Audited by

CredShields

Smart Contract Audit

April 17th, 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 April 7th, 2025, and April 8th, 2025. A retest was performed on April 16th, 2025.

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

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

Prepared for
LERN360

Table of Contents

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

LERN360 engaged CredShields to perform a smart contract audit from April 7th, 2025, to April 8th, 2025. During this timeframe, 8 vulnerabilities were identified. **A retest was performed on April 16th, 2025, and all the bugs have been addressed.**

During the audit, 2 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "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
LERNToken	0	2	0	2	1	3	**8**
	0	**2**	**0**	**2**	**1**	**3**	**8**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in LERNToken's 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 LERNToken smart contract presents no known security vulnerabilities as of the latest retest on April 16, 2025.
Based on industry-standard security assessment methodologies and manual code review, we are confident that the LERNToken contract is secure for deployment, assuming no changes are made post-audit.
Investors and ecosystem participants can rely on this audit as a mark of technical integrity and responsible security governance by the LERN360 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 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 a LERNToken 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 April 7th, 2025, to April 8th, 2025, was agreed upon during the preparation phase.

2.1.1 Scope

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

IN SCOPE ASSETS
https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#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 uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

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 the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, 8 security vulnerabilities were identified in the asset. At the time of the audit, the contract was using the Solidity Pragma version 0.8.28.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Approval race condition vulnerability in LERNToken contract	High	Approval Race Condition
Owner can burn tokens from any user's wallet without consent	High	Centralization Risk
Floating Pragma	Low	Floating Pragma (SWC-103)
Use Ownable2Step	Low	Missing Best Practices
Self-transfer results in unnecessary state changes	Informational	Gas Inefficiency
Large number literals	Gas	Gas & Missing Best Practices
Gas Optimization for State Variables	Gas	Gas Optimization
Cheaper Inequalities in require()	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0
SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use

SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found
SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.

SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

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

LERN360 is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on April 16th, 2025, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Approval race condition vulnerability in LERNToken contract	High	**Fixed** [April 16, 2025]
Owner can burn tokens from any user's wallet without consent	High	**Fixed** [April 16, 2025]
Floating Pragma	Low	**Fixed** [April 16, 2025]
Use Ownable2Step	Low	**Fixed** [April 16, 2025]
Self-transfer results in unnecessary state changes	Informational	**Fixed** [April 16, 2025]
Large number literals	Gas	**Fixed** [April 16, 2025]
Gas Optimization for State Variables	Gas	**Won't Fix** [April 16, 2025]
Cheaper Inequalities in require()	Gas	**Won't Fix** [April 16, 2025]

Table: Summary of findings and status of remediation

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

Bug ID #1 [Fixed]

Approval race condition vulnerability in LERNToken contract

Vulnerability Type
Approval Race Condition

Severity
High

Description
The issue is in the approve() function which does not protect against the approval race condition. Here's how it works:

1. Alice wants to change Bob's allowance from 1000 to 2000 tokens
2. Alice sends a transaction to call approve(Bob, 2000)
3. Bob sees this pending transaction in the mempool
4. Bob quickly sends a transaction to transferFrom(Alice, Bob, 1000) with a higher gas fee
5. Bob's transaction gets mined first, transferring 1000 tokens
6. Alice's approval transaction then gets mined, setting the allowance to 2000
7. Bob can now transfer another 2000 tokens, totaling 3000 tokens instead of the intended 2000

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L206

Impacts
By front-running approval transactions, an attacker could spend more tokens than the owner originally intended to authorize. In the worst-case scenario, if the token owner fails to carefully monitor allowance changes, this flaw could potentially allow an attacker to drain the entire approved balance

Remediation

It is recommended to add dedicated `increaseAllowance()` and `decreaseAllowance()` functions that allow users to safely modify existing approvals without exposing themselves to front-running attacks.

Retest

This issue has been fixed.

Bug ID #2 [Fixed]

Owner can burn tokens from any user's wallet without consent

Vulnerability Type
Centralization Risk

Severity
High

Description
The burn() function allows the contract owner to burn tokens from any user's address without requiring the user's permission. This is implemented through the onlyOwner modifier, which restricts function access to the contract owner, but doesn't require approval from the token holder. This design creates centralized control where the owner can reduce the token balance of any address.

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L235

Impacts
Users' tokens can be destroyed without their consent by the contract owner.

Remediation
It is recommended to add a validation to check the user has required approvals/balance of the msg.sender before burning.

Retest
This issue has been fixed.

Bug ID #3 [Fixed]

Floating Pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
The contract allowed floating pragma to be used, i.e., ^0.8.28. 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/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L160

Impacts
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 0.8.28 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest
This issue has been fixed.

Bug ID #4 [Fixed]

Use Ownable2Step

Vulnerability Type
Missing Best Practices

Severity
Low

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

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L163

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

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

Retest:
This issue has been fixed.

Bug ID #5 [Fixed]

Self-transfer results in unnecessary state changes

Vulnerability Type
Gas Inefficiency

Severity
Informational

Description
The transfer() and transferFrom() functions do not check if the sender and recipient are the same address. When a user transfers tokens to themselves, the contract still updates balances and emits a Transfer event. This has no effect on the actual balance but results in unnecessary gas usage and event emission.

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L186

Impacts
This behavior leads to avoidable gas costs and unnecessary log entries. It does not affect contract security but slightly impacts efficiency.

Remediation
Add validation to revert the transaction if the sender and recipient addresses are the same.

```
+ require(recipient != msg.sender, "Transfer to self not allowed");
```

Retest
This issue has been fixed.

Bug ID #6 [Fixed]

Large number literals

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

Description
Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations. The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation also supported by Solidity.

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L173

Impacts
Having a large number literals in the code increases the gas usage of the contract during its deployment and when the functions are used or called from the contract.
It also makes the code harder to read and audit and increases the chances of introducing code errors.

Remediation
Scientific notation in the form of 2e10 is also supported, where the mantissa can be fractional, but the exponent has to be an integer. The literal MeE is equivalent to M * 10**E. Examples include 2e10, 2e10, 2e-10, 2.5e1, as suggested in official solidity documentation. https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals
It is recommended to use numbers in the form "35 * 1e7 * 1e18" or "35 * 1e25".
The numbers can also be represented by using underscores between them to make them more readable such as "35_00_00_000"

Retest
This issue has been fixed.

Bug ID #7 [Won't Fix]

Gas Optimization for State Variables

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Plus equals (+=) costs more gas than the addition operator. The same thing happens with minus equals (-=). Therefore, x +=y costs more gas than x = x + y.

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L190
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L191
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L214
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L215
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L224
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L225
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L232
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L233

Impacts
Writing the arithmetic operations in x = x + y format will save some gas.

Remediation
It is suggested to use the format x = x + y in all the instances mentioned above.

Retest

Client Comments : This finding is not applicable to contracts compiled with modern versions of Solidity. The contract in question is compiled using Solidity v0.8.24, which includes compiler optimizations that eliminate the gas cost difference between compound assignment operators and their explicit arithmetic equivalents.
There is no measurable gas improvement from switching to the explicit arithmetic format.

Bug ID #8 [Won't Fix]

Cheaper Inequalities in require( )

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

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

Affected Code

- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L188
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L211
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L212
- https://testnet.bscscan.com/address/0xcAbd7dA9bBe77294A7F6d07DEC4237365E0Fd277#code#F1#L230

Impacts
Using non-strict inequalities inside "require" statements costs more gas.

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

Retest:
Client Comments :
These are necessary conditions to ensure correct logic and input validation.
There is no need to change the current comparison logic for the sake of gas optimization.
This issue can be marked as Resolved – Not an Issue.

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