CredShields

Smart Contract Audit

Sept 11th, 2023

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Capital Rock between Sept 8th, 2023, and Sept 10th, 2023. A retest was performed on Sept 11th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Capital Rock

Table of Contents

1. Executive Summary

Capital Rock engaged CredShields to perform a smart contract audit from Sept 8th, 2023, to Sept 10th, 2023. During this timeframe, Five (5) vulnerabilities were identified. **A retest was performed on Sept 11th, 2023, and all the bugs have been addressed.**

During the audit, Zero (0) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Capital Rock" and should be prioritized for remediation, and fortunately, none were found.

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
Smart Contract	0	0	0	0	2	3	**5**
	0	**0**	**0**	**0**	**2**	**3**	**5**

Table: Vulnerabilities Per Asset in Scope

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

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Capital Rock'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 Capital Rock can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Capital Rock can future-proof its security posture and protect its assets.

2. Methodology

Capital Rock engaged CredShields to perform a Capital Rock 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 Sept 8th, 2023, to Sept 10th, 2023, 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://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code

Table: List of Files in Scope

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

Capital Rock 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, 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	Note	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 around the engagement or this document.

3. Findings

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, Five (5) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Gas Optimization in Require Statements	Gas	Gas Optimization
Functions should be declared External	Informational	Best Practices
Large Number Literals	Gas	Gas & Missing Best Practices
Dead Code	Informational	Code With No Effects - SWC-135
Approve Frontrunning Attack	Informational	Frontrunning

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	Not 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

Capital Rock is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Sept 11th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Gas Optimization in Require Statements	Gas	**Won't Fix [12/09/2023]**
Functions should be declared External	Informational	**Won't Fix [12/09/2023]**
Large Number Literals	Gas	**Won't Fix [12/09/2023]**
Dead Code	Informational	**Won't Fix [12/09/2023]**
Approve Frontrunning Attack	Informational	**Won't Fix [12/09/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Won't Fix]

Gas Optimization in Require Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The **require()** 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.

Vulnerable Code

- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L157
- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L233-L234
- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L255
- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L274
- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L282-L283

- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L303

Impacts
Having longer require strings than 32 bytes costs a significant amount of gas.

Remediation
It is recommended to shorten the strings passed inside **require()** 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
The bug just helps to enhance gas optimization and is not a security threat and hence the team decided not to fix it. The CredShields team agrees with the decision.

Bug ID#2 [Won't Fix]

Functions should be declared External

Vulnerability Type
Best Practices

Severity
Informational

Description
Public functions that are never called by a contract should be declared external in order to conserve gas.
The following functions were declared as public but were not called anywhere in the contract, making public visibility useless.

Affected Code
The following functions were affected -

- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L264
- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L272

Impacts
Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.
"**public**" functions cost more Gas than "**external**" functions.

Remediation
Use the "**external**" state visibility for functions that are never called from inside the contract.

Retest

The bug is not exploitable and is just a best practice hence the bug won't be fixed. The Credshields team agrees with the decision of the team.

Bug ID#3 [Won't Fix]

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://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L178

Impacts
Having a large number literals in the code increases the gas usage of the contract while 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

The bug just helps to enhance gas optimization and is not a security threat and hence the team decided not to fix it. The CredShields team agrees with the decision.

Bug ID#4 [Won't Fix]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informational

Description
The contract inherits an Ownable contract and uses an owner variable during construction. It is not using the onlyOwner modifier functions or any owner-related functions anywhere in the code. Due to this, there's no point in having an Ownable contract in the ERC token.
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.

Vulnerable Code

- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L164

Impacts
Having an ownable contract won't have any effect since the owner variable or modifier is not used anywhere in the code.

Remediation
Consider removing the Ownable contract if it is not supposed to be used anywhere. Instead of using owner() when creating total supply, the owner address can directly be taken as a constructor parameter.

Retest

The bug is not exploitable and is just a best practice hence the bug won't be fixed. The Credshields team agrees with the decision of the team.

Bug ID#5 [Won't Fix]

Approve Frontrunning Attack

Vulnerability Type
Frontrunning

Severity
Informational

Description
Approve is well known to be vulnerable to front-running attacks. This may be exploited in cases where in case the user decides to modify the spending amount in quick succession, and the attacker sees the change and transfers more tokens than required.
The exploitation involves two parties - a victim and an attacker and the attacker must be monitoring the transactions to front-run the transaction.

Vulnerable Code

- https://bscscan.com/token/0x3542a28854c5243656FA5cfA1A2811a32E28C1c8#code#L243

Impacts
Front-running attacks might allow attackers to withdraw/transfer more tokens than the victim initially intended to allow.

Remediation
Instead of approve() to change the allowance, it is recommended to use increaseAllowance and decreaseAllowance functions which are meant for this use case.

Retest
The bug is not exploitable and is just a best practice hence the bug won't be fixed. The Credshields team agrees with the decision of the team.

6. Disclosure

The Reports provided by CredShields is 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.