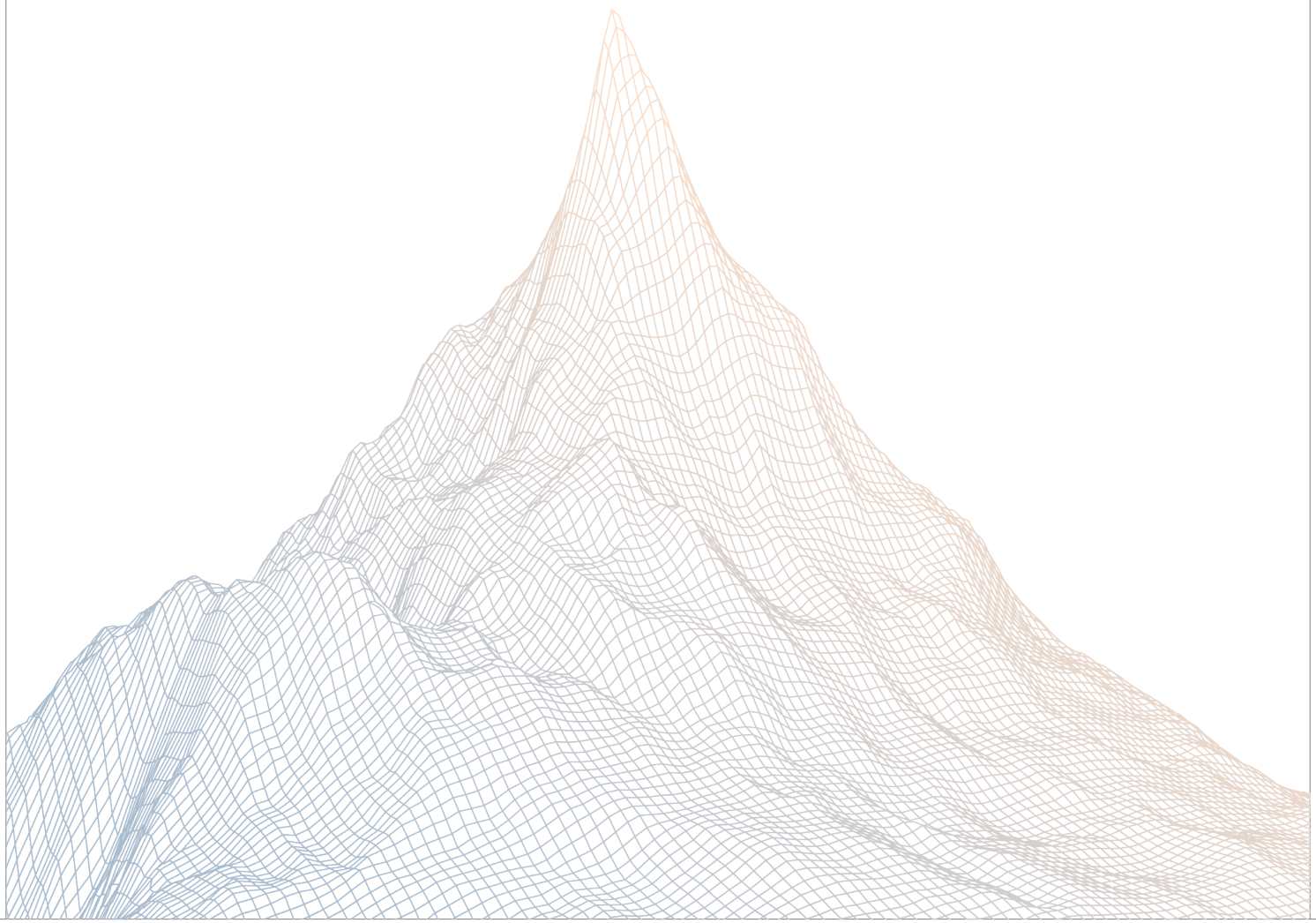
Zenith

Talus

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: October 7th to October 8th, 2025

AUDITED BY: 0x37
Christian Vari

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Talus

Talus has a vision of AI agents powering a new age of user-centric internet. We stand to serve the user. That is our goal. However, we realize that achieving this mission will require collaboration. Therefore we are building applications directly for users, but also the infrastructure and tooling that will empower an ecosystem of developers to create best-in-class AI-enhanced applications for the user.

To that end, Talus is the onchain platform for AI agents while Nexus is the developer framework for building these agents.

The AI agent space is broad and competitive, but we see the unique market opportunity to augment AI agents with the unique value proposition blockchains offer. A distinctive approach, the Talus way. For our community of users, by our community of developers, together with our partners.

2.2 Scope

The engagement involved a review of the following targets:

Target	talus-token
Repository	https://github.com/Talus-Network/talus-token
Commit Hash	`dbc4eb91e7964c96c3659cb942d3b4cc5c11e628`
Files	`deposit_pool.move` `reward_pool.move`

Target	Talus Mitigation Review
Repository	https://github.com/Talus-Network/talus-token
Commit Hash	`e12d33415f65b221c7ca4bdd76fc40f91df52b37 (v1.0.0-rc)`
Files	`deposit_pool.move` `reward_pool.move`

2.3 Audit Timeline

October 7, 2025	Audit start
October 8, 2025	Audit end
October 8, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	3
Informational	3
Total Issues	**7**

3

Findings Summary

ID	Description	Status
M-1	Users may claim less rewards than expected if we meet the exchange rate change	Resolved
L-1	reward_pool does not work with high decimal, low value reward token	Resolved
L-2	missing one apr protection in deposit	Resolved
L-3	Missing exchange rate validation leading to potential division-by-zero	Resolved
I-1	Fail to update one apy with one higher precision	Resolved
I-2	Suggest to prevent the term0 removed	Resolved
I-3	Lack of minimum claim threshold allows dust-level token burns	Resolved

4

Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] Users may claim less rewards than expected if we meet the exchange rate change

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- reward_pool.move

Description:

In reward_pool, users can choose to claim the related reward token via the loyalty token. The claimed reward token amount will be calculated based on the current exchange rate.

The problem here is that the exchange rate can be changed at any time by the related pool's owner. This will cause that users may fail to claim the expected reward.

```
public fun update_rate<Loyalty, Reward>(
    pool: &mut RewardPool<Loyalty, Reward>,
    admin_cap: &mut AdminCap,
    new_rate: u32,
) {
    assert!(pool.admin_cap_id == id(admin_cap), ENotAdmin);

    pool.exchange_rate = new_rate
}
```

Recommendations:

Add one expected reward token amount parameter.

Talus: Resolved in `8b3e13809395e3057f088bc779ad2a89e0f78e15`

Zenith: Verified

4.2 Low Risk

A total of 3 low risk findings were identified.

[L-1] reward_pool does not work with high decimal, low value reward token

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool.move

Description:

In reward_pool, users can claim reward token by the loyalty token. The claimed reward token amount will be calculated via the `exchange_rate`.

The problem here is that `exchange_rate` is u32 type. There is one assumption: one reward token's value must be larger than or equal to one loyalty token. If we want to distribute one high decimal, low value token as the reward token, we may fail to update the exchange rate with one expected value, e.g. 1 loyalty —> 2 reward token.

Recommendations:

Talus: Resolved in `8b3e13809395e3057f088bc779ad2a89e0f78e15`

Zenith: Verified

[L-2] missing one apr protection in deposit

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool.move

Description:

In deposit_pool, users will choose one term to deposit their base coin. They will earn the related apr for the chosen term.

The problem here is that the admin role may choose to update the related term's apy at any time. This will cause that users may get the less fixed apy than expected. What's more, users don't have the permission to re-choose one term because we don't allow to withdraw before the end of term.

```
public fun upsert_lock_term<Base, Loyalty>(
    pool: &mut DepositPool<Base, Loyalty>,
    admin: &mut AdminCap,
    days: u32,
    apy: u8,
) {
    assert!(pool.admin_cap_id == object::id(admin), ENotAdmin);
    assert!(pool.version == VERSION, EWrongVersion);
    // lock_term in ms
    // We will not
    if (pool.return_rates.contains(days)) {
        pool.return_rates.remove(days);
    };

    pool.return_rates.add(days, apy)
}
```

Recommendations:

Users can assign one expected APR. If the actual APR is less than the expected APY, we can revert this transaction, users can choose another prefer term and APR.

Talus: Resolved in 2c810d3f71cf19b5e5e4b92d832b615713fb8f41.

Zenith: Verified.

[L-3] Missing exchange rate validation leading to potential division-by-zero

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool/sources/reward_pool.move

Description:

In the reward pool implementation, the `claim()` function divides by `pool.exchange_rate` without verifying that the value is greater than zero.

The functions `new()` and `update_rate()` both allow the initialization or modification of the exchange rate to `0`, creating a potential for a division-by-zero panic if a claim is executed under this condition.

Recommendations:

We recommend adding explicit validation to ensure `exchange_rate > 0` in both the `new()` and `update_rate()` functions before assignment.

Talus: Resolved in `8b3e13809395e3057f088bc779ad2a89e0f78e15`

Zenith: Verified

4.3 Informational

A total of 3 informational findings were identified.

[I-1] Fail to update one apy with one higher precision

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool.move

Description:

In deposit_pool, the admin can set one fixed apy for one term. TAPY is u8 type, and we will calculate the actual apy via `apy / MAX_PCT`. This will decrease the configuration's flexibility. e.g. If the admin wants to set APY with 3.5%, we will fail to update this.

```
public fun upsert_lock_term<Base, Loyalty>(
    pool: &mut DepositPool<Base, Loyalty>,
    admin: &mut AdminCap,
    days: u32,
    apy: u8,
) {
    assert!(pool.admin_cap_id == object::id(admin), ENotAdmin);
    assert!(pool.version == VERSION, EWrongVersion);
    // lock_term in ms
    // We will not
    if (pool.return_rates.contains(days)) {
        pool.return_rates.remove(days);
    };

    pool.return_rates.add(days, apy)
}
```

Recommendations:

Talus: Resolved in 675bc97be8ec7c5420e036cfdfd1a46e791ba3e3.

Zenith:: Verified.

[I-2] Suggest to prevent the term0 removed

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool.move

Description:

In deposit_pool, we allow to delete one existing term via `delete_lock_term`. Based on current logic, if users input one incorrect term, we will choose the default term(term 0) for this deposit. So we should always keep one valid term 0.

Suggest to prevent the term0 removed.

```
public fun delete_lock_term<Base, Loyalty>(
    pool: &mut DepositPool<Base, Loyalty>,
    admin: &mut AdminCap,
    days: u32,
) {
    assert!(pool.admin_cap_id == object::id(admin), ENotAdmin);
    assert!(pool.version == VERSION, EWrongVersion);
    // lock_term in ms
    pool.return_rates.remove(days);
}
```

Recommendations:

Suggest to prevent the term0 removed.

Talus: Resolved in 2c810d3f71cf19b5e5e4b92d832b615713fb8f41

Zenith: Verified

[I-3] Lack of minimum claim threshold allows dust-level token burns

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- deposit_pool/sources/reward_pool.move

Description:

In the `claim()` function, if extremely small ("dust-level") token amounts are processed, due to rounding or integer flooring during conversion (`token.value() / exchange_rate`), the calculated payout may round down to zero.

Despite no actual token payout occurring, the function still proceeds to burn loyalty tokens, effectively removing user balances without compensation.

This behavior may result in minor user value loss and confusion when claiming negligible balances below the system's minimum convertible unit.

Recommendations:

We recommend implementing a minimum claim threshold or validation check to prevent burning loyalty tokens when the resulting payout is zero.

Talus: Resolved in `8b3e13809395e3057f088bc779ad2a89e0f78e15`

Zenith: Verified