CredShields

Smart Contract Audit

Feb 19th, 2024

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Wasset between Jan 31st, 2024, and Feb 8th, 2024. And a retest was performed on Feb 15th, 2024.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Wasset

Table of Contents

1. Executive Summary

Wasset engaged CredShields to perform a smart contract audit from Jan 31st, 2024, to Feb 8th, 2024. During this timeframe, Seventeen (17) vulnerabilities were identified. **A retest was performed on Feb 15th, 2024, and all the bugs have been addressed.**

During the audit, Two (2) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Wasset" 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	2	0	3	5	3	4	**17**
	2	**0**	**3**	**5**	**3**	**4**	**17**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

Wasset engaged CredShields to perform a Wasset 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 Jan 31st, 2024, to Feb 8th, 2024, 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/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/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

Wasset 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, Seventeen (17) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Funds Lockup in withdraw() Function	Critical	Funds Lockup
_calculateNextDrawTimestamp() should be external & onlyOwner	Critical	Business Logic
DoS in revealDraw() function	Medium	Wrong Implementation
Potential Overflow Vulnerability Due to Type Casting in _getRandomNumbersFromBytes32 Function	Medium	Overflow Vulnerability
Inefficient Use of Randomness in Winner Selection Algorithm	Medium	Business Logic
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing best practices

Missing Zero Address Validations	Low	Missing Input Validation
Missing Same Address Validation in _setFeeCollector() Function	Low	Missing Input Validation
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Use Ownable2Step	Low	Missing Best Practices
Missing NatSpec Comments	Informational	Missing best practices
Missing State Variable Visibility	Informational	Missing Best Practices
Functions should be declared External	Informational	Best Practices
Gas Optimization in Increments	Gas	Gas optimization
Gas Optimization in Require Statements	Gas	Gas Optimization
Variables should be Immutable	Gas	Gas Optimization
Array Length Caching	Gas	Gas Optimization

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

Wasset is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Feb 15th, 2024, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Funds Lockup in withdraw() Function	Critical	**Fixed [15/02/2024]**
_calculateNextDrawTimestamp() should be external & onlyOwner	Critical	**Fixed [15/02/2024]**
DoS in revealDraw() function	Medium	**Fixed [15/02/2024]**
Potential Overflow Vulnerability Due to Type Casting in _getRandomNumbersFromBytes32 Function	Medium	**Fixed [15/02/2024]**
Inefficient Use of Randomness in Winner Selection Algorithm	Medium	**Fixed [15/02/2024]**
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Fixed [15/02/2024]**
Missing Zero Address Validations	Low	**Fixed [15/02/2024]**
Missing Same Address Validation in _setFeeCollector() Function	Low	**Fixed [15/02/2024]**

Floating and Outdated Pragma	Low	**Fixed [15/02/2024]**
Use Ownable2Step	Low	**Fixed [15/02/2024]**
Missing NatSpec Comments	Informational	**Fixed [15/02/2024]**
Missing State Variable Visibility	Informational	**Fixed [15/02/2024]**
Functions should be declared External	Informational	**Fixed [15/02/2024]**
Gas Optimization in Increments	Gas	**Fixed [15/02/2024]**
Gas Optimization in Require Statements	Gas	**Fixed [15/02/2024]**
Variables should be Immutable	Gas	**Fixed [15/02/2024]**
Array Length Caching	Gas	**Fixed [15/02/2024]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Funds Lockup in withdraw() Function

Vulnerability Type
Funds Lockup

Severity
Critical

Description:
Funds can become locked up indefinitely in the withdraw() function. The withdraw() function is restricted to onlyOwner modifier, and if the owner does not have any winnings, the funds allocated to them will remain stuck, even if the owner has winnings, they can only withdraw a portion of their winnings while the rest remains locked up indefinitely within the contract and the winners will not able to withdraw their winnings.

Affected Variables and Line Numbers

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L289-L297

Impacts
It results in a partial funds lockup scenario, where users, including the owner, are unable to access their full winnings. This not only undermines trust in the contract but also causes financial loss and frustration to users unable to retrieve their funds.

Remediation

It is recommended to remove the onlyOwner modifier from the withdraw() function and make it accessible for all the other users to withdraw their winnings.

Retest

The onlyOwner modifier has been removed from the withdraw function.
https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L359-L366

Bug ID #2 [Fixed]

_calculateNextDrawTimestamp() should be external & onlyOwner

Vulnerability Type

Business Logic

Severity

Critical

Description

The `_calculateNextDrawTimestamp()` function determines when to end the current draw and when to start the next one based on the `_cycle` duration. However, the `_calculateNextDrawTimestamp()` function is marked as private and cannot be used to update the `_nextDrawTimestamp` variable. `_nextDrawTimestamp` will always remain default to zero. That will cause the DOS of the `draw()` function and the user's funds will get stuck in the contract.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L72

Impacts

Stagnant `_nextDrawTimestamp` from malfunctioning `_calculateNextDrawTimestamp()` could lead to repeated betting, causing a denial of service (DoS) to the `draw()` function. Betting of users will be stuck in the contract forever.

Remediation

It is recommended to mark `_calculateNextDrawTimestamp()` as external and onlyOwner so that the owner can change the `_nextDrawTimestamp` accordingly.

Retest

This has been remediated.
Comments from the Wasset Team: "Renamed all occurrences of nextDrawTimeStamp to endDrawTimestamp. Replaced _calculateNextDrawTimeStamp() with _calculateDrawEndTimestamp(). calculateDrawEndTimestamp() is called inside the constructor when there is a new draw."
https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L179

Bug ID #3 [Fixed]

DoS in revealDraw() function

Vulnerability Type

Wrong Implementation

Severity

Medium

Description

According to the Witnet Randomness [Documentation](), `block.number` should be passed to get the Randomness. However, in the `revealDraw()` function, block.timestamp is used instead of block.number.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L250-L287

Impacts

Passing `block.timestamp` instead of the intended `block.number` in the `revealDraw()` function results in an incorrect value being used for obtaining randomness. This deviation from the expected behaviour can lead to the transaction being reverted due to the inconsistency in the input parameters.

Remediation

It is recommended to use `_drawBlockNumber` variable instead of `_drawTimestamp`. Eg:

Retest

The block.timestamp has been replaced by block.number.

https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L261

Bug ID #4 [Fixed]

Potential Overflow Vulnerability Due to Type Casting in _getRandomNumbersFromBytes32 Function

Vulnerability Type

Overflow Vulnerability

Severity

Medium

Description:

The `_getRandomNumbersFromBytes32` function in the given contract performs type casting from a higher bytes data type (`uint256`) to a lower bytes data type (`uint64`). Specifically, it casts a `uint256` value `rb` to a `uint64` value during the calculation of `selectedNumber`. This type casting operation may lead to a potential overflow vulnerability.

In Solidity, when casting from a higher bytes data type to a lower bytes data type, the higher-order bits are truncated to fit into the lower bytes data type. If the value of `rb` is greater than the maximum value that can be represented by a `uint64`, the truncation of higher-order bits may result in an overflow condition. This overflow can lead to incorrect calculations and potentially revert transactions.

Affected Variables and Line Numbers

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L205-L236

Impacts

If an overflow occurs during the type casting operation in the `_getRandomNumbersFromBytes32` function, it can lead to incorrect calculations and potentially cause the contract to revert transactions. This vulnerability may result in unexpected behavior, loss of funds, or denial of service.

Remediation

Review the necessity of type casting from `uint256` to `uint64` in the `_getRandomNumbersFromBytes32` function. Ensure that the casting operation does not result in potential overflows. Consider using safe arithmetic libraries such as SafeMath to perform arithmetic operations safely, ensuring that no overflow or underflow occurs during calculations.

Retest

Witnet randomness is now used for calculating truly random unpredictable outcomes.
https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L254-L265

Bug ID #5 [Fixed]

Inefficient Use of Randomness in Winner Selection Algorithm

Vulnerability Type

Business Logic

Severity

Medium

Description:

In the given lottery contract, the `revealDraw` function is responsible for revealing the winners of the lottery draw. However, the method used to select winners is not truly random but rather sequential based on the outcome of the randomness.

The contract calculates the maximum number of winners using the `_calculateMaxNumberOfWinning` function. Then, it generates a random number using the `getWitnetRandomness()` function and the `_getRandomNumbersFromBytes32` function. However, instead of using this randomness to randomly select winners from the pool of participants, the contract sequentially selects the first `x` participants, where `x` is calculated based on the logarithm of the number of participants.

This method of winner selection is not truly random and may favour certain participants over others, especially those with lower ticket numbers. Additionally, it does not utilize the randomness obtained from the `getWitnetRandomness()` function effectively.

Affected Variables and Line Numbers

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L250-L287

Impacts

The inefficient use of randomness in the winner selection algorithm may lead to unfair results in the lottery. Participants may perceive the lottery as biased or rigged if they notice patterns in winner selection. Additionally, this approach does not provide the desired level of randomness and may not comply with regulatory requirements for fair lotteries.

Remediation

Revise the winner selection algorithm to utilize the randomness obtained from the `getWitnetRandomness()` function more effectively. Implement a truly random selection mechanism that ensures fairness and transparency in the lottery draw.

Retest

This is fixed. The winner selection logic now considers index positions generated by random numbers fetched from witnet.

Ref:
https://github.com/DFMlab/wasset/blob/501394ba6697738e067155040a2e20a4804ba11e/contracts/contracts/WassetLottery.sol#L343

Bug ID #6 [Fixed]

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom

Vulnerability Type

Missing best practices

Severity

Low

Description

The transfer() and transferFrom() method is used instead of safeTransfer() and safeTransferFrom(), presumably to save gas however OpenZeppelin's documentation discourages the use of transferFrom(), use safeTransferFrom() whenever possible because safeTransferFrom auto-handles boolean return values whenever there's an error.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L296
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L171

Impacts

Some ERC tokens do not revert on failure. This could prove fatal during an airdrop distribution because if any one of the transfer fails in the loop, it'll never revert, leading to wrong assumptions and inconsistent amount distribution.

Using safeTransferFrom has the following benefits -

- It checks the boolean return values of ERC20 operations and reverts the transaction if they fail,
- at the same time allowing you to support some non-standard ERC20 tokens that don't have boolean return values.

- It additionally provides helpers to increase or decrease an allowance, to mitigate an attack possible with vanilla approve.

Remediation

Consider using safeTransfer() and safeTransferFrom() instead of transfer() and transferFrom().

Retest

This is fixed. Safe ERC is now being used.

Bug ID #7 [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

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L92

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 validation has been added -
https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L108-L111

Bug ID #8 [Fixed]

Missing Same Address Validation in _setFeeCollector() Function

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
A function named `_setFeeCollector()` allows the contract `_feeCollector` to be changed. However, it lacks a validation check to verify whether the newly specified owner (`_feeCollector`) is the same as the current `_feeCollector`.
This omission raises a potential security concern, as it does not prevent unnecessary emissions of the `FeeCollectorChanged` event when the new `_feeCollector` is identical to the existing one.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L91

Impacts
Missing validation in `_setFeeCollector()` allows emitting `FeeCollectorChanged` events unnecessarily when the new fee collector address is the same as the current one, causing minor inefficiency and confusion.

Remediation
Add the same-address validation to the function where the address is being set.

Retest

Same address validation has been added - https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L73

Bug ID #9 [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., **^0.8.20.**
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 0.8.22 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

This is fixed. The contracts are now using a fixed pragma 0.8.22.

Bug ID #10 [Fixed]

Use Ownable2Step

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Affected Code
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L10
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetTokenDistribution.sol#L7
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetAirdrop.sol#L9

Impacts
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Remediation

It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

Retest

The contracts are now using Ownable2Step instead of Ownable for additional security.

Bug ID #11 [Partially 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
The lottery contract has been updated but others still don’t have NatSpec comments.

Bug ID #12 [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/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L134
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L144
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L145
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L146

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

This is fixed. The state variables are now explicitly defining their visibility.

Bug ID #13 [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

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L180

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 function is now external -
https://github.com/DFMlab/wasset/blob/1d7da88f2656c63a42cc8e5aaf90e266cb733267/contracts/contracts/WassetLottery.sol#L223

Bug ID #14 [Fixed]

Gas Optimization in Increments

Vulnerability Type

Gas optimization

Severity

Gas

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.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L214
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L242
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L272
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetTokenDistribution.sol#L24

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

Replaced **i++** with **++i.**

Bug ID #15 [Fixed]

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 lesser than 32 bytes saves a significant amount of gas.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetTokenDistribution.sol#L21

Impacts

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

Remediation

It is recommended to shorten the strings passed inside **require()** statements 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

Require string has been shortened.

Bug ID #16 [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:

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetAirdrop.sol#L13

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

This is fixed and the variable has been updated as immutable.

Bug ID #17 [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 favourable 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 favourable scenario, in which external calls are made during each iteration, the amount of gas wasted can be significant.

Affected Code

- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L242
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetLottery.sol#L272
- https://github.com/DFMlab/wasset/blob/d0f597d5829880dd48b95090c73a9c765fa9000d/contracts/contracts/WassetTokenDistribution.sol#L24

Impacts

Not storing reusable variables in memory costs more gas.

Remediation

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

Retest

This is fixed. The array lengths are now cached before the loops.

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.