Zellic

Prepared for
0xRipleys
Suilend

Prepared by
Sunwoo Hwang
Varun Verma
Zellic

November 19, 2024

Sui Token & Ticket Contracts

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 Suilend from November 15th to November 18th. During this engagement, Zellic reviewed Sui Token & Ticket contracts' code for security vulnerabilities, design issues, and general weaknesses in security posture. On December 4th scope was updated from c7c738b5 ↗ to 0cbb8ca3 ↗ and from fa95d498 ↗ to cfce38e0 ↗

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:

- Is the penalty calculation implemented correctly, ensuring no inaccuracies during linear interpolation of penalties over time?
- Can unauthorized users mint, redeem, or collect penalties? Are `AdminCap` and related access-control mechanisms robust enough to prevent unauthorized access or misuse?
- Could any malicious or faulty operations lead to incorrect or inconsistent states in `VestingManager` or `CapsuleManager` structures, such as unfunded managers?
- Are there potential economic vulnerabilities, such as rounding errors in penalty or reward distributions or improper decimal scaling between different coins?

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. During this assessment, time constraints prevented us from ensuring the complete accuracy of the decimal type used in the system.

1.4. Results

During our assessment on the scoped Sui Token & Ticket contracts, we discovered one finding, which was of low impact.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	0

2. Introduction

2.1. About Sui Token & Ticket Contracts

Suilend contributed the following description of Sui Token & Ticket contracts:

> MToken, standing for Maturing Token, is a smart contract that allows for the contingent vesting of a token for a given maturity. The receiver of an MToken can decide to unlock the underlying token by paying a penalty. The penalty decreases linearly over time.
>
> ClaimMSend is a second smart contract which allows users that own either Suilend Capsules or Suilend Points to exchange them for MSend (MToken from Suilend) for a given exchange rate.

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 contracts.

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 contracts 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:

Sui Token & Ticket Contracts

Type	Move
Platform	Sui

Target	mtoken
Repository	https://github.com/solendprotocol/mtoken ↗
Version	`c7c738b58013befa7eeb0b46f7df746cea8a58d8`
Programs	`mtoken.move`

Target	suilend_nft
Repository	https://github.com/solendprotocol/suilend_nft ↗
Version	`fa95d4984d372a39ba6628cde1bbba840195e2a6`
Programs	`points.move` `capsule.move`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 0.5 person-weeks. The assessment was conducted by two consultants over the course of two calendar days. Following the initial audit, the client has updated the code in both repositories, and the corresponding commit hashes for each repository have been provided and reviewed.

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:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Varun Verma
Engineer
varun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 15, 2024	Start of primary review period
November 18, 2024	End of primary review period
December 04, 2024	Scope updated from c7c738b5 to 0cbb8ca3
December 04, 2024	Scope updated from fa95d498 to cfce38e0

3. Detailed Findings

3.1. Sanity checks

Target	mtoken.move		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

Some sanity checks are missing in the `mint_mtokens` function in mtoken, which could lead to incorrect or unintended behavior:

1. **Penalty progression.** There is no check to ensure `start_penalty_numerator > end_penalty_numerator`. This is necessary to guarantee that penalties decrease over time as expected.
2. **Division by zero.** The `penalty_denominator` is not validated to be greater than zero. Since it is used as a divisor in the penalty calculation, a zero value would lead to a runtime error.

Impact

Without these sanity checks, the system may exhibit unintended behavior in the case of user-input error.

Recommendations

Implement the following checks in `mint_mtokens`:

1. Ensure `start_penalty_numerator > end_penalty_numerator`.
2. Ensure `penalty_denominator > 0` to avoid division by zero.

Remediation

This issue has been acknowledged by Suilend.

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. Admin caution for numerator and denominator

When redeeming mtoken, the penalty amount is calculated using the raw value of mtoken, which may have a different decimal precision than the penalty token. The admin can use the `numerator` and `denominator` parameters to scale this appropriately, but caution is required to ensure the scaling matches the decimal precision.

Users redeem mtoken in the following code:

```
 let withdraw_amount = mtoken_coin.value(); // AUDIT: Raw mtoken value
let current_time = clock::timestamp_ms(clock) / 1000;

// Ensure current time is within the valid range
assert!(current_time >= manager.start_time_s, ERedeemingBeforeStartTime);

// Interpolate penalty linearly
let end_penalty = decimal::from(manager.end_penalty_numerator)
        .mul(decimal::from(withdraw_amount)) //
        .div(decimal::from(manager.penalty_denominator));
```

This issue could lead to incorrect penalty deductions if the `numerator` and `denominator` are not scaled appropriately, resulting in potential overcharges for users redeeming their mtoken holdings. Therefore, the admin must exercise caution to ensure proper scaling.

Similarly, the `burn_points` function calculates the reward using the raw value of `points`, requiring proper configuration of the numerator and denominator ratios by the admin. Users may receive incorrect rewards if decimal differences between `points.value()` and `points_manager.balance` are not properly accounted for.

```
let amount = (points.value() * ratio.numerator) / ratio.denominator;
// Raw value used without checking decimal consistency
assert!(amount <= points_manager.balance.value(), EManagerUnfunded);
```

4.2. Zero-address burn

The current token burn uses `transfer::public_transfer(points, @0x0)`. While this removes tokens from circulation, it does not reduce the total supply of `Coin<SUILEND_POINT>`. However, since the protocol does not rely on the total supply for accounting, this method is sufficient, though it deviates slightly from standard conventions.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts 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. Module: capsule.move

Function: `new<T: drop>`

This creates a new `CapsuleManager` and its corresponding `AdminCap`. The `CapsuleManager` tracks balances and reward amounts for different capsule rarities.

Inputs

- `common_amount: u64`
    - **Validation**: None directly in this function — assumes a valid nonnegative value.
    - **Impact**: Sets the reward amount for burning a common capsule.
- `uncommon_amount: u64`
    - **Validation**: None directly in this function — assumes a valid nonnegative value.
    - **Impact**: Sets the reward amount for burning an uncommon capsule.
- `rare_amount: u64`
    - **Validation**: None directly in this function — assumes a valid nonnegative value.
    - **Impact**: Sets the reward amount for burning a rare capsule.

Branches and code coverage (including function calls)

Intended branches

- Create `CapsuleManager` with valid reward amounts, and create `AdminCap` linked to the correct `CapsuleManager`.
    - [x] Test coverage

Negative behavior

- Initializing `CapsuleManager` with edge values such as zero reward amounts for all capsule types.
    - [ ] Negative test

Function call analysis

- `new -> object::new(ctx)`

    - **External/Internal?** Internal.
    - **Argument control?** Caller provides `ctx`, which comes from `TxContext`, which is indirectly controllable.
    - **Impact**: Creates unique identifiers for `CapsuleManager` and `AdminCap`, guaranteeing their distinctness on chain.
- `new -> balance::zero()`
    - **External/Internal?** Internal.
    - **Argument control?** Not directly controlled by the caller.
    - **Impact**: Initializes the `CapsuleManager`'s balance to zero, ensuring no preexisting funds.
- `new -> AdminCap { id, manager_id }`
    - **External/Internal?** Internal.
    - **Argument control?** Caller does not control `id` or `manager_id` as they are derived internally from `object::new`.
    - **Impact**: Links the `AdminCap` to the corresponding `CapsuleManager`, establishing proper admin control.
- `new -> CapsuleManager { id, admin_id, balance, amounts }`
    - **External/Internal?** Internal.
    - **Argument control?** Caller controls `common_amount`, `uncommon_amount`, and `rare_amount` through function parameters.
    - **Impact**: Sets up the `CapsuleManager` with the correct reward structure for each capsule type, enabling reward management.

5.2. Module: mtoken.move

Function: `redeem_mtokens<MToken, Vesting, Penalty>`

This redeems `mtoken` coins from the `VestingManager`, applying a penalty based on the time elapsed since the vesting start. The redeemed amount is returned as `Vesting` coins, and the penalty is deducted from the `Penalty` coin balance.

Inputs

- `manager: &mut VestingManager<MToken, Vesting, Penalty>`
    - **Validation**: Ensures the current time is greater than or equal to `start_time_s`: `assert!(current_time >= manager.start_time_s, ERedeemingBeforeStartTime)`.
    - **Impact**: Guarantees that redemption only occurs during or after the vesting period, preventing early withdrawals.
- `mtoken_coin: Coin<MToken>`
    - **Validation**: Corresponds to an `mtoken` a vesting manager manages.
    - **Impact**: Represents the value to be redeemed, which determines the penalty and vested coin amount.

- `penalty_coin: &mut Coin<Penalty>`
  - **Validation**: Ensures that `penalty_coin.value()` is sufficient to cover the calculated penalty:
    `assert!(penalty_coin.value() >= penalty_amount, ENotEnoughPenaltyFunds)`.
  - **Impact**: Ensures that penalties are adequately funded.
- `clock: &Clock`
  - **Validation**: Provides the current timestamp for penalty calculations, which is indirectly validated by ensuring `current_time >= start_time_s` and must come from `use sui::clock::{Self, Clock}`.
  - **Impact**: Ensures accurate time-based calculations for vesting and penalty interpolation.

Branches and code coverage (including function calls)

Intended branches

- Redeem immediately after minting, redeem at midtime, and redeem at maturity.
  - ☑ Test coverage
- Edge case like a scenario where the calculated penalty approaches the total value of `penalty_coin`, ensuring no rounding errors in penalty computation.
  - ☐ Test coverage

Negative behavior

- Attempt to redeem before the vesting period starts (`ERedeemingBeforeStartTime`), and attempt to redeem with insufficient `penalty_coin` funds (`ENotEnoughPenaltyFunds`).
  - ☑ Negative test
- Redeem with invalid numerator/denominator ratios like `manager.penalty_denominator` equal to zero.
  - ☐ Negative test

Function call analysis

- `redeem_mtokens -> clock::timestamp_ms(clock)`
  - **External/Internal?** Internal.
  - **Argument control?** Caller controls `clock` by passing it to the function, but it must come from `use sui::clock::{Self, Clock}`, guaranteeing type safety.
  - **Impact**: Determines the current timestamp, essential for validating the redemption period and calculating penalties.
- `redeem_mtokens -> decimal::from(value)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled via `withdraw_amount` as value comes from `mtoken_coin`.
  - **Impact**: Converts values for decimal arithmetic in penalty calculation.

- `redeem_mtokens -> manager.mtoken_treasury_cap.burn(mtoken_coin)`
  - **External/Internal?** Internal.
  - **Argument control?** Caller controls `mtoken_coin` by passing it to the function.
  - **Impact**: Burns the redeemed `mtoken` amount, ensuring it cannot be reused.
- `redeem_mtokens -> penalty_coin.balance_mut().split(penalty_amount)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled by the caller via `penalty_coin` and the calculated `penalty_amount`.
  - **Impact**: Deducts the penalty amount from `penalty_coin`, ensuring proper fund transfer for penalties.
- `redeem_mtokens -> coin::from_balance(manager.vesting_balance.split(...)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled via `withdraw_amount`, which is based on `mtoken_coin`.
  - **Impact**: Converts the redeemed vesting balance back into a coin for the user.

5.3. Module: points.move

Function: `burn_points<T: drop>`

This burns `SUILEND_POINT` tokens in exchange for another token type based on a predefined ratio. This function validates that the manager has sufficient balance, calculates the reward, and burns the specified amount of points.

Inputs

- `points_manager: &mut PointsManager<T>`
  - **Validation**: Ensures `points_manager` has sufficient balance to cover the calculated reward: `assert!(amount <= points_manager.balance.value(), EManagerUnfunded)`. However, decimal equivalency between `points` and `points_manager.balance` is not checked.
  - **Impact**: Controls the reward calculation and deducts the corresponding amount from the `PointsManager` balance. Any decimal differences between the type `points_manager` uses and `points` are not accounted for.
- `points: Coin<SUILEND_POINT>`
  - **Validation**: After being used in the calculation, it must be less than or equal to `points_manager.balance.value()`.
  - **Impact**: Represents the amount of `SUILEND_POINT` tokens being burned, which determines the reward and that the reward is sufficiently funded.

Branches and code coverage (including function calls)

Intended branches

- Burn `SUILEND_POINT` with a valid ratio and sufficient balance, and correct reward calculation based on `Ratio`.
  - [x] Test coverage
- Scenario where `points.value()` is zero.
  - [x] Test coverage

Negative behavior

- Attempt to burn points with insufficient balance (`EManagerUnfunded`).
  - [x] Negative test
- Burn points with a malformed `Ratio`, such as zero denominator.
  - [ ] Negative test

Function call analysis

- `burn_points -> points.value()`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled by the caller via `points`.
  - **Impact**: Determines the amount of `SUILEND_POINT` tokens being burned, affecting the reward calculation.
- `burn_points -> points_manager.balance.split(amount)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled via calculated `amount`.
  - **Impact**: Splits the specified amount from the `PointsManager` balance, reducing it accordingly.
- `burn_points -> coin::from_balance(balance, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** Indirectly controlled via `balance.split`.
  - **Impact**: Converts the deducted balance into a reward coin for the user.

6. Assessment Results

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

During our assessment on the scoped Sui Token & Ticket contracts, we discovered one finding, which was of 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.