Audited by CRED SHiELDS

CredShields

Smart Contract Audit

January 08, 2026 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of HeyElsa between January 02, 2026, and January 07, 2026. A retest was performed on January 08, 2026.

Author

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

Reviewers

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

Prepared for

HeyElsa

Table of Contents

1. Executive Summary

HeyElsa engaged CredShields to perform a smart contract audit from January 02, 2026, to January 07, 2026. During this timeframe, 12 vulnerabilities were identified. A retest was performed on January 08, 2026, 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 "HeyElsa" 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
Staking Contract	0	2	3	2	2	3	12

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted a security audit to focus on identifying vulnerabilities within the Staking Contract's scope during the testing window, while adhering to the policies set forth by HeyElsa'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 HeyElsa'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 HeyElsa can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, HeyElsa can future-proof its security posture and protect its assets.

2. The Methodology

HeyElsa engaged CredShields to perform a Staking 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 January 02, 2026, to January 07, 2026, 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
Audited Commit: https://github.com/HeyElsa/staking-contract/tree/e2228f87383f2bf199a6cc775672ddc3fc9e10c7 **Retested Commit:** https://github.com/HeyElsa/staking-contract/tree/101cfddccf2c033bef79b49e2298fcb08432bd51

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

HeyElsa 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, 12 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SCWE | Vulnerability Type
H001 — Reward loss due to updating rewardDebt during insolvency	High	Business Logic (SC03-LogicErrors)
H002 — Emergency Withdraw Can Forfeit User Rewards	High	Inconsistent Accounting (SCWE-010)
M001 — Permit Front-Running Griefing in stakeWithPermit Can Cause Transaction Reverts	Medium	Denial of Service (SCWE-087)
M002 — User Can Permanently Block Own Account Interactions	Medium	Denial of Service (SCWE-087)
M003 — Admin Can Alter Rewards For Existing Stakers	Medium	Inconsistent Accounting (SCWE-010)
L001 — Outdated Pragma	Low	Outdated Compiler Version (SCWE-061)
L002 — Reactivating a Previously Deactivated Tier Does Not Reinsert It into activeTierIds Array	Low	Business Logic (SC03-LogicErrors)
I001 — Developers Can Mislead Auditors And Users	Informational	Documentation Error

I002 — Admin Can Trigger Contract Insolvency	Informational	Business Logic (SC03-LogicErrors)
G001 — Cheaper conditional operators	Gas	Gas Optimization (SCWE-082)
G002 — Cheaper Inequalities in if()	Gas	Gas Optimization (SCWE-082)
G003 — Gas Optimization in Increments	Gas	Gas Optimization (SCWE-082)

Table: Findings in Smart Contracts

4. Remediation Status

HeyElsa is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. A retest was performed on January 08, 2026, and all the issues have been addressed.

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
H001 — Reward loss due to updating rewardDebt during insolvency	High	Fixed [Jan 8, 2026]
H002 — Emergency Withdraw Can Forfeit User Rewards	High	Fixed [Jan 8, 2026]
M001 — Permit Front-Running Griefing in stakeWithPermit Can Cause Transaction Reverts	Medium	Fixed [Jan 8, 2026]
M002 — User Can Permanently Block Own Account Interactions	Medium	Fixed [Jan 8, 2026]
M003 — Admin Can Alter Rewards For Existing Stakers	Medium	Fixed [Jan 8, 2026]
L001 — Outdated Pragma	Low	Fixed [Jan 8, 2026]
L002 — Reactivating a Previously Deactivated Tier Does Not Reinsert It into activeTierIds Array	Low	Fixed [Jan 8, 2026]
I001 — Developers Can Mislead Auditors And Users	Informational	Fixed [Jan 8, 2026]
I002 — Admin Can Trigger Contract Insolvency	Informational	Fixed [Jan 8, 2026]
G001 — Cheaper conditional operators	Gas	Fixed [Jan 8, 2026]
G002 — Cheaper Inequalities in if()	Gas	Fixed [Jan 8, 2026]
G003 — Gas Optimization in Increments	Gas	Fixed [Jan 8, 2026]

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #H001 [Fixed]

Reward loss due to updating rewardDebt during insolvency

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
The _updateReward() function is responsible for updating global and user-specific reward state. It correctly checks whether the contract is solvent (rewardPool >= totalAccruedRewards) before accruing new rewards for a user. However, even when the contract is not solvent, the function still calls _updateStakeRewardDebts(account).
_updateStakeRewardDebts() updates each active stake's rewardDebt to the latest rewardPerTokenStored. When insolvency occurs, users are explicitly prevented from accruing rewards, but their rewardDebt is still advanced as if rewards were accrued. This creates a mismatch between actual earned rewards and accounting state.
As a result, when the contract later becomes solvent again, users cannot claim the rewards corresponding to the insolvent period because their rewardDebt was already moved forward. This violates expected reward accounting semantics: reward debt should only be updated when rewards are actually accrued.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L900-L921

Impacts
Permanent loss of user rewards for periods when the contract is insolvent. Users are unfairly penalized despite rewards not being credited to them.

Remediation
It is suggested to call _updateStakeRewardDebts(account) only when the contract is solvent, i.e., only when rewards are actually accrued. This ensures rewardDebt advances in sync with real reward accumulation

Retest

This vulnerability has been fixed by calling _updateReward() only when the solvent is true.

Bug ID #H002 [Fixed]

Emergency Withdraw Can Forfeit User Rewards

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
High

Description
The ELSAStaking.emergencyWithdraw function allows users to exit all active stakes when emergency mode is enabled and also attempts to pay out any pending rewards. Rewards are tracked in rewards[msg.sender] and are expected to represent already accrued user entitlements. However, when processing rewards, the function checks whether reward <= rewardPool and only transfers rewards if sufficient funds are available. If rewardPool is smaller than the user's accrued rewards, the function emits RewardsForfeited and silently drops the user's reward claim without preserving it for future settlement. The root cause is that accrued rewards are cleared or ignored instead of being recorded as outstanding liabilities when the reward pool is temporarily insufficient.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L396-L406

Impacts
Alice has accrued 1,000 ELSA in rewards but calls emergencyWithdraw during a period where the reward pool only contains 600 ELSA. Her full reward entitlement is forfeited and permanently lost, even if the reward pool is later replenished, resulting in an irreversible loss of user funds.

Remediation
Do not forfeit user rewards during emergency withdrawals; instead, preserve unpaid rewards as outstanding claims until the reward pool is replenished and can satisfy them.

Retest
This issue is fixed by not zeroing users rewards in emergencyWithdraw() function

Bug ID #M001 [Fixed]

Permit Front-Running Griefing in stakeWithPermit Can Cause Transaction Reverts

Vulnerability Type
Denial of Service (SCWE-087)

Severity
Medium

Description
The stakeWithPermit() function relies on an inline call to the permit() function to approve token spending before staking. Because permit() is a publicly callable function and uses a nonce that is incremented on successful execution, an attacker can observe a user's stakeWithPermit() transaction in the mempool, copy the signed permit parameters, and submit a standalone permit() transaction first.
This front-run call consumes the user's permit nonce and sets the allowance. When the original user transaction is later executed, the internal permit() call reverts due to an already-used nonce, causing the entire stakeWithPermit transaction to fail. While the attacker cannot steal funds or gain approval beyond what the user intended, this results in a temporary denial-of-service and gas griefing for the user

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L212

Impacts
A malicious actor can repeatedly front-run stakeWithPermit transactions and force them to revert, causing users to waste gas and preventing them from successfully staking via the permit-based flow.

Remediation
The issue can be mitigated by wrapping the permit() call in a try/catch block so that a failure to execute permit() does not automatically revert the entire stakeWithPermit transaction.

Retest
Fixed in cb63fc2.

Bug ID #M002 [Fixed]

User Can Permanently Block Own Account Interactions

Vulnerability Type
Denial of Service (SCWE-087)

Severity
Medium

Description
The staking system tracks user positions in an append-only userStakes array, where each call to stake appends a new element and unstake only marks the entry as inactive without removing it. Over time, this array monotonically grows for a user regardless of stake lifecycle. Multiple core functions, including _getActiveStakeCount, _calculatePendingRewards, and _updateStakeRewardDebts, iterate over the entire userStakes array on every user interaction. While the contract introduces MAX_STAKES_PER_USER to cap active stakes, the check itself relies on _getActiveStakeCount, which also performs a full linear scan of the unbounded array. As a result, a user who repeatedly stakes and unstakes can inflate their own userStakes array to the point where all subsequent interactions revert due to exceeding block gas limits.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L926-L941
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L845-L856
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L946-L956

Impacts
Alice repeatedly stakes and unstakes small amounts, growing her userStakes array to thousands of entries. Eventually, any call involving reward updates or stake counting runs out of gas, permanently preventing Alice from staking, unstaking, claiming rewards, or exiting the system, effectively locking her funds.

Remediation
It is suggested that remove unstaked entries from the userStakes array to keep its size bounded and ensure all user interactions remain within gas limits.

Retest

Fixed in cb63fc2.

Bug ID #M003 [Fixed]

Admin Can Alter Rewards For Existing Stakers

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
Medium

Description
The ELSAStaking.setLockTier function allows an admin to configure lock tier parameters such as lockDuration, rewardMultiplier, and earlyUnstakePenalty, and it can be called at any time for an active tier. Existing users may already have active stakes associated with a given tier, and their rewards are accrued over time based on the tier's rewardMultiplier.
However, the function directly overwrites tier.rewardMultiplier without first settling or snapshotting rewards for users who are already staking in that tier.
As a result, previously accrued rewards are implicitly recalculated using the new multiplier, because reward accounting relies on the current tier parameters rather than immutable values fixed at stake creation.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L657

Impacts
Alice stakes tokens in a tier expecting a higher reward multiplier over her lock period. An admin later reduces the multiplier via setLockTier, causing Alice's already accrued and future rewards to be lower than expected, breaking economic guarantees and undermining user trust in the staking system.

Remediation
It is recommended to store current rewardMultiplier in userStakes while stakes are created and use the same rewardMultiplier while calculating the reward.

Retest
Fixed in 101cfd.

Bug ID #L001 [Fixed]

Outdated Pragma

Vulnerability Type
Outdated Compiler Version (SCWE-061)

Severity Low

Description
The smart contract is using an outdated version of the Solidity compiler specified by the pragma directive i.e. 0.8.29. Solidity is actively developed, and new versions frequently include important security patches, bug fixes, and performance improvements. Using an outdated version exposes the contract to known vulnerabilities that have been addressed in later releases. Additionally, newer versions of Solidity often introduce new language features and optimizations that improve the overall security and efficiency of smart contracts.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L2
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/libraries/StakingErrors.sol#L2
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/libraries/StakingEvents.sol#L2

Impacts
The use of an outdated Solidity compiler version can have significant negative impacts. Security vulnerabilities that have been identified and patched in newer versions remain exploitable in the deployed contract.
Furthermore, missing out on performance improvements and new language features can result in inefficient code execution and higher gas costs.

Remediation
It is suggested to use the 0.8.30 pragma version.
Reference: https://scs.owasp.org/SCWE/SCSVS-CODE/SCWE-061/

Retest
Fixed in cb63fc2.

Bug ID #L002 [Fixed]

Reactivating a Previously Deactivated Tier Does Not Reinsert It into activeTierIds Array

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Low

Description
The contract maintains an activeTierIds array to track all currently active lock tiers, which is exposed through getActiveTierIds() and is expected to be the authoritative list of tiers available for staking. When a tier is first created via setLockTier, it is added to this array. If the tier is later deactivated using deactivateTier, its tierId is correctly removed from activeTierIds. However, when the same tier is reactivated by calling setLockTier again, the logic that determines whether a tier is "new" relies on the condition !tier.isActive && tier.rewardMultiplier == 0. Since rewardMultiplier is not reset on deactivation, this condition evaluates to false for previously created tiers. As a result, the tier is marked as active again but is not reinserted into activeTierIds, causing the contract to enter an inconsistent state where a tier is active but missing from the active tier list.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L654

Impacts
An active tier can become undiscoverable via getActiveTierIds(), leading frontends, off-chain services, and users to believe the tier is inactive even though staking into it is allowed. This creates inconsistent on-chain state, breaks the single source of truth for active tiers, and can prevent users from interacting with valid staking options unless they already know the tier ID.

Remediation
The logic for managing activeTierIds should rely solely on the tier's activation state and not infer tier existence from rewardMultiplier. When setLockTier is called, the contract should check whether the tier is currently inactive and whether its tierId is already present in activeTierIds; if the tier is being activated and is not in the array, it must be added regardless of its previous parameters.

Retest

Fixed in cb63fc2.

Bug ID #I001 [Fixed]

Developers Can Mislead Auditors And Users

Vulnerability Type
Documentation Error

Severity
Informational

Description
In the _calculateDynamicRewardRate, the inline comment describing the decay formula is incorrect and does not accurately reflect the implemented logic. The comment states that the decay is calculated as baseRate * (1 - decayFactor * stakingRatio / 10000), while the actual implementation computes a decay amount as baseRewardRate * rewardDecayFactor * stakingRatio / (BASIS_POINTS * BASIS_POINTS) and subtracts it from the base rate. This discrepancy creates a mismatch between documented behavior and real execution, with the root cause being outdated or improperly written documentation.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L879-L880

Impacts
Auditors, developers, and external reviewers may misunderstand the reward rate mechanics, leading to incorrect assumptions during security reviews, economic modeling, or governance decisions. This increases the risk of configuration errors, flawed audits, and loss of trust due to perceived inconsistencies between specification and implementation.

Remediation
Update the inline comment to precisely describe the implemented decay calculation and ensure future documentation changes are reviewed alongside code changes.

Retest
Fixed in cb63fc2.

Bug ID #I002 [Won't Fix]

Admin Can Trigger Contract Insolvency

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Informational

Description
The staking system tracks user reward entitlements through totalAccruedRewards, which represents the total rewards owed to users, and rewardPool, which represents available funds. The ELSAStaking.withdrawExcessRewards function allows an admin to withdraw tokens defined as "excess" rewards, calculated as rewardPool - totalAccruedRewards. This mechanism assumes that totalAccruedRewards always reflects all future liabilities accurately. However, reward accrual depends on user interaction and solvency checks, and timing mismatches or delayed updates can cause totalAccruedRewards to temporarily underrepresent actual obligations. As a result, an admin can legally withdraw rewards that will later be owed to users once rewards are updated, pushing the contract into an insolvent state without violating any explicit invariant.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L733

Impacts
Users interacting with the contract, accruing additional rewards that were economically implied but not yet accounted for, causing rewardPool to become insufficient and preventing users from claiming their full rewards.

Remediation
Restrict or remove the ability for admins to withdraw excess rewards, or introduce conservative accounting that buffers future reward accruals and prevents withdrawals that could jeopardize solvency

Retest
Client's comment: This function is protected by ADMIN_ROLE and only allows withdrawal of truly excess funds (rewardPool - totalAccruedRewards). Admin cannot withdraw funds that are owed to

users. For additional protection, consider implementing a timelock or multisig for admin operations in production.

Bug ID #G001 [Fixed]

Cheaper conditional operators

Vulnerability Type
Gas Optimization (SCWE-082)

Severity
Gas

Description
Upon reviewing the code, it has been observed that the contract uses conditional statements involving comparisons with unsigned integer variables. Specifically, the contract employs the conditional operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L389
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L396
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L408
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L913

Impacts
Employing x != 0 in conditional statements can result in reduced gas consumption compared to using x > 0. This optimization contributes to cost-effectiveness in contract interactions.

Remediation
Whenever possible, use the x != 0 conditional operator instead of x > 0 for unsigned integer variables in conditional statements.

Retest

Fixed in cb63fc2.

Bug ID #G002 [Partially Fixed]

Cheaper Inequalities in if( )

Vulnerability Type
Gas Optimization [(SCWE-082)]

Severity
Gas

Description
The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L389
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L396
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L408
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L430
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L458
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L649
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L699
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L700
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L701
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L746
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L890

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L891

Impacts

Using strict inequalities inside “if” statements costs more gas.

Remediation

It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest:

This issues is partially fixed.

Bug ID#G003 [Fixed]

Gas Optimization in Increments

Vulnerability Type
Gas optimization (SCWE-082)

Severity
Gas

Description
The contract uses two for loops**,** which use post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Affected Code

- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L330
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L366
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L506
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L679
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L932
- https://github.com/HeyElsa/staking-contract/blob/e2228f87383f2bf199a6cc775672ddc3fc9e10c7/contracts/ELSAStaking.sol#L951

Impacts
Using **i++** instead of **++i** costs the contract deployment around 600 more gas units.

Remediation
It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest

Fixed in cb63fc2.

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

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

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com