Jito Governance

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Jito Labs engaged OtterSec to assess the `governance` program. This assessment was conducted between November 22nd and November 24th, 2023. Followup reviews were conducted up until Dec 7th. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability regarding a potential overflow scenario while calculating maximum claimable tokens, wrapping around into a considerably smaller value. This denies the withdrawal of tokens whose amount exceeds this small limit (OS-JGV-ADV-00).

We also made recommendations around the lack of validation to ensure that vesting timestamps are greater than the current timestamp during distributor setup (OS-JGV-SUG-00). Additionally, we suggested extending the time frame between the end of a vesting period and the start of the clawback to ensure ample time for fund withdrawal (OS-JGV-SUG-01). Furthermore, we pointed out a typographical error in a specific function's name (OS-JGV-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/jito-foundation/distributor. This audit was performed against commit 5d27000.

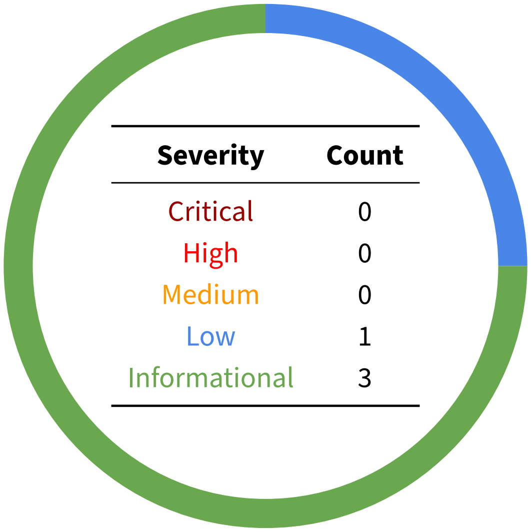
A brief description of the programs is as follows:

Name	Description
governance	A program designed to efficiently distribute tokens through uploading a Merkle root by deriving a 256-bit root hash from a tree of balances to mitigate the high rent cost associated with token distribution.

03 Findings

Overall, we reported 4 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	1
Informational	3

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-JGV-ADV-00	Low	Resolved	Calculating maximum claimable tokens may overflow, resulting in a small value of `max_total_claim`, restricting token claiming.

OS-JGV-ADV-00 [low] | Inability To claim Tokens

Description

MerkleTree::new creates a Merkle tree structure that includes the Merkle root, information about the number of nodes, the total claimable amount, and the individual tree nodes with their associated Merkle proofs. However, it lacks an overflow check during the calculation of max_total_claim, where it sums up the total claimable amounts of all the TreeNode instances in the tree_nodes vector in the Merkle tree.

MerkleTree.rs RUST

```rust
pub fn new(tree_nodes: &mut Vec<TreeNode>) -> Self {
    [...]
    let max_total_claim = tree_nodes.iter().map(|n| n.total_amount()).sum();
    MerkleTree {
        merkle_root: tree.get_root().unwrap().to_bytes(),
        max_num_nodes: tree_nodes.len() as u64,
        max_total_claim,
        tree_nodes: tree_nodes.to_vec(),
    }
}
```

Consequently, in MerkleTree::validate, a similar calculation of max_total_claim is performed to verify that the resulting sum matches the calculated value of max_total_claim. Thus, the sum overflows and correctly matches with max_total_claim, validating an inaccurate value.

MerkleTree.rs RUST

```rust
fn validate(&self) -> Result<(), MerkleValidationError> {
    [...]
    // validate that sum is equal to max_total_claim
    let sum: u64 = self.tree_nodes.iter().map(|n| (n.total_amount())).sum();
    if sum != self.max_total_claim {
        return Err(MerkleValidationError {
            msg: format!(
                "Tree nodes sum {} does not match max_total_claim {}",
                sum, self.max_total_claim
            ),
        });
    }
    [...]
}
```

As a result of the overflow, max_total_claim would contain a significantly smaller value than the actual claim amount. The on-chain program configures with this small value of max_total_claim, effectively resulting in a denial of service by barring users from claiming the tokens as the checks performed while claiming (total_claimed < max_total_claim) is not satisfied.

Remediation

Utilize `checked_add` to handle potential overflow by returning None if an overflow occurs.

Patch

Fixed in PR #26 by panicking when overflow occurs.

```diff
+ /// Given a set of tree nodes, get the max total claim amount. Panics on overflow
+ pub fn get_max_total_claim(nodes: &[TreeNode]) -> u64 {
+     nodes
+         .iter()
+         .try_fold(0, |acc: u64, n| acc.checked_add(n.total_amount()))
+         .unwrap()
+ }
```

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-JGV-SUG-00	Validation to ensure that `start_vesting_ts`, `end_vesting_ts`, and `clawback_start_ts` are greater than the current timestamp during distributor setup is absent.
OS-JGV-SUG-01	Narrow window between `end_vesting_ts` and `clawback_start_ts` may challenge users withdrawing their vested funds.
OS-JGV-SUG-02	Typographical error in `set_amin`.

OS-JGV-SUG-00 | Timestamp Validations

Description

In the process of setting up the distributor in `handle_new_distributor`, there is an absence of validations to ascertain that `start_vesting_ts`, `end_vesting_ts`, and `clawback_start_ts` are all greater than the current timestamp.

new_distributor.rs RUST

```rust
#[allow(clippy::too_many_arguments)]
pub fn handle_new_distributor(
    [...]
    start_vesting_ts: i64,
    end_vesting_ts: i64,
    clawback_start_ts: i64,
) -> Result<()> {
    require!(
        start_vesting_ts < end_vesting_ts,
        ErrorCode::InvalidTimestamp
    );
    require!(
        clawback_start_ts > end_vesting_ts,
        ErrorCode::ClawbackBeforeVestingEnd
    );

    let distributor = &mut ctx.accounts.distributor;
    [...]
}
```

This scenario may result in token users claiming vested tokens immediately after distributor setup, potentially affecting the planned token distribution. Furthermore, activating the ability to clawback tokens may occur earlier than intended, giving rise to unexpected consequences and potentially undesirable token movements.

Remediation

Enforce timestamp validations during distributor setup.

Patch

Fixed in PR #26 by enforcing that all the timestamps are set in the future.

new_distributor.rs DIFF

```
pub fn handle_new_distributor(
[...]
+    let curr_ts = Clock::get()?.unix_timestamp;
[...]
+    // New distributor parameters must all be set in the future
+    require!(
+        start_vesting_ts > curr_ts && end_vesting_ts > curr_ts &&
    ↪   clawback_start_ts > curr_ts,
+        ErrorCode::TimestampsNotInFuture
+    );
```

OS-JGV-SUG-01 | Withdrawal Window

Description

The issue concerns a potential timing constraint that may impact users' ability to withdraw the last chunk of vested funds. Specifically, suppose `clawback_start_ts` is set very close to `end_vesting_ts`. In that case, users may have a very short window to withdraw their tokens before the clawback functionality becomes active. This increases the risk that some users may be unable to submit withdrawal transactions in time.

Remediation

Adjust the timing parameters to allow for a more extended withdrawal window. This may involve increasing the time difference between `end_vesting_ts` and `clawback_start_ts`, providing users a more reasonable time frame to withdraw their funds.

Patch

Fixed in PR #26 by enforcing `clawback_start_ts` is at least one day after `end_vesting_ts`.

new_distributor.rs DIFF

```diff
pub fn handle_new_distributor(
[...]
+    // Ensure clawback_start_ts is at least one day after end_vesting_ts
+    require!(
+        clawback_start_ts >= end_vesting_ts + SECONDS_PER_DAY,
+        ErrorCode::InsufficientClawbackDelay
+    );
```

OS-JGV-SUG-02 | Typographical Error

Description

In `set_amin`, *admin* is incorrectly spelled as *amin* as shown below:

lib.rs RUST

```rust
pub fn set_amin(ctx: Context<SetAdmin>) -> Result<()> {
    handle_set_admin(ctx)
}
```

Remediation

Modify the function name to become `set_admin` instead of `set_amin`.

Patch

Fixed in PR #26.

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 section.

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.

Informational

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 the other 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.