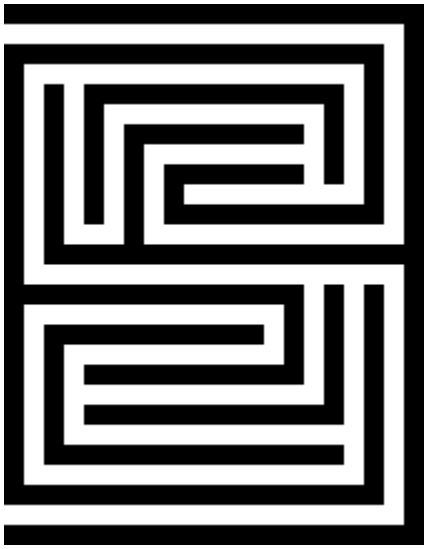


Squads Protocol V3 Public Report

PROJECT: Squads Protocol V3 Review

2022

Prepared For:

Squads Protocol

Prepared By:

Bramah Systems

info@bramah.systems

Table of Contents

Squads Protocol V3 Security Review

Executive Summary

Scope of Engagement

Bramah Systems was engaged in Spring of 2022 to perform a comprehensive security review of the Squads Protocol Rust repository. Our review was conducted over the period of two weeks.

Bramah's review pertains to Rust code (*.rs) as of commit **c367f219a230a4879286869f7135be4ccf9846c3.**

Engagement Goals

The primary scope of the engagement was to evaluate and establish the overall security of the Squads Protocol, with a specific focus on trading actions. In specific, the engagement sought to answer the following questions:

- Is it possible for an attacker to manipulate the code?
- Does the Rust code match the specification as provided?
- Is there a way to interfere with the software mechanisms?
- Are the arithmetic calculations trustworthy?

Protocol Specification

A basic specification document was compiled by the review team based upon review of the Squads Protocol code and discussion with the team.

Overall Assessment

Bramah Systems was engaged to evaluate and identify multiple security concerns in the codebase of the Squads Protocol architecture. During the course of our engagement, Bramah Systems identified multiple areas of security concern. The team acted rapidly and diligently in remediating surfaced concerns, and has introduced additional controls into their development process to surface potential vulnerabilities.

As the code does make heavy usage of third party library code, the Squads Protocol team should stay abreast of any security considerations of these protocols. This noted, the primary dependency the protocol possesses is for Anchor, which introduces numerous developer enhancements and is widely suggested for usage by the Solana ecosystem writ large.

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held, commercially reasonable understanding of Bramah Systems knowledge of security patterns as they relate to the Squads Protocol, with the understanding that distributed ledger technologies ("DLT") remain under frequent and continual development, and resultantly carry with them unknown technical risks and flaws. The scope of the review provided herein is limited solely to items denoted within "Scope of Engagement" and contained within "Directory Structure". The report does NOT cover, review, or opine upon security considerations unique to the Rust compiler, tools used in the development of the protocol, or distributed ledger technologies themselves, or to any other matters not specifically covered in this report.

The contents of this report must NOT be construed as investment advice or advice of any other kind. This report does NOT have any bearing upon the potential economics of the Squads Protocol or any other relevant product, service or asset of Squads Protocol or otherwise. This report is not and should not be relied upon by Squads Protocol or any reader of this report as any form of financial, tax, legal, regulatory, or other advice.

To the full extent permissible by applicable law, Bramah Systems disclaims all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. Bramah Systems makes no warranties, representations, or guarantees about the Squads Protocol. Use of this report and/or any of the information provided herein is at the users sole risk, and Bramah Systems. hereby disclaims, and each user of this report hereby waives, releases, and holds Bramah Systems harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Activities undertaken by Bramah Systems were part of a mutually agreed upon scope of work that was presented and signed prior to engagement commencement. Deviation from this scope of work was recorded where present.

Our engagements are explicitly time-boxed and rely upon claims made by clients, their partners, and their affiliates. As such, this report should not be considered a comprehensive list of security flaws, issues, or defects in the provided codebase.

Timeliness of Content

All content within this report is presented only as of the date published or indicated, to the commercially reasonable knowledge of Bramah Systems, as of such date, and may be superseded by subsequent events or for other reasons. The content contained within this report is subject to change without notice. Bramah Systems, does not guarantee or warrant

the accuracy or timeliness of any of the content contained within this report, whether accessed through digital means or otherwise.
Bramah Systems is not responsible for setting individual browser cache settings nor can it ensure any parties beyond those individuals directly listed within this report are receiving the most recent content as reasonably understood by Bramah Systems as of the date this report is provided to such individuals.

Specific Recommendations

Unique to Squads Protocol

Missing Instruction Account Key Validation in Execute Transaction

The execute transaction instruction is used to begin execution of an agreed upon multisig transaction. Transactions will have preconfigured instructions each marked with a predefined set of accounts that should be passed in when these instructions are executed.

During the actual execution of these instructions, the accounts required to perform the transaction are passed in using `ctx.remaining_accounts` . Because `remaining_accounts` include none of Anchor's typical guarantees, additional checks must be performed.

```
// check the instruction
if &ix_pda != ms_ix_account.key {
    return err!(MsError::InvalidInstructionAccount);
}
// get the instructions program account
let ix_program_info: &AccountInfo = next_account_info(ix_iter)?;
if &ms_ix.program_id != ix_program_info.key {
    return err!(MsError::InvalidInstructionAccount);
}
```

While some validation is performed that ensures that both the instruction and the program account are the expected accounts (*as seen above*), the actual instruction argument accounts passed into `invoke_signed` call are *not* validated. This means that the caller can change the accounts passed in to the call, changing the ultimate behavior of the instruction at execution

time.

```
for _ in 0..ms_ix.keys.len() {
    let ix_account_info: &AccountInfo<'info> = next_account_info(ix_iter)?;
    ix_account_infos.push(ix_account_info.clone());
}
```

Resolution: A fix was implemented by the team as of **fb6794e71f53d217615d5215203feff7e06544c1**.

Missing Constraint in Both MsAuth and MsAuthRealloc Allows Writing Arbitrary ms_change_index

A common constraint in the Squads program ensures that transactions cannot be edited, voted on, or executed that were created before a transaction that changes the Multisig in a way that impacts voting. This encompasses any transaction that changes the threshold or members in the Multisig and are referred to as Internal Transactions.

```
#[account(
    ..., // other conditions here
    constraint = transaction.transaction_index > multisig.ms_change_index
@MsError::DeprecatedTransaction,
)]
pub transaction: Account<'info, MsTransaction>,
```

The last internal transaction that changed a Multisig in a way that requires invalidating other transactions is by using the Multisig's `ms_change_index`. If the `transaction_index` is less than or equal to the `ms_change_index` on the Multisig, the transaction is considered invalid.

The instructions that change the Multisig in a way that would result in the `ms_change_index`

updating use `MsAuth` or `MsAuthRealloc` as guards.

```
pub fn add_member(ctx: Context<MsAuthRealloc>, new_member: Pubkey) -> Result<()> {
    if ctx.accounts.multisig.keys.len() >= usize::from(u16::MAX) {
        return err!(MsError::MaxMembersReached);
    }
    // ...
```

Neither `MsAuth` nor `MsAuthRealloc` correctly enforce that the Transaction is either the Transaction that triggered this change or even if that transaction is associated with the Multisig! This means that updates to members or threshold can be marked with an unexpected `ms_change_index` improperly invalidating or allowing future Transactions.

```
#[derive(Accounts)]
pub struct MsAuth<'info> {
    #[account(mut)]
    multisig: Box<Account<'info, Ms>>,
    #[account(
        constraint = transaction.status == MsTransactionStatus::ExecuteReady
@MsError::InvalidTransactionState,
        constraint = transaction.transaction_index > multisig.ms_change_index
@MsError::DeprecatedTransaction,
    )]
    transaction: Box<Account<'info, MsTransaction>>, // <-- doesn't validate this is associated
with multisig nor the change that actually triggered the change

    // other stuff
}
```

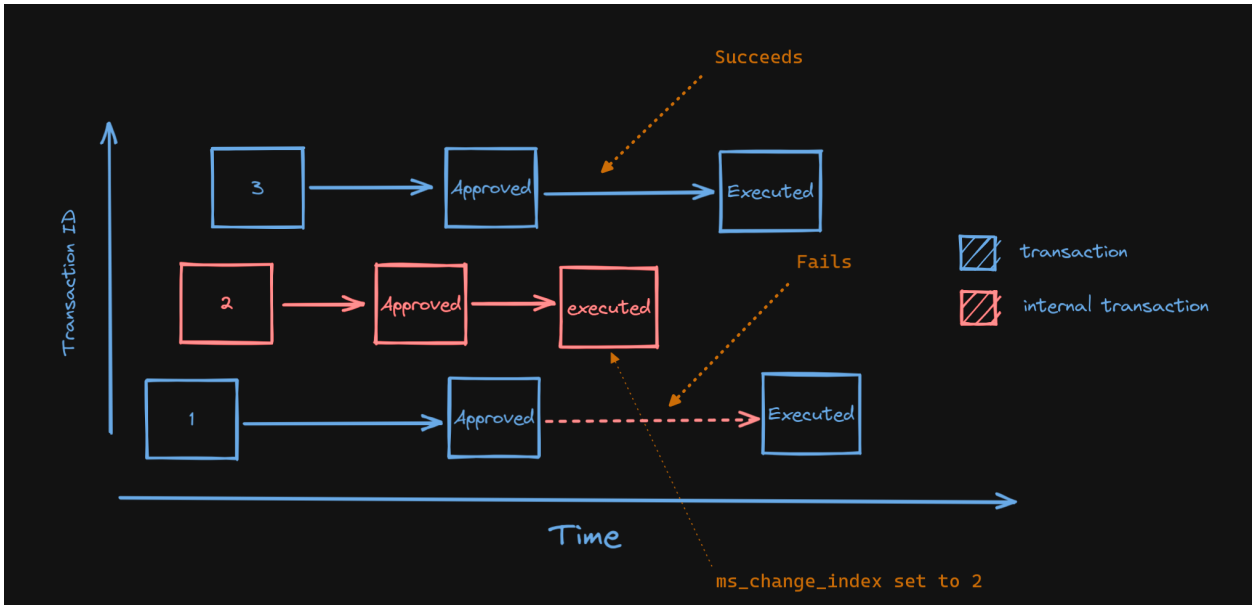
Resolution: A fix was implemented by the team as of

fb6794e71f53d217615d5215203feff7e06544c1.

Improper Transaction Invalidation When Performing Internal Transaction

Internal transactions are expected to change the `ms_change_index` which is used to invalidate other transactions. This is because internal transactions can change properties like membership and voting threshold, which would impact the result and validity of in-process transactions.

It appears that the design of this feature is flawed. Because it uses the assigned transaction index, regardless of when that internal transaction is actually executed, it can allow transactions created after the internal transaction to continue to be voted on. Here is a diagram demonstrating how transaction 3, despite a previous internal transaction executing, is still being allowed to execute even though transaction 1 *does* fail as expected.



The suggested solution is to change `ms_change_index` to the Multisig's current `transaction_index` instead of using the transaction index of the internal transaction. This way, all prior transactions are invalidated when an internal transaction is executed. This also has the side effect of making the fix for Improper Transaction Invalidation When Performing Internal Transaction significantly easier as the Transaction account can be removed from

`MsAuth` and `MsAuthRealloc`.

This appears to present the most fair and expected experience.

Resolution: A fix was implemented by the team as of **fb6794e71f53d217615d5215203feff7e06544c1**.

Missing Constraint Allows Any Party To Execute Transaction

Squad's documentation describes the typical process that participants can expect when interacting with Transactions. In the third step, "Execute", the following is said.

> *Once a transaction reaches the confirmation threshold it can be executed by any member of the Squad.*

Due to a missing constraint in the instruction guard, it appears that *any* party, regardless of whether or not they're a member, can execute the transaction.

```
#[derive(Accounts)]
pub struct ExecuteTransaction<'info> {
    #[account(
        mut,
        seeds = [
            b"squad",
            multisig.creator.as_ref(),
            b"multisig"
        ], // <-- missing constraint here!
        bump = multisig.bump,
    )]
    pub multisig: Box<Account<'info, Ms>>,
```

```
    // other stuff ...

    #[account(mut)]
    pub member: Signer<'info>, // <-- this can be anyone!
    pub system_program: Program<'info, System>
}
```

Resolution: A fix was implemented by the team as of **fb6794e71f53d217615d5215203feff7e06544c1**.

Potential Incorrect Space Reallocation In Add Member

It appears that excessive space is allocated in the `add_member` internal transaction instruction. The Multisig account starts out with enough space to hold 10 member keys without reallocation. When a new member is added, the following logic is used:

```
let multisig_account_info = ctx.accounts.multisig.to_account_info();
let curr_data_size = multisig_account_info.data.borrow().len();
let next_len = curr_data_size + 32;
if next_len > curr_data_size{
    let needed_len = curr_data_size + ( 10 * 32 );
```

This logic appears incorrect. The correct way to determine if the account needs to be resized would be taking the current size of the data field, subject `57`, divide by `32`, and subtract the number of current members to determine how many spots are available. `57` is because Multisig have the following fields and sizes:

```
pub threshold: u16,         // 2 bytes
pub authority_index: u16,   // 2 bytes
pub transaction_index: u32, // 4 bytes
pub ms_change_index: u32,   // 4 bytes
```

```
pub bump: u8,              // 1 byte
pub creator: Pubkey,       // 32 bytes
pub keys: Vec<Pubkey>,     // 4 + (32 * n) bytes
```

Anchor accounts also have an additional 8 bytes set aside for the discriminator.

Resolution: A fix was implemented by the team as of **fb6794e71f53d217615d5215203feff7e06544c1**.

Toolset Warnings

Unique to the Squads Protocol

Overview

In addition to our manual review, our process involves utilizing concolic analysis and dynamic testing in order to perform additional verification of the presence security vulnerabilities. An additional part of this review phase consists of reviewing any automated unit testing frameworks that exist.
The following sections detail warnings generated by the automated tools and confirmation of false positives where applicable, in addition to findings generated through manual inspection.

Compilation Warnings

Compilation warnings were not encountered.

Test Coverage

The contract repository possesses substantial unit test coverage. This testing provides a variety of unit tests which encompass the various operational stages of the protocol

Static Analysis Coverage

The contract repository underwent heavy scrutiny with multiple static analysis agents, including:

- Semgrep

Where applicable, Bramah manually performed validation checks based upon our understanding of the tool.

Directory Structure

At time of review, the directory structure of the Squads Protocol appeared as it does below. Our review, at request of Squads Protocol pertains to Rust code (*.rs) as of commit **c367f219a230a4879286869f7135be4ccf9846c3**.

.

```
├── Anchor.toml
├── Cargo.lock
├── Cargo.toml
├── app
│   ├── index.ts
│   ├── package.json
│   ├── tsconfig.json
│   └── yarn.lock
├── helpers
│   └── transactions.ts
├── migrations
│   └── deploy.ts
├── package.json
├── programs
│   └── squads-mpl
│       ├── Cargo.toml
│       ├── Xargo.toml
│       └── src
│           ├── errors.rs
│           ├── lib.rs
│           └── state
│               ├── mod.rs
│               └── ms.rs
```

Soteria Output

Cargo.toml: anchor_lang version: 0.24.2

anchor_lang_version: 0.24.2 anchorVersionTooOld: 0

EntryPoints:

entrypoint

detected 0 untrustful accounts in total.

detected 0 unsafe math operations in total.