Security Audit - Squads v4

conducted by Neodyme AG

Lead Auditor:	Sebastian Fritsch
Second Auditor:	Mathias Scherer
Administrative Lead:	Thomas Lambertz

December 10, 2024

Table of Contents

1 Executive Summary **3**

2 Introduction **4**

Summary of Findings 4

3 Scope **5**

4 Project Overview **6**

Instructions 6

Transaction Buffers 6

Custom allocator 7

5 Findings **8**

[ND-SQD3-LO-01] TransactionBuffer feature can be DoSed 9

Appendices

A About Neodyme **10**

B Methodology **11**

Select Common Vulnerabilities 11

C Vulnerability Severity Rating **13**

1 | Executive Summary

Neodyme audited **Squads'** on-chain multisig v4 program from October 2024 until November 2024. This audit report amends the report from February 6th 2024, .

The auditors found that the updates kept the clean design, security and far-above-standard code quality of the existing program. According to Neodymes Rating Classification, **1 security relevant** and **0 informational** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

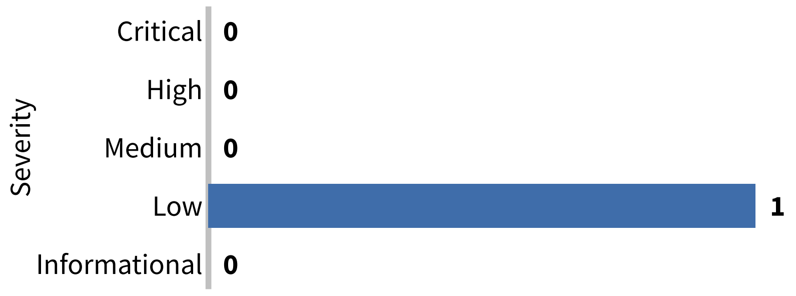


Figure 1: Overview of Findings

The auditors reported all findings to the Squads developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme. 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.

2 | Introduction

From October 2024 until November 2024, Squads engaged Neodyme to do a detailed security analysis of their Squads v4 multisig. Two senior security researchers from Neodyme conducted independent full audits of the contract between the 5th of October 2024 and the 11th of November 2024. Both auditors have a long track record of finding critical and other vulnerabilities in Solana programs, as well as in Solana's core code itself.

The audit focused on the security of the updates, that Squads implemented since the previous audit. In the following sections, the report details the audit's scope, gives a brief overview over the updates and the corresponding new instructions since the last audit . Lastly it goes into detail about the security relevant findings of this audit.

Neodyme would like to emphasize the high quality of Squads's work. Squads's team always responded quickly and **competently** to findings of any kind. Their **in-depth knowledge** of multisig programs was apparent during all stages of the cooperation, including excellent and crucial knowledge of the Solana CPI runtime. Evidently, Squads invested significant effort and resources into their product's security. Some technical debt was apparent in the contract, but only to a minor extent. 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.

Summary of Findings

All found issues were **quickly remediated**. In total, the audit revealed:

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

issues.

3 | Scope

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

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

Neodyme considers the source code, located at https://github.com/Squads-Protocol/v4/tree/main/programs/squads_multisig_program, in scope for this audit. Third-party dependencies are not in scope. Squads only relies on the Anchor library and the spl-token program, all of which are well-established. During the audit, minor changes and fixes were made by Squads, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `ca85338f6cfcfd619a1b4f5570f53d3d6d2d4335` · Start of the audit
- `dcac867070a3073929e2240a053780c324f4c29f` · Last reviewed revision

We verified that the last reviewed version corresponds to the deployed version with program hash `d48660833989ecea3145ff726164fe640bd90696f03ce00dfd0cda258cbf2fac`.

4 | Project Overview

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

Instructions

The updates since the last audit added a total of 4 instructions, which we briefly summarize here.

Instruction	Category	Summary
TransactionBufferCreate	Initiator-Only	Creates a new `TransactionBuffer` bound to the initiator (creator) with a predefined length and data hash
TransactionBufferExtend	Creator-Only	Writes data to the `TransactionBuffer`
TransactionBufferClose	Creator-Only	Closes the `TransactionBuffer`
VaultTransactionCreateFromBuffer	Creator-Only	Creates a new `VaultTransaction` with the `TransactionBuffer` data as transaction data

Transaction Buffers

Due to the way Solana formats transactions, only a limited number of bytes can be passed to a program as instruction arguments. This can result in situations where the proposed transaction for the multisig no longer fits within the original `VaultTransactionCreate` instruction.

To address this, Squads introduced the `TransactionBuffer` feature, allowing a multisig member to push proposed transaction data onto the chain in multiple iterations. When a member initializes a `TransactionBuffer` using `TransactionBufferCreate`, they must pre-define both the length and hash of the transaction data being pushed.

After that, the creator can incrementally add to the `TransactionBuffer` with `TransactionBufferExtend`. If all the data is correctly written – meaning the length and predefined hash match the actual written data – the creator can then use `VaultTransactionCreateFromBuffer` to convert the buffer into a proposed `VaultTransaction`.

A `TransactionBuffer` is derived using the following seeds:

```
[
  SEED_PREFIX, // b"multisig"
  multisig.key().as_ref(),
  SEED_TRANSACTION_BUFFER, // b"transaction_buffer"
```

```
    creator.key().as_ref(),
    &args.buffer_index.to_le_bytes() //u8
]
```

This means, every member with the `Initiate` permission (the `creator`) can maintain up to 256 buffers.

Custom allocator

In 2023, Solana introduced the `ComputeBudget` native program. One of its features, `RequestHeapFrame`, lets developers increase the heap size within the current transaction execution context. By default, the bump allocator allocates memory in reverse, starting from the default `HEAP_LENGTH` of 32 KiB. This means that a larger heap (and therefore a larger `HEAP_LENGTH`) requested via `RequestHeapFrame` can't be used by the default allocator.

To address this, Squads opted to implement a custom heap allocator for their program, which follows a forward allocation strategy. In this approach, allocation begins at the `HEAP_START_ADDRESS`, and subsequent allocations are made progressively higher in the address space. This forward allocation is fully compatible with `RequestHeapFrame`, but also works with the default heap layout.

5 | 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.

Identifier	Name	Severity	Status
ND-SQD3-LO-01	TransactionBuffer feature can be DoSed	LOW	Resolved

Table 2: Findings

[ND-SQD3-LO-01] TransactionBuffer feature can be DoSed

Severity	Impact	Affected Component	Status
LOW	TransactionBuffers are rendered unusable	TransactionBuffer	Resolved

Every multisig member with the `Initiate` permission can create `TransactionBuffer` accounts to build a transaction. Those accounts were derived in the following manner:

```
#[account(
    init,
    payer = rent_payer,
    space = TransactionBuffer::size(args.final_buffer_size)?,
    seeds = [
        SEED_PREFIX,
        multisig.key().as_ref(),
        SEED_TRANSACTION_BUFFER,
        &args.buffer_index.to_le_bytes(), //u8
    ],
    bump
)]
pub transaction_buffer: Account<'info, TransactionBuffer>,
```

This means, every `multisig` was able to have 256 different `TransactionBuffer` accounts.

These accounts can be closed in two ways:

1. By the buffer creator using the `TransactionBufferClose` instruction.
2. By any member with the `Initiate` permission via the `VaultTransactionCreateFromBuffer` instruction. For this method, the data in the buffer must also hash to a `final_buffer_hash` defined by the buffer creator as well as having the same size as the predefined `final_size`.

If a malicious member with the `Initiate` permission writes data to a buffer that does not match the predefined hash, no one except the malicious actor can close that account. By filling all 256 buffer accounts with invalid data, this effectively results in a denial-of-service (DoS) attack on the buffer account feature.

Resolution

Squads quickly remediated this issue in commit: `74fafb1ba7890d8e54cf13673c6cd7a3cf4c764a`. The `TransactionBuffer` PDA is now additionally derived with the `creator` pubkey, which allows every member with the `Initiate` permission to have 256 personal buffers instead of 256 multisig-wide buffers.

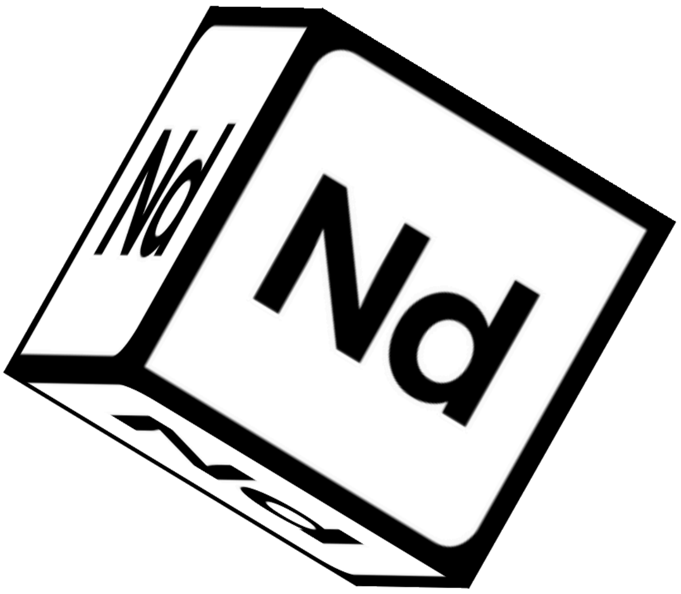
A | About Neodyme

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

We are not checklist auditors.

In fact, we pride ourselves on that. 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 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 here.

Select Common Vulnerabilities

Our most common findings are still specific to Solana itself. Among these are vulnerabilities such as the ones listed below:

- Insufficient validation, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Account confusions
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Dangerous user-controlled bumps
  - Insufficient Anchor account linkage
- Account reinitialization vulnerabilities
- Account creation DoS
- Redeployment with cross-instance confusion
- Missing rent exemption assertion
- Casting truncation
- Arithmetic over- or underflows
- Numerical precision and rounding errors
- Anchor pitfalls, such as accounts not being reloaded
- Non-unique seeds
- Issues arising from CPI recursion
- Log truncation vulnerabilities
- Vulnerabilities specific to integration of Token Extensions, for example unexpected external token hook calls

Apart from such Solana-specific findings, some of the most common vulnerabilities relate to the general logical structure of the contract. Specifically, such findings may be:

- Errors in business logic
- Mismatches between contract logic and project specifications
- General denial-of-service attacks
- Sybil attacks
- Incorrect usage of on-chain randomness
- Contract-specific low-level vulnerabilities, such as incorrect account memory management
- Vulnerability to economic attacks
- Allowing front-running or sandwiching attacks

Miscellaneous other findings are also routinely checked for, among them:

- Unsafe design decisions that might lead to vulnerabilities being introduced in the future
    - Additionally, any findings related to code consistency and cleanliness
- Rug pull mechanisms or hidden backdoors

Often, we also examine the authority structure of a contract, investigating their security as well as the impact on contract operations should they be compromised.

Over the years, we have found hundreds of high and critical severity findings, many of which are highly nontrivial and do not fall into the strict categories above. This is why our approach has always gone way beyond simply checking for common vulnerabilities. We believe that the only way to truly secure a program is a deep and tailored exploration that covers all aspects of a program, from small low-level bugs to complex logical vulnerabilities.

C | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
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.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io