CredShields

Smart Contract Audit

April 27th, 2023 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract Audit audit performed by CredShields Technologies PTE. LTD. on behalf of Uniscrow between April 7th, 2023, and April 11th, 2023. And a retest was performed on April 25th, 2023.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Uniscrow

Table of Contents

1. Executive Summary

Uniscrow engaged CredShields to perform a smart contract audit from April 7th, 2023, to April 11th, 2023. During this timeframe, 6 (six) vulnerabilities were identified. **A retest was performed on April 25th, 2023, and all the bugs have been addressed.**

During the audit, 0 (zero) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Uniscrow " 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
Smart Contract Audit	0	0	1	2	2	1	**6**
	0	**0**	**1**	**2**	**2**	**1**	**6**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Uniscrow engaged CredShields to perform a Uniscrow 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 April 7th, 2023, to April 11th, 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/uniscrow/escrow

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

Uniscrow 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, 6 (six) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Missing Zero Address Validations	Low	Missing input validation
Variables should be Immutable	Gas	Gas Optimization
Missing SPDX License	Informational	Best practices
Missing NatSpec Comments	Informational	Missing best practices
Missing Access Control	Medium	Missing Access Control

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	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

Uniscrow is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on April 25th, 2023, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating and Outdated Pragma	Low	**Fixed [25/04/2023]**
Missing Zero Address Validations	Low	**Fixed [25/04/2023]**
Variables should be Immutable	Gas	**Fixed [25/04/2023]**
Missing SPDX License	Informational	**Fixed [25/04/2023]**
Missing NatSpec Comments	Informational	**Fixed [25/04/2023]**
Missing Access Control	Medium	**Fixed [25/04/2023]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [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 was allowing floating or unlocked pragma to be used, i.e., **>=0.8 <0.9.0.**
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 -

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:
Pragma has been fixed to a single version and updated.

Bug ID #2 [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 Variables and Line Numbers

- Factory.constructor() - _arbitrator, _erc20, _feeRecipient

```
constructor(
    address _arbitrator,
    address _erc20,
    address _feeRecipient){
        arbitrator = _arbitrator;
        erc20 = _erc20;
        feeRecipient = _feeRecipient;
}
```

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
Zero address validations have been implemented.

Bug ID #3 [Fixed]

Variables should be Immutable

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
Declaring state variables that are not updated following deployment as immutable can save gas costs in smart contract deployments and function executions. Immutable state variables are those that cannot be changed once they are initialized, and their values are set permanently.

By declaring state variables as immutable, the compiler can optimize their storage in a way that reduces gas costs. Specifically, the compiler can store the value directly in the bytecode of the contract, rather than in storage, which is a more expensive operation.

Affected Code:

- EscrowTransaction.arbitrator, buyer, erc20, seller
- Factory.arbitrator, erc20, feeRecipient
- FeeCalculator.basepoints, feeRecipient, flatFee
- Managed.agent

Impacts:
Gas usage is increased if the variables that are not updated outside of the constructor are not set as immutable.

Remediation:
An "`immutable`" attribute should be added in the parameters that are never updated outside of the constructor to save the gas.

Retest

Variables have been marked as immutable.

Bug ID#4 [Fixed]

Missing SPDX License

Vulnerability Type
Best practices

Severity
Informational

Description:
The contracts were missing an SPDX License identifier in the source code. A smart contract whose source code is available can better establish trust. In order to minimize legal problems relating to copyright, Solidity encourages the use of machine-readable SPDX license identifiers.

Impacts:
SPDX Licenses help in identifying the legal owner of the software, therefore, helping in issues like copyright infringement.

Remediation:
Every source file should start with a comment indicating its license. Add a necessary license identifier in the contract code like the one shown below:

```
// SPDX-License-Identifier: MIT
```

Retest:
An SPDX license has been added to all the contracts.

Bug ID#5 [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 in the scope were missing these comments.

Impacts:
Without Natspec comments, it can be challenging for other developers to understand the code's intended behavior and purpose. This can lead to errors or bugs in the code, making it difficult to maintain and update the codebase. Additionally, it can make it harder for auditors to evaluate the code for security vulnerabilities, increasing the risk of potential exploits.

Remediation:
Developers should review their codebase and add Natspec comments to all relevant functions, variables, and events. Natspec comments should include a description of the function or event, its parameters, and its return values.

Retest:
NatSpec comments have been added to all the contracts.

Bug ID #6 [Fixed]

Missing Access Control

Vulnerability Type
Missing Access Control

Severity
Medium

Description
Access control is a critical aspect of developing secure and robust smart contracts in Solidity. It refers to the process of managing and restricting access to certain functionalities or data within a smart contract.
The Factory contract defines an external function called “reateEscrow()” that can be called by any user to create escrows for any buyer or seller.

```
function createEscrow(
    address seller,
    address buyer,
    uint256 flatFee,
    uint256 bpsFee,
    uint256 allowance
) external {
    PresetManagedWithFees escrow = new PresetManagedWithFees(
        buyer,
        seller,
        arbitrator,
        feeRecipient,
        flatFee,
        bpsFee,
        erc20,
        allowance
```

```
        );

        emit EscrowCreated(address(escrow));
    }
```

Impacts

Missing access control on the function to create escrows could lead to spam escrows being created. Due to the lack of owner validation or fees, regular users could create escrows for any address.

Remediation

It is recommended to either have onlyOwner validations on the function or deduct a fee when external users create an escrow to prevent unnecessary spam.

Retest:

OnlyOwner validation has been enforced on the function.

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.