CredShields

Smart Contract Audit

March 20, 2026 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD., prepared for WStaking (by Whale Venture), between March 10, 2026, and March 19, 2026. A retest was performed on March 20, 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

WStaking (by Whale Venture)

Table of Contents

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

WStaking engaged CredShields to perform a smart contract audit from March 10, 2026, to March 19, 2026. During this timeframe, 16 vulnerabilities were identified. **A retest was performed on March 20, 2026, and all the bugs have been addressed.**

During the audit, 3 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "WStaking" 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
WStaking Smart Contract	1	2	3	6	1	3	**16**

Table: Vulnerabilities Per Asset in Scope

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

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

WStaking engaged CredShields to perform a WStaking 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 March 10, 2026, to March 19, 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 Scope:** https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code • **Retested Scope:** https://testnet.bscscan.com/address/0x6B6b5eca778BcFdbC9D8b2d718e01233A1C1d5eD#code

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

WStaking 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, 16 security vulnerabilities were identified in the asset.

Vulnerability Title	Severity	Vulnerability Type	Status
Any User Can Access Developer-Only 10,000% APR Via Unguarded Fallback Branches in _getAPRRange()	Critical	Insufficient Authorization Checks (SCWE-016)	Fixed
Attacker Can Bypass All Unstake Penalties By Channeling Funds Through Auto-Renewed Stakes	High	Insecure Use of Storage (SCWE-044)	Fixed
Partial Unstake Unconditionally Resets APR And Lock Period Without Version Gate, Silently Reducing Yield For Threshold Stakers	High	Business Logic (SCWE-083)	Acknowledged
Missing systemNotDisabled Modifier on addFund()	Medium	Access Control (SCWE-016)	Fixed
Stakers Lose Yield Between Maturity And Late Claim Due To Timestamp Reset	Medium	Business Logic (SCWE-083)	Acknowledged
Owner Cannot Pause Contract During Emergency	Medium	Missing functionality (SCWE-006)	Fixed
Owner Can Silently Brick Existing Promo Stakes By Starting A New Promotion	Low	Insecure Use of Storage (SCWE-044)	Fixed
Owner Can Permanently Brick Protocol Via Single-Step Ownership Transfer	Low	Single-Step Ownership Transfer (SCWE-139)	Fixed

Administrators Cannot Rotate Hot Wallet Keys Without Full Proxy Upgrade	Low	Single EOA Admin Without Rotation (SCWE-129)	Fixed
Users Lose Gas On Silently Failed Core Operations Due To Missing Reverts	Low	Uncaught Exceptions (SCWE-004)	Fixed
Full Unstake Leaves Residual Rewards Phantom-Locked In Global Ungranted Counters	Low	State Inconsistency (SCWE-075)	Fixed
Floating and Outdated Pragma	Low	Floating Pragma (SCWE-060)	Fixed
Claim Receipt Hash Collision Risk For Same-Block Transactions	Informational	Data Integrity (SCWE-074)	Fixed
Public constants can be private	Gas	Gas Optimization (SCWE-082)	Fixed
Cheaper conditional operators	Gas	Gas Optimization (SCWE-082)	Fixed
Gas Optimization in Increments	Gas	Gas Optimization (SCWE-082)	Fixed

Table: Findings and Remediations

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

Bug ID #C001 [Fixed]

Any User Can Access Developer-Only 10,000% APR Via Unguarded Fallback Branches in _getAPRRange()

Vulnerability Type
Insufficient Authorization Checks (SCWE-016)

Severity
Critical

Description
The _getAPRRange() function gates developer test APRs (5,000%–10,000%) behind an isDev check at the top of the function, where isDev = (msg.sender == owner()). However, two unguarded fallback branches exist lower in the same function that return identical developer-level APR ranges to any caller without verifying isDev. This creates two independent exploit paths through a single root cause:

Path 1 (duration = 5): isNormalDuration(5) returns true, so any user can submit duration=5 to stake(). The isDev && durationInMinutes == 5 guard is bypassed for non-owners, and execution falls through to the unguarded else if (durationInMinutes == 5) fallback which returns (5000, 10000). This path is exploitable at any time with no preconditions.

Path 2 (duration = 6): isPromoDuration(6) returns true, so any user can submit duration=6 to stake() during an active promotion. The isDev && durationInMinutes == 6 guard is bypassed for non-owners, and execution falls through to the unguarded else if (durationInMinutes == 6) branch which returns (10000, 10000). This path is exploitable whenever a promotion is active — a routine protocol state.

In both paths, the resulting maxAPR flows directly into stake() where effectiveAPR = maxAPR * 1000, producing an effective APR of up to 10,000,000 (10,000%). The attacker needs only MINIMUM_STAKE (10 USDT) to open a position and can repeat every 5–6 minutes, compounding against the hot wallet.

Affected Code

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L658
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L672

Impact

Any user can open stakes at 5,000%–10,000% APR against the treasury hot wallet with zero special access. Path 1 is permanently available and requires no admin action. Path 2 is available during every promotional campaign, which is a routine protocol operation. Repeating the cycle every 5–6 minutes allows an attacker to drain the entire hot wallet balance within hours. The minimum capital required is 10 USDT.

Remediation

It is recommended to remove the unguarded else if (durationInMinutes == 5) and == 6 fallback branches inside _getAPRRange() that sit below the normal duration logic. These branches duplicate developer-only APR rates without access control, and their removal eliminates the public exploit path entirely.

Retest

This issue has been fixed by removing developer-only APR rates.

Client Comment :- We agree with this finding. The affected short-duration branches were included solely as internal technical testing paths for privileged operator use and were never intended to be exposed as public staking options. We have updated the duration-validation and APR-selection flow so these developer-only paths are no longer reachable by general users. Access is now explicitly restricted to privileged internal accounts only, removing the unintended public exposure while preserving controlled internal testing capability.

Bug ID #H001 [Fixed]

Attacker Can Bypass All Unstake Penalties By Channeling Funds Through Auto-Renewed Stakes

Vulnerability Type
Insecure Use of Storage (SCWE-044)

Severity
High

Description
The BSCSecureStakingV5 contract enforces early-unstake penalties inside unstake() using the condition block.timestamp < st.stakeEnd && st.stakingType < 2. When a user calls BSCSecureStakingV5.claimRewards() after stake maturity, the contract auto-renews the stake for an identical term and increments stakingType (e.g., from 1 to 2). Once stakingType >= 2, the penalty conditional in unstake() never evaluates to true. An attacker stakes the minimum amount, waits for maturity, triggers auto-renewal via claimRewards() to bump stakingType to 2, then injects massive capital via BSCSecureStakingV5.addFund(). The root cause is that addFund() resets stakedAt and stakeEnd but never resets stakingType back to 1 — despite the partial unstake path correctly resetting it, demonstrating awareness of this requirement. This omission allows the attacker to immediately call unstake() on the full balance with zero penalty, amplifying the attack from the original minimum stake to arbitrarily large capital.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1495

Proof of Concept
1. Alice calls BSCSecureStakingV5.stake(USDT, 43200, 10e18) with the minimum 10 USDT for a 30-day term (stakingType = 1).
2. After 30 days, Alice calls BSCSecureStakingV5.claimRewards(), triggering auto-renewal and setting stakingType = 2.
3. Alice calls BSCSecureStakingV5.addFund() with 100,000 USDT. The function resets stakedAt and stakeEnd but leaves stakingType = 2.
4. Alice immediately calls BSCSecureStakingV5.unstake() for the full 100,010 USDT. Because stakingType >= 2, the penalty check is skipped entirely.
5. Alice withdraws 100,010 USDT plus accrued rewards with zero penalty.

Impact

Attackers can farm maximum platform APYs over intraday horizons with arbitrarily large capital while completely circumventing the protocol's lockup penalty design. The treasury loses the economic guarantee that early withdrawers forfeit a portion of their principal, destroying liquidity stability.

Remediation

It is recommended to reset stakingType to 1 inside addFund() after resetting the staking timestamps. This forces freshly injected funds to restart the penalty evaluation cycle, preventing the auto-renew shell from shielding new capital.

Retest

This issue has been fixed by resetting stakingType to 1 inside addFund() after resetting the staking timestamps.

Client Comment :- We agree with this finding. The issue arose because addFund() refreshed the staking timestamps for renewed positions without restoring the stake to a penalty-bearing state. We have remediated this by resetting the relevant stake type during the addFund() flow so renewed stakes that receive additional funds are again subject to the intended early-unstake penalty model. This aligns the addFund() behavior with the protocol's intended treatment of refreshed stake positions.

Bug ID #H002 [Acknowledged]

Partial Unstake Unconditionally Resets APR And Lock Period Without Version Gate, Silently Reducing Yield For Threshold Stakers

Vulnerability Type
Business Logic (SCWE-083)

Severity
High

Description
Description When BSCSecureStakingV5.unstake() processes a partial withdrawal, the remaining principal always has its lock period and APR reset unconditionally via _refreshStakeAfterUpdate(). Two issues stem from this:

First, there is no version gate. BSCSecureStakingV5.claimRewards() auto-renewal explicitly checks stakeInfo.version >= 12 before triggering any stake refresh, protecting legacy migrated stakes (v10/v11) from having their original terms overwritten. The partial unstake path applies _refreshStakeAfterUpdate() to any stake regardless of version. For a legacy stake whose stored duration does not map to a valid entry in _getAPRRange(), the function reverts with Invalidtimerange(), permanently blocking partial unstake for that position. The user is forced into a full unstake and faces the early-exit penalty on their entire principal.

Second, APR is recalculated from the reduced remaining amount. The APR tier in this protocol is amount-dependent: stakes at or above THRESHOLD_AMOUNT earn maxAPR, while amounts below are interpolated. A user who originally staked at or above threshold to earn maxAPR will have their APR silently recalculated at the lower interpolated rate after a partial withdrawal that drops the remaining balance below threshold. The new lower APR is then locked in for a fresh full-duration cycle.

Affected Code

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1605
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1606

Impacts

A single partial unstake triggers a cascade of losses: early-exit penalty on the withdrawn amount, full lock period reset erasing all accrued time, and APR reduction if the remaining balance falls below threshold. If the user needs liquidity again before the new lock expires, they face a second early-exit penalty at the already-reduced APR. One partial withdrawal results in double penalties, lost lock progress, and reduced yield for the entire new cycle — compounding losses the user cannot anticipate or recover from.

Remediation

It is recommended to add a stakeInfo.version >= 12 check to the partial unstake refresh path, consistent with the gate already applied in claimRewards() auto-renewal. For the APR drop, consider preserving the original stakeInfo.APR for the refreshed cycle when the remaining amount drops below threshold due to the withdrawal, rather than recalculating from the new lower balance.

Retest

Client Comment :- We do not agree that this behavior should be classified as a vulnerability. In the intended protocol design, a partial unstake does not preserve the original economics for the remaining balance indefinitely. Instead, the remaining principal is intentionally refreshed into a new stake cycle, and APR is recalculated according to the protocol's standard duration- and amount-based pricing model. Because the APR structure is progressive and balance-sensitive up to the threshold amount, a lower remaining balance may qualify for a lower APR after partial withdrawal. This is expected protocol behavior and reflects the business rule that users earn yield based on the amount actually left staked.
Additionally, the report's statement regarding the absence of a version gate is not aligned with the current implementation, as the partial-unstake path already includes version-based handling for older positions.

Bug ID #M001 [Fixed]

Missing systemNotDisabled Modifier on addFund()

Vulnerability Type
Access Control (SCWE-016)

Severity
Medium

Description
The contract defines a systemNotDisabled modifier to block user-facing functions when the protocol is under maintenance. This modifier is applied to claimRewards() and unstake() , and stake() enforces an equivalent inline check. However, addFund() has neither the modifier nor any inline systemEnabled check exists anywhere in its body. As a result, when the owner disables the system via toggleSystemState(), users can still call addFund() to deposit additional tokens into an existing stake, trigger an unclaimed rewards payout from the hot wallet, and reset/recalculate their stake terms all while every other user-facing function is correctly blocked.

Affected Code

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1013

Impacts
When the system is disabled for maintenance, addFund() remains fully open. A user can still deposit tokens into an existing stake, receive unclaimed reward payouts transferred from the hot wallet, and have their stake duration and APR reset operations that the maintenance flag is explicitly designed to freeze. This undermines the protocol owner's ability to pause activity cleanly, and could result in unintended fund movements from the hot wallet during a maintenance or incident-response window.

Remediation
Add the systemNotDisabled modifier to addFund()'s function signature, consistent with claimRewards() and unstake().

Retest
This issue has been fixed by adding the systemNotDisabled modifier to addFund().

Client Comment :- We agree with this finding. The maintenance-disable control was already

enforced across the other user-facing flows, and addFund() should have respected the same operational restriction. We have updated the function so it now follows the same system-disable behavior as the rest of the user interaction surface, ensuring consistent emergency and maintenance handling across the staking lifecycle.

Bug ID #M002 [Acknowledged]

Stakers Lose Yield Between Maturity And Late Claim Due To Timestamp Reset

Vulnerability Type
Business Logic (SCWE-083)

Severity
Medium

Description
When a user calls BSCSecureStakingV5.claimRewards() after stake maturity, the auto-renewal logic sets the new st.stakedAt to block.timestamp — the current block time. Yield calculations for the new cycle begin from the claim timestamp, not from the original st.stakeEnd. If a user claims days or weeks after maturity, the gap between st.stakeEnd and the actual claim time produces zero yield. The user forfeits all potential earnings for the uncovered interval.

Affected Code

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1328

Proof of Concept

1. Alice's stake matures on March 1 (st.stakeEnd = March 1).
2. Alice calls BSCSecureStakingV5.claimRewards() on March 8.
3. The auto-renewal sets st.stakedAt = March 8 (block.timestamp), not March 1.
4. Alice earns zero yield for the 7-day gap between March 1 and March 8.

Impacts
Users who do not claim at the exact moment of maturity silently forfeit yield for the unclaimed gap period. The loss scales with the delay between maturity and claim days to weeks of yield on large positions can represent a meaningful financial loss. Users relying on auto-compounding strategies or off-chain claim bots are penalized for any network latency, gas conditions, or scheduling delays beyond their control.

Remediation
It is recommended to set the new st.stakedAt to the previous st.stakeEnd when auto-renewing non-promo stakes, chaining renewal intervals without gaps and ensuring no yield is silently lost between cycles.

Retest

Client Comment :- We acknowledge the observation but consider this an intentional product and economic design choice rather than a security issue. In the current staking model, when a user renews late after maturity, the new reward cycle begins from the actual renewal timestamp rather than retroactively covering the idle period between maturity and user action. This is consistent with the protocol's intended behavior: rewards accrue on actively staked capital during active staking periods, not during inactive user delay between cycles.

Bug ID #M003 [Fixed]

Owner Cannot Pause Contract During Emergency

Vulnerability Type
Missing functionality (SCWE-006)

Severity
Medium

Description
The contract inherits from `PausableUpgradeable`, which is designed to allow privileged actors to halt critical functionality during emergencies using internal functions such as `_pause` and `_unpause`. Core user-facing functions like `stake`, `addFund`, `claimRewards`, and `unstake` are protected using the `whenNotPaused` modifier, meaning their execution depends on the paused state.

However, the contract does not expose any external or public function that calls `_pause` or `_unpause`. As a result, although the pause mechanism is integrated at the modifier level, there is no way for the owner or any privileged role to actually trigger the paused state. The root cause is the absence of externally callable administrative functions to control the pause lifecycle.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L47

Impacts
In the event of a critical vulnerability, exploit, or abnormal system behavior, the owner cannot halt contract operations.

Remediation
Add owner-restricted external functions that invoke `_pause` and `_unpause` to enable emergency control over contract operations.

Retest
This issue has been fixed by adding owner-restricted external functions that invoke `_pause` and `_unpause`.

Client Comment :- We agree with the recommendation to make the emergency pause lifecycle explicitly owner-accessible. Although the contract already included a separate owner-controlled system-enable/system-disable mechanism for operational maintenance control, we have now also exposed owner-restricted external pause() and unpause() functions to activate the inherited PausableUpgradeable controls directly. This provides a clearer and more complete emergency-administration surface.

Bug ID #L001 [Fixed]

Owner Can Silently Brick Existing Promo Stakes By Starting A New Promotion

Vulnerability Type
Insecure Use of Storage (SCWE-044)

Severity
Low

Description
The BSCSecureStakingV5.startPromotion() function has no guard against being called while a promotion is already active. Re-calling it resets promoEndTime and increments currentPromoId. Existing promo stakes store the old promoId in stakeInfo.promoId. The addFund() function checks stakeInfo.promoId != currentPromoId and rejects fund additions when the IDs mismatch. When the owner starts a new promotion during an active one, all users with promo stakes from the prior promotion are silently locked out of addFund().

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L338

Proof of Concept
1. The owner calls BSCSecureStakingV5.startPromotion(). currentPromoId = 1.
2. Alice stakes during the promotion. stakeInfo.promoId = 1.
3. The owner calls BSCSecureStakingV5.startPromotion() again. currentPromoId = 2.
4. Alice calls BSCSecureStakingV5.addFund(). The check stakeInfo.promoId != currentPromoId (1 != 2) evaluates to true, and the call reverts.
5. Alice's promo stake is locked out of fund additions despite her original promotion window still being valid.

Impact
Legitimate promo stakers lose access to addFund() without warning. Protocol administrators can inadvertently brick active user positions by starting a new promotional campaign. Affected users have no recourse except waiting for maturity and unstaking.

Remediation
It is recommended to add a guard preventing startPromotion() from being called during an active promotion period.

Retest
This issue has been fixed by adding required check.

Client Comment :- We agree with this finding. The promotion lifecycle has been updated so a new promotion cannot be started while an existing promotion is still active. This prevents active promotional stake records from being invalidated by an overlapping promotion cycle and preserves expected behavior for users who entered under the earlier promotion state.

Bug ID #L002 [Fixed]

Owner Can Permanently Brick Protocol Via Single-Step Ownership Transfer

Vulnerability Type
Single-Step Ownership Transfer (SCWE-139)

Severity
Low

Description
The BSCSecureStakingV5 contract uses a single-transaction transferOwnership pattern for governance control. If the owner mistypes the new address or signs the transaction under a compromised key, ownership transfers irreversibly to an inaccessible or malicious address in a single block. All administrative functions — pausing, hot wallet routing, promotion management, admin configuration — become permanently inaccessible. No recovery mechanism exists.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L46

Impact
A single typo or key compromise permanently bricks all administrative access to the protocol. Hot wallet rotation, system pausing, promotion management, and admin changes all become impossible without a full UUPS proxy upgrade initiated from outside the contract's governance surface.

Remediation
It is recommended to adopt OpenZeppelin's Ownable2StepUpgradeable, which requires the new owner to explicitly accept ownership before the transfer completes.

Retest
This issue has been fixed.

Client Comment :- We agree with the governance-hardening objective behind this finding. Because the contract is upgradeable, replacing the ownership base contract directly with Ownable2StepUpgradeable was not the safest in-place remediation path for the live storage layout. Instead, we implemented a storage-safe custom two-step ownership transfer flow within the current architecture. Ownership transfers now require nomination followed by explicit acceptance

by the pending owner, providing the intended administrative protection without introducing unsafe storage-layout changes.

Bug ID #L003 [Fixed]

Administrators Cannot Rotate Hot Wallet Keys Without Full Proxy Upgrade

Vulnerability Type
Single EOA Admin Without Rotation (SCWE-129)

Severity
Low

Description
The upsertOrRenameToken() function that previously managed token registry and hot wallet addresses has been deprecated in V5. If the treasury team needs to rotate hot wallet keys due to standard security procedures or a key compromise, administrators have no direct method to update the tokenHotWallets mapping without initiating a full UUPS proxy upgrade.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L44

Impact
Hot wallet key rotation— a routine treasury security operation — requires deploying an entirely new implementation contract and executing a proxy upgrade. This delays incident response during a key compromise and adds unnecessary operational risk to a standard security procedure.

Remediation
It is recommended to add a dedicated hot wallet rotation function with appropriate access controls, separate from the deprecated token registry logic.

Retest
This issue has been fixed by adding updateTokenHotWallet().

Client Comment :- We agree with this finding. We have added a direct owner-controlled hot-wallet rotation function so the operational wallet for a supported token can be updated without requiring a full proxy upgrade. This improves treasury-operational flexibility and reduces administrative friction in normal key-rotation or emergency key-replacement scenarios.

Bug ID #L004 [Fixed]

Users Lose Gas On Silently Failed Core Operations Due To Missing Reverts

Vulnerability Type
Uncaught Exceptions (SCWE-004)

Severity
Low

Description
The BSCSecureStakingV5.addFund(), BSCSecureStakingV5.claimRewards(), and BSCSecureStakingV5.unstake() functions query required liquidity from the static tokenHotWallets address. When the hot wallet balance or ERC20 allowance is insufficient to cover the payout, the contract emits a Notification event with a "Network congestion" message and returns early via return; instead of reverting. Because the transaction succeeds on-chain with valid gas usage, block explorers, indexers, frontend integrations, and hardware wallets all display a successful transaction. The user's state change (e.g., reward claim, unstake) never executes, but every external system reports success.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1109
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1257
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1466
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1540
- https://testnet.bscscan.com/address/0x31FA35eaE6cd5C1E56Aa00fF3df2cB6080d655e4#code#F1#L1164

Proof of Concept
1. The protocol's hot wallet for USDT has 0 balance remaining.
2. Alice calls BSCSecureStakingV5.claimRewards() to collect 500 USDT in matured rewards.
3. The function checks the hot wallet balance, finds it insufficient, emits Notification("Network congestion"), and returns silently.
4. Alice's wallet shows a successful transaction. Her rewards remain unclaimed, her gas is spent, and she has no indication the operation failed.

Impact
Users repeatedly waste gas on operations that silently fail. Frontend dashboards show stale state because indexers record the no-op transaction as successful. During periods of hot wallet underfunding, every user interaction with the protocol becomes a deceptive no-op, eroding trust and causing material gas losses across the user base.

Remediation
It is recommended to replace the emit Notification(...); return; pattern with revert statements across all payout paths. This ensures failed operations produce explicit on-chain failures that wallets, indexers, and frontends correctly interpret as reverted.

Retest
This issue has been fixed by reverting instead of silent returning.

Client Comment :- We agree with this finding. The affected payout paths previously returned early after emitting a notification when the configured hot wallet lacked sufficient balance or allowance, which could create misleading successful transaction outcomes from the user's perspective. We have updated these flows to revert explicitly instead. This ensures the transaction outcome is unambiguous on-chain and across explorers, wallets, and frontend integrations.

Bug ID #L005 [Fixed]

Full Unstake Leaves Residual Rewards Phantom-Locked In Global Ungranted Counters

Vulnerability Type
State Inconsistency (SCWE-075)

Severity
Low

Description
When BSCSecureStakingV5.unstake() processes a full withdrawal, it pays out the time-accrued claimable rewards and correctly decrements totalUngrantedRewards by that amount. However, st.rewards still holds the residual unearned future rewards for the remaining lock period. The full unstake path marks the stake as unstaked = true and claimed = true but never subtracts this residual from totalUngrantedRewards or _tokenAgg[token].totalUngrantedRewards. The partial unstake path handles this correctly via _refreshStakeAfterUpdate and a delta calculation, confirming the requirement was understood. The full unstake path has no equivalent cleanup.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1565

Impacts
totalUngrantedRewards and _tokenAgg[token].totalUngrantedRewards inflate permanently with every early full unstake. Operators see phantom reward obligations that can never be claimed or cleared. If these aggregates are used to determine required hot wallet funding levels, the protocol continuously over-provisions liquidity against liabilities that no longer exist.

Remediation
It is recommended to subtract the residual st.rewards from both totalUngrantedRewards and _tokenAgg[token].totalUngrantedRewards inside the full unstake block, after the reward claim step and before marking the stake as unstaked.

Retest
This issue has been fixed.

Client Comment :- We agree with this finding. The full-unstake flow was updated so any residual

future reward amount remaining on the stake record is properly removed from both the global and per-token ungranted reward aggregates before the stake is finalized. This keeps the aggregate accounting aligned with the actual remaining protocol liability and prevents stale reward obligations from persisting after a position has been fully closed.

Bug ID #L006 [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://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#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 0.8.34 pragma version
Reference: https://scs.owasp.org/SCWE/SCSVS-CODE/SCWE-060/

Retest
This issue has been fixed.

Client Comment :- We agree with the general recommendation to avoid floating pragmas. The contract has been updated to use a fixed compiler pragma, ensuring deterministic compilation

behavior and improving audit reproducibility. We have pinned the contract to the compiler version used in the project's actual build pipeline rather than adopting a newer arbitrary version without full toolchain migration.

Bug ID #I001 [Fixed]

Claim Receipt Hash Collision Risk For Same-Block Transactions

Vulnerability Type
Data Integrity (SCWE-074)

Severity
Informational

Description
The claimedTxnHash computed in BSCSecureStakingV5.claimRewards(), BSCSecureStakingV5.addFund(), and BSCSecureStakingV5.unstake() is derived as keccak256(abi.encodePacked(msg.sender, block.timestamp, amount, token)). If the same user submits two transactions for the same token that produce an identical claimable amount within the same block, both generate the same hash. The second write to claimedRewardsByHash[hash] silently overwrites the first, corrupting the per-claim lookup record.

Affected Code
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1299

Impacts
Actual token transfers and all reward accounting state are unaffected. Only the claimedRewardsByHash lookup is corrupted, causing off-chain tracking and claim history queries to return incorrect data for the overwritten entry.

Remediation
It is recommended to include the stake's txnHash in the claimedTxnHash computation, since it is unique per position and ensures no two claim receipts can collide regardless of other parameters.

Retest
This issue has been fixed.

Client Comment :- We agree with this informational finding. The receipt-hash generation logic has been updated so the claim receipt key now incorporates the stake's unique transaction hash. This makes receipt identifiers position-specific and prevents same-block collisions across otherwise similar claim scenarios, improving the integrity of the off-chain lookup and claim-history surface without changing reward accounting behavior.

Bug ID #G001 [Fixed]

Public constants can be private

Vulnerability Type
Gas Optimization (SCWE-082)

Severity
Gas

Description
Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Affected Code

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L58
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L59
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L60
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L61
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L62
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L63
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L64
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L67
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L68
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L69
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L70

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L71

Impacts
Public constants are more costly due to the default getter functions created for them, increasing the overall gas cost.

Remediation
If reading the values for the constants is not necessary, consider changing the public visibility to private.

Retest
This issue has been fixed.

Client Comment :- We acknowledge this as a gas and bytecode optimization rather than a security issue. We adopted this optimization because it provides a meaningful reduction in deployed bytecode size while not affecting the current frontend integration flow. The affected constants are no longer exposed publicly where those getters were not operationally required.

Bug ID #G002 [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://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L697
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1085
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1436

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
This issue has been fixed.

Client Comment :- We acknowledge this as a gas optimization rather than a security issue. The cited unsigned integer checks have been normalized to the cheaper != 0 style at the relevant sites. This change does not affect storage, ABI, user behavior, or frontend integration.

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://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L356
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L501
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L767
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1822
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1880
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1883
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1891
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1921
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1939
- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1940

- https://testnet.bscscan.com/address/0xf7edfbf075b48daa47b3405c190b126fb6f02dcd#code#F1#L1950

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
This issue has been fixed.

Client Comment :- We acknowledge this as a gas optimization rather than a security issue. The cited loop increments have been updated to use pre-increment form. This is a micro-optimization only and does not change protocol behavior or external integration semantics.

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

AICPA SOC
aicpa.org/soc4so

SCS Advocates
OWASP
OWASP Smart Contract Top 10
Security Risks & Vulnerabilities

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com