CredShields

Smart Contract Audit

Aug 20th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of AssetChain between July 25th, 2024, and July 29th, 2024. A retest was performed on Aug 13th, 2024.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead), Shreyas Koli (Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor)

Prepared for

AssetChain

Table of Contents

CRED SHIELDS

1. Executive Summary

AssetChain engaged CredShields to perform a smart contract audit from July 25th, 2024, to July 29th, 2024. During this timeframe, 22 vulnerabilities were identified. **A retest was performed on Aug 13th, 2024, and all the bugs have been addressed.**

During the audit, 1 vulnerability was found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "AssetChain" 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 Contracts	0	1	4	6	3	8	**22**
	0	**1**	**4**	**6**	**3**	**8**	**22**

Table: Vulnerabilities Per Asset in Scope

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

2. Methodology

AssetChain engaged CredShields to perform a 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 July 25th, 2024, to July 29th, 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://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420 • https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42

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

AssetChain 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, and 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 about 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, 22 security vulnerabilities were identified in the asset.

Table: Findings in Smart Contracts

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Lack of Access Control in WithdrawTokens Function	High	Access Control
Unrestricted Modification of Active Presales	Medium	Business Logic
Missing Validation for Active Presale Status	Medium	Business Logic
Missing Price Feed Validation	Medium	Input Validation
Chainlink Oracle Min/Max price validation	Medium	Input Validation

Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	Missing Best Practices
Missing Access Control for Withdraw Function	Low	Access Control
Use Ownable2Step	Low	Missing best practices
Missing Zero Address Validations	Low	Input Validation
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Missing Events in Important Functions	Low	Missing Best Practices
Require with Empty Message	Informational	Code optimization
Dead Code	Informational	Code With No Effects - SWC-135
Use Call instead of Transfer	Informational	Missing Best Practices
Use Scientific Notations	Gas	Gas & Missing Best Practices
Boolean Equality	Gas	Gas Optimization
Custom error to save gas	Gas	Gas Optimization
Cheaper Inequalities in if()	Gas	Gas Optimization
Cheaper Inequalities in require()	Gas	Gas Optimization
Gas Optimization in Require/Revert Statements	Gas	Gas Optimization

Cheaper Conditional Operators	Gas	Gas Optimization
Variables should be Immutable	Gas	Gas Optimization

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	Bug ID #10
SWC-103	Floating Pragma	Vulnerable	Bug ID #10
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	Vulnerable	Bug ID #13

SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

AssetChain is actively partnering with CredShields from this engagement to validate the remediation of the discovered vulnerabilities. **A retest was performed on Aug 13th, 2024, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Lack of Access Control in WithdrawTokens Function	High	**Fixed [Aug 13th, 2024]**
Unrestricted Modification of Active Presales	Medium	**Won't Fix [Aug 13th, 2024]**
Missing Validation for Active Presale Status	Medium	**Won't Fix [Aug 13th, 2024]**
Missing Price Feed Validation	Medium	**Won't Fix [Aug 13th, 2024]**
Chainlink Oracle Min/Max price validation	Medium	**Won't Fix [Aug 13th, 2024]**
Use safeTransfer/safeTransferFrom instead of transfer/transferFrom	Low	**Partially Fixed [Aug 13th, 2024]**
Missing Access Control for Withdraw Function	Low	**Fixed [Aug 13th, 2024]**
Use Ownable2Step	Low	**Won't Fix [Aug 13th, 2024]**

Missing Zero Address Validations	Low	**Won't Fix [Aug 13th, 2024]**
Floating and Outdated Pragma	Low	**Won't Fix [Aug 13th, 2024]**
Missing Events in Important Functions	Low	**Won't Fix [Aug 13th, 2024]**
Require with Empty Message	Informational	**Fixed [Aug 13th, 2024]**
Dead Code	Informational	**Won't Fix [Aug 13th, 2024]**
Use Call instead of Transfer	Informational	**Won't Fix [Aug 13th, 2024]**
Use Scientific Notations	Gas	**Won't Fix [Aug 13th, 2024]**
Boolean Equality	Gas	**Won't Fix [Aug 13th, 2024]**
Custom error to save gas	Gas	**Won't Fix [Aug 13th, 2024]**
Cheaper Inequalities in if()	Gas	**Won't Fix [Aug 13th, 2024]**
Cheaper Inequalities in require()	Gas	**Won't Fix [Aug 13th, 2024]**
Gas Optimization in Require/Revert Statements	Gas	**Won't Fix [Aug 13th, 2024]**
Cheaper Conditional Operators	Gas	**Won't Fix [Aug 13th, 2024]**

Variables should be Immutable	Gas	**Won't Fix [Aug 13th, 2024]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Lack of Access Control in WithdrawTokens Function

Vulnerability Type
Access Control

Severity
High

Description
The `WithdrawTokens()` function in the sale contract allows any user to withdraw tokens from the contract and send them to the `fundReceiver` address. This function lacks proper access control, meaning it can be invoked by any user. As a result, a malicious actor could exploit this function to transfer all the tokens in the contract to the `fundReceiver` address, effectively emptying the contract's token balance.

Furthermore, if the contract allows users to buy tokens using USDT through the `buyWithUSDT` function, this lack of access control could prevent legitimate users from purchasing tokens, as the contract's token balance could be transferred to `fundReceiver`.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L927

Impacts

User can drain the contract of all tokens by invoking the WithdrawTokens() function, sending the tokens to the fundReceiver address. If tokens are drained, legitimate users attempting to buy tokens using USDT will be unable to buy tokens.

Remediation

It is recommended to restrict access to the `WithdrawTokens()` function by implementing an access control mechanism, such as the `onlyOwner` modifier or a similar role-based access control mechanism

Retest

This issue has been fixed by adding onlyOwner modifier

Bug ID #2 [Won't Fix]

Unrestricted Modification of Active Presales

Vulnerability Type
Logic Flow

Severity
Medium

Description
The `updatePresale()` function allows the modification of critical parameters (price, next stage price, tokens to sell, and hardcap) for any presale identified by `_id`, including those that are already active or have ended. This functionality can be abused to alter the terms of an ongoing or past presale, which can lead to unfair practices and potential financial manipulation.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L697

Impacts
Allowing modifications to active or past presales can undermine the integrity of the presale process. Participants in an ongoing presale might face unexpected changes in terms.

Remediation
Implement a validation check within the `updatePresale()` function to ensure that modifications are only allowed for upcoming presales that have not yet started. This can be done by checking the `Active` status of the presale before allowing any updates.

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #3 [Won't Fix]

Missing Validation for Active Presale Status

Vulnerability Type
Business Logic

Severity
Medium

Description
The `startPresale()` function in the provided smart contract code lacks validation to check whether the presale is already active. This allows the function to be called multiple times, each time overwriting the `startTime` and potentially manipulating the presale period.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L669

Impacts
If the `startPresale` function is called multiple times, it can reset the `startTime` to the current block timestamp and keep the presale active indefinitely.

Remediation
Implement a validation check within the `startPresale()` function to ensure that it cannot be called if the presale is already active. This can be done by adding a simple condition to verify the `Active` status before proceeding with the function logic. Eg:

```
function startPresale() public onlyOwner {
    require(!presale[presaleId].Active, "Presale is already active");
    presale[presaleId].startTime = block.timestamp;
    presale[presaleId].Active = true;
```

```
}
```

Retest

This issue has not been fixed. It is recommended to add a validation to make sure that presale is not active.

Bug ID #4 [Won't Fix]

Missing Price Feed Validation

Vulnerability Type
Input Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
The contract was found to be using `latestRoundData()` without proper input validations on the returned parameters which might result in a stale and outdated price.

Vulnerable Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L758

Impacts
Having oracles with functions to fetch price feed without any validation might introduce erroneous or invalid price values that could result in an invalid price calculation further in the contract.

Remediation

It is recommended to have input validations for all the parameters obtained from the Chainlink price feed. Here's a sample implementation:

```
(uint80 roundID ,int256 price, uint256 timestamp, uint80 answeredInRound) = AggregatorV3Interface(chainLinkAggregatorMap[underlying]).latestRoundData();
```

```
require(answer > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #5 [Won't Fix]

Chainlink Oracle Min/Max price validation

Vulnerability Type
Input Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset.

Vulnerable Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L758

Impacts
This would allow user to store their allocations with the asset but at the wrong price.

Remediation
The contract should check the returned answer/price against the minPrice/maxPrice and revert if the answer is outside of the bounds.

```
if (answer >= maxPrice or answer <= minPrice) revert(); // eg
```

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #6 [Partially 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://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L928
- https://bscscan.com/address/0xba4f808c6746f38e3833c63312dc4790c16fcb42#code#L52
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L833

Impacts
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 issue has been partially fixed.

Bug ID #7 [Fixed]

Missing Access Control for Withdraw Function

Vulnerability Type
Access Control

Severity
Low

Description
The `WithdrawContractFunds()` function lacks access control, allowing any external account to call this function and initiate the transfer of funds. Although the function transfers funds only to the `fundReceiver`, which is the owner.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L931

Impacts
In scenarios where the owner relies on specific triggers for withdrawals, this vulnerability could cause premature or unexpected fund transfers.

Remediation
In scenarios where the owner relies on specific triggers for withdrawals, this vulnerability could cause premature or unexpected fund transfers.

Retest
This issue has been fixed by adding onlyOwner modifier.

Bug ID #8 [Won't Fix]

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://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L537
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L13

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
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #9 [Won't Fix]

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://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L33
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L620
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L631
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L621
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L622
-

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
This issue has not been fixed. It is recommended to fix this as mentioned in Remediation.

Bug ID #10 [Won't Fix]

Floating and Outdated Pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.

The contract allowed floating or unlocked pragma to be used, i.e., ^0.8.0, ^0.8.19. 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 -

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L6
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L6

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 low.

Remediation
Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.25 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #11 [Won't Fix]

Missing Events in Important Functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code
The following functions were affected -

- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L36
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L40
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L44
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L48
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L55
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L630

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L634
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L669
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L674
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L684
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L697
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L717
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L726
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L782
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L927
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L931

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 important functions to keep track of them.

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #12 [Fixed]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple **require** statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the **require** statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L799

Impacts
Having a short descriptive message in the **require** statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation
It is recommended to add a descriptive message, no longer than 32 bytes, inside the **require** statement to give more detail to the user about why the condition failed.

Retest
This issue has been fixed by adding a message in require statement.

Bug ID #13 [Won't Fix]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informational

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L568

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement.
This reduces the overall size of the contracts and also helps in saving gas.

Remediation
If the library functions are not supposed to be used anywhere, consider removing them from the contract.

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #14 [Won't Fix]

Use Call instead of Transfer

Vulnerability Type
Best Practices

Severity
Informational

Description:
Using Solidity's transfer function has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:

- The withdrawer smart contract does not implement a payable fallback function.
- The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units.
- The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L907

Impacts
The transfer function has some restrictions when it comes to sending ETH to contracts in terms of gas which could lead to transfer failure in some cases.

Remediation
It is recommended to transfer ETH using the call() function, handle the return value using require statement, and use the nonreentrant modifier wherever necessary to prevent reentrancy.
Ref: https://solidity-by-example.org/sending-ether/

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #15 [Won't Fix]

Use Scientific Notations

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

Description
Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations. The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation also supported by Solidity.

Affected Code

- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L16
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L625
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L626

Impacts
Having a large number literals in the code increases the gas usage of the contract during its deployment and when the functions are used or called from the contract.
It also makes the code harder to read and audit and increases the chances of introducing code errors.

Remediation
Scientific notation in the form of 2e10 is also supported, where the mantissa can be fractional, but the exponent has to be an integer. The literal MeE is equivalent to M * 10**E. Examples include 2e10, 2e10, 2e-10, 2.5e1, as suggested in official solidity documentation.

https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals
It is recommended to use numbers in the form “35 * 1e7 * 1e18” or “35 * 1e25”.
The numbers can also be represented by using underscores between them to make them more readable such as “35_00_00_000”

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #16 [Won't Fix]

Boolean Equality

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be equating variables with a boolean constant inside a "require()" statement which is not recommended and is unnecessary. Boolean constants can be used directly in conditionals.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L649
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L798
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L806

Impacts
Equating the values to boolean constants in conditions cost gas and can be used directly.

Remediation
It is recommended to use boolean constants directly. It is not required to equate them to true or false.

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #17 [Won't Fix]

Custom error to save gas

Vulnerability Type
Gas Optimization

Severity
Gas

Description
During code analysis, it was observed that the smart contract is using the revert() statements for error handling. However, since Solidity version 0.8.4, custom errors have been introduced, providing a better alternative to the traditional revert(). Custom errors allow developers to pass dynamic data along with the revert, making error handling more informative and efficient. Furthermore, using custom errors can result in lower gas costs compared to the revert() statements.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L411

Impacts
Custom errors allow developers to provide more descriptive error messages with dynamic data. This provides better insights into the cause of the error, making it easier for users and developers to understand and address issues.

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

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #18 [Won't Fix]

Cheaper Inequalities in if()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L403
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L811

Impacts
Using strict inequalities inside "if" statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest
This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #19 [Won't Fix]

Cheaper Inequalities in require()

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be performing comparisons using inequalities inside the require statement. When inside the require statements, non-strict inequalities (>=, <=) are usually costlier than strict equalities (>, <).

Affected Code

- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L56
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L304
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L799
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L807
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L826
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L906
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L918

Impacts
Using non-strict inequalities inside "require" statements costs more gas.

Remediation

It is recommended to go through the code logic, and, **if possible**, modify the non-strict inequalities with the strict ones to save gas as long as the logic of the code is not affected.

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #20 [Won't Fix]

Gas Optimization in Require/Revert Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The **require/revert** 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://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L49
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L56
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L826

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

Remediation
It is recommended to shorten the strings passed inside **require/revert** 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

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #21 [Won't Fix]

Cheaper Conditional Operators

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Upon reviewing the code, it has been observed that the contract uses conditional statements involving comparisons with unsigned integer variables. Specifically, the contract employs the conditional operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L647
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L648
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L704
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L705
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L811

Impacts
Employing x != 0 in conditional statements can result in reduced gas consumption compared to using x > 0. This optimization contributes to cost-effectiveness in contract interactions.

Remediation

Whenever possible, use the x != 0 conditional operator instead of x > 0 for unsigned integer variables in conditional statements.

Retest

This issue has not been fixed. It is recommended to fix as mentioned in Remediation.

Bug ID #22 [Won't Fix]

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://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L14
- https://bscscan.com/address/0xbA4f808c6746F38E3833c63312dc4790c16fcB42#code#L15
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L539
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L540
- https://bscscan.com/address/0xb5e425dA77922E1Ae63B43B0B9a08c9ed87FE420#code#L563

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 issue has not been fixed. It is recommended to fix as mentioned in Remediation.

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.