount by dividing `total_lst_supply * sui_amount` by `total_sui_supply`. If the `sui_amount` is very small compared to `total_sui_supply`, the result of the division may round down to zero. This is problematic because it implies that the user effectively receives no tokens in exchange for their staked assets.

>_ contracts/sources/liquid_staking.move RUST

```rust
fun sui_amount_to_lst_amount<P>(
    self: &LiquidStakingInfo<P>,
    sui_amount: u64
): u64 {
    let total_sui_supply = self.total_sui_supply();
    let total_lst_supply = self.total_lst_supply();
    if (total_sui_supply == 0 || total_lst_supply == 0) {
        return sui_amount
    };
    let lst_amount = (total_lst_supply as u128)
     * (sui_amount as u128)
     / (total_sui_supply as u128);
    lst_amount as u64
}
```

Remediation

Prevent the minting of an invalid amount of `LST`.

Patch

Resolved in cbd2f94.

05 — General Findings

Here, we present a discussion of the general findings identified during our audit. While these findings do not pose an immediate security impact, they represent anti-patterns and could potentially lead to security issues in the future.

ID	Description
OS-SLS-SUG-00	There are several instances where additional validation may be added for improved functionality.
OS-SLS-SUG-01	Recommendation for modifying the codebase to ensure adherence to coding best practices.

Missing Validation Logic OS-SLS-SUG-00

Description

1. Add a check that ensures `sui_amount > MIN_STAKE_AMOUNT` at the beginning of `increase_validator_stake`. This ensures that if the `sui_amount` passed to the function is less than or equal to `MIN_STAKE_AMOUNT`, the function exits early, returning zero. This prevents further processing if the amount does not meet the minimum requirement.

>_ contracts/sources/liquid_staking.move RUST

```
public fun increase_validator_stake<P>(
    self: &mut LiquidStakingInfo<P>,
    _: &AdminCap<P>,
    system_state: &mut SuiSystemState,
    validator_address: address,
    sui_amount: u64,
    ctx: &mut TxContext
): u64 {
    self.refresh(system_state, ctx);
    let sui = self.storage.split_up_to_n_sui_from_sui_pool(sui_amount);
    if (sui.value() < MIN_STAKE_AMOUNT) {
        self.storage.join_to_sui_pool(sui);
        return 0
    };
    [...]
}
```

2. In the current implementation of `split_n_sui`, there is no check to ensure that the function only attempts to split when the requested amount of `SUI` is actually available in the `sui_pool`. Before calling `split_from_sui_pool`, a check should be added to verify whether the `sui_pool.value` is at least equal to `max_sui_amount_out`.

Remediation

Add the above-stated validations.

Patch

1. Issue #2 resolved in b7755a0.

Code Refactoring OS-SLS-SUG-01

Description

1. In `unstake_approx_n_sui_from_inactive_stake`, there is a condition that checks if the `staked_sui_amount` is less than or equal to the `target_unstake_sui_amount` plus a small buffer (`MIN_STAKE_THRESHOLD`). Replace the `<=` with `<` to favor partial unstaking (splitting) over full unstaking.

>_ contracts/sources/storage.move RUST

```
public(package) fun unstake_approx_n_sui_from_inactive_stake(
    [...]
): u64 {
    [...]
    let staked_sui_amount = validator_info.inactive_stake.borrow().staked_sui_amount();
    let staked_sui = if (staked_sui_amount <= target_unstake_sui_amount +
        ↪  MIN_STAKE_THRESHOLD) {
        self.take_from_inactive_stake(validator_index)
    }
    else {
        self.split_from_inactive_stake(validator_index, target_unstake_sui_amount, ctx)
    };
    [...]
}
```

2. Declare a constant value that represents the `max_bps` (`10_000`), and utilize it instead of hardcoding the value for improved clarity.

Remediation

Incorporate the above-mentioned refactor into the codebase.

Patch

1. Issue #1 resolved in dcc419f.
2. Issue #2 resolved in ab5d293.

A — Vulnerability Rating Scale

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

CRITICAL

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

Examples:

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

HIGH

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

Examples:

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

MEDIUM

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

Examples:

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

LOW

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

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

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

Examples:

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

B — Procedure

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

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

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

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

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

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

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