Audited by

CredShields

Smart Contract Audit

February 21st, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Artulabs Limited between February 3rd, 2025, and February 14th, 2025. A retest was performed on February 18th, 2025.

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

Artulabs Limited

Table of Contents

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

Artulabs Limited engaged CredShields to perform a smart contract audit from February 3rd, 2025, to February 14th, 2025. During this timeframe, 16 vulnerabilities were identified. **A retest was performed on February 18th, 2025, 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 "Artulabs Limited" and should be prioritized for remediation.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Artu Contracts	0	1	2	5	1	7	**16**
	0	**1**	**2**	**5**	**1**	**7**	**16**

Table: Vulnerabilities Per Asset in Scope

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

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

Artulabs Limited engaged CredShields to perform the Artu 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 February 3rd, 2025, to February 14th, 2025, was agreed upon during the preparation phase.

2.1.1 Scope

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

IN SCOPE ASSETS
https://github.com/artacle/artu-smartcontract/tree/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b

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

Artulabs Limited 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, 16 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
The mint() and mintBridge() functions always fails due to inconsistent maxCap validation	High	Logical Error
Failure to check transfer return value can lead to unnoticed withdrawal failures	Medium	Inadequate Error Handling for Low-Level Calls
Inconsistent timestamp comparison can allow invalid token minting	Medium	Inconsistent Time Validation Between Solana and Solidity Contracts
Floating and outdated pragma	Low	Floating Pragma
Missing events	Low	Missing Best Practices
Use Ownable2Step	Low	Missing Best Practices
Missing zero address validations	Low	Missing Input Validation
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing best practices

Redundant supply validation in constructor	Informational	Redundant Code
Public constants can be private	Gas	Gas Optimization
Unused Imports	Gas	Gas Optimization
Cheaper conditional operators	Gas	Gas Optimization
Gas optimization for state variables	Gas	Gas Optimization
Cheaper inequalities in if()	Gas	Gas Optimization
Gas optimization in increments	Gas	Gas Optimization
Cheaper inequalities in require()	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Vulnerable	Bug ID #4
SWC-103	Floating Pragma	Vulnerable	Bug ID #4
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 ------------------

Artulabs Limited is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on February 18th, 2025, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
The mint() and mintBridge() functions always fails due to inconsistent maxCap validation	High	**Fixed** [Feb 18, 2025]
Failure to check transfer return value can lead to unnoticed withdrawal failures	Medium	**Fixed** [Feb 18, 2025]
Inconsistent timestamp comparison can allow invalid token minting	Medium	**Fixed** [Feb 18, 2025]
Floating and outdated pragma	Low	**Fixed** [Feb 18, 2025]
Missing events	Low	**Fixed** [Feb 18, 2025]
Use Ownable2Step	Low	**Fixed** [Feb 18, 2025]
Missing zero address validations	Low	**Fixed** [Feb 18, 2025]
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed** [Feb 18, 2025]
Redundant supply validation in constructor	Informational	**Fixed** [Feb 18, 2025]
Public constants can be private	Gas	**Fixed** [Feb 18, 2025]
Unused Imports	Gas	**Fixed** [Feb 18, 2025]
Cheaper conditional operators	Gas	**Fixed** [Feb 18, 2025]

Gas optimization for state variables	Gas	**Fixed** [Feb 18, 2025]
Cheaper inequalities in if()	Gas	**Fixed** [Feb 18, 2025]
Gas optimization in increments	Gas	**Fixed** [Feb 18, 2025]
Cheaper inequalities in require()	Gas	**Fixed** [Feb 18, 2025]

Table: Summary of findings and status of remediation

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

Bug ID #1 [Fixed]

The mint() and mintBridge() functions always fails due to inconsistent maxCap validation

Vulnerability Type
Logical Error

Severity
High

Description
The vulnerability arises from a logical flaw in the mint() function, specifically related to the maxCap and totalSupply() values. In the constructor of the contract, the maxCap is initialized to the supply parameter, and the entire supply is minted to the contract creator (admin). However, the mint function contains a require statement that checks if totalSupply() + amount <= maxCap. Since the maxCap is initialized to the same value as the totalSupply at the time of contract creation, this condition will always evaluate to false for any further minting attempts, as the totalSupply will already equal maxCap. This results in the minting operation always failing with the message "Supply Reached."
The condition, therefore, effectively locks the minting process, even though the contract may be designed to allow minting under certain conditions.
The same above condition goes with mintBridge as well.

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L100
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L121

Impacts
The primary consequence of this vulnerability is that the minting functionality is permanently disabled after the initial minting to the admin account. Since the contract's design is based on minting additional tokens to users or accounts, the failure of the mint function prevents the contract from fulfilling its intended functionality.

Remediation

To resolve this issue, the contract logic should be adjusted to allow minting after the initial supply is set.

Retest

This issue has been fixed by mining initialSupply amount instead of maxCap amount.

Bug ID #2 [Fixed]

Failure to check transfer return value can lead to unnoticed withdrawal failures

Vulnerability Type
Inadequate Error Handling for Low-Level Calls

Severity
Medium

Description
In the contract, the function withdrawFunds() is designed to allow the contract owner to withdraw the entire balance of the contract and transfer it to a specified wallet address. The implementation of the function uses a low-level transfer call to send funds from the contract to the provided wallet address.

However, there is a critical issue in the way the return value of the transfer call is handled. The transfer function in Solidity returns a boolean indicating whether the transfer was successful or not, but in this case, the return value is not checked.
This omission means that any failure in the transfer operation, such as insufficient gas, a revert triggered by the receiver contract, or a failure due to other unforeseen conditions, will go undetected.

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L207

Impacts
If the transfer fails and the contract owner is unaware, the funds may remain locked in the contract indefinitely, leading to an effective denial of service. The failure to handle the return value of the low-level call means that there is no fallback mechanism to alert the contract owner about an unsuccessful withdrawal attempt, which could result in significant financial loss.

Remediation
The best practice in this scenario is to always check the return value of low-level calls, including transfer, call, and send.
The updated code should look as follows:

```
function withdrawFunds(address wallet) external onlyRole(ROLE_OWNER) {
 uint256 balanceOfContract = address(this).balance;
require(payable(wallet).transfer(balanceOfContract), "Transfer failed");
}
```

Retest

This issue has been fixed by using a call function with return value check.

Bug ID #3 [Fixed]

Inconsistent timestamp comparison can allow invalid token minting

Vulnerability Type
Inconsistent Time Validation Between Solana and Solidity Contracts

Severity
Medium

Description

The vulnerability arises due to the differing timestamp comparison logic between the Solidity and Solana implementations of the bridgeMint and mint_bridge_tokens functions. Both functions aim to ensure that a transaction is valid by checking whether the provided nonce (which represents an expiry timestamp) has passed. However, the comparison logic used in each contract differs significantly.

In the Solidity contract, the condition require(nonce > block.timestamp, "Nonce expired") ensures that the provided nonce must be strictly greater than the current block's timestamp. This means that once the timestamp reaches the provided nonce value, the transaction would fail, effectively disallowing any transaction that occurs when the timestamp exactly equals the nonce.

On the other hand, the Solana contract uses the comparison require!(current_timestamp<=nonce,error::ErrorCode::TimestampExpired), which allows the transaction to succeed if the current timestamp is less than or equal to the nonce. This difference in behavior allows transactions on Solana to succeed even when the current timestamp equals the nonce, which is not permitted in the Solidity implementation.

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/solana/token_contract/programs/token_program/src/lib.rs#L129
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L119
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L68
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/solana/airdrop/programs/airdrop/src/lib.rs#L58

Impacts
The main impact of this vulnerability is the risk of inconsistent behavior between the two chains. Since the timestamp validation in the Solidity contract is stricter than in the Solana contract, users or systems relying on both chains may experience inconsistencies in the minting process

Remediation
To resolve this inconsistency and ensure uniform behavior across both the Solidity and Solana contracts, the timestamp comparison logic should be aligned. One potential solution is to modify the solidity smart contract to match the Solana contract's logic of requiring the nonce to be greater than equal to(>=) the current timestamp.

Retest
This issue has been fixed.

Bug ID #4 [Fixed]

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., ^0.8.20. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L2
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L2
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L2

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 low.

Remediation
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.28 pragma version

Reference: https://swcregistry.io/docs/SWC-103

Retest

This issue has been fixed by removing the floating pragma version.

Bug ID #5 [Fixed]

Missing events

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

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L58-L60
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L125-L140
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L158-L172
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L189-L203
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L205-L209
-

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 important functions to keep track of them.

Retest

This issue has been fixed by emitting events in the functions.

Bug ID #6 [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://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L10

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 as recommended.

Bug ID #7 [Fixed]

Missing 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 Code
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L29-L34
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L33-L64
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L30-L38

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
This issue has been fixed by implementing zero address check.

Bug ID #8 [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://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L82
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L137
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L201
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L58
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L88
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L170

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 as recommended.

Bug ID #9 [Fixed]

Redundant supply validation in constructor

Vulnerability Type
Redundant Code

Severity
Informational

Description
The constructor of the contract includes a redundant validation check: require(supply <= maxCap, "Cannot mint more than maximum supply");. This check is unnecessary because maxCap is explicitly set to supply just before the require statement. Since both values are always the same, the condition will never evaluate to false, making the validation redundant and introducing minor inefficiencies in contract execution.

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L48

Impacts
While this issue does not introduce a security vulnerability, it unnecessarily increases gas costs during contract deployment. Any redundant computation within a smart contract should be avoided to optimize efficiency. Removing this validation check would slightly reduce gas consumption and improve contract readability without affecting functionality.

Remediation
To improve efficiency and maintain cleaner code, the redundant validation should be removed from the constructor.

Retest
This issue has been fixed by updating the require() condition check.

Bug ID #10 [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

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L15-L18
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L19-L22
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L16
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L17
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L10-L13
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L14-L17
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L25
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L26

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

This issue has been fixed by updating variables as recommended.

Bug ID #11 [Fixed]

Unused Imports

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract Airdrop.sol was importing contracts Strings.sol which was not used anywhere in the code. This increases the gas cost and overall contract's complexity.

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L5

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:
This issue has been fixed by removing the unused imports library.

Bug ID #12 [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://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L69
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L80
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L120

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 as recommended.

Bug ID #13 [Fixed]

Gas optimization for state variables

Vulnerability Type
Gas Optimization

Severity
Gas

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

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L60

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

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

Retest
This issue has been fixed as recommended.

Bug ID #14 [Fixed]

Cheaper inequalities in if( )

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

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

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L80
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L114
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L126
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L152

Impacts
Using strict inequalities inside "if" statements costs more gas.

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

Retest:
This issue has been fixed as recommended.

Bug ID#15 [Fixed]

Gas optimization in increments

Vulnerability Type
Gas optimization

Severity
Gas

Description
The contract uses two for loops**,** which use post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Vulnerable Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L66
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L98

Impacts
Using **i++** instead of **++i** costs the contract deployment around 600 more gas units.

Remediation
It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest
This issue has been fixed as recommended.

Bug ID #16 [Fixed]

Cheaper inequalities in require( )

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

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

Affected Code

- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L71
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Airdrop.sol#L134
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Vesting.sol#L82
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L48
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L93
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L100
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L105
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L121
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L185
- https://github.com/artacle/artu-smartcontract/blob/0bad32dbdd0f2d2effabb2ee264e9f40fcb80e1b/contracts/Artu.sol#L198

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 wherever possible as recommended.

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

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

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com