CredShields

Smart Contract Audit

Sept 24th, 2022 • CONFIDENTIAL

Description

This document details the process and result of the Wahed Token smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of Wahed Projects between Aug 9th, 2022, and Aug 15th, 2022. And a retest was performed on 20th Sept 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Wahed Projects

Table of Contents

1. Executive Summary **3**

State of Security 4

2. Methodology **5**

2.1 Preparation phase 5

2.1.1 Scope 6

2.1.2 Documentation 6

2.1.3 Audit Goals 6

2.2 Retesting phase 8

2.3 Vulnerability classification and severity 8

2.4 CredShields staff 12

3. Findings **13**

3.1 Findings Overview 13

3.1.1 Vulnerability Summary 13

3.1.2 Findings Summary 15

4. Remediation Status **19**

5. Bug Reports **20**

Bug ID#1 20

Floating Pragma 20

Bug ID#2 22

Functions should be declared External 22

Bug ID#3 24

Missing Multiple Zero Address Validations 24

Bug ID#4 26

Missing Events on important functions 26

Bug ID#5 28

Missing Constant Attribute in Variables 28

Bug ID#6 30

Gas Optimization in year and month 30

Bug ID#7 32

CRED SHiELDS

Gas Optimization due to redundant codes 32
Bug ID#8 34
Missing Timelock on critical state changes 34
Bug ID#9 36
Missing NatSpec Comments 36

6. Disclosure 37

1. Executive Summary

Wahed Projects engaged CredShields to perform a smart contract audit from Aug 9th, 2022, to Aug 15th, 2022. During this timeframe, eight (8) vulnerabilities were identified. **A retest was performed on 20th Sept 2022 and none of the bugs are in pending fix state.**

During the audit, zero (0) vulnerability was found that had a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Wahed Projects" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and 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
Wahed Token	0	0	0	4	1	4	**9**
	0	**0**	**0**	**4**	**1**	**4**	**9**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Wahed Project's scope during the testing window while abiding by the policies set forth by Wahed Project's team.

State of Security

Maintaining a healthy security posture requires constant review and refinement of existing security processes. Running a CredShields continuous audit allows Wahed Project's 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 Wahed Projects introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate Wahed Project's 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 Wahed Projects assets.

2. Methodology

Wahed Project's engaged CredShields to perform a Wahed Token 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 for 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 Aug 9th, 2022, to Aug 15th, 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://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18

Table: List of Files in Scope

2.1.2 Documentation

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

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

Wahed Projects 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.), and Technical and Business Impacts. The likelihood and the impact estimate are put together to calculate the overall severity of 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.

Considering the risk and volatility of smart contracts and how they use gas as a method of payment to deploy the contracts and interact with them, gas optimization becomes a major point of concern. To address this, CredShields also introduces another severity category called "**Gas Optimization**" or "**Gas**". This category deals with code optimization techniques and refactoring due to which Gas can be conserved.

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

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Functions should be declared External	Gas	Gas Optimization
Missing Multiple Zero Address Validations	Low	Missing Input Validation
Missing Events on important functions	Low	Missing Best Practices
Missing Constant Attribute in Variables	Gas	Gas Optimization
Gas Optimization in year and month	Gas	Gas Optimization
Gas Optimization due to redundant codes	Gas	Gas Optimization
Missing Timelock on critical state changes	Low	Missing best practices

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	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 used but the contract doesn't make strict check so not an issue.
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

Wahed Projects is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations.

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Accepted Risk**
Functions should be declared External	Gas	**Accepted Risk**
Missing Multiple Zero Address Validations	Low	**Accepted Risk**
Missing Events on important functions	Low	**Accepted Risk**
Missing Constant Attribute in Variables	Gas	**Accepted Risk**
Gas Optimization in year and month	Gas	**Accepted Risk**
Gas Optimization due to redundant codes	Gas	**Accepted Risk**
Missing Timelock on critical state changes	Low	**Accepted Risk**
Missing NatSpec Comments	Informational	**Accepted Risk**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1

Floating 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 contracts found in the repository allowed floating or unlocked pragma to be used, i.e., **^0.8.0**.
This allows the contracts to be compiled with all the solidity compiler versions above **0.8.0**.
The following contracts were found to be affected -

Affected Code

- WAHED.sol

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 strict 0.8.7 pragma version i.e

pragma solidity ^0.8.0;

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

Retest:

CredShields team agrees with the Wahed team that there are no exploiatble scenario with the range of compiler versions as they are latest and hence it is quite safe.

Bug ID#2

Functions should be declared External

Vulnerability Type
Gas Optimization

Severity
Gas

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 the public visibility useless.

Affected Code
The following functions were affected -

- **WAHED.sol**
  - pause() https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L876
  - unpause() https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L881
  - mint() https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L900

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:

The team is comfortable with the excess gas cost as the gas costs are negligible.

Bug ID#3

Missing Multiple Zero Address Validations

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
WAHED contract was 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

- Function “updateFounderAddress” -> variable “_addres” https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L868

```
function updateFounderAddress(address _addres) external onlyOwner{
    FOUNDERSAddress = _addres;
}
```

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:

There are no critical case scenarios at present and hence the Credshields team agrees with Wahed Team for not fixing the issue.

Bug ID#4

Missing Events on 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 -

- function **claim()**
  https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L859
- function **updateFounderAddress()**
  https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L865

Impacts
Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation

Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

Retest:
The Wahed team prefer less gas cost over logging and hence it is safe to leave this unfixed.

Bug ID#5

Missing Constant Attribute in Variables

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile time.
The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.
Compared to regular state variables, the gas costs of constant and immutable variables are much lower since no `SLOAD` is executed to retrieve constants from storage because they're interpolated directly into the bytecode.

Affected Code:

- WAHED.sol - https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L751

```
uint256 public FOUNDERS = 75_00_00_000;
```

Impacts:
Gas usage is increased if the variables that should be constants are not set as constants.

Remediation:

A "`constant`" attribute should be added in the parameters that never change to save the gas.

Retest:
The team is comfortable with the excess gas cost as the gas costs are negligible.

Bug ID#6

Gas Optimization in year and month

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
The contract defines “year” and “month” in days at the top of the “claim()” function. This is then used inside the function using their variable names.
While testing the gas efficiency of the contract, it was found that the deployment will save around 200 units of gas when these values are used directly like “365 days” and “30 days” as shown below.

Affected Code:

- https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L838

Impacts:
The current implementation consumes more gas than the suggested recommendation.

Remediation:
It is recommended to remove the “year” and “month” variables and use the values directly in the “claim()” function.

```
function claim() external {
    // uint256 year = 365 days;
    // uint256 month = 30 days;

    require(balanceOf(address(this)) > 0, "No token left");
```

```
        require(msg.sender == FOUNDERSAddress, "Only founder can
claim");
        require(
            block.timestamp > tokenDeployTimeStamp + 365 days,
            "token are currently locked"
        );
        uint256 nextTime = tokenDeployTimeStamp + 365 days + (30
days * claimCounter);
...
```

Retest:

The team is comfortable with the excess gas cost as the gas costs are negligible.

Bug ID#7

Gas Optimization due to redundant codes

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
The contract defines a separate function - “currentTime()” to return the value of the “block.timestamp”. This is not cost-effective.
While testing the gas efficiency of the contract, it was found that the deployment will save around 3600 units of gas when the value of “block.timestamp” is used directly without defining it inside a function.

Affected Code:

- https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L864

Impacts:
The current implementation consumes more gas than the suggested recommendation also it shows inconsistency in codes by using “block.timestamp” at some places and the function “currentTime()” at one place.

Remediation:
It is recommended to remove the function “currentTime()” and use the values of “block.timestamp” directly wherever required.

```
function claim() external {
    uint256 year = 365 days;
    uint256 month = 30 days;
```

```
        require(balanceOf(address(this)) > 0, "No token left");
        require(msg.sender == FOUNDERSAddress, "Only founder can
claim");
        require(
            block.timestamp > tokenDeployTimeStamp + year,
            "token are currently locked"
        );
        uint256 nextTime = tokenDeployTimeStamp + year + (month *
claimCounter);
        require(nextTime < block.timestamp, "no claim Available");
        uint256 counter = (block.timestamp - (nextTime)) / month;
...
```

Retest:

The team is comfortable with the excess gas cost as the gas costs are negligible.

Bug ID#8

Missing Timelock on critical state changes

Vulnerability Type
Missing best practices

Severity
Low

Description:
OpenZepplin includes some excellent modules for on-chain governance. One such module is called TimeLock. A timelock is a smart contract that delays function calls of another smart contract after a predetermined amount of time has passed.
The file **WAHED.sol** makes critical address changes and updates the Founder's address in the function "updateFounderAddress()" that lacks a timelock.

Affected Code:

- https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code#F1#L868

Impacts:
Having timelock on sensitive functions gives more control in the hands of the end-users. This gives them and the investors more time to note the essential changes before deciding on the course of action. Timelocks are mostly used in the context of governance to add delay in administrative actions and are generally considered a strong indicator that a project is legitimate and demonstrates a commitment to the project by project owners.

Remediation:
Implement a timelock mechanism in the smart contracts before making critical state-changing decisions, especially when updating the Founder's address.

```
      function updateFounderAddress(address _addres) external
onlyOwner{
       FOUNDERSAddress = _addres;
   }
```

Retest:

The contract owner is expected to be trusted and ethical and hence they team would like to skip the timelock implementation as this is not exploitable just an additional security measure.

Bug ID#9

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 contract **WAHED.sol** is missing NatSpec comments in the code which makes it difficult for the auditors and future developers to understand the code.

Affected Code:

- https://bscscan.com/address/0x733708a9869066a82b741410855aa756646c0f18#code

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

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

Retest:
Adding comments makes the smart contract more readable but it doesn't cause any case of exploitation hence this was agreed to be as it is.

6. 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.