Squads

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Akash Gurugunti Sud0u53r.ak@osec.io

Contents

01 **Executive Summary** 2

Overview 2

Key Findings 2

02 **Scope** 3

03 **Findings** 4

04 **Vulnerabilities** 5

OS-SQD-ADV-00 [low] [resolved] | Incorrect Threshold Checks 6

05 **General Findings** 8

OS-SQD-SUG-00 [resolved] | Improper Space Calculation at Creation 9

OS-SQD-SUG-01 [resolved] | Enforce Signed Multisig Program 11

OS-SQD-SUG-02 [resolved] | General Refactoring and Code Duplication 12

OS-SQD-SUG-03 [resolved] | Potential Unnecessary Call to Transfer 14

Appendices

A **Program Files** 15

B **Procedure** 16

C **Implementation Security Checklist** 17

D **Vulnerability Rating Scale** 19

01 | Executive Summary

Overview

Squads engaged OtterSec to perform an assessment of the `squads-mpl` program.

This assessment was conducted between July 12th and July 21st, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches July 25th, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 5 findings total
- No vulnerabilities which could lead to loss of funds

We also observed the following.

- Code quality of the program was high and overall design was solid
- The team was very knowledgeable and responsive to our feedback

02 | Scope

The source code was delivered to us in a git repository at github.com/squads-dapp/squads-mpl/. This audit was performed against commit `dea44c5`.

There was a total of one program included in this audit. A brief description of the program is as follows. A full list of program files and hashes can be found in Appendix A.

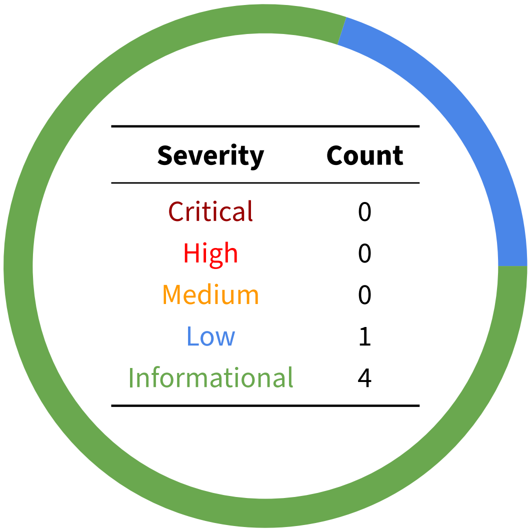
Name	Description
squads-mpl	Onchain multisig
program-manager	Utility program to manage program deployments

03 | Findings

Overall, we report 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix D.

ID	Severity	Status	Description
OS-SQD-ADV-00	Low	Resolved	Threshold checks do not account for duplicate members.

OS-SQD-ADV-00 [low] [resolved] | Incorrect Threshold Checks

Description

The threshold for multisig approval should never be greater than the number of members. The `create` instruction verifies this by comparing it against the length of the `members` vector, which is passed in as an argument.

lib.rs RUST

```rust
// since creator is considered a member, check we don't exceed u16 - very unlikely
let total_members = members.len();
if total_members < 1 {
    return err!(MsError::EmptyMembers);
}

//make sure we don't exceed on first call - not likely but this shoudl be here
if total_members > usize::from(u16::MAX) {
    return err!(MsError::MaxMembersReached);
}

//make sure threshold is valid
if !(1..=total_members).contains(&usize::from(threshold)) {
    return err!(MsError::InvalidThreshold);
}
```

However, these checks are conducted before duplicate members are removed, an action which would reduce the vector's length.

lib.rs RUST

```rust
// check that the creator isn't in the member list, they'll be added automatically
let mut members = members;
members.sort();
members.dedup();
```

If duplicate members are passed into the instruction, it is possible for the resulting multisig to have an impossibly high threshold.

Proof of Concept

Consider the following scenario:

1. Alice calls the `create` instruction with a threshold of 5 and a `members` vector consisting of one public key repeated 10 times.

The resulting multisig has 1 member, yet the threshold is 5.

Remediation

The `create` instruction should sort and remove duplicate members before performing other checks.

Patch

Resolved in 3c2139c.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-SQD-SUG-00	Resolved	Improper space calculations leads to confusing errors on creation
OS-SQD-SUG-01	Resolved	Missing `program_id` check for transactions with an `authority_index` equal to 0.
OS-SQD-SUG-02	Resolved	General refactoring to improve readability.
OS-SQD-SUG-03	Resolved	Avoid an unnecessary call to the system transfer for zero lamports

OS-SQD-SUG-00 [resolved] | Improper Space Calculation at Creation

Description

The space initially allocated for the Ms state account should depend on the number of members at creation. Furthermore, it does not check whether the provided data would exceed capacity.

However, the program allocates a fixed amount, which is implicitly enough for 10 members.

squads-mpl/src/state/ms.rs RUST

```rust
impl Ms {
    pub const SIZE_WITHOUT_MEMBERS: usize = 8 + // Anchor disriminator
    2 +         // threshold value
    2 +         // authority index
    4 +         // transaction index
    4 +         // processed internal transaction index
    1 +         // PDA bump
    32 +        // creator
    1 +         // allow external execute
    4;          // for vec length

    pub const MAXIMUM_SIZE: usize = (32 * 10) +
     ↪  Self::SIZE_WITHOUT_MEMBERS; // initial space for 10 keys
```

Remediation

Modify the space constraint in the Create struct to use a dynamic value that is dependent on members rather than a fixed constant.

Alternatively, return a more explicit error when exceeding the implicit 10 member threshold.

Patch

Dynamically calculate Ms space, fixed in 2220864.

squads-mpl/src/state/ms.rs RUST

```rust
#[derive(Accounts)]
#[instruction(threshold: u16, create_key: Pubkey, members:
     ↪  Vec<Pubkey>)]
pub struct Create<'info> {
    #[account(
        init,
        payer = creator,
        space = Ms::SIZE_WITHOUT_MEMBERS + (members.len() * 32),
```

OS-SQD-SUG-01 [resolved] | Enforce Signed Multisig Program

Description

When calling instructions inside a transaction created with an `authority_index` equal to 0, the multisig PDA account is sent as a signed account. This is designed to be used for calling internal instructions like adding or removing multisig members.

However, for these transactions, the program ID of the instructions is not checked. This leads to a possibility where you could drain the lamports out of the multisig account itself, causing it to become no-longer rent exempt.

Remediation

Add a constraint to the `ExecuteTransaction` instruction that ensures that `program_id == id()` if `transaction.authority_index = 0`.

Patch

Added `program_id` constraint, resolved in aa62d18 and d200f3d.

squads-mpl/src/state/lib.rs RUST

```rust
    // make sure internal transactions have a matching program id for
    ↪  attached instructions
    if tx.authority_index == 0 && &incoming_instruction.program_id !=
    ↪  ctx.program_id {
        return err!(MsError::InvalidAuthorityIndex);
    }
```

OS-SQD-SUG-02 [resolved] | General Refactoring and Code Duplication

Description

Some code refactoring can be made to improve readability and remove unnecessary code.

1. In the below examples, the commented out code can be replaced with the uncommented code.

   squads-mpl/src/lib.rs RUST

   ```rust
   // let max_ix_index = ctx.accounts.transaction.instruction_index
        ↪  + 1;
   // (1..max_ix_index)
   (1..=ctx.accounts.transaction.instruction_index)
   ```

   squads-mpl/src/lib.rs RUST

   ```rust
   // if !(1..=total_members).contains(&usize::from(threshold)) {
   if (threshold < 1 || threshold > total_members) {
   ```

   squads-mpl/src/lib.rs RUST

   ```rust
   // let acc = &ctx.remaining_accounts[index].clone();
   // acc.clone()
   &ctx.remaining_accounts[index].clone()
   ```

2. Refactor the `add_member_and_change_threshold` function to use the `change_threshold` function instead of duplicating the code.
3. The structs `ApproveTransaction` and `RejectTransaction` use the exact same fields with the exact same constraints. This redundant code can be eliminated by using a single struct for both the instructions.
4. Refactor `*_index` fields in state accounts that use u16/u32 to u64 to avoid any risk of overflows.
5. Refactor `execute_transaction` to use `execute_instruction`. Also reverify the `execute` flag on the instruction prior to executing the individual instruction an a transaction to be safe.
6. Consider having unique string seed prefixes to avoid any risk of collision between `program-manager` and `squads-mpl`

Remediation

Refactor the code as suggested.

Patch

Resolved in 850858c and 91ce590.

OS-SQD-SUG-03 [resolved] | Potential Unnecessary Call to Transfer

Description

In the AddMember instruction, a call to system_instruction::transfer is made to transfer lamports from the member account to the multisig account to allocate space for the extra slots required.

This call should be avoided if top_up_lamports is zero.

Remediation

Add a constraint that checks to ensure that top_up_lamports > 0 before invoking the transfer instruction.

Patch

Resolved in e7108e1.

squads-mpl/src/lib.rs RUST

```rust
    if top_up_lamports > 0 {
        invoke(
            &transfer(ctx.accounts.member.key,
 ↪  &ctx.accounts.multisig.key(), top_up_lamports),
            &[
                ctx.accounts.member.to_account_info().clone(),
                multisig_account_info.clone(),
                ctx.accounts.system_program.to_account_info().clone(),
            ],
        )?;
    }
```

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
program-manager
  Cargo.toml                                                48666c1bbb0cdeafb4a0c25cbe5f5db7
  Xargo.toml                                                815f2dfb6197712a703a8e1f75b03c69
  src
    lib.rs                                                  6f6202a1d0f4a9c14485211412693213
    state
      mod.rs                                                3064490501d53ed56365776b74b4c723
      pm.rs                                                 7793e10ef2606022677dae3af3db9826
squads-mpl
  Cargo.toml                                                a5e1d782f5af65155af6b1d1bab40dc1
  README.md                                                 dd5f911956eebeb7fc9b66d0c4c0ad8d
  Xargo.toml                                                815f2dfb6197712a703a8e1f75b03c69
  src
    errors.rs                                               54fb8b785b7ddb83652cb384ad54d58d
    lib.rs                                                  8aad0b5dd6c6fb338635707927cdeba0
    state
      mod.rs                                                8e28aa325e72c37057eff39700b750dd
      ms.rs                                                 d7934d74a2e3013ecc7c65a22305abfa
```

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix C.

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

C | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

D | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical

Vulnerabilities which immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority/token account validation
- Rounding errors on token transfers

High

Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions
- Exploitation involving high capital requirement with respect to payout

Medium

Vulnerabilities which could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input cause computation limit exhaustion
- Forced exceptions preventing normal use

Low

Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants
- Improved input validation
- Uncaught Rust errors (vector out of bounds indexing)