certora

Second report - Jito Restaking Security Assessment & Formal Verification

November 2024

Prepared for
Jito Labs

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform	Comments
Jito restaking	https://github.com/jito-foundation/restaking/releases/tag/v0.0.3	ecbe19a	Solana	Audit version
Jito restaking	https://github.com/jito-foundation/restaking	3fdcd88	Solana	Fix version

Project Overview

This document describes the **second** verification effort of Jito Restaking using manual code review. The work was undertaken from October 31, 2024, to November 11, 2024.

The following contract list is included in our scope: all files under `restaking_core/` , `restaking_program/`, `vault_core/` , and `vault_program/`.

The team performed a manual audit of all the Solana contracts in the repo**.** During the manual audit process, the Certora team discovered bugs in the Solana contracts code, as listed on the following page.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	2	2	2
Medium	3	3	2
Low	2	3	2
Informational	0	0	0
Total	**7**	**7**	**6**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Detailed Findings

High-Severity Issues

H-01 Zero Supply States Can Permanently Break VRT: Token Ratio		
Severity: High	Impact: High	Likelihood: Medium
Files: vault.rs	Category: Economic Model	Status: Fixed

Description:

The vault has a critical vulnerability in handling states where either deposited tokens or VRT supply reaches zero. When either supply becomes zero, the VRT:Token ratio becomes either infinite (0 tokens) or zero (0 VRT), creating an unrecoverable state that breaks the vault's economic model.

The current implementation only checks for zero token deposits and doesn't protect against zero-supply states and also doesn't properly handle 1:1 ratio cases.

This creates a scenario where the vault can enter an unrecoverable state where rational price discovery becomes impossible.

Impact:

- Permanent breakage of vault economics when zero-supply occurs
- Future slashing mechanisms could trigger this vulnerability
- Protocol becomes economically unstable in edge cases

Recommendation:

1. Mint initial virtual VRT and tokens at vault creation and never allow these virtual amounts to be burned or slashed
2. Add explicit handling for 1:1 ratios and implement guards against zero-supply states
3. Consider designing minimum supply protections thresholds and update slashing mechanism with this

Customer's response: PR150 fixes this, merged in the fix version: https://github.com/jito-foundation/restaking/pull/150/files

Fix review: The vulnerability appears properly remediated.

H-02 Incorrect Asset Unstaking Logic Leads to Over-Unstaking

Severity: High	Impact: High	Likelihood: Medium
Files: initialize_vault_update_state_tracker.rs	Category: Asset Management	Status: Fixed

Description: The `initialize_vault_update_state_tracker` contains a critical flaw in its unstaking logic. The function incorrectly assumes that a zero value of `additional_assets_need_unstaking` always indicates that the previous update state has completed. However, it is possible that all assets have been unstaked while not all operators have been processed. This would unstake more assets than necessary.

Current implementation:

```
C/C++
let additional_assets_need_unstaking = if
vault.additional_assets_need_unstaking() > 0 {
      // Assumes incomplete state and proceeds with unstaking
      // Without checking already scheduled operations
}
```

Two key issues:

1. No reliable way to detect incomplete crank operations
2. Immediate state updates of newly enqueued cooldown assets

Impact:

- Assets can skip required cooldown periods
- Impossible to reliably detect incomplete cranks
- Disruption of vault economics

Recommendation:

1. Add crank tracking mechanism
2. Modify state update logic to prevent updates of newly enqueued cooldown assets
3. Track crank progress explicitly
4. Handle incomplete cranks appropriately
5. Structural improvements like, status validation and implementing completion guarantees

Customer's response: Fixed in https://github.com/jito-foundation/restaking/pull/163/files

Fix review: The vulnerability appears properly remediated.

Medium-Severity Issues

M-01 Inconsistent Token Program Usage and Validation Creates Potential Vulnerabilities

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: mint_to.rs, burn_withdrawal_ticket.rs, enqueue_withdrawal.rs, delegate_token_account.rs, operator_delegate_token_account.rs, ncn_delegate_token_account.rs, loader.rs, initialize_vault.rs	Category: Input Validation	Status: Fixed

Description: The protocol inconsistently handles SPL Token and Token2022 program interactions, lacking proper validation of token program addresses and ownership checks. Multiple instruction handlers use a mix of both token program versions without consistent validation patterns.

Key issues:

1. Token program address validations are inconsistent across instructions
2. Missing ownership checks between token mints and accounts in some handlers
3. Inconsistent usage of transfer/mint/burn/close functions between token program versions

Impact:

- Potential for token program address spoofing
- Risk of accepting invalid token accounts or mints
- Inconsistent behavior between different token operations
- Possible transaction failures in edge cases
- Security assumptions may be violated when mixing token program versions

Recommendation:

1. Standardize token program validation across all handlers
2. Standardize on either SPL Token or Token2022 for all operations
3. Create helper functions for common token program validations
4. Add explicit tests for token program validation edge cases

Customer's response: Fixed in https://github.com/jito-foundation/restaking/pull/175

Fix review: The vulnerability appears properly remediated.

M-02 Inefficient Token Account State Unpacking and Missing Extension Validation

Severity: Medium	Impact: Medium	Likelihood: Low
Files: burn_withdrawal_ticket.rs, update_vault_balance.rs, loader.rs, delegate_token_account.rs, initialize_vault.rs	Category: Performance	Status: Confirmed, will not be fixed

Description: The protocol uses bad practices for unpacking token account state and lacks proper validation of Token2022 extensions. This creates both performance inefficiencies and potential security risks when handling tokens with extensions.

Key issues:

1. Using full `Account::unpack` when only basic state checks are needed
2. No handling of extension-specific behaviors like transfer fees
3. Inefficient state unpacking in high-frequency operations

Impact:

- Potential incompatibility with certain token extensions

- Risk of accounting errors with interest-bearing or fee-charging tokens

Recommendation:

1. Implement state unpacking based on requirements
2. Add extension validation at key points:
    a. Vault initialization
    b. Token deposits
    c. Withdrawals
    d. Delegations

Customer's response: We're not going to address this as Token2022 is no longer supported and we are okay with the inefficient unpack.

M-03 Missing Maximum Bounds Validation in Fee Rate Controls

Severity: Medium	Impact: Medium	Likelihood: Low
Files: vault.rs	Category: Input Validation	Status: Fixed

Description: Multiple fee-setting administrative functions in the vault lack maximum bounds validation against `MAX_BPS` (10,000 basis points = 100%). While these functions are admin-controlled, the lack of validation could lead to protocol dysfunction if misconfigured.

The affected functions lack validation to ensure input values don't exceed MAX_BPS (10,000)::

```C/C++
set_reward_fee_bps()
set_program_fee_bps()
set_withdrawal_fee_bps()
```

Impact:

- Protocol could become unusable if fees are set above 100%
- Transactions would fail due to arithmetic checks

Note that while the issue could disrupt protocol operation, direct fund loss is unlikely due to:

- Admin-only access to these functions
- Existing arithmetic checks preventing overflow
- Transaction failure rather than incorrect execution

Recommendation:

1. Add MAX_BPS validation to all fee-setting functions.
2. Implement consistent validation across all fee types
3. Document valid fee ranges in documentation

Customer's response: Fixed in https://github.com/jito-foundation/restaking/pull/184

Fix review: The vulnerability appears properly remediated.

Low-Severity Issues

L-01 `VaultStakerWithdrawalTicket` Uses Suboptimal PDA Address Verification Method

Severity: Low	Impact: Low	Likelihood: High

Files: vault_staker_withdrawal_ticket.rs	Best Practices	Status: Fixed

Description:
The `VaultStakerWithdrawalTicket::load()` function uses `find_program_address` instead of `create_program_address` for PDA verification, despite already storing the bump seed in the account data. While the current implementation remains secure through proper key comparison checks, this approach is suboptimal both from best practices and performance perspectives.

Current implementation redundantly searches for the bump seed:

```
C/C++
let expected_pubkey = Self::find_program_address(program_id, vault.key,
&base).0;
if vault_staker_withdrawal_ticket.key.ne(&expected_pubkey) {
      msg!("Vault staker withdrawal ticket is not at the correct PDA");
      return Err(ProgramError::InvalidAccountData);
}
```

Impact:

- Significantly higher compute unit consumption per verification due to `find_program_address` being more expensive in terms of compute units than `create_program_address`
- Redundant storage of bump seed that isn't utilized in the verification process

Recommendation:

- Modify the load() function to use create_program_address with the stored bump.
- Add explicit tests verifying PDA address validation using stored bump

Customer's response: Fixed in https://github.com/jito-foundation/restaking/pull/154

Fix review: The vulnerability appears properly remediated.

L-02 Inconsistent Epoch Calculation Methods

Severity: Low	Impact: Low	Likelihood: Medium
Files:		Status: Fixed

Description:
The codebase shows inconsistent patterns for calculating epochs from slots:
1. Some components use direct slot/epoch_length division
2. Others use helper functions (get_epoch_from_slot)
3. Different error types are returned for the same failure case

Impact:
- Inconsistent error handling between components
- Potential for subtle bugs in epoch boundary conditions

Example:

```C/C++
// Direct calculation:
let current_epoch =
slot.checked_div(epoch_length).ok_or(VaultError::DivisionByZero)?;


// Helper function:
let current_epoch = config.get_epoch_from_slot(slot)?;
```

Recommendation:
1. Normalize all epoch calculations through a single helper function

2. Create consistent error types for epoch-related failures

3. Consider adding a trait for epoch-related operations

Customer's response: Fixed in https://github.com/jito-foundation/restaking/pull/185

Fix review: The vulnerability appears properly remediated.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.