Zellic

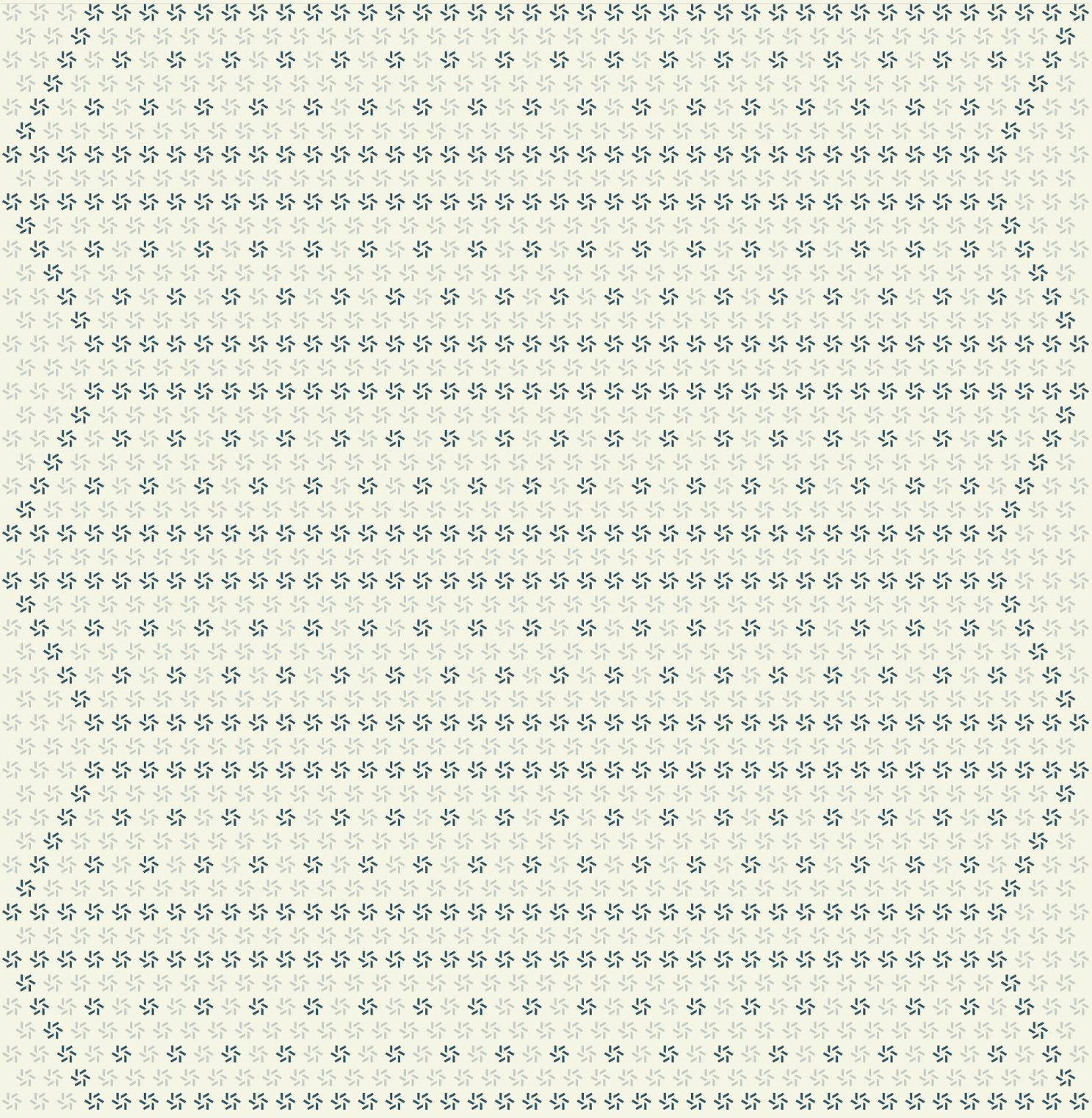
Prepared for
Nikhil Suri
Wormhole Foundation

Prepared by
Jasraj Bedi
Dimitri Kamenski
Zellic

February 4, 2025

MultiGov

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Wormhole Foundation from January 14th to January 31st, 2025. During this engagement, Zellic reviewed MultiGov's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could users interact with other staked accounts in inappropriate manners?
- Could an attacker modify configuration settings designated solely for the admin?
- Are there any denial-of-service (DOS) conditions that could be exploited?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

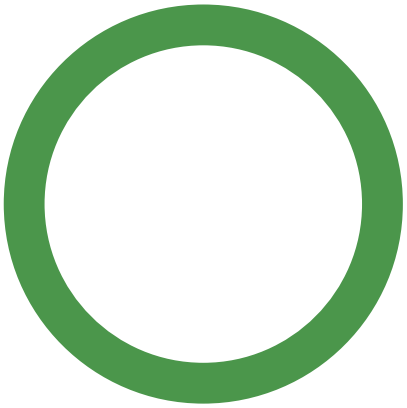
1.4. Results

During our assessment on the scoped MultiGov programs, we discovered two findings, both of which were low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Wormhole Foundation in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	2
Informational	0



2. Introduction

2.1. About MultiGov

Wormhole Foundation contributed the following description of MultiGov:

> MultiGov is a cross-chain governance system that extends traditional DAO governance across multiple blockchain networks. By leveraging Wormhole's interoperability infrastructure, MultiGov enables seamless voting and proposal mechanisms across various chains.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the programs.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped programs itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

MultiGov Programs

Type	Rust
Platform	Solana
Target	MultiGov
Repository	https://github.com/wormhole-foundation/multigov ↗
Version	`d899f66a7d2c11c2c077ff3fdcde130d46e2dfd7`
Programs	`solana/programs/staking/**`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-weeks. The assessment was conducted by two consultants over the course of 1.5 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi
Co-founder
jazzy@zellic.io

Dimitri Kamenski
Engineer
dimitri@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 14, 2025	Kick-off call
January 14, 2025	Start of primary review period
January 31, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect account bindings allows admin to deduct from vester

Target	lib.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `cancel_vesting()` instruction allows an admin signer to cancel the vesting on a `vester_ta.owner`. Since the `vest` account is not properly bound to the `config` and `vester_ta`, it is possible for an admin to cancel the `Vesting` acount for an unrelated vester, deducting the incorrect amount from that vester's total.

The `cancel_vesting()` instruction subtracts the vested amount from config as follows:

```
pub fn cancel_vesting(&mut self) -> Result<()> {
        self.config.vested = self
            .config
            .vested
            .checked_sub(self.vest.amount)
            .ok_or(VestingError::Underflow)?;

        self.vesting_balance.total_vesting_balance = self
            .vesting_balance
            .total_vesting_balance
            .checked_sub(self.vest.amount)
            .ok_or(VestingError::Underflow)?;

        Ok(())
    }
```

However, the `self.vest.amount` can be used from an unrelated vester since the `vest` can refer to itself for the `vester_ta` used in the account seed.

```
#[account(
        mut,
        close = admin,
        has_one = config, // This check is arbitrary, as ATA is baked into the
    PDA
        seeds = [VEST_SEED.as_bytes(), config.key().as_ref(),
    vest.vester_ta.key().as_ref(), vest.maturation.to_le_bytes().as_ref()],
```

```
        bump = vest.bump
    )]
    vest: Account<'info, Vesting>,

pub struct Vesting {
    pub vester_ta: Pubkey,
    pub config: Pubkey,
    pub amount: u64,
    pub maturation: i64,
    pub bump: u8,
}
```

If an admin uses a `vest` account with the wrong `vester_ta`, the incorrect amount would be subtracted from the vesting balance.

Impact

The issue requires administrative access, lowering its likelihood. The impact is High since balances can be incorrectly deducted.

Recommendations

Add a constraint in the account macro on the `vest` to ensure `has_one = vester_ta`.

Remediation

This issue has been acknowledged by Wormhole Foundation, and a fix was implemented in commit `2e111968` ↗.

3.2. Broken uniqueness invariant in binary search

Target	lib.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `update_vote_weight_window_length` is responsible for passing in what timestamp ranges the voting checkpoints should use during `cast_vote()`. The instruction for reading window length uses a binary search algorithm similar to the logic used in checkpoint searches. However, unlike the checkpoint searches, it pushes a new window length the program does not validate for uniqueness in the search criterion. This is a critical invariant required by binary search to ensure determinism (i.e., the same search criteria returns the same value result every time).

If two window lengths can exist for the same timestamp, when reading from the set of (window length, timestamp) tuples, we will receive different values for the same fixed timestamp query depending on the length of the data set. This is strictly because the midpoint may fall on one of those duplicate values; upon adding values, the midpoint shifts to the duplicate, changing the outcome of the returned value.

Consider the following example:

```
(value, timestamp): [(1,0), (50,1), (100,2), (150,2), (200, 3), (250, 4)]
Search criteria: Less than or equal to timestamp of 2
Return value: 100
```

If in the next iteration the set increases by a size of one more value, the original midpoint changes, and we will have the following search:

```
(value, timestamp): [(1,0), (50,1), (100,2), (150,2), (200, 3), (250, 4),
    (300, 5)]
Search criteria: Less than or equal to timestamp of 2
Return value: 150
```

Impact

This behavior could result in nondeterministic results for searches against the same timestamp.

Recommendations

Ensure a duplicate timestamp overwrites the existing value or reverts.

Remediation

This issue has been acknowledged by Wormhole Foundation, and a fix was implemented in commit 508585ea ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Precarious associated-token-account constraints

It was noted during the audit that a few of the instruction contexts had used constraint practices that were considered precarious. Take the `TransferVesting` instruction context, which involves a `VestingBalance` account, a current and new `Vesting` account, and a current and new vester associated token account, as well as a signature of the vester. It is imperative that constraints are present that enforce that the signer owns the vesting `TokenAccount`, the associated `Vesting` account, and the `VestingBalance` account. If any of these constraints are not held, then an attacker might be able to pass a victim's `VestingBalance`, rather than their own.

Part of our conclusion that the design of this instruction context was precarious was the pattern repeated throughout the codebase where associated token accounts (ATAs) used their own fields as constraints. This then requires *other* accounts to step in and provide constraints to tie permissions together.

Here is an example:

```
#[account(
    mut,
    associated_token::mint = mint,
    associated_token::authority = vester_ta.owner,
    associated_token::token_program = token_program
)]
vester_ta: Box<InterfaceAccount<'info, TokenAccount>>,
```

It is recommended that if an appropriate ATA authority is available it be used as a constraint instead of the ATA itself.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the programs and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Crate: wormhole-staking-program

The **wormhole-staking-program** program is a Solana program built using the Anchor framework. It handles multiple aspects of protocol state including governance configuration, stake account creation and management, delegation and vote-casting, vesting operations, and cross-chain message execution (via Wormhole). The contract maintains various on-chain states:

Global Configuration: Stored in a configuration account, this includes:

- `bump`: Seed bump for proper PDA derivation.
- `governance_authority`: The address authorized to make governance-level changes.
- `voting_token_mint`: The mint address for the token used in voting.
- `vesting_admin`: The administrator permitted to perform vesting operations.
- `max_checkpoints_account_limit`: A bound on checkpoint entries to mitigate potential account bloat/DoS.

Stake Account State: For each user, stake account metadata and checkpoint data track:

- Ownership, delegated addresses, and recorded balances.
- Historical checkpoints for stake and delegation amounts, which are critical for calculating voting power over time.

Vesting State: Multiple vesting-related accounts and configuration parameters are maintained to control token vesting, claims, transfers, and cancellations.

Spoke Message Execution State: This includes executor, airlock, and metadata collector accounts used to securely process cross-chain messages.

Below is a function-by-function walkthrough with key safety and threat-mitigation considerations.

Function: `init_config`

Description:

- Initializes the global configuration.
- Stores the governance authority, voting token mint, vesting admin, and a checkpoint limit.

- Enforces that the `max_checkpoints_account_limit` does not exceed 655,000 (to prevent account bloat and DoS).

Safety Considerations:

- **DoS Mitigation:** Limiting checkpoint size prevents an attacker from filling the account.

Function: `update_governance_authority`

Description:

- Updates the governance authority stored in the configuration.
- Critical for transferring administrative control.

Safety Considerations:

- **Authorization:** Must be callable only by the current governance authority.

Function: `update_vesting_admin`

Description:

- Updates the vesting administrator.
- Allows transfer of vesting operational control.

Safety Considerations:

- **Authorization:** Access is restricted to authorized parties.

Function: `create_stake_account`

Description:

- Creates a new stake account for a user.
- Initializes stake account metadata and checkpoint data.

Safety Considerations:

- **State Initialization:** Correctly sets up user stake and checkpoint state for later operations.

Function: `create_checkpoints`

Description:

- Creates and appends a checkpoint entry reflecting the current stake balance.
- Loads the existing checkpoint data and pushes a new checkpoint.

Safety Considerations:

- **Checkpoint Limits:** Verifies that the number of checkpoints does not exceed the configured limit.
- **Account Safety:** Uses safe account initialization via anchor's init annotation to avoid state corruption.

Function: `delegate`

Description:

- Updates a staker's delegation to a delegatee.
- Adjusts checkpoint data for both the previous delegate and the new delegatee.
- Emits events (`DelegateChanged`) to signal delegation updates.

Safety Considerations:

- **Checkpoint Boundaries:** Enforces limits to prevent account bloat.
- **Balance Validation:** Updates both stake and recorded vesting balances carefully to avoid miscounting.

Function: `withdraw_tokens`

Description:

- Allows users to withdraw tokens from their stake custody.
- Updates checkpoint data to reflect a changed balance.
- Verifies that the withdrawal amount is non-zero and that the destination is controlled by the signer.

Safety Considerations:

- **SPL Token Transfer:** Uses the SPL token program with `invoke_signed` and proper signer seeds.
- **Account Ownership:** Checks that the destination account is owned by the signer to prevent unauthorized transfers.

Function: `cast_vote`

Description:

- Enables users to cast votes on proposals.
- Determines voting weight by traversing historical checkpoints within a defined time window.
- Validates that the vote does not exceed the available weight and updates proposal totals.

Safety Considerations:

- **Historical Integrity:** Uses checkpoint history to secure against vote manipulation.
- **Bounds Checking:** Thorough arithmetic and bounds checks prevent overflows and replay of stale checkpoints.
- **Data Consistency:** Handles transitions between checkpoint accounts safely.

Function: `initialize_vesting_config`

Description:

- Initializes vesting configuration (mint, vault, admin, etc.).

Safety Considerations:

- **Access Control:** Only authorized (vesting-admin) accounts should perform this operation.

Function: `create_vesting_balance`

Description:

- Creates a vesting balance account for a user.
- Intended to be used when the vesting configuration is not yet finalized.

Safety Considerations:

- **Account Initialization:** Ensures that vesting balance accounts are correctly associated and initialized.

Function: `finalize_vesting_config`

Description:

- Finalizes the vesting configuration, preventing further vesting account creation or cancellation.
- Restricted to the vesting admin.

Safety Considerations:

- **Locking Critical State:** Once finalized, no further vesting changes are allowed. This is strictly enforced.
- **Privilege Control:** Ensures only the vesting admin can finalize to prevent accidental or malicious configuration changes.

Function: `create_vesting`

Description:

- Opens a new vesting account and deposits tokens into the vesting vault.
- Accepts parameters for maturation time and vesting amount.

Safety Considerations:

- **Parameter Validation:** Ensures maturation and amount parameters are valid and won't cause overflow.
- **State Consistency:** Must integrate correctly with vesting configuration state (especially in non-finalized mode).

Function: `claim_vesting`

Description:

- Allows a user to claim tokens from a vesting account and close it.
- Emits events to signal vesting balance and delegated vote adjustments.

Safety Considerations:

- **Atomic Claims:** Updates vesting state and transfers tokens atomically to prevent double-claims.
- **State Cleanup:** Correctly removes vesting accounts to free storage and avoid dangling state.

Function: `transfer_vesting`

Description:

- Transfers vesting tokens from one account to another.
- Updates the stake account metadata and emits events for vesting balance changes.

Safety Considerations:

- **State Synchronization:** Both source and destination vesting accounts are updated to avoid fund loss.
- **Arithmetic Safety:** Uses safe math and validated state updates.

Function: `cancel_vesting`

Description:

- Cancels and closes a vesting account for configurations that are not finalized.
- Restricted to the vesting admin.

Safety Considerations:

- **Conditional Access:** Ensures vesting cancellation is only permitted when configuration is not finalized.
- **Fund Recovery:** Must correctly refund any remaining tokens and update state.

Function: `withdraw_surplus`

Description:

- Enables the vesting admin to withdraw surplus tokens beyond the total vested amount.
- Intended to recover tokens that are not allocated to vesting obligations.

Safety Considerations:

- **Privilege Restriction:** Only callable by the vesting admin.
- **Amount Validation:** Must ensure that withdrawal amounts are correctly computed to avoid draining user funds.

Function: `initialize_spoke_message_executor`

Description:

- Sets up the executor for processing cross-chain messages (using Wormhole).
- Initializes key state variables such as the hub chain ID, spoke chain ID, and Wormhole core program ID.

Safety Considerations:

- **Proper Initialization:** Uses bump seeds and strict account derivation to prevent misrouting of messages.
- **Access Control:** Only authorized accounts should set up the executor.

Function: `receive_message`

Description:

- Processes an incoming Wormhole message.
- Validates that the message comes from the expected Wormhole (spoke) chain.
- Iterates over embedded instructions and calls them via `invoke_signed` with the correct signer seeds.
- Ensures that lamport usage remains within the allowed maximum and that the payer's ownership is unchanged after execution.

Safety Considerations:

- **Cost Limitation:** Verifies lamport differences to prevent draining the payer's account.
- **Payload Validation:** Checks that the Wormhole chain ID and message payload are as expected to block spoofing.

Function: `initialize_spoke_airlock`

Description:

- Initializes the airlock and its self-call counterpart used in cross-chain message execution.

Function: `initialize_spoke_metadata_collector`

Description:

- Sets up a metadata collector to receive proposal metadata from the hub chain.
- Stores details such as the hub chain ID, hub proposal metadata, and Wormhole core program ID.

Safety Considerations:

- **Data Authenticity:** Ensures that only messages from legitimate hub sources are recorded.

Function: `update_hub_proposal_metadata`

Description:

- Updates the hub proposal metadata stored in the metadata collector.
- Enforces that either governance or an authorized airlock self-call account may perform the update, based on a configuration flag.

Safety Considerations:

- **Access Verification:** Validates the caller's authority (either governance or via a valid self-call).
- **Data Integrity:** Prevents unauthorized modifications that could misdirect proposal data.

Function: `relinquish_admin_control_over_hub_proposal_metadata`

Description:

- Permanently disables governance control over updating hub proposal metadata.
- Shifts control solely to the airlock self-call mechanism.

Function: `initialize_vote_weight_window_lengths`

Description:

- Initializes the vote weight window lengths, which are used to determine historical vote weight based on checkpoint data.
- Sets an initial window length in a dedicated account.

Safety Considerations:

- **State Integrity:** Correctly initializes historical data needed for secure vote weight calculations.
- **Boundary Enforcement:** Uses safe initialization methods to avoid overflow or misconfiguration.

Function: `update_vote_weight_window_lengths`

Description:

- Updates (by appending) the window lengths used in vote weight calculations.
- Records the new window length along with a timestamp.

Safety Considerations:

- **Bounds Checking:** Ensures that new window lengths do not cause account bloat or exceed limits.

Function: `post_signatures`

Description:

- Appends guardian signatures to a `GuardianSignatures` account.
- Supports multiple invocations if the number of required signatures exceeds what can fit in a single transaction.

Safety Considerations:

- **Authorization:** Verifies that the caller (the refund recipient) is authorized to append signatures.

Function: `close_signatures`

Description:

- Allows the initial payer to close the `GuardianSignatures` account if the associated query was invalid.
- Returns the funds by refunding the payer.

Safety Considerations:

- **Access Restriction:** Only callable by the initial payer.

Function: `add_proposal`

Description:

- Adds a new proposal after verifying external proposal metadata obtained via a Wormhole query response.
- Parses the query response, validates the Ethereum call data (including the expected function signature and proposal ID), and updates the proposal state.
- Emits events (`ProposalCreated`) on successful addition.

Safety Considerations:

- **Strict Validation:** Checks chain IDs, function signatures, and proposal IDs to prevent spoofing.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Solana Mainnet.

During our assessment on the scoped MultiGov programs, we discovered two findings, both of which were low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.