Zellic

Prepared for
Arun Kirubarajan
Magna

Prepared by
Juchang Lee
Varun Verma
Zellic

November 12, 2025

Airlock

Sui Move Application 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 Magna from October 30th to October 31th, 2025. During this engagement, Zellic reviewed Airlock'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:

- Are calculations involving token transfers predictable and accurate?
- Could tokens be permanently frozen or drained under any circumstances?
- Is access control implemented correctly and effectively?
- Is the contract upgrade mechanism implemented securely and correctly?

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.

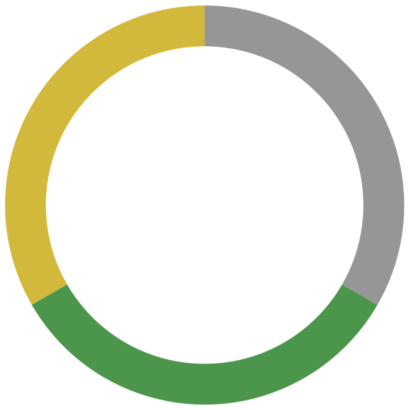
1.4. Results

During our assessment on the scoped Airlock modules, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	1
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About Airlock

Magna contributed the following description of Airlock:

> Airlock is Magna's protocol for managing customizable token unlocks at scale.

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 modules.

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 modules 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:

Airlock Modules

Type	Move
Platform	Sui
Target	protocol-vesting-sui
Repository	https://github.com/magna-eng/protocol-vesting-sui ↗
Version	`60dc5b29b83a052b4f026e64a5e6a0f44664701c`
Programs	`vesting_merkle.move` `utils/merkle_proof.move`

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Juchang Lee
Engineer
lee@zellic.io ↗

Varun Verma
Engineer
varun@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 30, 2025	Start of primary review period
October 31, 2025	End of primary review period

3. Detailed Findings

3.1. Allocation ID Collision Enables Beneficiary Hijacking

Target	vesting_merkle.move		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

`DistributionState` is keyed only by `allocation_id`, not by the unique leaf content. If two merkle leaves share the same `allocation_id`, they share the same beneficiary. The first withdrawal for a given `allocation_id` permanently locks the beneficiary address, and subsequent withdrawals with different leaves but the same ID will use that locked beneficiary.

The issue:

```
ofield::add(
    vesting_uid,
    allocation.allocation_id,  // ← Only keyed by allocation_id
    DistributionState {
        beneficiary: allocation.original_beneficiary,  // ← First caller sets
    this
        withdrawn: 0,
        terminated_timestamp: 0,
    },
);

// Later access retrieves shared state:
let distribution_state = ofield::borrow_mut<u256, DistributionState>(
    &mut vesting.id,
    allocation.allocation_id,  // ← Same ID = same state, regardless of leaf
);
```

If Alice and Bob have different leaves but both have `allocation_id = 999`, whoever withdraws first becomes the beneficiary for both allocations.

Impact

Without uniqueness validation, an operational error can create a race condition where the first withdrawal locks the beneficiary and the second person loses their entire allocation. Since merkle roots cannot be removed, there's no way to fix it once discovered.

Recommendations

Bind state to the unique leaf content instead of just `allocation_id`:

Primary fix: Key state by leaf hash

```
let leaf_hash = keccak256(&serialized_allocation);
ofield::add(vesting_uid, leaf_hash, DistributionState { ... })
```

This makes collisions impossible - each unique leaf gets its own state by design.

Additional safeguards:

- Build off-chain validation to check for duplicate `allocation_ids` before generating merkle roots
- Document that `allocation_id` must be globally unique across all leaves
- Add a view function to validate roots for duplicates before calling `add_merkle_root()`

Remediation

Magna provided the following response to this finding:

> We have strong backend guarantees (Unique key DB constraints) that prevents any duplicate allocation-ids (across all projects and all trees)

3.2. Revoke Function Forfeits Already-Vested Tokens Contrary to Documentation

Target	vesting_merkle.move		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `revoke()` function's documentation states it "forfeits all unvested tokens," implying that already-vested tokens remain claimable. However, the implementation sets `terminated_timestamp = 1` (a sentinel value), which causes all subsequent `withdraw()` calls to abort with `ERR_ALREADY_TERMINATED`, preventing withdrawal of any tokens including those already vested.

Code evidence:

In `terminate_allocation()`:

```
distribution_state.terminated_timestamp = if (is_being_revoked)
    1 else (clock.timestamp_ms() / 1000) as u32;
```

In `withdraw()`:

```
assert!(distribution_state.terminated_timestamp != 1, ERR_ALREADY_TERMINATED);
```

This differs from `cancel()`, which correctly sets `terminated_timestamp` to the current time, allowing beneficiaries to claim tokens vested up to the cancellation point.

Impact

Beneficiaries lose access to tokens they have already earned through time passage. For example:

- A 4-year vesting schedule is 90% complete (3.6 years elapsed)
- Beneficiary has not yet withdrawn
- Benefactor calls `revoke()`
- Beneficiary permanently loses 90% of their allocation despite those tokens being fully vested

This violates standard vesting semantics where "revoke" typically means "stop future vesting but preserve earned amounts".

Recommendations

Option 1: Match documentation behavior

1. Allow withdrawal of tokens vested up to the revocation timestamp:

Option 2: Update documentation to match implementation

If the current behavior is intentional, update the comment to:

```
/// @notice Revokes a vesting allocation (forfeits ALL tokens, including
    already vested)
```

And clearly communicate this behavior to beneficiaries.

Remediation

This issue has been acknowledged by Magna, and a fix was implemented in commit `e95f43bf` ↗.

Magna confirmed the function documentation was inaccurate and that the behavior is intentional.

3.3. Missing Validation Checks in Allocation Deserialization

Target	vesting_merkle.move		
Category	Code Maturity	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `deserialize_allocation()` function performs BCS deserialization of allocation data without validating the integrity of the deserialized values. This creates multiple denial-of-service vectors that will manifest at withdrawal time rather than at merkle root creation time.

Missing validations:

1. **Calendar schedule:** No check that `unlock_timestamps` and `unlock_amounts` arrays have equal length
2. **Interval schedule:** No check that `period_length > 0` or `number_of_periods > 0`
3. **Amount consistency:** No check that `allocation.amount` matches sum of unlock amounts or piece amounts
4. **Array bounds:** No maximum size limits on timestamps/amounts/pieces vectors

#1 - Array length mismatch:

In `deserialize_allocation()`:

```
Schedule::Calendar {
    unlock_timestamps: stream.peel_vec!(|stream| stream.peel_u32()),
    unlock_amounts: stream.peel_vec!(|stream| stream.peel_u64()),
}
// No validation that lengths match
```

Later in `calc_vested_amount_calendar()`:

```
let timestamps_length = unlock_timestamps.length();
while (i < timestamps_length) {
    if (unlock_timestamps[i] <= final_timestamp) {
        amount = amount + unlock_amounts[i]; // ← Runtime panic if
    unlock_amounts.length() < timestamps_length
    };
    i = i + 1;
};
```

#2 Division by zero:

In `deserialize_allocation()`:

```
Piece {
    start_time: stream.peel_u32(),
    period_length: stream.peel_u32(),  // ← No check that this > 0
    number_of_periods: stream.peel_u32(),  // ← No check that this > 0
    amount: stream.peel_u64(),
}
```

Later in `calc_vested_piece_amount()`:

```
let mut fully_vested_periods = elapsed_time / piece.period_length;  // ←
    Division by zero panic

// ...later
piece.amount * (fully_vested_periods as u64) / (piece.number_of_periods as
    u64)  // ← Division by zero panic
```

#3 - No array size caps:

```
let pieces = stream.peel_vec!(|stream| { /* ... */ });
// No check on pieces.length() - could be 10,000+ entries
```

Used in gas-intensive loops:

```
fun calc_vested_amount_interval(final_timestamp: u32, pieces: &vector<Piece>):
    u64 {
    let pieces_length = pieces.length();  // Could be unbounded
    let mut amount = 0;
    let mut i = 0;
    while (i < pieces_length) {  // ← Gas exhaustion with large arrays
        amount = amount + calc_vested_piece_amount(&pieces[i],
    final_timestamp);
        i = i + 1;
    };
    amount
}
```

Impact

Permanent withdrawal DoS:

- Mismatched array lengths cause runtime panics in `calc_vested_amount_calendar()` during withdrawal
- Zero values in `period_length` or `number_of_periods` cause division-by-zero panics during withdrawal
- Once a merkle root containing malformed data is added, affected allocations become permanently unclaimable

Fund locking:

- If `sum(unlock_amounts) < allocation.amount`, excess funds remain permanently locked in the contract
- If `sum(piece.amount) < allocation.amount`, beneficiaries cannot claim their full allocation

Gas exhaustion:

- Unbounded array sizes (e.g., 10,000 unlock timestamps) can make withdrawal transactions consistently fail due to gas limits, effectively locking funds

Operational risk:

- These issues stem from off-chain tooling errors: CSV export bugs, spreadsheet formula errors, script typos, or copy-paste mistakes
- Without validation, innocent operational mistakes permanently brick user allocations
- Discovery happens at withdrawal time (potentially months/years after merkle root creation), making remediation impossible since merkle roots cannot be removed

Recommendations

Add comprehensive validation checks in `deserialize_allocation()`:

1. **Array length validation:** Assert that calendar `unlock_timestamps` and `unlock_amounts` have equal length

2. **Zero-value guards:** Assert that `period_length > 0` and `number_of_periods > 0` for all interval pieces

3. **Array size caps:** Limit maximum entries to reasonable bounds to prevent gas exhaustion

4. **Non-empty validation:** Assert that calendar schedules have at least one unlock and interval schedules have at least one piece

5. **Amount validation:** Optionally verify that sum of unlock_amounts or piece amounts equals the total allocation amount

These validations should occur during deserialization to fail-fast at withdrawal time, or ideally add a view function to validate allocations before calling `add_merkle_root()`.

Remediation

Magna provided the following response to this finding:

> This is a conscious design decision, we wouldn’t like to waste gas by doing those checks on-chain, those checks are done offchain when building the tree. All checks are implicitly ‘done’ on-chain by verifying the merkle proof against the root. The root is assumed to be correct.

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. Centralization Risk: Benefactor Privileges

The protocol grants significant control to benefactors with no mechanism for removal or governance oversight. Benefactors can add merkle roots (enabling potential allocation_id collisions), fund or defund the contract at will, and terminate allocations through cancel or revoke operations. While this centralized trust model is common for vesting protocols where benefactors represent the token-issuing organization, it concentrates substantial power without checks or time delays. Users must trust that benefactors will act honestly and maintain secure operational practices, as compromised benefactor credentials enable direct fund manipulation through defunding or malicious merkle root injection.

5. Assessment Results

During our assessment on the scoped Airlock modules, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

5.1. 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.