Soroban Governor

Security Assessment

July 4th, 2024 — Prepared by OtterSec

Andreas Mantzoutas andreas@osec.io

Robert Chen r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Script3 engaged OtterSec to assess the `soroban-governor` program. This assessment was conducted between June 10th and June 18th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 8 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the balance transfer functionality sends tokens to the user if the `from` and `to` addresses are the same, due to incorrect caching (OS-SGR-ADV-00). Furthermore, the Time To Live for total supply checkpoints is set to eight days, which may not be sufficient for scenarios where the voting period plus the grace period exceeds this duration (OS-SGR-ADV-02). Additionally, we highlighted a centralization risk where the governor's contract allows the council cancel any proposal, including settings changes (OS-SGR-ADV-05).

We also recommended rewriting the voting calculations by utilizing multiplication and integer conversion to avoid precision loss (OS-SGR-SUG-01). Additionally, we advised emitting an event when a proposal expires (OS-SGR-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/script3/soroban-governor. This audit was performed against commit 0a77889.

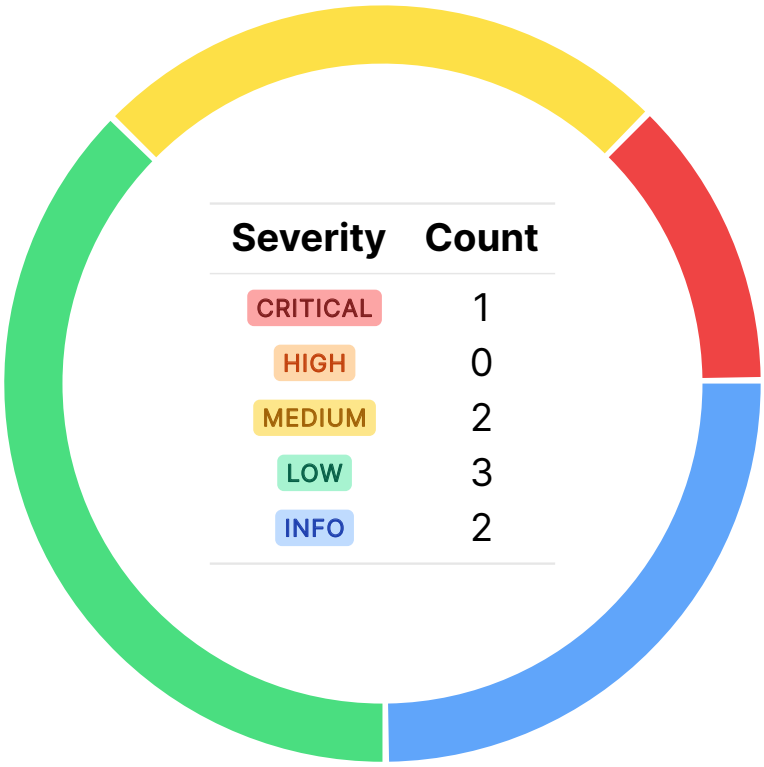
A brief description of the programs is as follows:

Name	Description
soroban-governor	A comprehensive governance framework based on the OpenZeppelin Governance system, optimized for the Soroban environment and supporting Stellar Classic assets.

03 — Findings

Overall, we reported 8 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	0
MEDIUM	2
LOW	3
INFO	2

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 A.

ID	Severity	Status	Description
OS-SGR-ADV-00	CRITICAL	RESOLVED	`transfer_balance` sends tokens to the user if the `from` and `to` addresses are same, due to incorrect caching.
OS-SGR-ADV-01	MEDIUM	RESOLVED	The current logic in `checkpoints` incorrectly assumes `vote_ledgers` are sorted, performing a binary search relying on this assumption, which introduces inconsistencies in how votes are counted or verified.
OS-SGR-ADV-02	MEDIUM	RESOLVED	The Time To Live for total supply checkpoints is set to eight days, which may not be sufficient for scenarios where the voting period, plus the grace period, exceeds this duration.
OS-SGR-ADV-03	LOW	RESOLVED	The current validation allows settings with a zero grace period, which may permanently stall proposals since `close` relies on the grace period to differentiate between ongoing voting and proposals taking too long.
OS-SGR-ADV-04	LOW	RESOLVED	`allowance` in the Soroban contract does not consider expiration times for SEP-41 allowances. This means it may incorrectly report a non-zero allowance even if it has expired.
OS-SGR-ADV-05	LOW	RESOLVED	The governor's contract allows the council to cancel any proposal, including setting changes.

Balance Cache Inconsistency CRITICAL OS-SGR-ADV-00

Description

There is a potential vulnerability in `balance::transfer_balance` related to caching balances before and after the transfer, especially when the sender (`from`) and receiver (`to`) addresses are the same. The function retrieves the balances (`from_balance` and `to_balance`) for the sender and receiver addresses before performing the actual transfer.

>_ *votes/src/balance.rs* rust

```rust
pub fn transfer_balance(e: &Env, from: &Address, to: &Address, amount: i128) {
    let from_balance = storage::get_balance(e, from);
    if from_balance < amount {
        panic_with_error!(e, TokenVotesError::BalanceError);
    }
    let to_balance = storage::get_balance(e, to);
    [...]
    storage::set_balance(e, from, &(from_balance - amount));
    storage::set_balance(e, to, &(to_balance + amount));
}
```

However, there is no check to ensure that these cached values are utilized consistently throughout the transfer process. If `from` and `to` are the same address (self-transfer), the vulnerability arises. The function subtracts the transfer amount (`amount`) from the cached `from_balance` . Since `from` and `to` are the same, it then adds the same amount back to the same cached `to_balance` . In effect, this discrepancy allows the user to obtain free tokens.

Remediation

Refrain from caching the balance for the recipient address (`to`) if `from` and `to` are the same.

Patch

Fixed in 7ab57c7.

Faulty Vote History Retrieval MEDIUM OS-SGR-ADV-01

Description

`checkpoint` inherently assumes that the `vote_ledgers` vector is sorted, as it relies on this to efficiently locate relevant checkpoints utilizing binary search. However, `add_vote_ledger` appends new entries to the `vote_ledgers` vector via `push_back`, which does not guarantee that the vector remains sorted. This will result in issues if a new vote ledger sequence is lower than the previous one, as such an entry will invalidate the binary search method. Binary search will potentially find incorrect positions for checkpoints, resulting in inconsistencies due to inaccurate voting history retrieval.

votes/src/storage.rs rust

```rust
pub fn add_vote_ledger(e: &Env, sequence: u32) {
    [...]
    if let Some(last) = vote_ledgers.last() {
        if last == sequence {
            return;
        }
    }
    vote_ledgers.push_back(sequence);
    storage::set_vote_ledgers(&e, &vote_ledgers);
}
```

Remediation

Modify `add_vote_ledger` to sort the `vote_ledgers` vector after appending the new sequence. This ensures the vector remains sorted for efficient binary search operations.

Patch

Fixed in 0a1be86.

Insufficient TTL for Checkpoints MEDIUM OS-SGR-ADV-02

Description

The vulnerability lies in the way `storage::set_total_supply_checkpoints` sets the Time To Live (TTL) for the total supply checkpoints. The function stores an array of total supply values (`balance`) at different points in time. These checkpoints are utilized to calculate voting power during governance proposals. The TTL is set to `MAX_CHECKPOINT_AGE_LEDGERS` (which is equal to eight) to ensure checkpoints are available for the entire voting period (a maximum of seven days) and a little extra time for potential ledger inconsistencies.

>_ *votes/src/storage.rs* rust

```rust
pub fn set_total_supply_checkpoints(e: &Env, balance: &Vec<u128>) {
    e.storage()
        .temporary()
        .set(&TOTAL_SUPPLY_CHECK_KEY, balance);
    // Checkpoints only need to exist for at least 7 days to ensure that correct
    // vote periods can be tracked for the entire max voting period of 7 days.
    // TTL is 8 days of ledgers, providing some wiggle room for fast ledgers.
    e.storage().temporary().extend_ttl(
        &TOTAL_SUPPLY_CHECK_KEY,
        MAX_CHECKPOINT_AGE_LEDGERS,
        MAX_CHECKPOINT_AGE_LEDGERS,
    );
}
```

However, the total lifetime a checkpoint needs to be available may exceed `MAX_CHECKPOINT_AGE_LEDGERS` , when `vote_period + grace_period` surpasses eight days. Considering a scenario with a maximum vote period of seven days and a three-day grace period for proposal execution, a proposal will close after ten days. If a proposal closes after ten days, the relevant checkpoint for calculating voting power may have expired from temporary storage, or `add_checkpoint` may have pruned the checkpoint during its execution. This results in `contract::close` either aborting or utilizing an incorrect total supply checkpoint.

Remediation

Increase `MAX_CHECKPOINT_AGE_LEDGERS` to ensure that it is long enough to accommodate the maximum voting period plus the desired grace period. Consider a buffer for ledger inconsistencies as well.

Patch

Fixed in 0a1be86.

Governor Proposal Stalling LOW OS-SGR-ADV-03

Description

The settings parameter validation in `settings::require_valid_settings` is lax. The current validation in `require_valid_settings` allows settings with `vote_threshold == 0` and `grace_period == 0`. A value of zero for `vote_threshold` means no minimum number of votes is required to pass a proposal, rendering the governance process meaningless. When `grace_period` is zero, it removes the buffer period after voting ends, resulting in a situation where successfully closing the proposal becomes impossible. This occurs because `contracts::close` relies on the `grace_period` to determine if a proposal has taken too long to close as explained below:

>_ *governor/src/settings.rs* rust

```rust
pub fn require_valid_settings(e: &Env, settings: &GovernorSettings) {
    if settings.vote_period > MAX_VOTE_PERIOD {
        panic_with_error!(&e, GovernorError::InvalidSettingsError)
    }
    if settings.vote_delay + settings.vote_period + settings.timelock + settings.grace_period *
        ↪ 2
        > MAX_PROPOSAL_LIFETIME
    {
        panic_with_error!(&e, GovernorError::InvalidSettingsError)
    }
    if settings.proposal_threshold < 1
        || settings.counting_type > 0b111
        || settings.quorum > 9999
        || settings.vote_threshold > 9999
    {
        panic_with_error!(&e, GovernorError::InvalidSettingsError)
    }
}
```

If the current ledger sequence is less than or equal to the proposal's vote end time (`proposal_data.vote_end`), it means voting is still ongoing, and the function panics with a `GovernorError::VotePeriodNotFinishedError`. If the current ledger sequence is greater than the vote end time plus the `grace_period`, it signifies the proposal took too long to close and is marked as expired (`ProposalStatus::Expired`). The problem arises when `grace_period` is zero.

>_ *governor/src/contract.rs* rust

```rust
fn close(e: Env, proposal_id: u32) {
    [...]
    if e.ledger().sequence() <= proposal_data.vote_end {
        panic_with_error!(&e, GovernorError::VotePeriodNotFinishedError)
    }

    let settings = storage::get_settings(&e);
    let vote_count = storage::get_proposal_vote_count(&e, proposal_id).unwrap_optimized();

    if e.ledger().sequence() > proposal_data.vote_end + settings.grace_period {
        // proposal took too long to be closed. Mark expired and close.
        proposal_data.status = ProposalStatus::Expired;
    }
    [...]
}
```

After the voting period ends, the condition `e.ledger().sequence() > proposal_data.vote_end + settings.grace_period` will always be true because `grace_period` is zero. This will always trigger the "*proposal took too long*" logic, marking the proposal as expired even if it may have passed the voting requirements. Consequently, the proposal will never be closed as `successful` since either the voting period check or the expired check will always be true, creating a situation where the protocol becomes bricked as finalizing proposals with a successful outcome becomes impossible.

Remediation

Enforce stricter validation on the setting parameters within `require_valid_settings`.

Patch

Fixed in 1a6f8e1.

Expired Allowance Miscalculation LOW OS-SGR-ADV-04

Description

The SEP-41 standard allows the optional specification of an expiration time for allowances. This means an account (`from`) may grant spending permission to another account (`spender`) for a limited time. The current implementation of allowance simply retrieves the stored allowance data utilizing `storage::get_allowance(&e, &from, &spender)` . It then returns only the amount field from the retrieved data. This approach ignores the expiration time associated with the allowance.

>_ *votes/src/contract.rs* rust

```rust
fn allowance(e: Env, from: Address, spender: Address) -> i128 {
    let result = storage::get_allowance(&e, &from, &spender);
    result.amount
}
```

If an allowance has expired (the expiration time has passed), the retrieved data might still contain a non-ero amount value. The allowance function, by simply returning this amount, would incorrectly indicate that the spender can still spend that amount. This may mislead users about the true spending power of authorized spenders.

Remediation

`allowance` should consider the expiration time when retrieving and returning allowance information.

Patch

Fixed in 7ab57c7.

Centralization Risk LOW

OS-SGR-ADV-05

Description

Within `contract::cancel`, the current implementation allows the council to cancel any proposal, including those proposing changes to the governor's settings. This grants the council significant power and introduces a centralization risk.

>_ *governor/src/contract.rs* rust

```rust
fn cancel(e: Env, from: Address, proposal_id: u32) {
    [...]
    // require from to be the creator or the council
    if from != proposal_data.creator {
        let settings = storage::get_settings(&e);
        if from != settings.council {
            panic_with_error!(&e, GovernorError::UnauthorizedError);
        }
    }
    [...]
}
```

Remediation

Prevent the council from canceling proposals that modify the governor's settings. This ensures voters retain control over the governor's settings even if the council attempts to block changes. Voters may still remove the council from power (if the council's address is stored in a setting) by proposing and passing a settings change.

Patch

Fixed in b616d59.

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 represent anti-patterns and may result in security issues in the future.

ID	Description
OS-SGR-SUG-00	`contract::close` fails to emit an event if the proposal is expired.
OS-SGR-SUG-01	The division in `is_over_quorum` and `is_over_threshold` may result in inaccurate results due to integer truncation, incorrectly assessing the proposals.

Failure To Emit Event

OS-SGR-SUG-00

Description

`contract::close` checks if the current ledger sequence is beyond the grace period after the voting ends. If so, it marks the proposal as expired by setting `proposal_data.status = ProposalStatus::Expired`. However, it does not call `events::proposal_expired` to emit an event for this specific expiration scenario.

>_ *governor/src/contract.rs* rust

```rust
fn close(e: Env, proposal_id: u32) {
    [...]
    let settings = storage::get_settings(&e);
    let vote_count = storage::get_proposal_vote_count(&e, proposal_id).unwrap_optimized();
    if e.ledger().sequence() > proposal_data.vote_end + settings.grace_period {
        // proposal took too long to be closed. Mark expired and close.
        proposal_data.status = ProposalStatus::Expired;
    }
    [...]
}
```

External observers or other parts of the application relying on governance events might not be notified about proposals expiring within the close function. This may result in incomplete information about the lifecycle of proposals and a lack of transparency in the governance process.

>_ *governor/src/events.rs* rust

```rust
pub fn proposal_expired(e: &Env, proposal_id: u32) {
    let topics = (Symbol::new(&e, "proposal_expired"), proposal_id);
    e.events().publish(topics, ());
}
```

Remediation

Ensure to emit an event when the proposal expires within `close`.

Patch

Fixed in ccfc2d3.

Precision Loss In Vote Calculation OS-SGR-SUG-01

Description

The current implementation of `is_over_quorum` and `is_over_threshold` within `VoteCount` is susceptible to precision loss due to integer division. The calculation of `quorum_requirement_floor` and `for_votes` in `is_over_quorum` and `is_over_threshold` respectively involves dividing the product of two integers by a third integer.

>_ *governor/src/vote_count.rs* rust

```rust
pub fn is_over_quorum(&self, quorum: u32, counting_type: u32, total_votes: i128) -> bool {
    let quorum_votes = self.count_quorum(counting_type);
    let quorum_requirement_floor = (total_votes * quorum as i128) / BPS_SCALAR;
    quorum_votes > quorum_requirement_floor
}

pub fn is_over_threshold(&self, vote_threshold: u32) -> bool {
    let against_and_for_votes = self.against + self._for;
    if against_and_for_votes == 0 {
        return false;
    }
    let for_votes = (self._for * BPS_SCALAR) / against_and_for_votes;
    for_votes > vote_threshold as i128
}
```

Integer division truncates any fractional remainder, potentially leading to inaccurate results, especially when dealing with percentages (represented as basis points here).

Remediation

Rewrite the calculations utilizing multiplication to avoid precision loss.

Patch

Fixed in 2e4a9c6.

A — Vulnerability Rating Scale

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

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a 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 that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

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

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

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 on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, 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 comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have 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.