Steward

Security Assessment

October 28th, 2024 — Prepared by OtterSec

Kevin Chow kchow@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

01 — Executive Summary

Overview

Jito Foundation engaged OtterSec to assess the `steward` program. This assessment was conducted between June 19th and July 29th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability concerning a potential desynchronization between the internal state of the Steward Program and the stake pool's validator list when a validator is marked for deactivation but not immediately removed (OS-SWD-ADV-00). Additionally, the number of pool validators is decremented incorrectly when a validator is added and removed within the same cycle, resulting in an off-by-one error that prevents the re-balancing of the last validator (OS-SWD-ADV-01).

Furthermore, we highlighted an inconsistency in allowing state transition operations to execute in the new epoch, utilizing internal state from the previous epoch (OS-SWD-ADV-02).

We also recommended relaxing the strict invariant check to allow the system to continue operating even if some validators are in the deactivation state (OS-SWD-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/jito-foundation/stakenet. This audit was performed against commit f4ea93a.

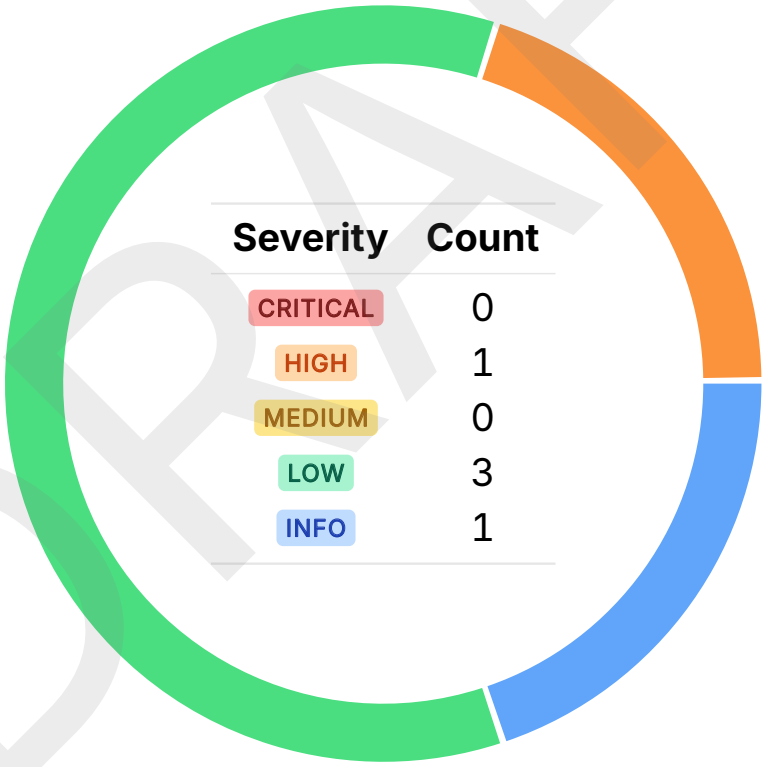
A brief description of the program is as follows:

Name	Description
steward	The system manages staking authority for an SPL Stake Pool by using on-chain validator history to select high-performing validators. It maintains the desired stake levels over time and continuously monitors and re-evaluates the validator set at regular intervals.

03 — Findings

Overall, we reported 5 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	1
MEDIUM	0
LOW	3
INFO	1

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-SWD-ADV-00	HIGH	RESOLVED	There is a potential desynchronization between the internal state of the Steward Program and the stake pool's validator list when a validator is marked for deactivation but not immediately removed.
OS-SWD-ADV-01	LOW	RESOLVED	`num_pool_validators` is decremented incorrectly when a validator is added and removed within the same cycle, resulting in an off-by-one error, preventing the re-balancing of the last validator.
OS-SWD-ADV-02	LOW	RESOLVED	Allowing state transition operations to execute in the new epoch utilizing internal state from the previous epoch will result in inconsistencies.
OS-SWD-ADV-03	LOW	RESOLVED	`remove_validator` incorrectly sets the removal status of a validator to false after its index has already been shifted to the left.

Validator State Desynchronization HIGH OS-SWD-ADV-00

Description

Removing validators immediately after a successful `EpochMaintenance` may result in the desynchronization of the internal state of the Steward Program with the external state of the stake pool validator list, particularly when handling delinquent validators. A validator may be removed from the list in the same epoch if there is no transient stake.

The issue arises because the validator is marked for deactivation in one epoch but is not removed until the next. If the internal state of the Steward program does not update synchronously with the stake pool's external state, the Steward may think it still has the validator in its active pool while the stake pool has already removed it, creating a discrepancy between the two systems.

Proof of Concept

1. `deactivate_delinquent` is called on a validator's stake account during epoch 1, marking it as delinquent but not immediately removing it from the validator list.
2. `epoch_maintenance` is executed during epoch 2, which performs necessary checks and updates the internal state of the Steward Program.
3. `auto_remove_validator_from_pool` is called during the maintenance process, which attempts to remove the delinquent validator from the pool.
4. The stake pool operations may subsequently execute, which will remove the validator from the validator list, resulting in state desynchronization.

Remediation

Implement a mechanism to flag delinquent validators for immediate removal, ensuring that the state machine halts further progress until these validators are removed within the same epoch.

Patch

Resolved in 0425db0.

Mismatch in Validator Count LOW OS-SWD-ADV-01

Description

In `steward_state::compute_score`, `num_pool_validators` is updated positively only once at the beginning of a cycle. During this update, it reflects the current number of validators in the pool. This adjustment ensures that new validators added to the pool at the start of the cycle are properly counted.

>_ *steward/src/state/steward_state.rs* RUST

```
pub fn compute_score(
    [...]
) -> Result<Option<ScoreComponents>> {
    if matches!(self.state_tag, StewardStateEnum::ComputeScores) {
        [...]
        if self.progress.is_empty()
            || current_epoch > self.current_epoch
            || slots_since_scoring_started > config.parameters.compute_score_slot_range
        {
            [...]
            self.num_pool_validators = num_pool_validators;
            self.validators_added = 0;
        }[...]
    }
    [...]
}
```

If a validator is added and then removed within the same cycle, the `num_pool_validators` gets decremented when the validator is removed, but this happens in `epoch_maintenance` during the next epoch. The decrement is done via `remove_validator`. However, `num_pool_validators` remains one lower than it should be after the validator removal because `compute_score` does not update `num_pool_validators` again until the next compute cycle starts (in the following epoch).

>_ *steward/src/instructions/epoch_maintenance.rs* RUST

```
pub fn handler([...]) -> Result<()> {
    [...]
    {[...]
        if let Some(validator_index_to_remove) = validator_index_to_remove {
            state_account
                .state
                .remove_validator(validator_index_to_remove)?;
        }
    }[...]
}
```

In `delegation::decrease_stake_calculation` (which adjusts stake distribution based on yield scores), it utilizes `num_pool_validators` to determine the range of validators that are eligible for rebalancing. Because `num_pool_validators` is one lower than it should be, the final validator may be excluded from the rebalancing process.

Remediation

Ensure that `validators_added` is decremented properly.

Patch

Resolved in 0275585.

Improper State Transition LOW OS-SWD-ADV-02

Description

In the Steward program, internal state transitions manage how the program behaves at different stages (`Rebalance` , `InstantUnstake` , and `Idle`). During these transitions, the program may carry internal states from the previous epoch into the new one. When a new epoch begins, the program is expected to reset. Currently, it is possible to execute an extra call in the current state (`Rebalance` or `InstantUnstake`) to return to idle during a new epoch in `transition_rebalance` , and another call to return to `compute_score` during a new cycle.

>_ *steward/src/state/steward_state.rs* RUST

```rust
fn transition_rebalance(
    &mut self,
    current_epoch: u64,
    current_slot: u64,
    num_epochs_between_scoring: u64,
) -> Result<()> {
    if current_epoch >= self.next_cycle_epoch {
        self.reset_state_for_new_cycle(
            current_epoch,
            current_slot,
            num_epochs_between_scoring,
        )?;
    } else if self.has_flag(RESET_TO_IDLE) {
        self.state_tag = StewardStateEnum::Idle;
        self.progress = BitMask::default();
        // NOTE: RESET_TO_IDLE is cleared in the Idle transition
    }[...]
}
```

If the program performs a `Rebalance` (or `InstantUnstake`) operation utilizing the internal state from the previous epoch without properly resetting, it will result in inappropriate adjustments of stakes based on stale data, which may not reflect the current validator performance or network conditions.

Remediation

Force a transition to the `Idle` state when a new epoch starts.

Patch

Resolved in 9585307.

Index Mismanagement LOW OS-SWD-ADV-03

Description

After a validator is marked for removal, and the state is updated by shifting the remaining validators to the left, the original index of the removed validator is no longer valid. The entries at that index are replaced by the next validator in the list. Therefore, retaining of a reference to the removed index or attempting to clear its marked state is problematic.

>_ *steward/src/state/steward_state.rs* RUST

```rust
/// Update internal state when a validator is removed from the pool
pub fn remove_validator(&mut self, index: usize) -> Result<()> {
    [...]
    if marked_for_regular_removal {
        self.validators_to_remove.set(index, false)?;
    } else {
        self.validators_for_immediate_removal.set(index, false)?;
    }
    Ok(())
}
```

The current logic in `remove_validator` attempts to set the validator at the index to false in `validators_to_remove` and `validators_for_immediate_removal`, even though the validator at the `index` has been effectively removed.

Remediation

Remove the lines that set the validator's index to false after the left shift.

Patch

Resolved in 1428eea and 59176f8.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SWD-SUG-00	The validator removal process in the epoch maintenance module may stall if the global 25% stake deactivation limit is reached.

Stalling Due to Inconsistency in Validator Removal OS-SWD-SUG-00

Description

`epoch_maintenance` includes a requirement that all validators in the list must have a `StakeStatus::Active` status before proceeding. However, if 25% or more of the total global stake is deactivated at once, this will result in delays in fully removing those validators within a single epoch. In such a scenario, `epoch_maintenance` would fail to pass the invariant check for multiple epochs, as there would still be validators in the deactivation state, stalling the entire maintenance process.

>_ *steward/src/instructions/epoch_maintenance.rs* RUST

```rust
pub fn handler(
    ctx: Context<EpochMaintenance>,
    validator_index_to_remove: Option<usize>,
) -> Result<()> {
    [...]
    // Ensure there are no validators in the list that have not been removed, that should be
    require!(
        !check_validator_list_has_stake_status_other_than(
            &ctx.accounts.validator_list,
            &[StakeStatus::Active]
        )?,
        StewardError::ValidatorsHaveNotBeenRemoved
    );
    [...]
}
```

Remediation

Relax the strict invariant check to allow the system to continue operating even if some validators are in a deactivation state.

Patch

This situation has been acknowledged by the Steward team but not addressed, as it would require 25% of the total global stake to be deactivated at once, which is considered an uncommon event. The team also mentioned that the program would only stall for a single epoch, assuming that the validators would eventually be removed in subsequent epochs once deactivation has been fully processed.

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.