Security Audit – Marinade Validator Bonds

conducted by Neodyme AG

Lead Auditor:	Nico Gründel
Second Auditor:	Simon Klier

April 09th 2024

Table of Contents

1 Introduction

Neodyme audited **Marinades** on-chain Validator Bond program during March and April of 2024. The scope of this audit was focused on technical security, with further considerations about operational security. The auditors found that Marinade's Validator Bond program comprised a clean design and above-standard code quality, relying on the industry-standard Anchor framework. According to Neodymes Rating Classification, **no critical**, **one high**, and one **low-severity issue** was found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

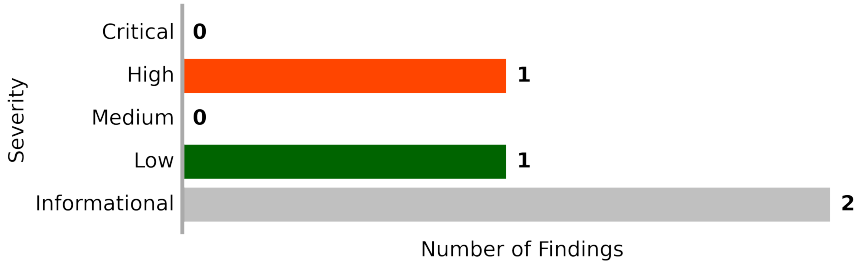

Figure 1: Overview of Findings

All findings were reported to the Marinade developers and addressed promptly. The security fixes were verified for completeness by Neodyme. In addition to these findings, Neodyme delivered the Marinade team a list of nit-picks and additional notes that are not part of this report.

Summary of Findings

During the audit, **two security-relevant** and **two informational** findings were identified. Marinade remediated all of those findings.

In total, the audit revealed:

0 critical • **1** high-severity • **0** medium-severity • **1** low-severity • **2** informational

issues.

The high-severity finding addresses a vulnerability, where anyone could withdraw settlement receivers claims to locked stake accounts, making those funds inaccessible and opening up the possibility of extortion.

All findings are detailed in section Findings.

2 | Scope

The contract audit's scope is focused on the **Implementation** security of the contract's source code.

Neodyme considers the source code, located at https://github.com/marinade-finance/validator-bonds, in scope for this audit. Third-party dependencies are not in scope.

The relevant source code revisions are:

- `afbdad4742dd21678aa0ee7052793b59e30eb597` • Start of the audit
- `7e6d35e8337174bfe6fcf2691914ac65427f6095` • Last reviewed revision

3 | Project Overview

This section briefly outlines Marinade's Validator Bonds functionality, design, and architecture, followed by a detailed discussion of all related authorities.

Functionality

Marinade's Validator Bond program allows for Solana validators to put up stake accounts as a bond. A global operator authority can create settlements against that bond, which marinade plans to trigger in response to so-called protected events, namely slashing or underperformance of the validator. Stakers can then claim their share of the settlement to recuperate losses caused by the protected event.

On-Chain Accounts and Authorities

The root of the Validator Bond programs account structure are the `Config` accounts. The creation of these is not permissioned, and they can be created at any address (requiring a signature, of course). A `Config` holds the following values:

```
/// Root account that configures the validator bonds program
#[account]
#[derive(Debug)]
pub struct Config {
    /// Admin authority that can update the config
    pub admin_authority: Pubkey,
    /// Operator authority (bot hot wallet)
    pub operator_authority: Pubkey,
    /// How many epochs permitting to claim the settlement
    pub epochs_to_claim_settlement: u64,
    /// How many epochs before withdraw is allowed
    pub withdraw_lockup_epochs: u64,
    /// Minimum amount of lamports to be considered for a stake account
    ↪  operations (e.g., split)
    pub minimum_stake_lamports: u64,
    /// PDA bonds stake accounts authority bump seed
    pub bonds_withdrawer_authority_bump: u8,
    /// Authority that can pause the program in case of emergency
    pub pause_authority: Pubkey,
    // Defines if the program is paused
    pub paused: bool,
    /// reserved space for future changes
    pub reserved: [u8; 479],
}
```

The `admin authority` is able to update the config, while the `operator authority` is able to create settlements. The `pause authority` is able to pause and resume the contract.

Validators can create Bond accounts. The address of a Bond account is derived from the `Config` key that will manage this bond and the vote account of the validator.

```
/// Bond account for a validator vote address
#[account]
#[derive(Debug)]
pub struct Bond {
```

```
    /// Program root config address. Validator bond is created for this
     ↪ config as PDA
    // saving the address here for easier access with getProgramAccounts
     ↪ call
    pub config: Pubkey,
    /// Validator vote address that this bond account is crated for
    /// INVARIANTS:
    ///  - one bond account per validator vote address
    ///  - this program does NOT change stake account delegation
     ↪ voter_pubkey to any other validator vote account
    pub vote_account: Pubkey,
    /// Authority that may close the bond or withdraw stake accounts
     ↪ associated with the bond
    /// The same powers has got the owner of the validator vote account
    // https://github.com/solana-
     ↪ labs/solana/blob/master/vote/src/vote_account.rs
    pub authority: Pubkey,
    /// Cost per mille per epoch
    pub cpmpe: u64,
    /// PDA Bond address bump seed
    pub bump: u8,
    /// reserve space for future extensions
    pub reserved: [u8; 142],
}
```

Both the bond `authority` and the vote account of the validator can create withdraw requests for the bond:

```
/// Request from a validator to withdraw the bond
#[account]
#[derive(Debug)]
pub struct WithdrawRequest {
    /// Validator vote account that requested the withdrawal
    pub vote_account: Pubkey,
    /// Bond account that the withdraw request is for (has to match with
     ↪ vote_account)
    pub bond: Pubkey,
    /// Epoch when the withdrawal was requested, i.e., when this "ticket"
     ↪ is created
    pub epoch: u64,
    /// Amount of lamports to withdraw
```

```
    pub requested_amount: u64,
    /// Amount of lamports withdrawn so far
    pub withdrawn_amount: u64,
    /// PDA account bump
    pub bump: u8,
    /// reserve space for future extensions
    pub reserved: [u8; 93],
}
```

These are only claimable after a certain amount of time, as configured in the `Config` account. Only a single `WithdrawRequest` can exist for each Bond at one time. Both the vote account and the bond `authority` can cancel the withdraw request again.

The `operator authority` can create settlements against bond accounts:

```
/// Settlement account for a particular config and merkle root
/// Settlement defines that a protected event happened and it will be
↪  settled
#[account]
#[derive(Debug)]
pub struct Settlement {
    /// the settlement belongs under this bond, i.e., under a particular
    ↪  validator vote account
    pub bond: Pubkey,
    /// settlement authority used as the 'staker' stake account authority
    /// of stake accounts funded to this settlement
    pub staker_authority: Pubkey,
    /// 256-bit merkle root to check the claims against
    pub merkle_root: [u8; 32],
    /// maximum number of funds that can ever be claimed
    pub max_total_claim: u64,
    /// maximum number of merkle tree nodes that can ever be claimed
    pub max_merkle_nodes: u64,
    /// total lamports funded
    pub lamports_funded: u64,
    /// total lamports that have been claimed
    pub lamports_claimed: u64,
    /// number of nodes that have been claimed
    pub merkle_nodes_claimed: u64,
    /// what epoch the Settlement has been created for
    pub epoch_created_for: u64,
    /// address that collects the rent exempt from the Settlement account
    ↪  when closed
```

```
    pub rent_collector: Pubkey,
    /// address that collects rent exempt for "split stake account"
     ↪  possibly created on funding settlement
    pub split_rent_collector: Option<Pubkey>,
    /// amount of lamports that are collected for rent exempt for "split
     ↪  stake account"
    pub split_rent_amount: u64,
    /// PDA bumps
    pub bumps: Bumps,
    /// reserve space for future extensions
    pub reserved: [u8; 99],
}
```

A merkle tree is used to store all claims. Claims are only able to be claimed for a certain amount of epochs, and funds associated with a settlement are no longer available for withdrawal. In order to ensure that no claim is claimed twice, an account is created once each claim is executed.

Instructions

The contract has a total of 20 instructions, which we briefly summarize here.

Instruction	Category	Summary
InitConfig	Permissionless	Initializes a `Config` account
ConfigureConfig	Admin only	Changes the configuration on a `Config`
EmergencyPause	Pause authority only	Pauses all relevant functions of the program
EmergencyResume	Pause authority only	Resumes all relevant functions of the program
InitBond	Permissionless	Initializes a Bond account.
FundBond	Permissionless	Funds a stake account to a validators Bond by changing the withdraw and staking authorities
ConfigureBond	Bond authority only	Changes the config of a Bond
MintBond	Bond authority only	Mints a special SPL token to the vote account, that can be used to change the Bond config
ConfigureBondWithMint	Config token holder only	Burns a config token that was previously minted using `MintBond` and changes the Bond config
InitWithdrawRequest	Bond authority only	Initializes a `Withdraw Request`
CancelWithdrawRequest	Bond authority only	Cancels a `Withdraw Request`
ClaimWithdrawRequest	Bond authority only	Claims a `Withdraw Request`
InitSettlement	Operator only	Initializes a `Settlement`
FundSettlement	Operator only	Funds a stake account that is managed by the corresponding Bond to the `Settlement`
CloseSettlement	Permissionless	Closes a `Settlement`, only possible after it expired and no more claims can be claimed
ClaimSettlement	Permissionless	Executes a claim against a `Settlement`, by withdrawing the stake into a stake account owned by the claimant

Instruction	Category	Summary
CloseSettlementClaim	Permissionless	Closes a `SettlementClaim` account after the `Settlement` it belongs to is expired, in order to claw back the rent
MergeStake	Permissionless	Merges two stake accounts that are managed by the Validator Bond program
ResetStake	Permissionless	Resets the stake authority of a stake account after a `Settlement` has expired
WithdrawStake	Operator only	Withdraws funded stake accounts that belong to an expired `Settlement`

4 | Findings

This section outlines all of our findings. They are classified into one of five severity levels, detailed in Appendix C. In addition to these findings, Neodyme delivered the Marinade team a list of nit-picks and additional notes which are not part of this report.

All findings are listed in Table 2 and further described in the following sections.

Table 2: Findings

Name	Severity	State
[ND-MND3-H1] Settlements Can Be Claimed To Locked Stake Accounts	High	Resolved
[ND-MND3-L1] No Safeguards for Compromised Operator Hot Wallet	Low	Resolved
[ND-MND3-I1] Unreliable Events	Info	Resolved
[ND-MND3-I2] Vote Account Version Is Not Checked	Info	Resolved

ND-MND3-H1 – Settlements Can Be Claimed To Locked Stake Accounts

Severity	Impact	Affected Component	Status
High	Anyone can withdraw every claim in all settlements into locked stake accounts, potentially extorting the receiver of the claim	Settlements	Resolved

Settlement claims can be executed by anyone. However, they can only be withdrawn into stake accounts that are owned by the receiver of the claim. There is, however, no check that the stake account is not locked. That means an attacker can create a locked stake account that belongs to the claim receiver, but set themselves as the lockup authority, and then execute the claim. Now the funds are inaccessible to the claim receiver, until the attacker releases the funds, opening up possibilities of extortion.

Suggestion

The receiving stake account ought to not have an active lockup.

Remediation

The issue was fixed in commit `803ca0a8c6b36834f640596cea3fd8d60e5ae433` by adding a check to the `ClaimSettlement` instruction that fails the transaction if the receiving stake account has an active lockup.

ND-MND3-L1 – No Safeguards for Compromised Operator Hot Wallet

Severity	Impact	Affected Component	Status
Low	A single compromised hot wallet can be used to withdraw all of the funds managed by the contract instantaneously	Settlements	Resolved

The operator has full control over all funds managed by the contract, and can withdraw them at will. As Marinade plans to use a hot wallet as the operator, safeguards should be put in place to limit the damage an exploiter can do in this scenario.

Suggestion

A mechanism to delay the execution of a settlement, or limit the amount of outflows out of the protocol per epoch should be implemented.

Remediation

The issue was fixed in commit `2d0e0926fee643b4b5e11d7d210f7993ec91c9ca` by adding a configuration option to `Config`, which introduces a delay between creation and execution of a settlement, as well as a way to abort a settlement before it is executed. This way either the admin or the operator can react to any settlements that were created by an attacker before any funds are lost.

ND-MND3-I1 – Unreliable Events

Severity	Impact	Affected Component	Status
Info	Events can be dropped in case of log truncation	Events	Resolved

Events are exclusively logged to the programs log. This log, however, gets truncated in case it gets too long. This means events can't be relied upon to pick up on all state changes.

Suggestion

Use CPI for event logging.

Remediation

The issue was fixed by using Anchor's `emit_cpi` feature.

ND-MND3-I2 – Vote Account Version Is Not Checked

Severity	Impact	Affected Component	Status
Info	The program can break in unpredictable ways if vote accounts layout should ever change	Everything	Resolved

Vote accounts are parsed by directly accessing and copying raw bytes out of the vote account, without a version check. As vote accounts are explicitly versioned, there's no guarantee that their layout won't change.

Suggestion

Parse the vote accounts version and assert that it's one of the known ones.

Remediation

The issue was fixed in commit `15a9b42a5feae4dd7c3d6b83ffbbabb122961a13` by following our suggestion and failing if the vote account version is not known.

A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

B | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated. We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to even find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list below.

- Rule out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Solana account confusions
  - Redeployment with cross-instance confusion
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Missing rent exemption assertion
  - Casting truncation
  - Arithmetic over- or underflows
  - Numerical precision errors
- Check for unsafe design decisions that might lead to vulnerabilities being introduced in the future
- Check for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain
- Ensure that the contract logic correctly implements the project specifications
- Examine the code in detail for contract-specific low-level vulnerabilities
- Rule out denial of service attacks
- Rule out economic attacks
- Check for instructions that allow front-running or sandwiching attacks
- Check for rug pull mechanisms or hidden backdoors

C | Vulnerability Severity Rating

Critical Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.

High Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.

Medium Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.

Low Bugs that do not have a significant immediate impact and could be fixed easily after detection.

Info Bugs or inconsistencies that have little to no security impact.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io