Audited by

CredShields

Smart Contract Audit

September 6th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Protop Vesting Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Protop between July 16th, 2024, and July 30th, 2024. A retest was performed on September 3rd, 2024.

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

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

Prepared for
Protop

Table of Contents

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

Protop engaged CredShields to perform a smart contract audit from July 16th, 2024, to July 30th, 2024. During this timeframe, 8 vulnerabilities were identified. **A retest was performed on September 3rd, 2024, and all the bugs have been addressed.**

During the audit, 1 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Protop" 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
Protop Vesting	1	0	0	3	0	4	**8**
	1	**0**	**0**	**3**	**0**	**4**	**8**

Table: Vulnerabilities Per Asset in Scope

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

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

Protop engaged CredShields to perform a Protop Vesting 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 July 16th, 2024, to July 30th, 2024, 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
Testnet: https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948 **Mainnet:** https://polygonscan.com/address/0xBD66Ca6a4e65d0120a491A3D78feaE8A85cD38e4

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

Protop 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.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Incorrect Time Calculation Leading to Vesting Logic Flaws	Critical	Business Logic Issue
Outdated Pragma	Low	Outdated Compiler Version (SWC-102)
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing Best Practices
Use Ownable2Step	Low	Missing Best Practices
Cheaper Inequalities in require()	Gas	Gas Optimization
Cheaper Conditional Operators	Gas	Gas Optimization
Custom Errors to Save Gas	Gas	Gas Optimization
Gas Optimization in Require/Revert Statements	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 -------------------

Protop is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on September 3rd, 2024, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Incorrect Time Calculation Leading to Vesting Logic Flaws	Critical	**Fixed** [September 3rd, 2024]
Outdated Pragma	Low	**Fixed** [September 3rd, 2024]
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed** [September 3rd, 2024]
Use Ownable2Step	Low	**Fixed** [September 3rd, 2024]
Cheaper Inequalities in require()	Gas	**Fixed** [September 3rd, 2024]
Cheaper Conditional Operators	Gas	**Fixed** [September 3rd, 2024]
Custom Errors to Save Gas	Gas	**Fixed** [September 3rd, 2024]
Gas Optimization in Require/Revert Statements	Gas	**Won't Fix** [September 3rd, 2024]

Table: Summary of findings and status of remediation

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

Bug ID #1 [Fixed]

Incorrect Time Calculation Leading to Vesting Logic Flaws

Vulnerability Type

Business Logic Issue

Severity

Critical

Description

The vest() and claim() functions contain significant logic errors in the calculation of time intervals. The LockDuration and nextClaim parameters are intended to be calculated in months but the current implementation incorrectly calculates these values into minutes. This allows users to bypass the intended monthly vesting schedule by making claims every minute.

Affected Code

- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L62
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L64
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L93

Impacts

Users can exploit the incorrect time calculations to claim tokens prematurely and repeatedly in short intervals rather than adhering to the monthly schedule.

Remediation

To resolve this issue, the time calculations should be adjusted to accurately reflect months instead of minutes.

Example:

```
function vest(address account, uint256 _amount,uint256 lockDurationInMonth, uint256
vestDurationInMonth) external onlyOwner nonReentrant returns(bool) {
    ...
```

```
    vests[account] = VestDetail(
      account,
      _amount,
      block.timestamp,
@>    block.timestamp + lockDurationInMonth * 30 * 24 * 60 * 60,
      vestDurationInMonth,
@>    block.timestamp + lockDurationInMonth * 30 * 24 * 60 * 60,
      0,
      _amount,
      true
    );
}
```

```
function claim() external nonReentrant returns(bool){

  ...

  if(timeDiff > 1 minutes) {
@>    count = timeDiff / (30*24*60*60);
      count += 1;
      claimAmount = amountPerMonth * count;
    }
  ...
}
```

Retest

This vulnerability has been fixed by updating the calculation process from minutes to months.

Bug ID #2 [Fixed]

Outdated Pragma

Vulnerability Type
Outdated Compiler Version (SWC-102)

Severity
Low

Description
The smart contract is using an outdated version of the Solidity compiler specified by the pragma directive i.e. 0.8.24. Solidity is actively developed, and new versions frequently include important security patches, bug fixes, and performance improvements. Using an outdated version exposes the contract to known vulnerabilities that have been addressed in later releases. Additionally, newer versions of Solidity often introduce new language features and optimizations that improve the overall security and efficiency of smart contracts.

Affected Code
The following contracts were found to be affected:

- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L2

Impacts
The use of an outdated Solidity compiler version can have significant negative impacts. Security vulnerabilities that have been identified and patched in newer versions remain exploitable in the deployed contract.
Furthermore, missing out on performance improvements and new language features can result in inefficient code execution and higher gas costs.

Remediation
It is suggested to use the 0.8.25 pragma version.
Reference: https://swcregistry.io/docs/SWC-103

Retest
This issue has been fixed by updating the pragma version to 0.8.25

Bug ID #3 [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://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L70
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L98

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
Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest
This issue has been fixed by updating to safeTransfer() and safeTransferFrom().

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://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L8

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 by updating to Ownable2Step or Ownable2StepUpgradeable.

Bug ID #5 [Fixed]

Cheaper Inequalities in require()

Vulnerability Type
Gas Optimization

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://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L77
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L78

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:
This issue has been fixed by updating the non-strict inequalities with the strict ones.

Bug ID #6 [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 operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L53
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L55
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L56

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

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

Retest
This issue has been fixed by updating the 'x > 0' to 'x != 0'.

Bug ID #7 [Fixed]

Custom error to save gas

Vulnerability Type
Gas Optimization

Severity
Gas

Description
During code analysis, it was observed that the smart contract is using the revert() statements for error handling. However, since Solidity version 0.8.4, custom errors have been introduced, providing a better alternative to the traditional revert(). Custom errors allow developers to pass dynamic data along with the revert, making error handling more informative and efficient. Furthermore, using custom errors can result in lower gas costs compared to the revert() statements.

Affected Code

- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L105

Impacts
Custom errors allow developers to provide more descriptive error messages with dynamic data. This provides better insights into the cause of the error, making it easier for users and developers to understand and address issues.

Remediation
It is recommended to replace all the instances of revert() statements with error() to save gas.

Retest
This issue has been fixed by replacing revert() with a custom error.

Bug ID #8 [Won't Fix]

Gas Optimization in Require/Revert Statements

Vulnerability Type
Gas Optimization

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://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L29
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L53
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L55
- https://amoy.polygonscan.com/address/0x4955775504A1c4F2E2B98B0cE1988B48fD94b948#code#F1#L56

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 not been resolved. It is recommended to fix it as mentioned in Remediation.

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