CredShields

Smart Contract Audit

Feb 7th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Zenland Token Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Zenland between Feb 1st, 2023, and Feb 6th, 2023. **And a retest was performed on Feb 7th, 2023.**

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Zenland

Table of Contents

CRED SHIELDS

1. Executive Summary

Zenland engaged CredShields to perform a smart contract audit from Feb 1st, 2023, to Feb 6th, 2023. During this timeframe, Fifteen (15) vulnerabilities were identified. **A retest was performed on Feb 7th, 2023, and all the bugs have been addressed.**

During the audit, Two (2) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Zenland" 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
Zenland Token Contract	0	2	1	4	2	6	**15**
	0	**2**	**1**	**4**	**2**	**6**	**15**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Zenland engaged CredShields to perform a Zenland 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 Feb 1st, 2023, to Feb 6th, 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://github.com/zenland-dao/token_contracts

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

Zenland 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, Fifteen (15) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Outdated Pragma versions	Low	Floating Pragma (SWC-103)
Unnecessary Require Statement	Gas	Code Optimization
DeadCode	Informative	Code With No Effects - SWC-135
Missing Events in important functions	Low	Missing Best Practices
Use Require instead of If & Revert	Gas	Gas optimization
Functions should be declared External	Gas	Gas Optimization
Unused Imports	Gas	Gas Optimization

Gas Optimization in Require Statements	Gas	Gas Optimization
Missing Multiple Zero Address Validations	Low	Missing Input Validation
Gas Optimization in _cliff Validation	Gas	Gas Optimization
Missing Validations in Timestamp and Duration [Singlevesting]	Medium	Missing Input Validation
Missing Validations in Duration [Multivesting]	Low	Missing Input Validation
Missing NatSpec Comments	Informational	Missing best practices
Business Logic Issue in Vesting Duration	High	Business Logic
Locked Ether	High	Locked Ether

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

Zenland is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Feb 7th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Outdated Pragma versions	Low	**Fixed [07/02/2023]**
Unnecessary Require Statement	Gas	**Fixed [07/02/2023]**
DeadCode	Informative	**Fixed [07/02/2023]**
Missing Events in important functions	Low	**Fixed [07/02/2023]**
Use Require instead of If & Revert	Gas	**Fixed [07/02/2023]**
Functions should be declared External	Gas	**Fixed [07/02/2023]**
Unused Imports	Gas	**Fixed [07/02/2023]**
Gas Optimization in Require Statements	Gas	**Fixed [07/02/2023]**

Missing Multiple Zero Address Validations	Low	**Fixed [07/02/2023]**
Gas Optimization in _cliff Validation	Gas	**Fixed [07/02/2023]**
Missing Validations in Timestamp and Duration [Singlevesting]	Medium	**Fixed [07/02/2023]**
Missing Validations in Duration [Multivesting]	Low	**Fixed [07/02/2023]**
Missing NatSpec Comments	Informational	**Fixed [07/02/2023]**
Business Logic Issue in Vesting Duration	High	**Fixed [07/02/2023]**
Locked Ether	High	**Fixed [07/02/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Outdated Pragma versions

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Using an outdated compiler version can be problematic, especially if there are publicly disclosed bugs and issues that affect the current compiler version.
The contracts found in the repository were allowing an old compiler version to be used, i.e., 0.8.0

Affected Code

- 0.8.0 - Multivesting.sol (https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L4)
- 0.8.0 - Singlevesting.sol (https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L5)
- 0.8.0 - ZENF.sol (https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/ZENF.sol#L6)

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 Low severity.

Remediation
Keep the compiler versions updated in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.9 pragma version which is stable and not too recent.
Reference: https://swcregistry.io/docs/SWC-103

Retest:
Strict pragma is in use with a stable pragma version 0.8.9

Bug ID#2 [Fixed]

Unnecessary Require Statement

Vulnerability Type
Code Optimization

Severity
Gas

Description
The contract **Multivesting.sol** had defined a function called addVesting that takes an input "**uint256 _cliff**". It also validates the parameter inside a "**require()**" statement on line 34 which is not necessary since the **_cliff** is an unsigned integer and can never be negative.

Affected Code

- Multivesting.sol - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L34

```
  /// Creates vesting for beneficiary, with a given amount of funds to allocate
  function addVesting(address _beneficiary, uint256 _amount, uint256 _cliff, uint256 _duration) public onlyOwner {
      require(_cliff >= 0, "Cliff cannot be negative.");
```

Impacts
Using unnecessary require statements might incur more gas cost when calling the function or deploying the contract.

Remediation
It is recommended to remove the "**require()**" statement since it'll never be false.

Retest:

The statement has been removed.

Bug ID#3 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informative

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.
The contract **Singlevesting.sol** had defined a variables called "**_released**" which is not used anywhere in the code.

Affected Code

- Singlevesting.sol - Line 27 - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L27

```
contract VestingWallet is Context {
   event ERC20Released(address indexed token, uint256 amount);

   uint256 private _released;
```

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
If the variables and constants are not supposed to be used anywhere, consider removing them from the contract.

Retest:
The variable has been removed.

Bug ID#4 [Fixed]

Missing Events in important functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code
The following functions were affected -

- Multivesting.sol - addVesting()
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L33-L50
- Multivesting.sol - withdraw()
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L53-L70

```
    function addVesting(address _beneficiary, uint256 _amount, uint256
_cliff, uint256 _duration) public onlyOwner {}

   function withdraw() external {}
```

Impacts

Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation

Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

Retest:

Events have been added for both addVesting() and withdraw() function.

Bug ID#5 [Fixed]

Use Require instead of If & Revert

Vulnerability Type
Gas optimization

Severity
Gas

Description
The contract **Multivesting.sol** is using a combination of **if** and **revert** statements on Line 37. This is unnecessary and increases gas usage. This can be optimized by using a **require** statement instead of **revert**.

Vulnerable Code

- Multivesting.sol - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L37

```
   if(vestingMap[_beneficiary].exists) revert("Vesting object for this beneficiary already exists.");
```

Impacts
Using both **if** and **revert** simultaneously costs more than than using a simple **require** statement. If **require** was used, the contract would have saved approximately 216 gas units.

Remediation
It is recommended to switch to a **require** statement as shown below:

```
require(!vestingMap[_beneficiary].exists), "Vesting object for
this beneficiary already exists.");
```

Retest

The **if** and **revert** statements have been changed to a **require** statement.

Bug ID#6 [Fixed]

Functions should be declared External

Vulnerability Type
Gas Optimization

Severity
Gas

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 the public visibility useless.

Affected Code
The following functions were affected -

- Singlevesting.sol - Line 84 - release() - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L84-L89
- Multivesting.sol - Line 33 - addVesting() - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L33-L50

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:
Public functions release() and addvesting() has been marked as external to save gas.

Bug ID#7 [Fixed]

Unused Imports

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract **Singlevesting.sol** was importing an OpenZeppelin's contract **Address.sol** which was not used anywhere in the code. This increases the gas cost and overall contract's complexity.

Affected Code
The following functions were affected -

- Singlevesting.sol - Line 8
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L8

Impacts
Unused imports in smart contracts can lead to an increase in the size of the code, making it more difficult to verify and potentially slowing down its execution. Moreover, having unused code in a smart contract can also increase the attack surface by potentially introducing vulnerabilities that can be exploited by malicious actors. This can lead to security issues and compromise the integrity of the contract.

Additionally, including unused imports in smart contracts can also increase deployment and gas costs, making it more expensive to deploy and run the contract on the Ethereum network.

Remediation
It is recommended to remove the import statement if the external contracts or libraries are not used anywhere in the contract.

Retest:
The **Address.sol** has been removed from the imports.

Bug ID#8 [Fixed]

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

- Singlevesting.sol - Line 37
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L37
- Multivesting.sol - Line 35
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L35

```
   require(beneficiaryAddress != address(0), "VestingWallet: beneficiary
is zero address");

   require(_cliff <= 60 * 60 * 24 * 365 * 2, "Cliff cannot be more than
2 years.");
```

Impacts

Having longer require strings than 32 bytes cost 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 strings inside require statements have been shortened to be less than 32 bytes.

Bug ID#9 [Fixed]

Missing Multiple Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contracts were found to be setting new addresses without proper validations for zero addresses.
Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.
Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Variables and Line Numbers

- Multivesting.sol - Line 28 - IERC20 _token
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L28
- Multivesting.sol - Line 33 - address _beneficiary
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L33
- Multivesting.sol - Line 66 - address _receiver
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L66
- Singlevesting.sol - Line 84 - address token
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L84

```
    constructor(IERC20 _token) {
        token = _token;
        owner = msg.sender;
    }

     function addVesting(address _beneficiary, uint256 _amount, uint256
_cliff, uint256 _duration) public onlyOwner {}

    function release(address token) public virtual {}

    function release(address token) public virtual {}
```

Impacts

If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation

Add a zero address validation to all the functions where addresses are being set.

Retest

All the mentioned function have zero address validation implemented.

Bug ID#10 [Fixed]

Gas Optimization in _cliff Validation

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
The contract **Multivesting.sol** was using calculations in the code to find the number of seconds in 2 years.
This could be optimized by directly calculating and using the value to save some gas.

Affected Variables and Line Numbers

- Multivesting.sol - https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L35

```
    require(_cliff <= 60 * 60 * 24 * 365 * 2, "Cliff cannot be more than
2 years.");
```

Impacts
Alowing the code to calculate the value which is static costs some extra gas.

Remediation
It is recommended to hard-code the value instead of calculations and mention in the comments that it refers to “2 years” to notify the users.

```
require(_cliff <= 63072000, "Cliff cannot be more than 2 years."); //2
```

```
years
```

Retest

The validation in **_cliff** has been updated to "**63072000**".

Bug ID#11 [Fixed]

Missing Validations in Timestamp and Duration [Singlevesting]

Vulnerability Type
Missing Input Validation

Severity
Medium

Description:
The contract **Singlevesting.sol** was accepting the start timestamp and the duration for the vesting. However, there's no input validation on the parameters allowing the timestamp to be set in the past and the duration to be infinite or 0.

Affected Variables and Line Numbers

- Singlevesting.sol - Line 39, 40
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Singlevesting.sol#L39-L40

```
  constructor(address beneficiaryAddress, uint64 startTimestamp, uint64
durationSeconds) payable {
            require(beneficiaryAddress != address(0), "VestingWallet:
beneficiary is zero address");
       _beneficiary = beneficiaryAddress;
       _start = startTimestamp;
       _duration = durationSeconds;
   }
```

Impacts
If there's no validation on the start timestamp the release amount could be incorrect and erroneous.

Remediation

It is recommended to have input validation on the start timestamp so that it can not be set in the past.

The "**_duration**" should also have an input validation where it is set within an acceptable range and not set to 0.

Retest

Validations have been added in the timestamp and duration parameters.

Bug ID#12 [Fixed]

Missing Validations in Duration [Multivesting]

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
The contract **Multivesting.sol** was accepting the duration for the vesting in the function . However, there's no input validation on the parameter allowing the duration to be set to be infinite or 0.

Affected Variables and Line Numbers

- Multivesting.sol - Line 33
  https://github.com/zenland-dao/token_contracts/blob/135d4a282756277448b5b15323be9febd667b8e0/Multivesting.sol#L33

```
    function addVesting(address _beneficiary, uint256 _amount, uint256
_cliff, uint256 _duration) public onlyOwner {}
```

Impacts
If the duration is set to 0 it might result in incorrect release amount calculations.

Remediation
The "**_duration**" should have an input validation where it is set within an acceptable range and not set to 0.

Retest
Input validation has been added in the duration parameter.

Bug ID#13 [Fixed]

Missing NatSpec Comments

Vulnerability Type
Missing best practices

Severity
Informational

Description
Solidity contracts use a special form of comments to document code. This special form is named the Ethereum Natural Language Specification Format (NatSpec).
The document is divided into descriptions for developers and end-users along with the title and the author.
The contracts were missing NatSpec comments in the code which makes it difficult for the auditors and future developers to understand the code.

Affected Code

- Multivesting.sol

Impacts
Missing NatSpec comments and documentation about a library or a contract affect the audit and future development of the smart contracts.

Remediation
Add necessary NatSpec comments inside the contracts along with documentation specifying what it's for and how it's implemented.

Retest
NatSpec comments have been added to multivesting contract.

Bug ID#14 [Fixed]

Business Logic Issue in Vesting Duration

Vulnerability Type
Business Logic

Severity
High

Description
The Singlevesting.sol contract requires users to vest any tokens into the contract and the release timer starts according to the duration specified in the constructor.
Users are also allowed to deposit multiple ERC20 tokens into the contract.
There is an improper logic where if a user deposits multiple ERC20 tokens, the duration is always calculated from the initial deposit time of the first token.
This allows them to reduce the staking time for all the other tokens in the contract by depositing a fake ERC20 token initially.

Affected Code
- Singlevesting.sol

Impacts
This business logic issue allows users to reduce the vesting time for their tokens by depositing a fake ERC20 token initially and when the vesting duration gets over, they can deposit the actual token to release them immediately.

Remediation
It is recommended to have a duration calculation from the deposit time of each token into the contract so the vesting time period is constant for every token.

Retest
A fix has been added to return extra tokens back to the user. Also this was partially working as intended.

Bug ID#15 [Fixed]

Locked Ether

Vulnerability Type
Locked Ether

Severity
High

Description
The Singlevesting.sol contract has a payable constructor that can accept ETH at the time of deployment. The comment mentions that the contract is also able to handle ETH vesting but there's no such logic.
Due to this, if a user deposits ETH during the contract's deployment, their ETH will be locked forever in the contract since there's no function to withdraw the Ether.

Affected Code
- Singlevesting.sol

Impacts
This vulnerability could lead to locked ETH in the contract.

Remediation
It is recommended to implement the logic for ETH vesting if it's required, or a function to safely transfer all the ETH to the user/beneficiary.
If this is not required, the consider removing the "payable" keyword from the constructor.

Retest
This has been fixed. Constructor is not payable.

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.