CredShields

Smart Contract Audit

Oct 9th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Claim Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of StationX between Sept 18th, 2023, and Sept 26th, 2023. A retest was performed on Oct 6th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

StationX

Table of Contents

1. Executive Summary

StationX engaged CredShields to perform a Claim smart contract audit from Sept 18th, 2023, to Sept 26th, 2023. During this timeframe, Eleven (11) vulnerabilities were identified. **A retest was performed on Oct 6th, 2023, and all the bugs have been addressed.**

During the audit, One (1) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "StationX" 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
Claim Smart Contract	0	1	1	4	1	4	**11**
	0	**1**	**1**	**4**	**1**	**4**	**11**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

StationX engaged CredShields to perform a StationX Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from Sept 18th, 2023, to Sept 26th, 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/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/

Table: List of Files in Scope

2.1.2 Documentation

Below was the document provided by the StationX team for the audit of the claim contract.

https://stnx.notion.site/Contracts-in-scope-for-audit-77d3581e4b0b49dbaa38c723f8bcb974?pvs=4

2.1.3 Audit Goals

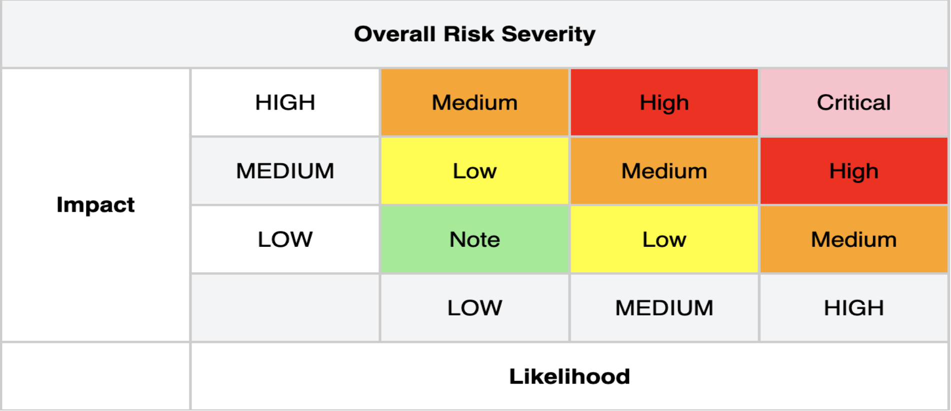
CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

StationX 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, Eleven (11) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Incorrect Validation for PendingClaimDetails in the claim()	High	Validation Logic Flaw
Unreturned Excess Fee in claim Function	Medium	Finance Logic Flaw
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Missing Events in Important Functions	Low	Missing Best Practices
Lack of validation for _newMaxClaimAmount in changeMaxClaimAmount()	Low	Lack of Input Validation
Lack of Validation for StartTime & EndTime in changeStartAndEndTime()	Low	Lack of Input Validation

Gas Optimization for State Variables	Gas	Gas Optimization
Array Length Caching	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas optimization
Require With Empty Message	Gas	Gas Optimization
Missing NatSpec Comments	Informational	Missing best practices

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

StationX is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Oct 6th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Incorrect Validation for PendingClaimDetails in the claim()	High	**Fixed [06/10/2023]**
Unreturned Excess Fee in claim Function	Medium	**Fixed [06/10/2023]**
Floating and Outdated Pragma	Low	**Fixed [06/10/2023]**
Missing Events in Important Functions	Low	**Fixed [06/10/2023]**
Lack of validation for _newMaxClaimAmount in changeMaxClaimAmount()	Low	**Fixed [06/10/2023]**
Lack of Validation for StartTime & EndTime in changeStartAndEndTime()	Low	**Fixed [06/10/2023]**
Gas Optimization for State Variables	Gas	**Fixed [06/10/2023]**
Array Length Caching	Gas	**Won’t Fix**

Gas Optimization in Increments	Gas	**Won’t Fix**
Require With Empty Message	Gas	**Fixed [06/10/2023]**
Missing NatSpec Comments	Informational	**Fixed [06/10/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Incorrect Validation for PendingClaimDetails in the claim()

Vulnerability Type

Validation Logic Flaw

Severity

High

Description

In the claim function, there is a validation check for PendingClaimDetails[msg.sender].length to ensure that it does not exceed a limit of 20. This check is intended to limit the number of pending claims that a user can have. However, the validation is based on msg.sender, which represents the user's address.

The issue here is that the PendingClaimDetails array is being used to store cooldown details, and the validation should be based on the _receiver address because the details are pushed into the _receiver array, not the msg.sender's array. As a result, if a user specifies a different _receiver address when making a claim, the validation based on msg.sender will always fail, even if the _receiver has not reached the limit of 20 pending claims.

Affected Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L128

Impacts

Users who specify a different _receiver address for their claims may encounter a "MaxReached" error even if the _receiver has not reached the limit of 20 pending claims. This could lead to an incorrect rejection of valid claims.

Remediation

To address this issue, you should update the validation check for the PendingClaimDetails array to be based on the _receiver address rather than msg.sender. This change will ensure that the validation accurately reflects the state of the _receiver's pending claims, regardless of whether a different _receiver address is specified when making a claim.

Retest

The function has been updated to validate the PendingClaimDetails for _receiver rather than msg.sender.

Bug ID#2 [Fixed]

Unreturned Excess Fee in claim Function

Vulnerability Type

Finance Logic Flaw

Severity

Medium

Description

In the claim function, the contract expects users to send a fee in Ether (msg.value) to cover the cost of the claim. If the sent fee (msg.value) is less than the required claim fee, the function reverts with an "InvalidAmount" error. However, there is no mechanism in place to refund any excess Ether sent by the user if the sent fee exceeds the required amount.This issue can lead to a loss of Ether for users who accidentally send more Ether than required for the claim, as the excess Ether is not returned to them

Vulnerable Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L78-L82
- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/ClaimFactory.sol#L48

Impacts

Users who accidentally send more Ether than required for the claim will lose the excess Ether, as it is not returned to them.

Remediation

To address this issue and improve user experience, you should implement a mechanism to refund any excess Ether sent by users when making a claim. This mechanism should ensure that users receive the exact amount of Ether required for the claim, and any excess Ether should be promptly returned to the sender.

Retest

The functions now revert if the user tries to send an incorrect fee amount. Care should be taken that the users know the fee amount and pass the exact value otherwise the function will always revert.

Bug ID #3 [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., ^0.8.7, ^0.8.16.

This allows the contracts to be compiled with all the solidity compiler versions above the limit specified.

Impacts

If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.

Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.

The likelihood of exploitation is really low therefore this is only informational.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.18 pragma version

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

Retest

The pragma version has been updated and fixed.

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 -

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L198-L228

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 After changing Merkle root in depositTokens()

Retest

The function depositTokens() has been updated to emit an event.

Bug ID #5 [Fixed]

Lack of validation for _newMaxClaimAmount in changeMaxClaimAmount()

Vulnerability Type

Lack of Input Validation

Severity

Low

Description

Allowing the maxClaimable value to be set to zero could lead to unexpected behavior within the contract. If this value is set to zero, it might effectively disable the ability to claim any amounts, which could disrupt the intended functionality of the contract.

Affected Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L263-L271

Impacts

Allowing the maxClaimable value to be set to zero could lead to unexpected behavior within the contract. If this value is set to zero, it might effectively disable the ability to claim any amounts, which could disrupt the intended functionality of the contract.

Remediation

To prevent this issue, you should add a validation check at the beginning of the function to ensure that _newMaxClaimAmount is not set to zero. If _newMaxClaimAmount is zero, you can choose to reject the change or handle it in a way that aligns with the intended behavior of your contract. Adding this validation will help ensure that the contract operates as expected and avoids potential disruptions caused by a zero maxClaimable value.

Retest

Input validation has been added for the _newMaxClaimAmount variable so that it can’t be set to 0.

Bug ID #6 [Fixed]

Lack of Validation for StartTime & EndTime in changeStartAndEndTime()

Vulnerability Type

Lack of Input Validation

Severity

Low

Description

The changeStartAndEndTime function allows the MODERATOR role to change the start and end times for claimSettings without any validation checks. This means that the _startTime and _endTime values can be set to any arbitrary values, including values that are in the past or values that don't align with the intended functionality of the contract.

Affected Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L300-L311

Impacts

If _startTime is set to a value in the past or _endTime is set to a value earlier than _startTime, it could disrupt the intended functionality of the contract. Setting invalid times could lead to inconsistent behavior within the contract, potentially affecting the ability to claim rewards or perform other actions.

Remediation

To mitigate these potential issues, you should add validation checks to ensure that _startTime is less than or equal to _endTime and that both values align with the intended usage of the contract. Additionally, consider validating that the times are not set to values in the past unless such functionality is explicitly desired. Adding these validation checks will help ensure the contract operates as intended and prevent unintended disruptions.

Retest

Input validations have been added to the start and end times.

Bug ID #7 [Fixed]

Gas Optimization for State Variables

Vulnerability Type

Gas Optimization

Severity

Gas

Description

In the Claim.sol contract, there are two instances where the -= operator is used to subtract _amount from the claimBalance state variable. While this operation is correct in terms of logic, it's worth noting that the -= operator can consume more gas compared to using the = operator in conjunction with addition. In other words, x -= y may consume more gas than x = x - y.

Affected Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L243
- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L192

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

The operation has been optimized in the form of x = x + y to save some gas.

Bug ID #8 [Won't Fix]

Array Length Caching

Vulnerability Type

Gas Optimization

Severity

Gas

Description

During each iteration of the loop, reading the length of the array uses more gas than is necessary. In the most favorable scenario, in which the length is read from a memory variable, storing the array length in the stack can save about 3 gas per iteration. In the least favorable scenario, in which external calls are made during each iteration, the amount of gas wasted can be significant.

Affected Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L156

Impacts

Reading the length of an array multiple times in a loop by calling .length costs more gas.

Remediation

Consider storing the array length of the variable before the loop and use the stored length instead of fetching it in each iteration.

Retest

Array length is not cached in a variable and is used directly in the loop.

Bug ID #9 [Won't Fix]

Gas Optimization in Increments

Vulnerability Type

Gas optimization

Severity

Gas optimization

Description

The contract uses **for** loops that 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/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L163

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 saves some gas.

Retest

The contract is using ++i but j++ instead of ++j. This won't have any security impact at all but the loop could have been a bit more cost effective.

Bug ID#10 [Fixed]

Require With Empty Message

Vulnerability Type

Gas Optimization

Severity

Informational

Description

During code analysis, it has been observed that some require statements lack descriptive messages, which provide crucial information to users when conditions are not met. These messages, limited to 32 bytes, improve user understanding of why a transaction was reverted.

Vulnerable Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f23
- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/Claim.sol#L121C7-L124C11

Impacts

Users may be left without clear context when a transaction is reverted due to unmet conditions, leading to confusion and frustration

Remediation

Add concise, informative messages to require statements, explaining why the condition failed. Ensure messages are clear and within the 32-byte limit.

Retest

Require statements have been updated with descriptive messages.

Bug ID#11 [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.

Vulnerable Code

- https://github.com/StationX-Network/smartcontract/blob/9fb58109b37bd6f2351ab8bc6f0f495bd517860b/contracts/Claim/ClaimFactory.sol

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

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

Retest
NatSpec comments are added to the contract.

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.