CredShields

Smart Contract Audit

Sept 21st, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Just Farming Smart Contracts audit performed by CredShields Technologies PTE. LTD. on behalf of Justfarming GmbH & Co. OHG between Sept 6th, 2023, and Sept 12th, 2023. A retest was performed on September 19th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Justfarming GmbH & Co. OHG

Table of Contents

CRED SHiELDS

1. Executive Summary

Justfarming GmbH & Co. OHG engaged CredShields to perform a smart contract audit from Sept 6th, 2023, to Sept 12th, 2023. During this timeframe, Ten (10) vulnerabilities were identified. **A retest was performed on Sept 19th, 2023, and all the bugs have been addressed.**

During the audit, Ten (10) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Justfarming GmbH & Co. OHG" and should be prioritized for remediation, and fortunately, none were found.

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

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Just Farming Smart Contracts	0	0	1	0	3	6	**10**
	0	**0**	**1**	**0**	**3**	**6**	**10**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Justfarming GmbH & Co. OHG engaged CredShields to perform a Justfarming GmbH & Co. OHG 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 6th, 2023, to Sept 12th, 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/justfarming/contracts/tree/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts

Table: List of Files in Scope

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

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

2.2 Retesting phase

Justfarming GmbH & Co. OHG 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, Ten (10) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Array Length Caching	Gas	Gas Optimization
Gas Optimization in Require Statements	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas Optimization
Custom Errors instead of Revert	Gas	Gas Optimization
Functions should be declared External	Informational	Best Practices
Missing State Variable Visibility	Informational	Missing Best Practices
Unnecessary Checked Arithmetic in Loops	Gas	Gas Optimization

Uint Underflow in StakingRewards	Medium	Arithmetic Underflow
Gas Optimization for State Variables	Gas	Gas Optimization
Floating Pragma	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

Justfarming GmbH & Co. OHG is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Sept 19th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Array Length Caching	Gas	**Fixed [20/09/2023]**
Gas Optimization in Require Statements	Gas	**Won't Fix**
Gas Optimization in Increments	Gas	**Fixed [20/09/2023]**
Custom Errors instead of Revert	Gas	**Fixed [20/09/2023]**
Functions should be declared External	Informational	**Fixed [20/09/2023]**
Missing State Variable Visibility	Informational	**Fixed [20/09/2023]**
Unnecessary Checked Arithmetic in Loops	Gas	**Fixed [20/09/2023]**
Uint Underflow in StakingRewards	Medium	**Fixed [20/09/2023]**

Gas Optimization for State Variables	Gas	**Fixed [20/09/2023]**
Floating Pragma	Informational	**Fixed [20/09/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

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/justfarming/contracts/blob/main/contracts/lib/StakingRewards.sol#L165

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

This is fixed. The array length is being cached outside of the loop.

https://github.com/justfarming/contracts/blob/v1.1.0/contracts/lib/StakingRewards.sol#L165

Bug ID#2 [Won't Fix]

Gas Optimization in Require Statements

Vulnerability Type

Gas Optimization

Severity

Gas

Description

The **require()** statement takes an input string to show errors if the validation fails.

The strings inside these functions that are longer than **32 bytes** require at least one additional MSTORE, along with additional overhead for computing memory offset and other parameters. For this purpose, having strings less than 32 bytes saves a significant amount of gas. Once such example is given below:

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L42
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L70-L73
- https://github.com/justfarming/contracts/blob/4ec14e3ce0f96750eda92844d2192ead355a8744/contracts/lib/BatchDeposit.sol#L126-L130
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L136
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L140
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L72C1-L72C1
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L76

Impacts

Having longer require strings than 32 bytes cost a significant amount of gas.

Remediation

It is recommended to go through all the **require()** statements present in the contract and shorten the strings passed inside them to fit under **32 bytes**. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Retest

This won’t be fixed in order to display better descriptive errors.

Bug ID #3 [Fixed]

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/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.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
This is fixed by using ++i.
https://github.com/justfarming/contracts/blob/v1.1.0/contracts/lib/StakingRewards.sol#L179

Bug ID#4 [Fixed]

Custom Errors instead of Revert

Vulnerability Type

Gas Optimization

Severity

Gas

Description

The contract was found to be using a revert() statement. Since Solidity v0.8.4, custom errors have been introduced which are a better alternative to the revert.

This allows the developers to pass custom errors with dynamic data while reverting the transaction and also makes the whole implementation a bit cheaper than using revert.

Vulnerable Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L43

Impacts

Using revert() instead of error() costs more gas.

Remediation

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

Retest

This if fixed by implementing custom errors.
https://github.com/justfarming/contracts/blob/v1.1.0/contracts/lib/BatchDeposit.sol#L49

Bug ID#5 [Fixed]

Functions should be declared External

Vulnerability Type
Best Practices

Severity
Informational

Description
Public functions that are never called by a contract should be declared external in order to conserve gas.
The following functions were declared as public but were not called anywhere in the contract, making public visibility useless.

Affected Code
The following functions were affected -

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L187-L199
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L208-L213

Impacts
Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.
"**public**" functions cost more Gas than "**external**" functions.

Remediation
Use the "**external**" state visibility for functions that are never called from inside the contract.

Retest
This is fixed. Both the functions have been made external.

Bug ID #6 [Fixed]

Missing State Variable Visibility

Vulnerability Type
Missing Best Practices

Severity
Informational

Description
In Solidity, the visibility of state variables is important as it determines how those variables can be accessed and modified by other contracts or functions.
The contract defined state variables that were missing a visibility modifier.

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L18

Impacts
If the visibility of a state variable is accidentally left out, it can cause unexpected behavior and security vulnerabilities. For example, if a state variable is supposed to be private and is accidentally declared without any visibility keyword, it will be treated as "internal" by default, which may lead to it being accessible by other contracts or functions outside the intended scope. This can lead to a potential attack vector for malicious actors.

Remediation
Explicitly define visibility for all state variables. These variables can be specified as public, internal, or private.

Retest
The state variable visibility has been set.
https://github.com/justfarming/contracts/blob/v1.1.0/contracts/lib/BatchDeposit.sol#L18

Bug ID #7 [Fixed]

Unnecessary Checked Arithmetic in Loops

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Loops are in most cases bounded by definition (the bounding is represented by the exit condition). Therefore in the vast majority of cases, checking for overflows is really not needed, and can get very gas expensive. Here's an example:

```
pragma solidity ^0.8.0;

contract Test1 {


    function loop() public pure {

        for(uint256 i = 0; i < 100; i++) {
        }

    }
}
```

```
pragma solidity ^0.8.0;

contract Test {

    function loop() public pure {

        for(uint256 i = 0; i < 100;) {

            unchecked {

                i++;

            }

        }

    }

}
```

loop() in Test1 costs more than 31K gas, vs 25.5K gas for loop() in Test2.

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L75
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L143

Impacts

Removing overflow validations using unchecked blocks will save gas in the loops.

Remediation

It is recommended to implement unchecked blocks in for loops wherever possible since they are already bounded by an upper length and there's a very rare chance that it might overflow.

Retest

This is fixed. Loops have been made unchecked to save gas.

Bug ID #8 [Fixed]

Uint Underflow in StakingRewards

Vulnerability Type
Arithmetic Underflow

Severity
Medium

Description
Upon reviewing the contract's code, it has been identified that there is a potential underflow issue in the releasable() function. This issue arises if the condition *(totalBalance >= _exitedStake)* evaluates to false, leading to a subtraction operation that could result in an underflow when calculating *totalFees - releasedFees*.

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L208-L229
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L128-L148

Impacts
In this scenario, the user's funds will be stuck in the contract and when the user tries to withdraw funds, the transaction will revert.

Remediation
When `account == _feeRecipient`, before performing the subtraction operation `(totalFees - releasedFees)`, it is recommended to check if `billableRewards` is greater than zero.

Retest

This has been fixed by validating billableRewards - https://github.com/justfarming/contracts/commit/1222ebacc6baf37ff3872c9d8fdd55d75788bc0d#diff-aa586f47914318fcad9e5cbe35726632fc31df638637800749a7ca4b736e3545

Bug ID #9 [Fixed]

Gas Optimization for State Variables

Vulnerability Type
Gas Optimization

Severity
Gas

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

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L175

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 is fixed. The code is now using the format x = x + y.
https://github.com/justfarming/contracts/blob/v1.1.0/contracts/lib/StakingRewards.sol#L182

Bug ID #10 [Fixed]

Floating Pragma

Vulnerability Type
Missing Best Practices

Severity
Informational

Description
Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Affected Code

- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/BatchDeposit.sol#L2
- https://github.com/justfarming/contracts/blob/ca92d2097a0e28b366d63cfbfd7ed091dad775a5/contracts/lib/StakingRewards.sol#L2

Impacts
A contract with a floating pragma could be compiled with any version up and above the version specified. Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

Remediation
Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Retest

The pragma has been fixed and locked to 0.8.21.

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.