CredShields

Smart Contract Audit

Sept 18th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Rove Token audit performed by CredShields Technologies PTE. LTD. on behalf of Rove Finance between Sept 15th, 2023, and Sept 18th, 2023. And a retest was performed on Sept 18th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Rove Finance

Table of Contents

1. Executive Summary

Rove Finance engaged CredShields to perform a smart contract audit from Sept 15th, 2023, to Sept 18th, 2023. During this timeframe, Eight (8) vulnerabilities were identified. **A retest was performed on Sept 18th, 2023, and all the bugs have been addressed.**

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

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Rove Finance engaged CredShields to perform a Rove Finance 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 15th, 2023, to Sept 18th, 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
RoveToken.sol

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

Rove Finance 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, Eight (8) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Minting Failure	High	Business Logic
Faulty Pausable Logic Implementation	Medium	Business Logic
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Use Ownable2Step	Informational	Missing Best Practices
Functions should be declared External	Informational	Best Practices
Large Number Literals	Gas	Gas & Missing Best Practices
Public Constants can be Private	Gas	Gas Optimization

Cheaper Conditional Operators	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	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

Rove Finance is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Sept 18th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Minting Failure	High	**Fixed [18/09/2023**
Faulty Pausable Logic Implementation	Medium	**Fixed [18/09/2023**
Floating and Outdated Pragma	Low	Won't Fix
Use Ownable2Step	Informational	Won't Fix
Functions should be declared External	Informational	**Fixed [18/09/2023**
Large Number Literals	Gas	**Fixed [18/09/2023**
Public Constants can be Private	Gas	**Fixed [18/09/2023**
Cheaper Conditional Operators	Gas	**Fixed [18/09/2023**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Minting Failure

Vulnerability Type

Business Logic

Severity

High

Description

The contract intends to allow owners to mint new tokens by calling the mint() function. This function validates that the total tokens after minting do not go over the totalSupplyCap defined during construction.

Since there's no burn functionality and the ERC transfers are not allowed to 0 addresses, the total supply will always be constant due to which the mint() function will always revert on the validation: "totalSupply() + amount <= totalSupplyCap".

Affected Code

- RoverToken.mint()

Impacts

Due to the always-reverting function, the contract will never be able to mint new tokens and the total supply will be constant. The mint function will be rendered useless.

Remediation

If the contract's intention is to allow owners to mint new tokens, consider having a burn functionality. Otherwise, adjust the initial mint supply in the constructor so there's some threshold left to mint at a later stage.

Retest

The contract is now minting 80% of the total supply cap due to which the mint function will work as it should. This is fixed.

https://github.com/Rover-Finance/ROVE-Token/blob/bad63a231634e3efa5654adb26403dce225ca6a7/RoverToken.sol#L25

Bug ID #2 [Fixed]

Faulty Pausable Logic Implementation

Vulnerability Type

Business Logic

Severity

Medium

Description

The RoverToken contract implements Pausable and defines two functions to pause and unpause the contract. However, it should be noted that the pausable feature does not work by simply defining these functions. The developer has to implement modifiers in functions they want to prevent from being called when the contract is paused (whenNotPaused). Without these modifiers and their implementations, the contract functionalities will never be paused.

Affected Code

- RoverToken.pause(), unpause()

Impacts

The pause and unpause functions won't have any effect on the token transactions in the contract.

Remediation

It is recommended to override the internal function _beforeTokenTransfer from Openzeppelin's ERC20 contract and add the whenNotPaused modifier to the function. This will be called before every token transaction essentially pausing the contract when the owner wants. E.g.:

```
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal override whenNotPaused {
    super._beforeTokenTransfer(from, to, amount);
}
```

Retest

This is fixed by overriding _beforeTokenTransfer and implementing whenNotPaused.

https://github.com/Rover-Finance/ROVE-Token/blob/bad63a231634e3efa5654adb26403dce225ca6a7/RoverToken.sol#L80-L86

Bug ID #3 [Won't Fix]

Floating and Outdated Pragma

Vulnerability Type

Floating Pragma (SWC-103)

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., >= 0.8.9. 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

- RoverToken

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 really low therefore this is only informational.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.20 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

The pragma version is fixed but the contract is using an outdated pragma. Since it is not exploitable it is left as it is and CredShields' team agrees with the decision.

Bug ID #4 [Won’t Fix]

Use Ownable2Step

Vulnerability Type

Missing Best Practices

Severity

Informational

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

- RoverToken

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

The team decided not to implement the additional security measure and CredShields' team agrees that it is optional.

Bug ID #5 [Fixed]

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

- RoverToken.mint()

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 functions that are not called inside the contract have been made external.

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

- RoverToken.totalSupplyCap

Impacts
Having 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

Large numbers have been updated in the scientific notation form.

https://github.com/Rover-Finance/ROVE-Token/blob/bad63a231634e3efa5654adb26403dce225ca6a7/RoverToken.sol#L14

Bug ID #7 [Fixed]

Public Constants can be Private

Vulnerability Type
Gas Optimization

Severity
Gas

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

Affected Code

- RoverToken.totalSupplyCap

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

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

Retest
The variable has been updated as private.
https://github.com/Rover-Finance/ROVE-Token/blob/bad63a231634e3efa5654adb26403dce225ca6a7/RoverToken.sol#L14

Bug ID #8 [Fixed]

Cheaper Conditional Operators

Vulnerability Type

Gas Optimization

Severity

Gas

Description

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

Affected Code

```
require(amount > 0, "Amount must be greater than 0");
```

Impacts

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

Remediation

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

Retest

This is fixed. The validation has been updated to x != 0.
https://github.com/Rover-Finance/ROVE-Token/blob/bad63a231634e3efa5654adb26403dce225ca6a7/RoverToken.sol#L53

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.