Audited by CRED SHIELDS

CredShields

Smart Contract Audit

November 7, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of LERN360 between October 8th, 2025, and October 31st, 2025. A retest was performed on November 7th, 2025.

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
LERN360

Table of Contents

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

LERN360 engaged CredShields to perform a smart contract audit from October 8th, 2025, to October 31st, 2025. During this timeframe, 33 vulnerabilities were identified. All the 4 contracts that we audited are related to the staking and vesting smart contracts that are required to give effect to LERN360's Tokenomics.

During the audit, 16 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "LERN360" 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
ICO Contracts	6	10	8	5	2	2	**33**

Table: Vulnerabilities Per Asset in Scope

A retest was performed on November 7th, 2025, and all the vulnerabilities/bugs have been satisfactorily addressed.

The CredShields team conducted the security audit to focus on identifying vulnerabilities in the ICO Contracts' scope during the testing window while abiding by the policies set forth by LERN360's team.

Investor Summary

Following a thorough audit and successful remediation of all critical and high-severity findings, CredShields is satisfied that the smart contracts presents no known security vulnerabilities as of the latest retest on November 7th, 2025.

Based on industry-standard security assessment methodologies and manual code review, we are confident that the contracts are secure for deployment, assuming no changes are made post-audit.

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 LERN360'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 LERN360 can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, LERN360 can future-proof its security posture and protect its assets.

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

LERN360 engaged CredShields to perform the ICO 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 October 8th, 2025, to October 31st, 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
The following commits were in scope: • https://github.com/Lern360/ico-smart-contract/tree/96f21c0d66c477eee754971cf218b3f02e94b5c0/ • https://github.com/Lern360/ico-smart-contract/tree/de40a2292de127890939af03ce4626ed4066e25d/ • https://github.com/Lern360/ico-smart-contract/tree/93f923c013b3001677322eefa199cc5037d6827d/ **From the commits above, the following contracts were in scope:** • **Staking.sol** • **MainStaking.sol**

- **StakingVault.sol**
- **Vesting.sol**
- **Timelock.sol**

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

LERN360 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, 33 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SCWE | Vulnerability Type
User cannot claim from any vault if one vault has no stake	Critical	Business Logic (SC03-LogicErrors)
Inactive vaults can receive rewards and lock user funds	Critical	Lack of Input Validation (SC04-Lack Of Input Validation)
User cannot claim Merkle rewards due to a mismatched recipient check	Critical	Business Logic (SC03-LogicErrors)
Malicious User Can Drain Rewards When useAPYSystem is True	Critical	Fund Loss
Reward Fund Flow Causing Permanent Token Lock in Staking Vaults	Critical	Business Logic (SC03-LogicErrors)
The user can claim rewards repeatedly for the same block	Critical	Inconsistent Accounting (SCWE-010)
Reward Cooldown Bypass via Auto-Claim in _stake and withdrawStake Function	High	Business Logic (SC03-LogicErrors)

Staker can freeze time-weight and overearn when APY is re-enabled	High	Business Logic (SC03-LogicErrors)
User can Stake, Claim, and Unstake instantly to get free rewards	High	Business Logic (SC03-LogicErrors)
User Can Replay Signature To Bypass Authorization	High	Signature Replay Attacks (SCWE-055)
EIP-7702 EOAs Can Stake But Cannot Withdraw Or Claim	High	Access Control (SCWE-016)
Inconsistent userLastClaimTime Update and Unconditional canUserClaim Check in Non-APY Mode	High	Business Logic (SC03-LogicErrors)
Misuse of canUserClaimAfterStake and Missing Cooldown Enforcement Leading to Potential Reward Bypass and Staking Restrictions	High	Reward Distribution Vulnerability
User Can Permanently Lose Unpaid Rewards During Unstake	High	Inconsistent Accounting (SCWE-010)
Early User Can Drain Later User Stake Balance	High	Inconsistent Accounting (SCWE-010)
Reward Debt Updated Incorrectly Under APY System	High	Business Logic (SC03-LogicErrors)
First vault consistently receives the remainder, causing unfair distribution	Medium	Business Logic (SC03-LogicErrors)
Vault pause/unpause functions inaccessible due to factory-admin role assignment	Medium	Access Control (SCWE-016)
Inconsistent Reward State Due to Missing TotalRewards Deduction on Claims	Medium	Inconsistent Accounting (SCWE-010)
Manual Rewards Could Be Lost on Emergency Withdraw	Medium	Business Logic (SC03-LogicErrors)
Unclaimed Rewards Lost on Unstake Due to Missing Claim Enforcement	Medium	Business Logic (SC03-LogicErrors)

Timelock cannot receive vested tokens due to missing vesting integration	Medium	Missing functionality (SCWE-006)
User can reuse one signature to call many actions	Medium	Signature Replay Attacks (SCWE-055)
Inconsistent Reward Payment Logic Between Unstake and Claim Functions	Medium	Business Logic (SC03-LogicErrors)
Vesting Can Be Created With Cliff Greater Than Duration	Low	Lack of Input Validation (SC04-Lack Of Input Validation)
Floating and Outdated Pragma	Low	Floating Pragma (SCWE-060)
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing best practices
Missing Upper Bound on APY Rate Allows Arbitrary Reward Inflation	Low	Unchecked Parameter
Admin Can Grant Vesting Role To Arbitrary Address	Low	Business Logic (SC03-LogicErrors)
Dead Code	Informational	Code With No Effects (SCWE-062)
apyPercentage Not Reset When Disabling APY System	Informational	Inconsistent Accounting (SCWE-010)
Gas Optimization in Require/Revert Statements	Gas	Gas Optimization (SCWE-082)
Splitting Require/Revert Statements	Gas	Gas Optimization (SCWE-082)

Table: Findings in Smart Contracts

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

LERN360 is actively partnering with CredShields on this engagement to validate the remediations for the discovered vulnerabilities. **A retest was performed on November 7th, 2025, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
User cannot claim from any vault if one vault has no stake	Critical	Fixed [Nov 7, 2025]
Inactive vaults can receive rewards and lock user funds	Critical	Fixed [Nov 7, 2025]
User cannot claim Merkle rewards due to a mismatched recipient check	Critical	Fixed [Nov 7, 2025]
Malicious User Can Drain Rewards When useAPYSystem is True	Critical	Fixed [Nov 7, 2025]
Reward Fund Flow Causing Permanent Token Lock in Staking Vaults	Critical	Fixed [Nov 7, 2025]
The user can claim rewards repeatedly for the same block	Critical	Fixed [Nov 7, 2025]
Reward Cooldown Bypass via Auto-Claim in _stake and withdrawStake Function	High	Fixed [Nov 7, 2025]
Staker can freeze time-weight and overearn when APY is re-enabled	High	Fixed [Nov 7, 2025]
User can Stake, Claim, and Unstake instantly to get free rewards	High	Fixed [Nov 7, 2025]
User Can Replay Signature To Bypass Authorization	High	Acknowledged [Nov 7, 2025]
EIP-7702 EOAs Can Stake But Cannot Withdraw Or Claim	High	Acknowledged [Nov 7, 2025]

Inconsistent userLastClaimTime Update and Unconditional canUserClaim Check in Non-APY Mode	High	Acknowledged [Nov 7, 2025]
Misuse of canUserClaimAfterStake and Missing Cooldown Enforcement Leading to Potential Reward Bypass and Staking Restrictions	High	Acknowledged [Nov 7, 2025]
User Can Permanently Lose Unpaid Rewards During Unstake	High	Fixed [Nov 7, 2025]
Early User Can Drain Later User Stake Balance	High	Fixed [Nov 7, 2025]
Reward Debt Updated Incorrectly Under APY System	High	Fixed [Nov 7, 2025]
First vault consistently receives the remainder, causing unfair distribution	Medium	Acknowledged [Nov 7, 2025]
Vault pause/unpause functions inaccessible due to factory-admin role assignment	Medium	Fixed [Nov 7, 2025]
Inconsistent Reward State Due to Missing TotalRewards Deduction on Claims	Medium	Fixed [Nov 7, 2025]
Manual Rewards Could Be Lost on Emergency Withdraw	Medium	Fixed [Nov 7, 2025]
Unclaimed Rewards Lost on Unstake Due to Missing Claim Enforcement	Medium	Acknowledged [Nov 7, 2025]
Timelock cannot receive vested tokens due to missing vesting integration	Medium	Fixed [Nov 7, 2025]
User can reuse one signature to call many actions	Medium	Acknowledged [Nov 7, 2025]
Inconsistent Reward Payment Logic Between Unstake and Claim Functions	Medium	Fixed [Nov 7, 2025]
Vesting Can Be Created With Cliff Greater Than Duration	Low	Fixed [Nov 7, 2025]
Floating and Outdated Pragma	Low	Fixed [Nov 7, 2025]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Fixed [Nov 7, 2025]
Missing Upper Bound on APY Rate Allows Arbitrary Reward Inflation	Low	Fixed [Nov 7, 2025]
Admin Can Grant Vesting Role To Arbitrary Address	Low	Fixed [Nov 7, 2025]
Dead Code	Informational	Fixed [Nov 7, 2025]
apyPercentage Not Reset When Disabling APY System	Informational	Fixed [Nov 7, 2025]
Gas Optimization in Require/Revert Statements	Gas	Fixed [Nov 7, 2025]
Splitting Require/Revert Statements	Gas	Fixed [Nov 7, 2025]

Table: Summary of findings and status of remediation

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

Bug ID #C001 [Fixed]

User cannot claim from any vault if one vault has no stake

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Critical

Description
claimAllRewards() loops through every vault and calls claimAPYRewards(user, user). Inside each vault, claimAPYRewards enforces require(userStake.amount > 0 && userStake.active, "No active stake"). If the user has no stake in any single vault, that call reverts, which bubbles up and reverts the entire claimAllRewards() transaction. As a result, rewards successfully claimable from other vaults are also lost in the reverted batch. This makes claiming brittle and forces users to maintain at least a minimal position in every vault to avoid global failure.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/MainStaking.sol#L314-L336
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L243-L262

Impacts
Users are blocked from receiving rewards from eligible vaults whenever any one vault has zero stake.

Remediation
It is suggested to make claimAllRewards() resilient to vaults with zero stake. Change vault behavior to return 0 instead of reverting when userStake.amount == 0

Retest
This vulnerability has been fixed by removing the require check from the claimAPYRewards() function.

Bug ID #C002 [Fixed]

Inactive vaults can receive rewards and lock user funds

Vulnerability Type
Lack of Input Validation (SC04-Lack Of Input Validation)

Severity
Critical

Description
distributeRewards() withdraws tokens from the timelock and loops over vaultNames to send each corresponding vaultAddress an equal share, then calls vault.addRewards(vaultAmount). The code assumes vaults in vaultNames are active, but it never checks an "active" flag or queries the vault's status before transferring. If a vault is paused, deprecated, or otherwise inactive, rewards can be routed to it and become inaccessible for distribution to stakers as intended.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/MainStaking.sol#L181-L214

Impacts
Rewards can become stranded in inactive or misconfigured vaults, resulting in delayed or failed payouts to users, inconsistent accounting of distributed amounts, and increased manual recovery overhead.

Remediation
It is recommended to check if the vault is active before sending funds to vaults.

Retest
This issue is resolved by verifying whether the vault is active before sending rewards to it.

Bug ID #C003 [Fixed]

User cannot claim Merkle rewards due to a mismatched recipient check

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Critical

Description
claimMerkleRewards() verifies the user's Merkle proof and then pulls tokens from the timelock by calling timelock.withdrawToStaking(finalReward, msg.sender). Inside the timelock, withdrawToStaking() enforces require(to == stakingContract, "Invalid recipient");. Because claimMerkleRewards() passes msg.sender (the user) as to, the require check always fails and the transaction reverts. As a result, no user can successfully claim their Merkle rewards even with a valid proof.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/MainStaking.sol#L345-L395
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Timelock.sol#L81-L88

Impacts
All Merkle reward claims revert, blocking legitimate users from receiving rewards and effectively freezing the reward program.

Remediation
To fix this issue, it is recommended to pass the staking contract as the recipient and perform the user payout from the staking contract.

Retest
This issue has been fixed by sending rewards to Mainstaking and then to the user.

Bug ID #C004 [Fixed]

Malicious User Can Drain Rewards When useAPYSystem is True

Vulnerability Type
Fund Loss

Severity
Critical

Description
The withdrawStake(...) function allows users to withdraw staked tokens and claim accumulated rewards. When useAPYSystem is enabled, reward accrual depends on elapsed time since userLastClaimTime.

In the claimRewards(...) function, this timestamp is properly updated to block.timestamp after each successful claim to enforce the cooldown between reward withdrawals.

```
function claimRewards(...) external whenNotPaused nonReentrant notContract {
    //..

    // Update last claim time for cooldown
>>  userLastClaimTime[msg.sender] = block.timestamp;
    emit RewardsClaimedWithCooldown(msg.sender, pending, getNextClaimTime(msg.sender));

    //..
}
```

However, in the withdrawStake(...) function, this update is missing. As a result, a malicious user can call withdrawStake(0, signature) repeatedly within the same transaction to claim rewards multiple times without advancing userLastClaimTime. This effectively bypasses the intended cooldown mechanism.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/Staking.sol#L356-L392

Impacts

An attacker can repeatedly invoke withdrawStake(...) with zero amount and drain all available rewards without waiting for the cooldown period. Honest stakers experience accelerated pool depletion and reduced future rewards.

Remediation

To fix this issue update userLastClaimTime to the current block timestamp after rewards are claimed in withdrawStake and _stake to align behavior with claimRewards.

For eg:

```
+    userLastClaimTime[msg.sender] = block.timestamp;
```

Retest

This issue has been resolved by updating the userLastClaimTime in the relevant locations.

Bug ID #C005 [Fixed]

Reward Fund Flow Causing Permanent Token Lock in Staking Vaults

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Critical

Description
The distributeRewards function incorrectly transfers tokens to individual StakingVault contracts while the reward payment mechanism operates from the MainStaking contract's balance. This creates a critical mismatch where tokens sent to vaults become permanently inaccessible since vaults lack any mechanism to transfer funds back to MainStaking or use them for reward payments.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L282

Impacts
Permanent locking of reward tokens in vault contracts, inability to pay user rewards despite sufficient system funds, and eventual system failure when MainStaking exhausts its token balance while vaults hold undistributed rewards.

Remediation
Remove the token transfer to vaults in distributeRewards and only update reward tracking through addRewards calls. Keep all reward tokens in MainStaking contract where reward payments actually occur.

Retest
The distributeRewards function no longer transfers tokens to vault contracts. Rewards are now retained in MainStaking and only tracked via addRewards, preventing reward token locking.

Bug ID #C006 [Fixed]

The user can claim rewards repeatedly for the same block

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
Critical

Description
claimRewards() calls pendingRewards() to compute how much a user can claim. pendingRewards() correctly computes an *up-to-date* local value _accRewardsPerShare by simulating the per-block reward accrual since lastRewardBlock — but it does not write that updated value back to contract storage accRewardsPerShare. Later in claimRewards() the contract sets the staker's rewardDebt using the stored accRewardsPerShare. Because accRewardsPerShare in storage is stale, rewardDebt is updated with a smaller value than it should be. The next time the same user calls claimRewards(), pendingRewards() will simulate accrual again (including the same blocks) and compute a larger pending reward — effectively paying the user the same block rewards more than once (double-counting the un-applied accrual). This causes overpayment of rewards.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/Staking.sol#L467-L497

Impacts
Users can call claim repeatedly across blocks or coordinate to extract extra rewards. If exploited at scale, the protocol's token reserves could be drained.

Remediation
It is recommended to call updatePool() at the start of claimRewards() before computing pending, so the storage accRewardsPerShare and lastRewardBlock are up-to-date.

Retest
This issue has been fixed by adding the updatePool() function in claimRewards().

Bug ID #H001 [Fixed]

Reward Cooldown Bypass via Auto-Claim in _stake and withdrawStake Function

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
The Staking contract implements a cooldown mechanism for claiming rewards using claimRewards(...), which ensures users wait a specific time between consecutive claims. However, both _stake(...) and withdrawStake(...) functions automatically transfer pending rewards to users without enforcing the cooldown check or updating userLastClaimTime.

As a result, a malicious user can continuously stake or perform zero-amount withdrawals to claim rewards multiple times within the cooldown period. This effectively renders the cooldown logic meaningless and can lead to excessive reward distribution, bypassing emission rate controls and reward pacing mechanisms.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L322-L354
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L356-L391

Impacts
Users can bypass the cooldown system and claim rewards multiple times in a short period. By repeatedly staking small amounts or calling withdrawStake(0, sig), they can drain the reward pool faster than intended. This breaks the emission pacing, creates unfair reward distribution, and may cause token supply inflation or economic imbalance within the staking system.

Remediation
To fix this issue add cooldown validation and timestamp updates in all functions that transfer rewards. Before sending pending rewards in _stake() and withdrawStake(), check canUserClaim(user) and update userLastClaimTime[user] after a successful claim.

Retest

This vulnerability is fixed by updating `userLastClaimTime` in `_stake()`

Bug ID #H002 [Fixed]

Staker can freeze time-weight and overearn when APY is re-enabled

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
On stake, the contract always increments userWeightedTime[user] by amount * block.timestamp and userTotalStaked[user] by amount, regardless of useAPYSystem. On withdrawStake, those values are only decremented if useAPYSystem is true. If a user withdraws while useAPYSystem is false, the weight variables are not reduced, so they no longer match staker.amount. When APY mode is later turned back on, the user's average time (userWeightedTime / userTotalStaked) is artificially old because it still includes already-withdrawn size and time.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/Staking.sol#L370

Impacts
A user can intentionally withdraw during an APY-off window to "freeze" a large, old time-weight and, after APY is re-enabled, earn more rewards than their real stake and holding time justify. This dilutes honest stakers and can over-distribute rewards.

Remediation
Keep time-weighting in sync with actual stake regardless of the APY toggle. Remove the useAPYSystem guard on withdrawal so weights are always decremented proportionally, or gate both deposit and withdrawal symmetrically. If APY can be toggled at runtime, normalize state at toggle time so stale data cannot carry forward.

Retest
This issue has been fixed by removing useAPYSystem guard.

Bug ID #H003 [Fixed]

User can Stake, Claim, and Unstake instantly to get free rewards

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
The contract allows a user to call stake() and then immediately call claimAllRewards() without any delay. Since there is no cooldown or reward-debt update at the manager level, the user can receive rewards that were generated before they actually staked. After claiming, they can instantly call unstake() and walk away with free rewards.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/MainStaking.sol#L314-L336

Impacts
An attacker can repeat this process — stake, claim, unstake — to steal unearned rewards. This reduces the reward pool and unfairly affects honest stakers.

Remediation
Add a cooldown or "warm-up" period before a user can claim rewards after staking. For example, record lastStakeTime[user] and require some minimum time to pass before allowing claims. Also make sure each vault updates the user's reward debt to the latest state at the time of staking so no old rewards can be claimed.

Retest
This issue has been fixed by adding stakeWarmupPeriod.

Bug ID #H004 [Acknowledged]

User Can Replay Signature To Bypass Authorization

Vulnerability Type
Signature Replay Attacks (SCWE-055)

Severity
High

Description
The contract's withdrawStake() and claimRewards() functions use _verifySignature(msg.sender, signature) to authenticate user actions based on an off-chain signature produced by a backend signer. The verification process only signs and checks abi.encodePacked(user), meaning the same signature remains valid for any function call and any number of invocations.
As a result, a user can reuse (replay) the same signature to perform multiple withdrawals or reward claims without requiring a fresh signed message. The root cause is that the signed payload does not include context parameters—such as the specific function selector, withdrawal amount, or a nonce—and there is no tracking of whether a signature has already been used.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L356
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L393

Impacts
An attacker can reuse a single valid signature to repeatedly call withdrawStake and claimRewards, This completely breaks backend authorization guarantees.

Remediation
Include the function selector, parameters, and a unique nonce in the signed message, and maintain on-chain nonce tracking to prevent signature reuse.

Retest
This issue has not been fixed as it uses a nonce from a function parameter, but it is recommended to fix this issue as above remediation.

Client Comment

We will give the reward based on KYC done at the backend so that signature module will be modified at later stages. At the moment its completely paused till KYC functionality is implemented.

Bug ID #H005 [Acknowledged]

EIP-7702 EOAs Can Stake But Cannot Withdraw Or Claim

Vulnerability Type
Access Control (SCWE-016)

Severity
High

Description
The sale contract allows staking via allocateTokens(...) when tx.origin == msg.sender (no code-size check). The staking contract accepts stakes but uses a notContract modifier on withdraw/claim that blocks addresses with non-zero code (via extcodesize). Under Pectra / EIP-7702, EOAs may have code attached; such users can stake successfully through the sale contract but later be unable to withdraw or claim because they are blocked by notContract.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/Sale.sol#L439C5-L442C6
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/Staking.sol#L227C5-L239C6

Impacts
Affected users will be unable to withdraw principal or claim rewards; funds remain recorded but inaccessible, creating a denial-of-service against legitimate users and potential financial and legal risk for the platform.

Remediation
Align the caller-type validation across both contracts to ensure consistent behavior. Specifically, modify the onlyEOA modifier in the Sale contract to match the logic of the notContract modifier used in the Staking contract. This will prevent EIP-7702-enabled EOAs from being able to stake if they would later be blocked from withdrawing or claiming. Alternatively, remove both modifiers and replace them with consistent, authority-based access control (e.g., signature or role verification) for long-term compatibility.

Retest
The acknowledged points are as per team logic and not imapcting flow.
Note: This issue has not been fixed. It is recommended to fix this issue as per the above remediation.

Client Comment

Sale phase is already ended, so this point is of no use.

Bug ID #H006 [Acknowledged]

Inconsistent userLastClaimTime Update and Unconditional canUserClaim Check in Non-APY Mode

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
The staking contract contains an inconsistency in how it handles claim-related logic between APY-based and non-APY reward modes. Specifically, the issue arises from how the contract updates the userLastClaimTime variable and performs claim eligibility checks using canUserClaim().

In both the stake() and withdrawStake() functions, the contract updates userLastClaimTime only when the APY system is active. This behaviour ensures that the timestamp reflects the last claim time relevant to APY-based rewards. However, in the claimRewards() function, userLastClaimTime is updated unconditionally, regardless of whether useAPYSystem is true or false. As a result, even when the APY system is disabled and no APY rewards are being calculated or distributed, the user's userLastClaimTime is still updated to the current block timestamp. This behaviour can lead to desynchronization between the recorded claim time and the actual reward mechanism in use. Consequently, when the APY system is later re-enabled, users might face unexpected cooldown restrictions or inconsistencies in reward eligibility because their userLastClaimTime does not accurately represent their APY claim history.

A related inconsistency exists with the canUserClaim() checks. This function is designed to enforce cooldown for APY-based reward claims. In the stake() and withdrawStake() functions, these checks are executed only when the APY system is active. However, in the claimRewards() function, checks are performed unconditionally, even when the APY system is disabled. This leads to unintended claim restrictions, where users attempting to claim non-APY rewards are still subject to APY cooldown conditions.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/93f923c013b3001677322eefa199cc5037d6827d/smart_contracts/contracts/Staking.sol#L476

- https://github.com/Lern360/ico-smart-contract/blob/93f923c013b3001677322eefa199cc5037d6827d/smart—contracts/contracts/Staking.sol#L464

Impacts

When the APY reward system is disabled, users can still be blocked from claiming rewards because of the unconditional cooldown and warmup checks. Additionally, the unconditional update of userLastClaimTime introduces incorrect state tracking, which can interfere with future reward calculations once the APY mode is re-enabled.

Remediation

The claim logic should be made consistent across all functions and modes. Both the cooldown checks and the userLastClaimTime update should only occur when the APY system is active. When useAPYSystem is false, these operations should be skipped entirely to prevent unnecessary state changes and claim restrictions.

Retest

The acknowledged points are as per team logic and not imapcting flow.

Client comment: As stake will called only via sale contract and sale is already disabled. So we not added much validation regarding that as stake is already completed and no new stake will happen in this contract.

Bug ID #H007 [Acknowledged]

Misuse of canUserClaimAfterStake and Missing Cooldown Enforcement Leading to Potential Reward Bypass and Staking Restrictions

Vulnerability Type
Reward Distribution Vulnerability

Severity
High

Description
The VestingStakingManager contract contains multiple inconsistencies in the enforcement of the staking warm-up period, controlled via the canUserClaimAfterStake function. This function is intended to prevent users from claiming rewards immediately after staking, enforcing a stake-based cooldown. However, its current implementation causes unintended and inconsistent behavior across different functions.

Firstly, in the _stake function, canUserClaimAfterStake is incorrectly used to restrict additional staking. This means that a user cannot stake again until the cooldown period has passed, which is not the intended behavior. Users should always be able to stake, but the claiming of rewards should respect the cooldown.

Secondly, in the withdrawStake and emergencyWithdraw functions, canUserClaimAfterStake is not invoked at all. This allows users to bypass the cooldown period and claim their rewards immediately, which undermines the purpose of the stake warm-up period. Specifically, in emergencyWithdraw, the function transfers both the staked amount and the manual rewards without checking if the cooldown period has elapsed. While the staked amount withdrawal is acceptable, allowing manual rewards to be claimed during the cooldown violates the intended reward distribution rules.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/93f923c013b3001677322eefa199cc5037d6827d/smart_contracts/contracts/Staking.sol#L375

Impacts
The inconsistent cooldown enforcement can lead to several issues. Users may be prevented from staking additional tokens due to the improper restriction in _stake, reducing flexibility and

engagement. Conversely, users may withdraw or emergency-withdraw their stake and claim rewards immediately, bypassing intended cooldown restrictions.

Remediation

The contract should be updated to enforce cooldowns consistently across all reward-claiming pathways. In the _stake function, the call to canUserClaimAfterStake should be removed as a staking restriction; instead, it should be used only to determine whether pending rewards are distributed during staking. Pending rewards should only be auto-claimed if the cooldown period has passed, and otherwise deferred until the next eligible claim. In withdrawStake and emergencyWithdraw, reward claiming should respect canUserClaimAfterStake, such that rewards cannot be claimed if the cooldown period has not elapsed. Emergency withdrawals should allow withdrawal of the staked principal at any time but must skip manual rewards if the user is still within the cooldown period. This approach ensures that staking, reward claiming, and cooldown enforcement are consistent, secure, and aligned with the intended reward logic.

Retest

The acknowledged points are as per team logic and not imapcting flow.

Client comment: As stake will called only via sale contract and sale is already disabled. So we not added much validation regarding that as stake is already completed and no new stake will happen in this contract.

Bug ID #H008 [Fixed]

User Can Permanently Lose Unpaid Rewards During Unstake

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
High

Description
In the MainStaking.unstake function, a user's pending rewards are fetched using vault.getPendingRewards(msg.sender) before unstaking. The contract then attempts to transfer the full pending reward amount to the user. If MainStaking has insufficient token balance, it tries to withdraw from the timelock contract. However, when both sources lack sufficient tokens, the function transfers only the remaining available balance (mainStakingBalance) and does not record the unpaid portion.

As a result, the user's staking data inside the vault is updated as if the full reward was paid, even though only a partial amount was actually transferred. The unpaid rewards (the delta between expected and actual payment) are permanently lost from the user's perspective, since there is no mechanism to track or repay the deficit later. The root cause is the absence of state accounting for partially paid rewards when the contract's liquidity is insufficient.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L196

Impacts
If MainStaking and Timelock lack enough reward tokens, users who unstake may receive only a fraction of their earned rewards. The system loses track of the unpaid portion, leading to permanent reward loss and inaccurate accounting between users and the protocol.

Remediation
Record any unpaid reward amounts for each user and ensure these deferred rewards are settled once sufficient tokens are available.

Retest

This issue has been fixed by adding the required check to revert in case of the partially payments of the rewards.

Clients Comment :- We have implemented these checks as admin will always prefund the timelock/staking contract.

Bug ID #H009 [Fixed]

Early User Can Drain Later User Stake Balance

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
High

Description
In the current design of MainStaking, user rewards and principal stakes are both paid directly from the contract's ERC20 token balance. The unstake and claimRewards functions rely on whatever tokens exist in MainStaking or its timelock to process transfers. However, the contract itself does not isolate user deposits from the pool of distributable rewards.

This means that when the contract runs low on reward tokens—before the admin explicitly replenishes the reward pool—subsequent unstake or reward claims may end up transferring tokens that belong to other users' staked balances. The underlying accounting in IStakingVault continues to assume that each user's principal is fully backed, but in reality, tokens are being drained from the shared balance.

The root cause is that both staking principal and rewards share the same balance source without separation or guaranteed coverage, allowing principal tokens to be unintentionally spent as reward liquidity.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L165
- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L207

Impacts
If the contract's token balance becomes insufficient to cover both pending rewards and total staked principal, users unstaking later will be unable to retrieve their full deposits. This creates a situation where early users drain liquidity that should have been reserved for later unstakers. Funds effectively become locked until the admin manually replenishes the reward pool, resulting in partial loss or denial of service for legitimate stakers.

Remediation
Separate the accounting of staked principal and reward liquidity to ensure user stakes are never used to fund reward payouts.

Retest
This issue has been fixed by implementing proper checks to prevent the withdrawing the rewards from the users principle.

Bug ID #H010 [Fixed]

Reward Debt Updated Incorrectly Under APY System

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
High

Description
In the withdrawStake function, after processing withdrawals and potential reward claims, the contract updates both staker.rewardDebt and userManualRewardDebt unconditionally. These variables are used to track reward distribution in the block-based reward system, but they are irrelevant when the APY-based reward system (useAPYSystem) is enabled.

When useAPYSystem is true, rewards are calculated based on time and APY percentage rather than block reward shares. However, the contract still recalculates and overwrites staker.rewardDebt and userManualRewardDebt, introducing inconsistencies between the two reward systems. This may lead to incorrect pending reward calculations once the APY system is disabled or switched back, as user reward tracking would have been incorrectly reset under an unrelated accounting model.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/Staking.sol#L461-L462

Impacts
While operating under the APY system, users' reward debt and manual reward tracking are erroneously modified, leading to inaccurate reward accounting and potential loss of claimable rewards when toggling between APY-based and block-based reward modes.

Remediation
Update staker.rewardDebt and userManualRewardDebt only when the APY system is disabled (useAPYSystem == false).

Retest
This issue has been fixed by staker.rewardDebt and userManualRewardDebt when the APY system is disabled.

Bug ID #M001 [Acknowledged]

First vault consistently receives the remainder, causing unfair distribution

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Medium

Description
distributeRewards() splits the amount equally across vaults and assigns the entire remainder to the first vault in the loop. This deterministic behavior advantages the first vault every distribution cycle. Over time, that vault accrues more rewards than others, diverging from the intended "equal" distribution policy. While not an immediate security risk, it creates persistent allocation bias and user dissatisfaction.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/MainStaking.sol#L181-L214

Impacts
Rewards are skewed toward the first vault, leading to unfair outcomes, distorted APYs across vaults, and potential reputational risk. Users in later-indexed vaults receive systematically fewer tokens than expected.

Remediation
Distribute the remainder fairly using a round-robin offset that rotates which vault receives the extra token(s) each cycle. Persist an index (e.g., nextRemainderIndex) and advance it modulo vaultCount after each distribution so the remainder is spread evenly over time.

Retest
The acknowledged points are as per team logic and not imapcting flow.
Note: This issue still persists — the remainder is still being directed to the first active vault.

Bug ID #M002 [Fixed]

Vault pause/unpause functions inaccessible due to factory-admin role assignment

Vulnerability Type
Access Control (SCWE-016)

Severity
Medium

Description
In the StakingVault contract, the functions pause() and unpause() are restricted to the DEFAULT_ADMIN_ROLE. During deployment via VaultFactory, the factory contract itself is assigned as the vault's admin.

However, VaultFactory does not provide any functions to call pause() or unpause on the vaults. As a result, no externally controlled account can pause or unpause vaults, rendering the emergency stop mechanism ineffective.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L265C3-L271C6
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/VaultFactory.sol#L81

Impacts
The vaults pause and unpause functions cannot be used, leaving user funds exposed during emergencies or system misconfigurations. This creates operational risk, as the protocol cannot halt staking or reward distribution when needed.

Remediation
Add administrative functions in VaultFactory to forward pause() and unpause() calls to individual vaults

Retest
This issue has been fixed by adding pause() and unpause() functionality in the VaultFactory contract.

Bug ID #M003 [Fixed]

Inconsistent Reward State Due to Missing TotalRewards Deduction on Claims

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
Medium

Description
The StakingVault.addRewards() function allows an authorized account with STAKING_MANAGER_ROLE to increase vaultInfo.totalRewards by any arbitrary amount. This function is intended to record the total amount of rewards available for distribution within the vault.
However, there is no corresponding deduction or adjustment of vaultInfo.totalRewards when users later claim rewards via claimAPYRewards. As a result, the vault's on-chain accounting diverges from the actual reward token balance and the true remaining rewards.

The root cause is that addRewards updates vaultInfo.totalRewards unidirectionally without maintaining synchronization with actual reward outflows, causing totalRewards to permanently overstate the available reward supply.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L173

Impacts
Over time, the vaultInfo.totalRewards field becomes inaccurate, overstating the vault's token liabilities and misleading on-chain analytics or off-chain systems relying on this field for accounting or reporting.

Remediation
Deduct claimed reward amounts from vaultInfo.totalRewards inside claimAPYRewards to maintain accurate state consistency.

Retest
This issue has been resolved by subtracting the rewards from vaultInfo.totalRewards within the claimAPYRewards() function.

Bug ID #M004 [Fixed]

Manual Rewards Could Be Lost on Emergency Withdraw

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Medium

Description
The contract allows users to perform an emergencyWithdraw of staked tokens without claiming pending rewards. While block-based staking rewards are forfeited by the user (expected behavior), manual rewards allocated via addManualReward remain stuck in the contract because manualRewardPerShare is not adjusted and there is no mechanism for recovery.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L440C4-L459C6

Impacts
Users performing emergency withdraw will permanently forfeit their pending manual rewards, and these tokens will remain locked in the contract. This could result in unclaimed rewards accumulating over time and reducing overall reward efficiency for other users.

Remediation
Implement a function for the admin to reclaim unclaimed manual rewards for users who have emergency withdrawn.

Retest
This issue has been resolved by returning pendingManualRewards() to the user during the emergency withdrawal process.

Bug ID #M005 [Acknowledged]

Unclaimed Rewards Lost on Unstake Due to Missing Claim Enforcement

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Medium

Description
In the StakingVault contract, when a user unstakes their tokens via the unstake() function, the contract does not automatically claim or credit any unclaimed APY rewards that may have accrued before the unstake.

The unstake() function simply reduces userStake.amount and updates vault totals, but it does not call claimAPYRewards() or preserve the pending reward value anywhere. As a result, once the user's stake amount becomes 0, all reward calculation data (averageStakedAt, unlockTime, active) is reset to zero.

Because _calculateFixedAPYRewards() uses these fields to compute pending rewards, any unclaimed rewards become irrecoverable once the stake is reset.

This leads to loss of user rewards if the main staking manager does not enforce a pre-claim before unstaking.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L106C5-L130C6
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L209C5-L218C6

Impacts
Users permanently lose their accrued APY rewards if they unstake without first claiming them. Since the unstake() function resets the user's staking data and does not automatically distribute pending rewards, any unclaimed rewards become irrecoverable.

Remediation
Modify the unstake() function to automatically calculate and send pending rewards.

Retest

This issue has been partially resolved in StakingVault::unstake() by calculating pending rewards there. Especially since StakingVault::unstake() is called within it using vault.unstake(msg.sender, amount);, the MainStaking::unstake() function should also include pending rewards instead of returning only the unstaked amount.

Client Comment

MainStaking contract is the one that will be live but here too reward logic is not 100 percent, after the APY calculation some of the metrics will be calculated at backend level how user interact with main application.

Main purpose of implementing this partial solution is to have the smart contract basic structure so that we can easily upgrade and be transparent to users that your funds are safe at contract.

Bug ID #M006 [Fixed]

Timelock cannot receive vested tokens due to missing vesting integration

Vulnerability Type
Missing functionality (SCWE-006)

Severity
Medium

Description
Timelock.receiveFromVesting(uint256 amount) is restricted to the vesting contract (onlyRole(VESTING_CONTRACT_ROLE)) and pulls tokens with safeTransferFrom(msg.sender, address(this), amount). However, the current Vesting contract does not implement any call path that approves the timelock to spend its tokens and invokes receiveFromVesting(). Without this integration, the timelock can never receive vested tokens, even though the function exists and access control is configured.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Timelock.sol#L70-L74

Impacts
Vested tokens remain stuck in the Vesting contract or an intermediary wallet instead of moving into the Timelock. The vesting contract won't be able to add funds to the timelock contract.

Remediation
It is suggested to implement the transfer flow in Vesting to approve and call the receiveFromVesting() function in the Timelock contract

Retest
This issue has been fixed by implementing transferToTimelock() at vesting.sol

Bug ID #M007 [Acknowledged]

User can reuse one signature to call many actions

Vulnerability Type
Signature Replay Attacks (SCWE-055)

Severity
Medium

Description
_verifySignature() builds the message hash with only msg.sender (keccak256(abi.encodePacked(user))). This same signature is accepted by claimRewards(), claimVested(), and emergencyWithdraw(). Because the signed data does not include what action is being approved, the amount, a nonce, expiry, chainId, or the contract address, the same signature can be replayed across different functions and called multiple times. An attacker who obtains one valid signature for their address can repeatedly use it to trigger other sensitive flows until funds are drained.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L216-L225
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L357
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L394
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L418
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L441

Impacts
A user (or anyone who can submit their signature) can re-use that single signature to claim rewards multiple times, release vested tokens again, or perform an emergency withdrawal without fresh approval. This can cause unauthorized payouts, double claims, and treasury loss.

Remediation
Add strong domain separation and replay protection. Use EIP-712 typed data or, at a minimum, hash all critical fields: function intent, user, amount, user-specific nonce, deadline, chainId, and this

contract's address. Verify the signature against that structured hash, and consume the nonce on success.

Retest

Client Comment

We will give the reward based on KYC done at the backend so that signature module will be modified at later stages. At the moment its completely paused till KYC functionality is implemented.

Bug ID #M008 [Fixed]

Inconsistent Reward Payment Logic Between Unstake and Claim Functions

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Medium

Description
The contract implements different reward payment mechanisms for unstake and claimRewards functions, where unstake includes comprehensive balance checking with fallback to timelock and partial payments, but claimRewards uses a simple direct transfer that will revert entirely if MainStaking has insufficient balance. This inconsistency creates unreliable user experience and makes reward claiming more prone to failure than unstaking operations.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L221
- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart_contracts/contracts/MainStaking.sol#L181C9-L201C1

Impacts
Reward claiming becomes less reliable than unstaking, users may face transaction reverts when claiming rewards that would partially succeed during unstaking, and inconsistent user experience for similar reward payment operations.

Remediation
Implement the same comprehensive payment logic from unstake function into claimRewards, including balance checks, timelock fallback, and partial payment capability to ensure consistent and reliable reward distribution across all user actions.

Retest
This issue has been fixed by adding the required checks.

Bug ID #L001 [Fixed]

Vesting Can Be Created With Cliff Greater Than Duration

Vulnerability Type
Lack of Input Validation (SC04-Lack Of Input Validation)

Severity
Low

Description
The VestingManager contract supports creating vesting schedules via VestingManager.createVesting, VestingManager.createVestingMultiple (which uses VestingManager._createVestingWithPreFunded), and the internal helper VestingManager._createVesting. A vesting schedule is expected to specify a start, a cliff (seconds after start when vesting begins), and a duration (total vesting period).

However, the contract does not validate that cliff is less than or equal to duration. As a result, an admin can create a schedule where cliff > duration. The root cause is missing input validation in VestingManager._createVesting and VestingManager._createVestingWithPreFunded that should enforce cliff <= duration. The vesting calculation in _vestedAmount first checks if (block.timestamp < vest.start + vest.cliff) return 0; and only later returns vest.amountTotal when block.timestamp >= vest.start + vest.duration. When cliff > duration that ordering causes tokens to remain unreleasable until the cliff passes, and then the entire amount vests at once when the cliff is reached.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/Vesting.sol#L225C1-L237C55

Impacts
An administrator can intentionally (or accidentally) deploy vestings that withhold tokens from beneficiaries for longer than the declared duration, causing beneficiaries to be unable to claim any vested tokens until the (longer) cliff passes; at that point the full remaining balance vests instantly. This can lead to prolonged loss of liquidity for beneficiaries, unexpected lump-sum token releases that break downstream assumptions or accounting, and potential reputation or regulatory risk for the project. Because createVestingMultiple uses the same unchecked internal routine, batched vestings are also affected.

Remediation
To fix this issue apply require condition cliff <= duration when creating vestings in the internal creation functions.

Retest
This issue has been fixed by a require check to make sure that the cliff is less than duration.

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., ^0.8.19. 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/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Timelock.sol#L2
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Vesting.sol#L2
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/MainStaking.sol#L2
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L2
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.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

It is suggested that 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
This issue has been fixed by using fixed pragma.

Bug ID #L003 [Fixed]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom

Vulnerability Type
Missing best practices

Severity
Low

Description
The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L437
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L575
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L534

Impacts
Using safeTransferFrom has the following benefits -
- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.
- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Remediation
It is recommened that consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest

This issue has been fixed by using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Bug ID #L004 [Fixed]

Missing Upper Bound on APY Rate Allows Arbitrary Reward Inflation

Vulnerability Type
Unchecked Parameter

Severity
Low

Description
The StakingVault.updateApyRate() function allows the account with DEFAULT_ADMIN_ROLE to set a new APY rate without any upper bound. The vault uses this APY rate to calculate user rewards via _calculateFixedAPYRewards, which directly multiplies the staked amount by the APY rate and staking duration. As a result, if the admin by mistake sets an excessively high APY value, reward calculations can overflow, deplete vault funds, or create unbacked liabilities. The root cause is the absence of a maximum cap on newApyRate within updateApyRate.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart_contracts/contracts/StakingVault.sol#L224C4-L227C6

Impacts
An admin can set an arbitrary APY (e.g., newApyRate = 1e18), causing claimAPYRewards to transfer disproportionate token amounts to users addresses, leading to immediate vault insolvency.

Remediation
Restrict newApyRate to a safe and predefined maximum.

Retest
This issue has been fixed by adding a cap of 100% in newApyRate function.

Bug ID #L005 [Fixed]

Admin Can Grant Vesting Role To Arbitrary Address

Vulnerability Type
Business Logic (SC03-LogicErrors)

Severity
Low

Description
The contract allows the admin to manage privileged roles such as STAKING_CONTRACT_ROLE and VESTING_CONTRACT_ROLE. In updateStakingContract, the implementation enforces strict validation by ensuring the new address is non-zero, not the same as the old one, and by revoking the old role before granting the new one.
However, in grantVestingRole(), no such validation is performed—the function directly grants VESTING_CONTRACT_ROLE to any address supplied by the admin. The root cause is missing parity in role-granting logic, which could allow configuration errors or accidental assignment to unintended addresses.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Timelock.sol#L120-L123

Impacts
While only an authorized admin can call this function, a misconfiguration (e.g., granting the vesting role to an EOA or unrelated contract) could lead to unintended access to vesting operations, potentially affecting token distribution logic.

Remediation
Validate the target address similarly to how the staking contract is handled before granting the vesting role.

Retest
This issue has been fixed by adding require checks for target addresses as suggested.

Bug ID #I001 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects (SCWE-062)

Severity
Informational

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L136-L139
- https://github.com/Lern360/ico-smart-contract/blob/de40a2292de127890939af03ce4626ed4066e25d/smart—contracts/contracts/Staking.sol#L243-L252

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion
When the code is sent to other developers or auditors to understand and implement.
This reduces the overall size of the contracts and also helps in saving gas.

Remediation
It is suggested that if the library functions are not supposed to be used anywhere, consider removing them from the contract.

Retest
The issue has been fixed.

Bug ID #I002 [Fixed]

apyPercentage Not Reset When Disabling APY System

Vulnerability Type
Inconsistent Accounting (SCWE-010)

Severity
Informational

Description
The enableAPYSystem function allows an administrator to activate the APY-based reward system by setting useAPYSystem = true and configuring apyPercentage. When disabling the system through disableAPYSystem, the function only flips the useAPYSystem flag to false but does not reset the stored apyPercentage value to zero. This creates a state inconsistency where the APY system is marked as disabled, yet a non-zero APY percentage remains stored in contract state.

Affected Code
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L159C4-L164C6

Impacts
When the APY system is disabled, apyPercentage retains its previous non-zero value which will create state inconsistency.

Remediation
It is suggested that reset apyPercentage to zero when disabling the APY system to maintain a consistent and predictable state.

Retest
This issue has been fixed by reseting apyPercentage to zero in disableAPYSystem.

Bug ID #G001 [Fixed]

Gas Optimization in Require/Revert Statements

Vulnerability Type
Gas Optimization (SCWE-082)

Severity
Gas

Description
The **require/revert** statement takes an input string to show errors if the validation fails.
The strings inside these functions that are longer than **32 bytes** require at least one additional MSTORE, along with additional overhead for computing memory offset and other parameters. For this purpose, having strings lesser than 32 bytes saves a significant amount of gas.
Affected Code

- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L625

Impacts
Having longer require/revert strings than **32 bytes** cost a significant amount of gas.

Remediation
It is recommended to shorten the strings passed inside **require/revert** statements to fit under **32 bytes**. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Retest
This issue has been fixed by shorten the strings passed inside require statement.

Bug ID #G002 [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/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Vesting.sol#L163
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/StakingVault.sol#L248
- https://github.com/Lern360/ico-smart-contract/blob/96f21c0d66c477eee754971cf218b3f02e94b5c0/smart—contracts/contracts/Staking.sol#L150

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
This issue is fixed.

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

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com