# SPL Governance v3

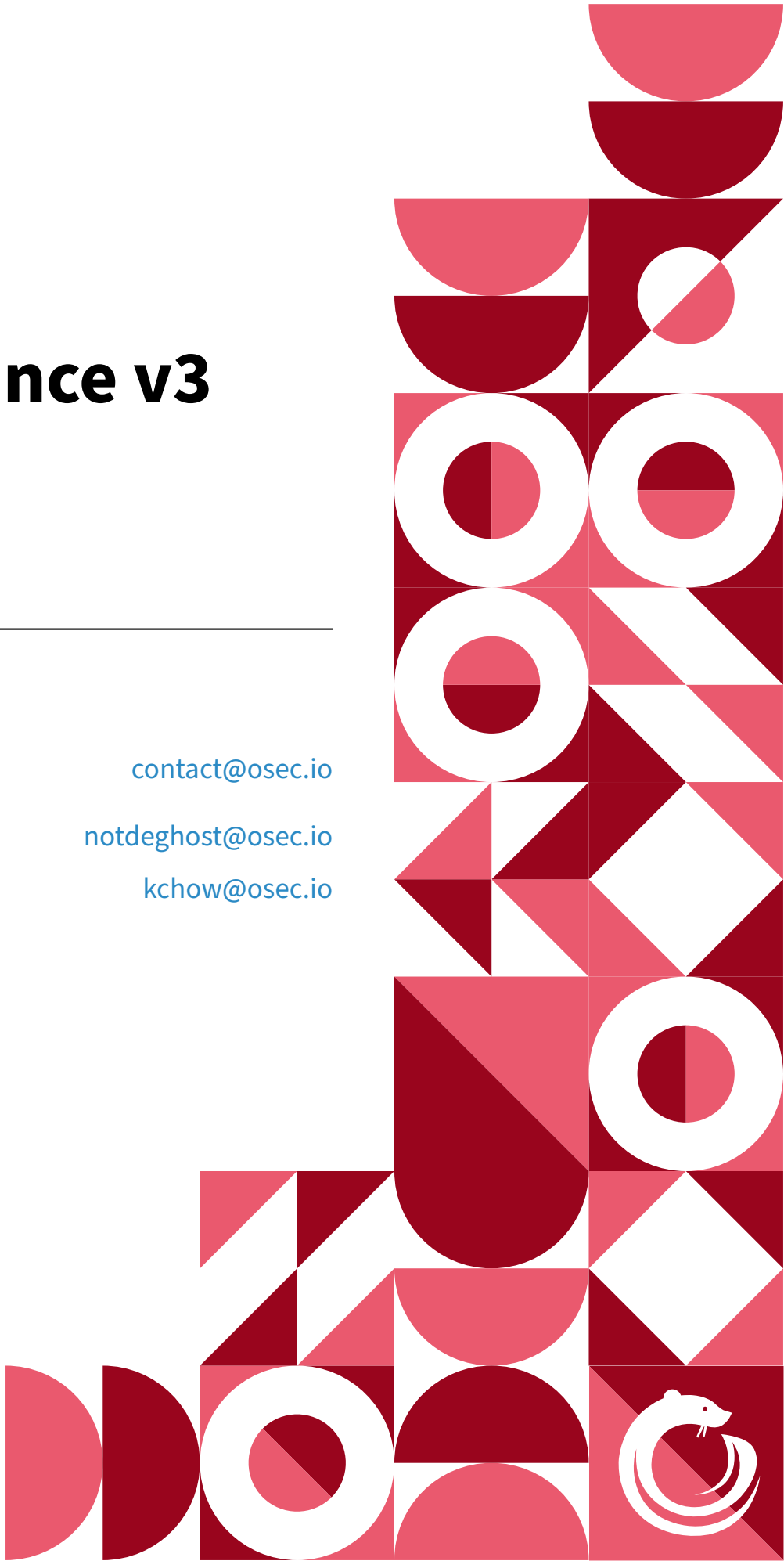
# Audit


Presented by:

**OtterSec** contact@osec.io

**Robert Chen** notdeghost@osec.io

**Kevin Chow** kchow@osec.io

# Contents



## Appendices

# 01 | Executive Summary

## Overview

Solana Labs engaged OtterSec to perform an assessment of `spl-governance v3`.

This is an ongoing assessment, but we delivered this intermediate report on September 12th, 2022.

## Key Findings

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

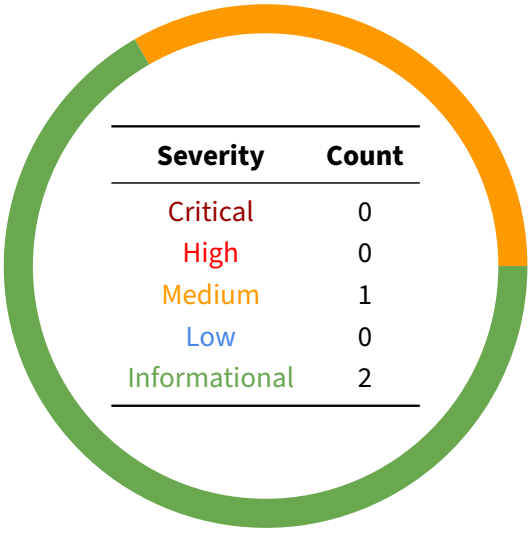
- 3 findings total
- 0 vulnerabilities which could lead to loss of funds

# 02 | Findings

Overall, we report 3 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 | 1 |
| Low | 0 |
| Informational | 2 |

# 03 | 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 C.

| ID | Severity | Status | Description |
|---|---|---|---|
| OS-GOV-ADV-00 | Medium | Resolved | Voter weight manipulation by burning |

## OS-GOV-ADV-00 [med] [resolved] | Voter weight manipulation by burning after vote

### Description

A voter can influence their vote weight after voting ends and before proposal finalization. The voter relinquishes his/her vote, withdraws their governing tokens, burns to lower the mint supply (and therefore `max_voter_weight`) and finalizes.

### Proof of Concept

1. Deposit 33% of mint supply
2. Create a proposal and vote on it.
3. Between voting time ending and proposal finalization, relinquish vote and withdraw tokens. The vote persists.
4. Burn withdrawn tokens (33%) and finalize. 33/66 = 50%

*governance/program/src/processor/process_relinquish_vote.rs* RUST

```
} else {
    vote_record_data.is_relinquished = true;
    vote_record_data.serialize(&mut
    ↪  *vote_record_info.data.borrow_mut())?;
}

// If the Proposal has been already voted on then we only have to
    ↪  decrease unrelinquished_votes_count
token_owner_record_data.unrelinquished_votes_count =
    ↪  token_owner_record_data
    .unrelinquished_votes_count
    .checked_sub(1)
    .unwrap();

token_owner_record_data.serialize(&mut
    ↪  *token_owner_record_info.data.borrow_mut())?;

Ok(())
```

### Remediation

Prevent vote relinquishment before the vote is finalized. Resolved in #3210.

*governance/program/src/processor/process_relinquish_vote.rs* RUST

```rust
} else {
    // After Proposal voting time ends and it's not tipped then it
     ↪  enters implicit (time based) Finalizing state
    // and releasing tokens in this state should be disallowed
    // In other words releasing tokens is only possible once Proposal
     ↪  is manually finalized using FinalizeVote
    if proposal_data.state == ProposalState::Voting {
        return
     ↪  Err(GovernanceError::CannotRelinquishInFinalizingState.into());
    }

    vote_record_data.is_relinquished = true;
    vote_record_data.serialize(&mut
     ↪  *vote_record_info.data.borrow_mut())?;
}
// If the Proposal has been already voted on then we only have to
     ↪  decrease unrelinquished_votes_count
token_owner_record_data.unrelinquished_votes_count =
     ↪  token_owner_record_data
    .unrelinquished_votes_count
    .checked_sub(1)
    .unwrap();
token_owner_record_data.serialize(&mut
     ↪  *token_owner_record_info.data.borrow_mut())?;
Ok(())
```

# 04 | 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 | Description |
| --- | --- |
| OS-GOV-SUG-00 | Add safeguards/warnings to membership |
| OS-GOV-SUG-01 | Revoke membership should check and revoke votes |

## OS-GOV-SUG-00 | Add safeguards/warnings to membership token minting/burning

### Description

`spl-governance` issues membership tokens that never interact with a voter via an authority minting and burning tokens directly into a holding account. However, these actions are sensitive because `max_voter_weight` is mint supply by default. Ideally these actions only occur when there are no live proposals, and the ability to create new proposals is disabled. Additionally, a mint and/or burn series should occur in a single transaction, or a minority could have more vote weight than intended.

*governance/src/processor/process_deposit_governing_tokens.rs* RUST

```
    if is_spl_token_account(governing_token_source_info) {
        // If the source is spl-token token account then transfer
        ↪ tokens from it
        transfer_spl_tokens(
            governing_token_source_info,
            governing_token_holding_info,
            governing_token_source_authority_info,
            amount,
            spl_token_info,
        )?;
    } else if is_spl_token_mint(governing_token_source_info) {
        // If it's a mint then mint the tokens
        mint_spl_tokens_to(
            governing_token_source_info,
            governing_token_holding_info,
            governing_token_source_authority_info,
            amount,
            spl_token_info,
        )?;
    } else {
        return
        ↪ Err(GovernanceError::InvalidGoverningTokenSource.into());
    }
```

### Remediation

The Solana Labs team communicated that they intend to support a queuing mechanism to enforce proposal order in the future.

## OS-GOV-SUG-01 | Revoke membership should check and revoke votes

### Description

When `mint_authority` revokes membership tokens, there should be a check for any live votes, and the ability for a DAO to relinquish those votes.

*governance/src/processor/process_revoke_governing_tokens.rs* RUST

```
token_owner_record_data.governing_token_deposit_amount =
    ↪  token_owner_record_data
    .governing_token_deposit_amount
    .checked_sub(amount)
    .ok_or(GovernanceError::InvalidRevokeAmount)?;

token_owner_record_data.serialize(&mut
    ↪  *token_owner_record_info.data.borrow_mut())?;

burn_spl_tokens_signed(
    governing_token_holding_info,
    governing_token_mint_info,
    realm_info,
    &get_realm_address_seeds(&realm_data.name),
    program_id,
    amount,
    spl_token_info,
)?;
```

### Remediation

The Solana Labs team has noted they intend to support vote relinquishment for DAOs in the future.

# A | 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 B.

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.

# B | 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. |

# C | 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<br>Examples:<br>• Misconfigured authority/token account validation<br>• Rounding errors on token transfers |
| **High** | Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit.<br>Examples:<br>• Loss of funds requiring specific victim interactions<br>• Exploitation involving high capital requirement with respect to payout |
| **Medium** | Vulnerabilities which could lead to denial of service scenarios or degraded usability.<br>Examples:<br>• Malicious input cause computation limit exhaustion<br>• Forced exceptions preventing normal use |
| **Low** | Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.<br>Examples:<br>• Oracle manipulation with large capital requirements and multiple transactions |
| **Informational** | Best practices to mitigate future security risks. These are classified as general findings.<br>Examples:<br>• Explicit assertion of critical internal invariants<br>• Improved input validation<br>• Uncaught Rust errors (vector out of bounds indexing) |