Security Audit – Squads Multisig v4 Updates

Lead Auditor: Robert Reith

Second Auditor: Sebastian Fritsch

Administrative Lead: Thomas Lambertz

February 6th 2024

Table of Contents

Executive Summary

Neodyme audited several updates to **Squads'** on-chain Multisig v4 program during January 2024. This audit report amends the report from September 22nd 2023. The auditors found that the updates kept the clean design and far-above-standard code quality of the existing program. According to Neodymes Rating Classification, **no critical, high, or medium vulnerabilities** and only **two low-severity issues** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

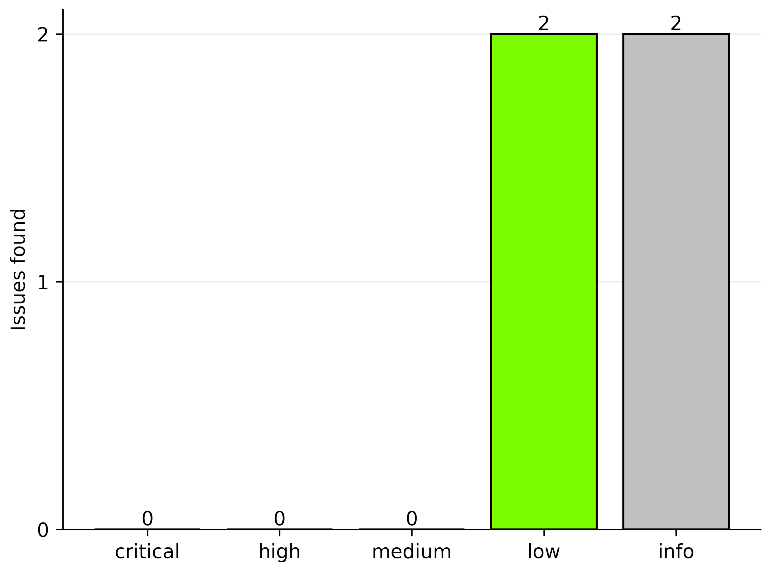


Figure 1: Overview of Findings

All findings were reported to the Squads developers, but due to the low severity they have decided with our approval to leave them as-is. In addition to these findings, Neodyme delivered the Squads team a list of nit-picks and additional notes that are not part of this report.

1 | Introduction

During January 2024, Squads commissioned Neodyme to conduct a detailed security analysis of multiple updates regarding Squads' on-chain Multisig v4 program. Previously, Neodyme has conducted a full audit of the program in September 2023.

Two senior auditors performed the audit between 11^{th} of January and 12^{th} of January. This report details all findings from this time span.

After the introduction, this report details the audit's Scope, gives a brief Overview of the Updates, then goes on to document Findings.

Neodyme would like to emphasize the high quality of Squads' work. Squads' team always responded quickly and **competent** to findings of any kind. Their **in-depth knowledge** about multisig programs was apparent during all stages of the cooperation. Evidently, Squads invested significant effort and resources into their product's security. Their **code quality is far above standard**, as the code is well documented, naming schemes are clear, and the overall architecture of the program is **clean and coherent**. The contract's source code has no unnecessary dependencies, relying mainly on the well-established anchor framework.

Findings Summary

During the audit, **2 security-relevant** and **2 informational** findings were identified. Squads accepted those findings before the protocol's launch.

In total, the audit revealed:

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

issues.

All findings are detailed in section Findings.

2 | Scope

The contract audit's scope comprised of 2 major components:

- **Implementation** security of the source code
- Security of the **overall design** including the updates

All of the source code, located at https://github.com/Squads-Protocol/v4, is in scope of this audit. However, third-party dependencies are not. As Squads only relies on the well-established Anchor library, the security-txt standard, and the address-lookup program provided by Solana, this does not seem problematic.

Relevant source code revisions are:

- `86b60cb6db82a70722ff6a9d4f2f9b0d175b0e38` • Start of the audit
- `3742e5521a3e833f24a4c6bc024dd1aa5385d010` • Latest reviewed on-chain revision
- `bfa43ad6b38487b98224c542ae85691a88917fad` • Latest reviewed revision

3 | Project Overview

This section briefly outlines the Updates proposed to the Squads Multisig v4. A more comprehensive overview of the program functionality can be found in our previous report of the Squads Multisig v4.

Rent Reclamation

The updates to the Squads Multisig v4 program add the feature of rent reclamation for accounts that users may want to close. For instance, users may want to close old transaction accounts for transactions that have been rejected. In such cases, the rent shall be returned to an account that is configured for each multisig as the `rent_collector`. This allows the closing transactions to be permissionless. This account is saved within the `Multisig` account as an `Option<Pubkey>` type. Previous versions of the Multisig had an empty padding byte at this offset, which corresponds to a `None` value within the new layout. This means that old multisigs will simply have no `rent_collector` configured, which they can change using a config transaction.

Program Config

The second feature added to the Squads Multisig v4 program in this update is the introduction of the Program Config. This is a configuration of the entire deployed Squads program. This configuration is saved at the `"program_config"` PDA. It is used to implement a multisig creation fee. To achieve this, the configuration records the public key of an authority which can modify this setting, the new multisig creation fee in Lamports, and the public key of the treasury designated to collect the fees.

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 Squads 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 1: Findings

Name	Severity
[ND-SQD2-L1] ProgramConfig Invariant Checks are Insufficient to Avoid Losing Fees	Low
[ND-SQD2-L2] multisig_remove_spending_limit Allows Anyone to Collect Rent When rent_collector is Configured	Low
[ND-SQD2-I1] Inconsistency in Spending Limit Invariants When Members are Removed	Info
[ND-SQD2-I2] Spending Limit PDAs can't be Easily Enumerated	Info

We also presented the Squads team with a list of nit-picks and other notes we had on the contract, which we omit here.

ND-SQD2-L1 – ProgramConfig Invariant Checks are Insufficient to Avoid Losing Fees

Severity	Impact	Affected Component	Status
Low	User Error	Program Config	Accepted

Description

The invariant checks of the `ProgramConfig` struct currently check that neither the authority, nor the treasury accounts are set to `Pubkey::default()`. The intention seems to be to avoid a case where these accounts are configured to be invalid which would break the program when fees are transfered. However, there is an infinite amount of other invalid options that users may set by mistake, for instance a Pubkey such as `0xffffffffffffffff`, or anything else that is not a usable Solana account.

Location

- src/state/program_config.rs#L21C5-L37C6

Relevant Code

```rust
impl ProgramConfig {
    pub fn invariant(&self) -> Result<()> {
        // authority must be non-default.
        require_keys_neq!(
            self.authority,
            Pubkey::default(),
            MultisigError::InvalidAccount
        );

        // treasury must be non-default.
        require_keys_neq!(
            self.treasury,
            Pubkey::default(),
            MultisigError::InvalidAccount
        );

        Ok(())
    }
}
```

Mitigation Suggestion

It is hard to ensure that a sensible authority account is given with the current scheme. One approach to ensure a new authority account is valid would be to enforce that it is passed as a Signer in `program_config_init` and in `program_config_set_authority`. To make sure that the given treasury account is valid, it should be a system account or a squads account. However, if it's a system account, it isn't ensured that it's a valid usable account. The simplest solution would be to let it sign the transactions when it is configured too.

Remediation

This issue has been acknowledged without a remediation, due to a lack of a good general approach to fix this issue while being low in severity.

ND-SQD2-L2 – multisig_remove_spending_limit Allows Anyone to Collect Rent When rent_collector is Configured

Severity	Impact	Affected Component	Status
Low	Inconsistency	Spending Limit	Accepted

Description

When a `rent_collector` is configured, the `multisig_remove_spending_limit` instructions still allows for anyone to collect rent, not just the configured account. This clashes with the stated purpose of the `rent_collector` which is that only this account is allowed to collect rent for accounts that are being closed.

Location

- src/instructions/multisig_remove_spending_limit.rs#L23-L29

Relevant Code

```rust
#[derive(Accounts)]
pub struct MultisigRemoveSpendingLimit<'info> {
    #[account(
        seeds = [SEED_PREFIX, SEED_MULTISIG, multisig.create_key.as_ref()],
        bump = multisig.bump,
    )]
    multisig: Account<'info, Multisig>,

    /// Multisig `config_authority` that must authorize the configuration change.
    pub config_authority: Signer<'info>,

    #[account(mut, close = rent_collector)]
    pub spending_limit: Account<'info, SpendingLimit>,

    /// This is usually the same as `config_authority`, but can be a different account if needed.
    /// CHECK: can be any account.
    #[account(mut)]
    pub rent_collector: AccountInfo<'info>,
}
```

Mitigation Suggestion

Enforce that only the configured `rent_collector` can reclaim the rent from closed spending limits.

Remediation

This issue has been acknowledged without a remediation, due to the low impact and the high complexity of fixing the inconsistency in `config_transaction_execute`.

ND-SQD2-I1 – Inconsistency in Spending Limit Invariants When Members are Removed

Severity	Impact	Affected Component	Status
Info	Inconsistency	Spending Limit	Accepted

Description

When a member is removed from the multisig, this member is not removed from existing spending limits if they were also a member of them. This creates a state where a spending limit has a member who's not a member of the multisig. This is inconsistent with the invariant that is enforced when creating a new spending limit. Furthermore, it creates the possibility that a member is removed from the multisig while being member of a spending limit, and then later being added back to the multisig. When they are added back to the multisig, they can use the old spending limit again, which may be unexpected.

Location

- src/instructions/config_transaction_execute.rs#L119-L123
- src/instructions/multisig_add_spending_limit.rs#L77-L82

Relevant Code

```
// in config_transaction_execute.rs
ConfigAction::RemoveMember { old_member } => {
    multisig.remove_member(old_member.to_owned())?;
    multisig.invalidate_prior_transactions();
}

// in multisig_add_spending_limit.rs
    fn validate(&self) -> Result<()> {
        // [...]
        // SpendingLimit members must all be members of the multisig.
        for sl_member in self.spending_limit.members.iter() {
            require!(
                self.multisig.is_member(*sl_member).is_some(),
                MultisigError::NotAMember
            );
        }
        // [...]
```

Mitigation Suggestion

We recommend enforcing that when a member leaves a multisig, they also get removed from any spending limit. Unfortunately this seems to be a tricky suggestion, because then all the relevant spending limit accounts need to be passed whenever a member is removed. This could even be impossible if too many spending limits exist for a multisig. A different way of implementing this could be a record of removed members. This could be another account that then needs to be passed to spending limit instructions, which will then remove any members on that list from the spending limit the next time it is used. Another more elegant solution could be to merge this information into the permissions of a member , to indicate a former member of a multisig.

Remediation

This issue has been acknowledged without a remediation, due to the low impact and the high complexity and reduced usability of migrating to a different approach.

ND-SQD2-I2 – Spending Limit PDAs can't be Easily Enumerated

Severity	Impact	Affected Component	Status
Info	Usability	Spending Limit	Accepted

Description

The spending limit PDA is constructed with the following seeds: `SEED_PREFIX, multisig.key().as_ref(), SEED_SPENDING_LIMIT, args.create_key.as_ref(),` where `create_key` can be arbitrary bytes. This means that a multisig can have many spending limits, and to find all of them, it is not enough to just know the address of the multisig. Instead, one needs to either enumerate all Squads program accounts, which are a lot, or look at all recorded Squads transactions that created new spending limits. As we understand, Squads is using an off-chain indexer which records such transactions to be able to easily list all spending limits for a multisig in its frontend. While this approach may work, it is not fully reliable as users will have to trust that the indexer indeed catches all spending limits. It is easy to imagine some scenarios where such an indexer may fail to correctly index a new spending limit. For instance if it encounters network problems, or when Solana crashes, or maybe when a user creates a spending limit, deletes it again, and then creates it again at the same address within the same transaction. The indexer may very well be handling these scenarios correctly, but as it is off-chain, this is hard to verify. An attack would try to create a spending limit that's not picked up by indexing during configuration of a controlled multisig. Then the attacker could have a multisig with a spending limit where the spending limit is not visible in the UI because it hasn't been indexed. This would effectively be a backdoor in a multisig.

Location

- src/instructions/multisig_add_spending_limit.rs#L44-L56

Relevant Code

```
#[account(
        init,
        seeds = [
            SEED_PREFIX,
            multisig.key().as_ref(),
            SEED_SPENDING_LIMIT,
            args.create_key.as_ref(),
        ],
        bump,
        space = SpendingLimit::size(args.members.len(), args.
           destinations.len()),
        payer = rent_payer
    )]
    pub spending_limit: Account<'info, SpendingLimit>,
```

Mitigation Suggestion

It would be better if spending limits for a multisig could be enumerated easily. This means that a user could look up all possible spending limit accounts for a multisig within constant time without relying on an off-chain indexer. This could be achieved by replacing the `create_key` seed in the spending limit PDA with something like a `u8` or a `u16` that should grow incrementally. Then it should be easy to check all the spending limits for a multisig.

Remediation

This issue has been acknowledged without a remediation, due to the low impact and the high complexity of migrating to a different approach.

A | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program up front, exploring its expected behaviour, 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 designs which might lead to common 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

B | 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.

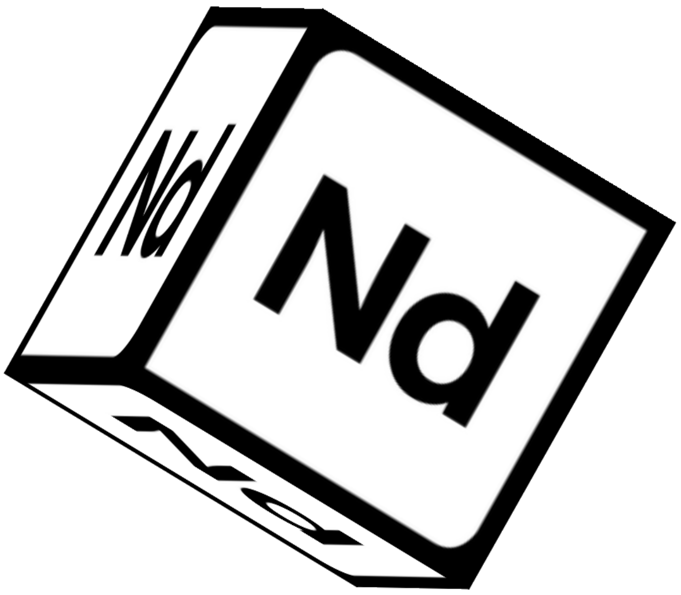
C | About Neodyme

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 we have the most qualified auditors for Solana programs in our company. 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.

Our team met as participants in hacking competitions called CTFs. There, 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 of 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.



Neodyme AG

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

https://neodyme.io