Audited by

CredShields

# Smart Contract Audit

14th August, 2025 • CONFIDENTIAL

**Description**

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Zodor between August 6th, 2025, and August 7th, 2025. A retest was performed on August 14th, 2025.

**Author**
Shashank (Co-founder, CredShields) shashank@CredShields.com

**Reviewers**
Aditya Dixit (Research Team Lead), Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor), Yash Shah (Auditor), Prasad Kuri (Auditor)

**Prepared for**
Zodor

# Table of Contents

## 1. Executive Summary --------------------

Zodor engaged CredShields to perform a smart contract audit from August 6th, 2025, to August 7th, 2025. During this timeframe, 7 vulnerabilities were identified. **A retest was performed on August 14th, 2025, and all the bugs have been addressed.**

During the audit, 2 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Zodor" and should be prioritized for remediation.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

| Assets in Scope | Critical | High | Medium | Low | info | Gas | Σ |
|---|---|---|---|---|---|---|---|
| Zodor RWA Contract | 2 | 0 | 1 | 2 | 0 | 2 | **7** |
| | **2** | **0** | **1** | **2** | **0** | **2** | **7** |

*Table: Vulnerabilities Per Asset in Scope*

The CredShields team conducted the security audit to focus on identifying vulnerabilities in the Zodor RWA Contract's scope during the testing window while abiding by the policies set forth by Zodor's team.

## State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Zodor's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Zodor can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Zodor can future-proof its security posture and protect its assets.

# 2. The Methodology --------------------

Zodor engaged CredShields to perform a Zodor RWA Smart Contract audit. The following sections cover how the engagement was put together and executed.

## 2.1 Preparation Phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from August 6th, 2025, to August 7th, 2025, was agreed upon during the preparation phase.

### 2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

| IN SCOPE ASSETS |
|---|
| https://github.com/ZodorRWA/Zodor_staking/tree/bf653405b83026c82406c194032d1e3060671e38 |

### 2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

### 2.1.3 Audit Goals

CredShields employs a combination of in-house tools and thorough manual review processes to deliver comprehensive smart contract security audits. The majority of the audit involves manual inspection of the contract's source code, guided by OWASP's Smart Contract Security Weakness Enumeration (SCWE) framework and an extended, self-developed checklist built from industry best practices. The team focuses on deeply understanding the contract's core logic, designing targeted test cases, and assessing business logic for potential vulnerabilities across OWASP's identified weakness classes.

CredShields aligns its auditing methodology with the OWASP Smart Contract Security projects, including the Smart Contract Security Verification Standard (SCSVS), the Smart Contract Weakness Enumeration (SCWE), and the Smart Contract Secure Testing Guide (SCSTG). These frameworks, actively contributed to and co-developed by the CredShields team, aim to bring consistency, clarity, and depth to smart contract security assessments. By adhering to these OWASP standards, we ensure that each audit is performed against a transparent, community-driven, and technically robust baseline. This approach enables us to deliver structured, high-quality audits that address both common and complex smart contract vulnerabilities systematically.

## 2.2 Retesting Phase

Zodor is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

## 2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

| **Overall Risk Severity** | | | | |
|---|---|---|---|---|
| **Impact** | HIGH | 🟡 Medium | 🟠 High | 🔴 Critical |
| | MEDIUM | 🟢 Low | 🟡 Medium | 🟠 High |
| | LOW | ⚪ None | 🟢 Low | 🟡 Medium |
| | | LOW | MEDIUM | HIGH |
| **Likelihood** | | | | |

Overall, the categories can be defined as described below -

## 1. Informational

We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

## 2. Low

Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

## 3. Medium

Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. **High**

   High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. **Critical**

   Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

## 2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- Shashank, Co-founder CredShields shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

# 3. Findings Summary -------------------

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by asset and OWASP SCWE classification. Each asset section includes a summary highlighting the key risks and observations. The table in the executive summary presents the total number of identified security vulnerabilities per asset, categorized by risk severity based on the OWASP Smart Contract Security Weakness Enumeration framework.

## 3.1 Findings Overview

### 3.1.1 Vulnerability Summary

During the security assessment, 7 security vulnerabilities were identified in the asset.

| VULNERABILITY TITLE | SEVERITY | SCWE \| Vulnerability Type |
|---|---|---|
| Owner Can Arbitrarily Cap Rewards for All Stakers Mid-Staking | Critical | Business Logic Issue (SCWE-001) |
| User can permanently lose staked funds due to same-block timestamp vulnerability | Critical | Business Logic Issue (SCWE-001) |
| Contract can permanently lock wei amounts due to precision loss in refund calculations | Medium | Business Logic Issue (SCWE-001) |
| Use Ownable2Step | Low | Missing Best Practices |
| Floating and Outdated Pragma | Low | Floating Pragma (SCWE-060) |
| Splitting Require/Revert Statements | Gas | Gas Optimization (SCWE-082) |
| Inefficient x > 0 Check Increases Gas Cost | Gas | Gas Optimization (SCWE-082) |

*Table: Findings in Smart Contracts*

## 4. Remediation Status ------------------

Zodor is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on August 14th, 2025, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

| VULNERABILITY TITLE | SEVERITY | REMEDIATION STATUS |
|---|---|---|
| Owner Can Arbitrarily Cap Rewards for All Stakers Mid-Staking | Critical | Fixed [August 14, 2025] |
| User can permanently lose staked funds due to same-block timestamp vulnerability | Critical | Fixed [August 14, 2025] |
| Contract can permanently lock wei amounts due to precision loss in refund calculations | Medium | Fixed [August 14, 2025] |
| Use Ownable2Step | Low | Fixed [August 14, 2025] |
| Floating and Outdated Pragma | Low | Fixed [August 14, 2025] |
| Splitting Require/Revert Statements | Gas | Fixed [August 14, 2025] |
| Inefficient x > 0 Check Increases Gas Cost | Gas | Fixed [August 14, 2025] |

*Table: Summary of findings and status of remediation*

## 5. Bug Reports ------------------------

### Bug ID #C001 [Fixed]

### Owner Can Arbitrarily Cap Rewards for All Stakers Mid-Staking

**Vulnerability Type**
Business Logic Issue (SCWE-001)

**Severity**
Critical

**Description**
When refundMode is activated mid-staking period, user rewards are capped based on the time elapsed at the moment of refund activation. Even if users complete the full lock duration, they only receive partial rewards, which is unintuitive and can lead to user dissatisfaction

For example, if a user stakes on Day 0 into Plan 1 (duration: 129,600 minutes or 90 days) and the owner activates refund on Day 45, the logic executes during claim:Since elapsedMinutes = 64800 and plan.durationMinutes = 129600, the condition elapsedMinutes >= plan.durationMinutes fails, leading to reward proration. The user receives only 50% of the intended reward, even if they wait and claim after the full 90-day period.

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L136-L137

**Impacts**
Users receive reduced rewards even after fully completing their lock period, leading to financial loss and broken staking expectations.

**Remediation**
It is recommended to allow users who complete the full staking period to receive full rewards, even in refundMode.
Suggested fix: Compare current time against startTimestamp + duration and grant full reward if staking is fully completed.

**Retest**

The issue is fixed as per the recommended changes.

## Bug ID #C002 [ Fixed ]

### User can permanently lose staked funds due to same-block timestamp vulnerability

**Vulnerability Type**
Business Logic Issue (SCWE-001)

**Severity**
Critical

**Description**
The ZodorStaking contains a critical vulnerability in its refund mode implementation that can result in permanent fund loss for users. The issue stems from the strict inequality check in the claim function at line 156: require(refundActivationTime > pos.startTimestamp, "Refund before stake"). This condition creates a scenario where users who stake in the same block as when refund mode is activated are completely unable to withdraw their funds.

The vulnerability manifests when the following sequence occurs within a single block: a user calls the stake function, and subsequently, the owner calls activateRefund in the same block. Since both transactions share identical block.timestamp values, the staked position's startTimestamp equals the refundActivationTime. When the user later attempts to claim their position, the strict greater-than comparison fails because refundActivationTime is not greater than pos.startTimestamp when they are equal, causing the transaction to revert with "Refund before stake" error.

This creates an impossible scenario where the user's funds become permanently locked in the contract. The position cannot be claimed in refund mode due to the failed timestamp check, and it cannot be claimed in normal mode either since refund mode, once activated, remains permanent with no mechanism to deactivate it.

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L132

**Impacts**
Users affected by this edge case experience complete and permanent loss of their staked principal amounts, with no available recovery mechanism within the contract's current implementation.

**Remediation**
The vulnerability can be resolved by modifying the timestamp comparison logic to use a greater-than-or-equal-to operator instead of the strict greater-than comparison.

**Retest**
The bug has been fixed by following the recommended steps.

Bug ID #M001 [ Fixed ]

## Contract can permanently lock wei amounts due to precision loss in refund calculations

**Vulnerability Type**
Business Logic Issue (SCWE-001)

**Severity**
Medium

**Description**
The ZodorStaking contract contains a precision loss vulnerability in its refund reward calculation logic that results in small amounts of wei being permanently locked within the contract. The issue occurs in the claim function between where integer division operations cause rounding down of calculated values, creating discrepancies in the reward pool accounting.

The vulnerability manifests through a two-step division process. First, elapsedMinutes is calculated using (refundActivationTime - pos.startTimestamp) / 60, which truncates any fractional seconds. Second, the proportional reward calculation (fullReward * elapsedMinutes) / plan.durationMinutes performs another division that can result in additional precision loss. When unusedReward is calculated as fullReward - reward, it fails to account for the cumulative precision loss from both division operations.

This creates a scenario where the sum of distributed rewards plus returned unused rewards is less than the original fullReward amount that was deducted from the reward pool during staking. The difference, typically consisting of small wei amounts, remains untracked and becomes permanently inaccessible. The withdrawReward function's strict condition amount <= rewardPool prevents the owner from withdrawing these locked funds, as they exist in the contract's token balance but are not reflected in the rewardPool accounting variable.

Over time, as multiple users claim their refund positions, these precision losses accumulate, creating an increasingly significant amount of permanently locked tokens that serve no economic purpose and cannot be recovered through normal contract operations.

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L136

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L142

**Impacts**
The financial impact of this vulnerability is cumulative and progressive. Each refund claim operation potentially locks small wei amounts permanently within the contract, and these losses compound over time as more users interact with the refund mechanism.

**Remediation**
A simpler but more centralized approach is to remove the strict validation in the withdrawReward function by eliminating the condition require(amount > 0 && amount <= rewardPool, "Invalid withdraw amount"). This would allow the owner to withdraw any tokens present in the contract, including the locked precision loss amounts. While this remediation introduces centralization risks by granting the owner broader withdrawal capabilities beyond the tracked reward pool, it provides an immediate solution to recover locked funds and prevents permanent capital inefficiency.

**Retest**
This bug has been fixed by using Math.muldiv() to avoid precision

## Bug ID #L001 [ Fixed ]

### Use Ownable2Step

**Vulnerability Type**
Missing Best Practices

**Severity**
Low

**Description**
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L10

**Impacts**
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

**Remediation**
It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

**Retest**:
The bug is fixed by using Ownable2Step .

Bug ID #L002 [ Fixed ]

## Floating and Outdated Pragma

**Vulnerability Type**
Floating Pragma (SCWE-060)

**Severity**
Low

**Description**
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contract allowed floating or unlocked pragma to be used, i.e., >= 0.8.24. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L2

**Impacts**
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.
Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is low.

**Remediation**
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.29 pragma version
Reference: https://scs.owasp.org/SCWE/SCSVS-CODE/SCWE-060/

**Retest**
The issue is fixed as per the recommended changes.

## Bug ID #G001 [Fixed]

### Splitting Require/Revert Statements

**Vulnerability Type**
Gas Optimization (SCWE-082)

**Severity**
Gas

**Description**
Require/Revert statements when combined using operators in a single statement usually lead to a larger deployment gas cost but with each runtime calls, the whole thing ends up being cheaper by some gas units.

**Affected Code**

- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L179

**Impacts**
The multiple conditions in one **require/revert** statement combine require/revert statements in a single line, increasing deployment costs and hindering code readability.

**Remediation**
It is recommended to separate the **require/revert** statements with one statement/validation per line.

**Retest**
The bug has been fixed by following the recommended steps.

Bug ID #G002 [ Fixed ]

## Inefficient x > 0 Check Increases Gas Cost

**Vulnerability Type**
Gas Optimization (SCWE-082)

**Severity**
Gas

**Description**
Using x > 0 on unsigned integers is functionally correct but less gas-efficient than x != 0. The EVM handles != 0 more efficiently, making it the preferred choice in conditions and require statements.

**Affected Code**
- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L74
- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L121
- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L170
- https://github.com/ZodorRWA/Zodor_staking/blob/bf653405b83026c82406c194032d1e3060671e38/ZodorStaking.sol#L179

**Impacts**
Slightly higher gas usage for users, especially in frequently called functions.

**Remediation**
Replace x > 0 with x != 0 for all unsigned integer checks to reduce gas costs.

**Retest**
The issue is fixed as per the recommended changes.

## 6. The Disclosure ----------------------

The Reports provided by CredShields are not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.

YOUR SECURE FUTURE STARTS HERE

CRED SHiELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHiELDS