



CredShields Smart Contract Audit

October 25th, 2022 • CONFIDENTIAL

Description

This document details the process and result of the Phase 2 Smart-Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Capx Global between Oct 15th, 2022, and Oct 21st, 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Capx Global

Table of Contents

1. Executive Summary	2
State of Security	3
2. Methodology	4
2.1 Preparation phase	4
2.1.1 Scope	4
2.1.2 Documentation	5
2.1.3 Audit Goals	8
2.2 Retesting phase	9
2.3 Vulnerability classification and severity	9
2.4 CredShields staff	11
3. Findings	12
3.1 Findings Overview	12
3.1.1 Vulnerability Summary	13
3.1.2 Findings Summary	14
3.1.3 Testing and Observations:	17
4. Remediation Status	26
5. Bug Reports	27
Multiple Zero Address Validations Missing	27
6. Appendix 1	30
6.1 Files in scope:	29
6.2 Disclosure:	35

1. Executive Summary

Capx Global engaged CredShields to perform a Phase 2 smart contract audit from October 15th, 2022, to October 21st, 2022, after introducing new features and refactoring the code. During this timeframe, One (1) vulnerability was identified. **A retest was performed by the Credshields team between 19th Oct 2022 to 21st Oct 2022, and all the vulnerabilities were found to be fixed.**

During the audit, zero (0) vulnerability was found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Capx Global" 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	Σ
Capx Global	0	0	0	0	0	0
	0	0	0	0	0	0

Table: Post remediation per asset

The CredShields team conducted the Phase 2 security audit to focus on identifying vulnerabilities in Capx Global's scope after the new updates in the code during the testing window while abiding by the policies set forth by "Capx Global."

State of Security

Maintaining a healthy security posture requires constant review and refinement of existing security processes. Running a CredShields continuous audit allows “Capx Global” internal security team and development team to not only uncover specific vulnerabilities but gain a better understanding of the current security threat landscape.

We recommend running regular security assessments to identify any vulnerabilities introduced after Capx Global introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate “Capx Global” internal development and security teams and allow manual or automated procedures to be put in place to eliminate entire classes of vulnerabilities in the future. This proactive approach helps contribute to future-proofing the security posture of Capx Global assets.

2. Methodology

Capx Global engaged CredShields to perform a Phase 2 smart contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

CredShields team read all the provided documents and comments in the smart-contract code to understand the contract's features and functionalities. The team reviewed all the functions and prepared a mind map to review possible security vulnerabilities in the order of the function with more critical and business-sensitive functionalities for the refactored code.

The team deployed a self-hosted version of the smart contract to verify the assumptions and validation of the vulnerabilities during the audit phase.

A testing window from Oct 15th, 2022, to Oct 21st, 2022, 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/capx-Global/Smart-Vesting <ul style="list-style-type: none">- Controller.sol- ERC20CloneContract.sol- ERC20Factory.sol

<ul style="list-style-type: none">- Master.sol- UUPSUpgradeable.sol- Vesting.sol- timestampToDateLibrary.sol

Table: In-scope assets

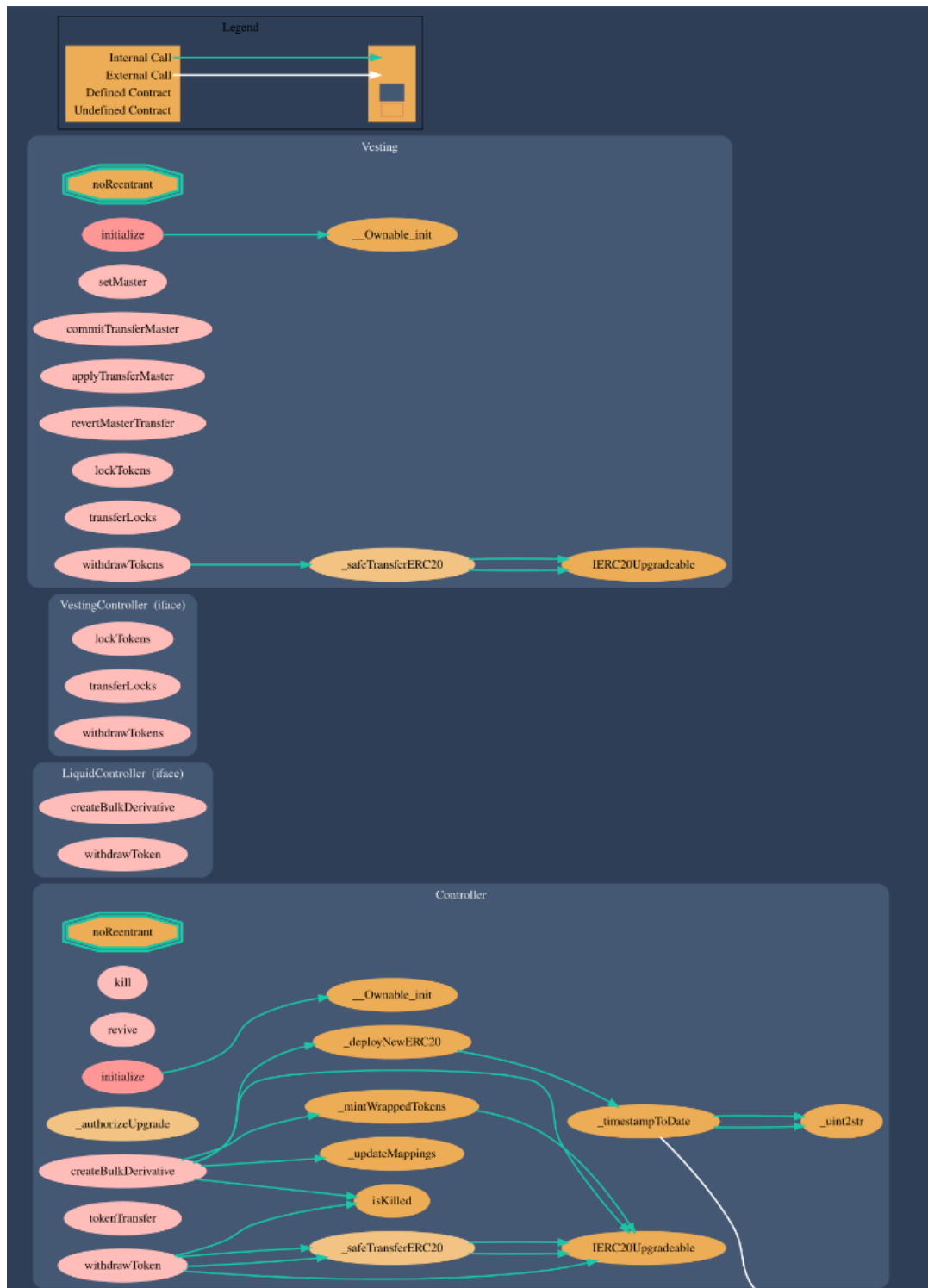
2.1.2 Documentation

The following documentation was available to the audit team.

<https://github.com/Capx-Global/Smart-Vesting/tree/master/ContractDocumentation>

The audit team also mapped all the user flows and privileges and mapped new functionalities. The control flow graph can be found below

Control Flow Graph:



2.1.3 Audit Goals

CredShields' methodology uses individual tools and methods; however, tools are just used for aids. The majority of the audit methods involve manually reviewing the smart contract source code. The team followed the standards of the [SWC registry](#) for testing along with an extended self-developed checklist based on industry standards, but it was not limited to it. The team focused heavily on understanding the core concept behind all the functionalities along with preparing test and edge cases. Understanding the business logic and how it could have been exploited.

The audit's focus was to verify that the smart contract system is secure, resilient, and working according to its specifications. Breaking the audit activities into the following three categories:

- **Security** - Identifying security-related issues within each contract and the system of contracts.
- **Sound Architecture** - Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
- **Code Correctness and Quality** - A full review of the contract source code. The primary areas of focus include:
 - Correctness
 - Readability
 - Sections of code with high complexity
 - Improving scalability
 - Quantity and quality of test coverage

2.2 Retesting phase

Capx Global is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

Discovering vulnerabilities is important, but estimating the associated risk to the business is just as important.

To adhere to industry guidelines, CredShields follows OWASP's Risk Rating Methodology. This is calculated using two factors - **Likelihood** and **Impact**. Each of these parameters can take three values - **Low**, **Medium**, and **High**.

These depend upon multiple factors such as Threat agents, Vulnerability factors (Ease of discovery and exploitation, etc.), Technical and Business Impacts. The likelihood and the impact estimate are put together to calculate an overall severity for the risk.

CredShields also define an **Informational** severity level for vulnerabilities that do not align with any of the severity categories and usually have the lowest risk involved.

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 believe in the importance of technical excellence and pay a great deal of attention to its details. Our coding guidelines, practices, and standards help ensure that our software is stable and reliable.

Informational vulnerabilities should not be a cause for alarm but rather a chance to improve the quality of the codebase by emphasizing readability and good practices. They do not represent a direct risk to the Contract but rather suggest improvements and the best practices that can not be categorized under any of the other severity categories.

Code maintainers should use their own judgment as to whether to address such issues.

2. Low

Vulnerabilities in this category represent a low risk to the Smart Contract and the organization. The risk is either relatively small and could not be exploited on a recurring basis, or a risk that the client indicates is not important or significant, given the client's business circumstances.

3. Medium

Medium severity issues are those that are usually introduced due to weak or erroneous logic in the code.

These issues may lead to exfiltration or modification of some of the private information belonging to the end-user, and exploitation would be detrimental to the client's reputation under certain unexpected circumstances or conditions. These conditions are outside the control of the adversary.

These issues should eventually be fixed under a certain timeframe and remediation cycle.

4. High

High severity vulnerabilities represent a greater risk to the Smart Contract and the organization. These vulnerabilities may lead to a limited loss of funds for some of the end-users.

They may or may not require external conditions to be met, or these conditions may be manipulated by the attacker, but the complexity of exploitation will be higher.

These vulnerabilities, when exploited, will impact the client's reputation negatively.

They should be fixed immediately.

5. Critical

Critical issues are directly exploitable bugs or security vulnerabilities. These issues do not require any external conditions to be met.

The majority of vulnerabilities of this type involve a loss of funds and Ether from the Smart Contracts and/or from their end-users.

The issue puts the vast majority of, or large numbers of, users' sensitive information at risk of modification or compromise.

The client's reputation will suffer a severe blow, or there will be serious financial repercussions.

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, one (1) security vulnerability was identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC Vulnerability Type
Multiple Zero Address Validation Missing	Low	Improper Input Validation

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 v0.8.4)
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 0.8.4 is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses fixed pragma v0.8.4
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used anywhere in the code
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	<p>The contract uses require & assert statements or modifier functions needed in case of accessing functions.</p> <p>require is checking if a wrapped token is invalid and if withdraw time is before the vest time while token withdrawal.</p>

			<p>noReentrant prevents reentrancy.</p> <p>A check is implemented to stop the withdrawal if the contract is killed.</p>
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	<code>selfdestruct()</code> is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	Notable functions such as <code>createBulkDerivative</code> , <code>withdrawToken</code> , <code>lockTokens</code> , <code>transferLocks</code> , <code>withdrawTokens</code> , etc, use reentrancy guard
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, <code>v0.5.0</code>
SWC-110	Assert Violation	Not Vulnerable	Asserts are used properly.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like <code>block.blockhash()</code> , <code>msg.gas</code> , <code>throw</code> , <code>sha3()</code> , <code>callcode()</code> , <code>suicide()</code> , <code>constant</code> are in use

SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	The smart contract safely uses UUPS Proxy. OnlyProxy() is implemented and all the state variables are unaffected after the upgrade
SWC-113	DoS with Failed Call	Not Vulnerable	The logic is implemented in such a way that the external calls if they fail, they fail gracefully, i.e. without having a major impact on the functionality of the contract. External calls are not being made to other contracts.
SWC-114	Transaction Order Dependence	Not Vulnerable	Contract uses SafeERC20
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 used which is better than block.number and doesn't affect for longer duration which is common for vesting logics.
SWC-117	Signature Malleability	Not Vulnerable	ecrecover function is not used anywhere in the code

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	ecrecover is not used anywhere in the code
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	The contract uses a check on the length of the arrays.

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 unused variables were found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	<code>abi.encodePacked()</code> is in use but not exploitable because date can't be controlled in the caller
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

3.1.3 Testing and Observations:

1. Solidity Versions and Pragma:

The contract uses 0.8.4 which is time tested and a stable version.

2. Uses of modifiers:

The contract uses modifiers only to implement checks and does not make state changes. The observed modifiers in use were OpenZeppelin's modifiers such as *noReentrant*, *onlyOwner*, *onlyProxy*, *initializer*, etc., and were used properly.

3. Controlled `delegateCall`:

delegateCall() or *callcode()* to an address controlled by the user allows execution of malicious contracts in the context of the caller's state. A contract needs to ensure trusted destination addresses for such calls. The UUPS proxy is implemented securely and *onlyProxy()* is implemented and all the state variables are unaffected after the upgrade.

4. Reentrancy vulnerabilities:

Untrusted external contract calls could callback lead to unexpected results such as multiple withdrawals or out-of-order events. Use check-effects-interactions pattern or reentrancy guards. The contract uses *noReentrant* modifier on all sensitive functions like *createBulkDerivative()*, *withdrawWrappedVestingToken()*, *withdrawVestingLock()*, *transferVestingLock()*, *createBulkDerivative()*, *withdrawToken()*, *lockTokens()*, *transferLocks()*, *withdrawTokens()*

The application also uses *safeERC20* transfers rather than *transfer()* and *send()*. Also, the ERC777 callbacks and hooks are not used.

5. Private on-chain data:

Marking variables private does not mean that they cannot be read on-chain. Private data should not be stored unencrypted in contract code or state but instead stored encrypted or off-chain. Fortunately, no sensitive unencrypted data is stored on the chain.

6. Block values as time proxies and weak PRNG's:

block.timestamp is used for vesting periods demarcation; however, *block.timestamp* is only exploitable for a short duration when the timing is around 15 seconds because of issues

with synchronization, miner manipulation, and changing block times. The contract ensures that the vesting time is in the future and more than one day from the current time.

7. Integer overflow/underflow:

The contract uses solidity version 0.8.0+, which auto-handles overflows and underflows.

8. Divide before multiply:

Performing multiplication before the division is generally better to avoid loss of precision because Solidity integer division might truncate. The contract uses it at line [168](#) & [198](#). However, it is used to normalize the vesting time period and hence has no impact.

9. Transaction order dependency:

No such vulnerability was found related to a race condition or transaction order dependency. The contract avoids the use of *approve()* and does not make assumptions about transaction orders.

10. Signature malleability:

ECRecover function is not used.

11. Dangerous strict equalities:

The use of strict equalities with tokens/Ether can accidentally/maliciously cause unexpected behavior. Consider using `>=` or `<=` instead of `==` for such variables depending on the contract logic. The contract just uses strict equalities while minting, which is necessary as the token minted should be equal.

12. Dangerous usage of *tx.origin*:

The contract does not rely on *tx.origin* anywhere.

13. Deleting a *mapping* within a *struct*:

Deleting a struct that contains a mapping will not delete the mapping contents, which may lead to unintended consequences. The contract does use mapping. However, there was no delete functionality.

14. Tautology, contradiction, and boolean handling:

All the `&&` and `||` statements were carefully evaluated to check if they would always render as true or false due to faulty logic. However, none were found.

The use of Boolean constants (true/false) in code (e.g., conditionals) is indicative of flawed logic; however, no boolean constants were found.

15. State-modifying functions:

Functions that modify the state (in assembly or otherwise) but are labeled constant/pure/view revert in solc $\geq 0.5.0$ (work in prior versions) because of the use of `STATICCALL` opcode. We carefully reviewed the contract and noticed

- *pure* is used only in the timestamp library
- *view* is used in *isKilled()*, *symbol()*, *name()*, *decimals()*, *getFactory()*, *getProposal()* and is not modifying the state

Hence no such concerns were found.

16. Low-level calls and threats with inline assembly:

No low-level calls, inline assembly, shift operators like `(shl(x,y))`, or arbitrary “jump” with functions were in use.

17. Dangerous shadowing:

No instance of dangerous shadowing of state variables was found.

18. Dangerous loops and array inconsistency:

Costly operations or infinite operation inside a loop may lead to out-of-gas errors. The contract does use loops but limits the loop entries to a length of 300 entries - Line [181](#)

The arrays are properly checked for inconsistency when multiple array inputs are used for a function call.

19. Missing events and unindexed event parameters:

Events are present for all important function calls with indexed attributes if required, and they are emitted. None of the events are without *emit* calls.

20. Dangerous unary expressions:

Unary expressions such as `x += 1` are likely errors where the programmer meant to use `x += 1`. Unary `+` operator was deprecated in solc v0.5.0. These operators are used in the contract, but they are correctly used.

21. Critical address change:

The application ensures two step process using commit and apply scheme to make sure critical address change doesn't happen directly and the owner gets a chance to revert if anything goes wrong. Also a time lock is added to the commit scheme. Line [94-111 \(vesting.sol\)](#) & Line [147-177 \(Master.sol\)](#)

22. assert()/require() state change:

Invariants in `assert()` and `require()` statements should not modify the state per best practices. The contract safely uses the `assert`.

23. Deprecated keywords:

Use of deprecated functions/operators such as `block.blockhash()` for `blockhash()`, `msg.gas` for `gasleft()`, `throw` for `revert()`, `sha3()` for `keccak256()`, `callcode()` for `delegatecall()`, `suicide()` for `selfdestruct()`, `constant` for `view` or `var` for *actual type name* should be avoided to prevent unintended errors with newer compiler versions. No deprecated keywords were used in the code.

24. Missing or incorrect inheritance ordering:

Contracts inheriting from multiple contracts with identical functions should specify the

correct inheritance order. However, no such case was observed. It is also important to not make spelling mistakes or miss out on inheritance. The code was carefully reviewed, and everything was found to be in place.

25. Hash collisions with multiple variable-length arguments:

Using *abi.encodePacked()* with multiple variable-length arguments can, in certain situations, lead to a hash collision. Do not allow users access to parameters used in *abi.encodePacked()*, use fixed-length arrays or use *abi.encode()*. (see [here](#) and [here](#))

The contract did use *abi.encodePacked()*. However, the situation was not exploitable as the date was concatenated along with the token address at lines [339](#), [346](#), [367](#), [374](#) and [526](#) (**Controller.sol**)

26. Constant state variables:

Constant state variables should be declared constant to save gas which was implemented in the contract.

27. Security issues with variables:

No long number of literals were used, which can cause confusion and possible business logic errors. No similar variable names are used as they can cause confusion.

28. Uninitialized state/local variables:

The variables were initialized wherever required, unless default values were expected.

29. Uncalled public functions:

Public functions that are never called from within the contracts should be declared *external* to save gas. However, no such functions were found.

30. ERC20 decimals return a uint8:

Several tokens incorrectly return a uint256. If this is the case, ensure the value returned is below 255. The contract (**controller.sol**) at line [354](#) & [382](#) uses *uint8*, and hence there is no vulnerability here.

31. ERC20 name, decimals, and symbol functions:

These functions are optional in the ERC20 standard and might not be present. However, the contract has all three in use.

32. System documentation:

Capx team has well-written documentation and ensures that the entire system's roles, functionalities, and interactions are well documented to the greatest detail possible.

33. Input validation:

- *setLiquidFactory, upgradeTo, upgradeToAndCall* - 0 address not allowed
- *withdrawToken* -
 - Cannot withdraw before vest time
 - 0 address is checked
- *tokenTransfer* - 0 address is checked
- *assetAddressesToProjectOwner* - validations are in place to make *msg.sender* the owner and 0 address is checked.
- *vestingTimeOfTokenId* - should be less than *block.timestamp* is in place
- *lockTokens* -
 - *_amount* array length should be exactly 2
 - *_withdrawalAddress* array length cannot be more than 100
 - All array length match is done.
- *createBulkDerivative* -
 - *_name* = 2 to 26 length limit
 - *_documentHash* - should be of length 46
 - *_tokenAddress* - 0 address is checked
 - length check between arrays is ensured
 - *_distTime* is greater than the current time and should be more than 1 day.
(not allowed to be in past)

- `_distAddress` cannot be 0 address as it minting is not allowed to it.
- Array length cannot be more than 300

34. Zero Address Check:

Functions that are setters for critical addresses should always use a zero address check to prevent the loss and burn of funds. This could also lead to lack of ownership if the addresses are wrong.

The following functions were using zero address checks -
setLiquidFactory, upgradeTo, upgradeToAndCall

The following Address variables lacked zero address checks -






















- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/ERC20Factory.sol#L28> (`_implementaiton, _controller`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/ERC20Factory.sol#L35> (`_lender`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Master.sol#L164> (`_newProposal`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/4ac8a5dd3e687c0ef6c899eb5da60b26c7277a24/contracts/Master.sol#L147> (`_newFactory`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Vesting.sol#L94> (`_newMaster`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/4ac8a5dd3e687c0ef6c899eb5da60b26c7277a24/contracts/ERC20CloneContract.sol#L76> (`_minter`)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Controller.sol#L164> (`_caller`)

35. ByteCode verification:

Bytecode-level checking helps to ensure that the code behaves correctly for all input values. In this audit, we used [Mythx](#) deep analysis to verify a small number of basic

properties on the weight rebalancing and conversion functions and to detect conditions that would cause runtime exceptions.

MythX uses symbolic execution and input fuzzing to explore a large number of possible inputs and program states. The team performed the assessment of all the alerts by the scanner, and flags were found to be false positives.

61b9885ec779...	Standard	15/12/2021, 11:47:02 AM	StorageSlotUpgradeable.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b9885d310d...	Standard	15/12/2021, 11:47:01 AM	Master.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b9885b4406...	Standard	15/12/2021, 11:46:59 AM	timestampToDateLibrary.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b98856310d...	Standard	15/12/2021, 11:46:56 AM	ERC20CloneContract.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b98856957d...	Standard	15/12/2021, 11:46:54 AM	AddressUpgradeable.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b988542970...	Standard	15/12/2021, 11:46:52 AM	Controller.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project
61b98851c779...	Standard	15/12/2021, 11:46:49 AM	Vesting.sol	  	In Progress: 0 Queued: 0 Finished: 1 Total: 1	PDF report Add to Project

4. Remediation Status

Capx Global is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. The table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDICATION STATUS
Multiple zero validations missing	Low	Fixed [21/10/2022]

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Multiple Zero Address Validations Missing

Vulnerability Type

Missing Input Validation

Severity

Low

Description:

Multiple Solidity contracts were found to be setting new addresses without proper validations for zero address.

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

- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/ERC20Factory.sol#L28> (_implementaiton, _controller)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/ERC20Factory.sol#L35> (_lender)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Master.sol#L164> (_newProposal)
- <https://github.com/Capx-Global/Smart-Vesting/blob/4ac8a5dd3e687c0ef6c899eb5da60b26c7277a24/contracts/Master.sol#L147> (_newFactory)
- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Vesting.sol#L94> (_newMaster)
- <https://github.com/Capx-Global/Smart-Vesting/blob/4ac8a5dd3e687c0ef6c899eb5da60b26c7277a24/contracts/ERC20CloneContract.sol#L76> (_minter)

- <https://github.com/Capx-Global/Smart-Vesting/blob/master/contracts/Controller.sol#L164>
(_caller)

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 has been fixed and zero address checks are implemented on all the functionalities and line numbers mentioned above.

6. Appendix 1











6.1 Files in scope:

Contracts Description Table:









Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
ERC20Properties	Interface			
	symbol	External !		NO !
	name	External !		NO !
	decimals	External !		NO !
ERC20Clone	Interface			
	mintbyControl	External !	⛔	NO !
	burnbyControl	External !	⛔	NO !
Master	Interface			

	getFactory	External !		NO !
	getProposal	External !		NO !
AbsERC20Factory	Interface			
	createStorage	External !	⛔	NO !
Controller	Implementation	Initializable, UUPSUpgradable, OwnableUpgradable		
	isKilled	Internal 🔒		
	kill	External !	⛔	onlyOwner
	revive	External !	⛔	onlyOwner
	initialize	Public !	⛔	initializer
	_authorizeUpgrade	Internal 🔒	⛔	onlyOwner
	createBulkDerivative	External !	⛔	noReentrant
	_updateMappings	Internal 🔒	⛔	



	_safeTransferERC20	Internal 🔒	🛑	
	_deployNewERC20	Internal 🔒	🛑	
	_mintWrappedTokens	Internal 🔒	🛑	
	tokenTransfer	External !	🛑	NO !
	withdrawToken	External !	🛑	noReentrant
	_timestampToDate	Internal 🔒		
	_uint2str	Internal 🔒		
controllerProperties	Interface			
	tokenTransfer	External !	🛑	NO !
factoryLender	Interface			
	lender	External !		NO !

ERC20CloneContract	Implementation	Context, IERC20, IERC20Metadata		
	initializer	Public !		NO !
	name	Public !		NO !
	symbol	Public !		NO !
	decimals	Public !		NO !
	totalSupply	Public !		NO !
	balanceOf	Public !		NO !
	transfer	Public !		NO !
	allowance	Public !		NO !
	approve	Public !		NO !
	transferFrom	Public !		NO !
	increaseAllowance	Public !		NO !
	decreaseAllowance	Public !		NO !
	_transfer	Internal 		
	_mint	Internal 		

	_burn	Internal 🔒	🛑	
	mintbyControl	External !	🛑	NO !
	burnbyControl	External !	🛑	NO !
	_approve	Internal 🔒	🛑	
ERC20Clone	Implementatio n			
	initializer	Public !	🛑	NO !
ERC20Factory	Implementatio n	Ownable		
Vesting	Implementatio n	Initializable, UUPSUpgrade able, OwnableUpgr adeable		
	_authorizeUpgra de	Internal 🔒	🛑	onlyOwner
	initialize	Public !	🛑	initializer
	setMaster	External !	🛑	onlyOwner

	commitTransferMaster	External !		onlyOwner
	applyTransferMaster	External !		onlyOwner
	revertMasterTransfer	External !		onlyOwner
	_safeTransferERC20	Internal 		
	lockTokens	External !		noReentrant
	transferLocks	External !		noReentrant
	withdrawTokens	External !		noReentrant

Legend

Symbol	Meaning
	Function can modify state
	Function is payable

6.2 Disclosure:

The Reports neither endorse nor condemn any specific project or team, nor do they guarantee the security of any specific project. The contents of this report do not, and should not be interpreted as having any bearing on, the economics of tokens, token sales, or any other goods, services, or assets.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. There is no warranty or representation made by this report to any Third Party in regards to the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business.

In no way should a third party use these reports to make any decisions about buying or selling a token, product, service, or any other asset. It should be noted that this report is not investment advice, is not intended to be relied on as investment advice, and has no endorsement of this project or team. It does not serve as a guarantee as to the project's absolute security.

CredShields Audit team owes no duty to any third party by virtue of publishing these Reports.