CredShields

Smart Contract Audit

Oct 31st, 2022 • CONFIDENTIAL

Description

This document details the process and result of the BuscemiBeats NFT Token smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of BuscemiBeats NFT between Oct 21st, 2022, and Oct 27th, 2022. And a retest was performed on 30th Oct 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

BuscemiBeats NFT

Table of Contents

1. Executive Summary

BuscemiBeats NFT engaged CredShields to perform a smart contract audit from Oct 21, 2022, to Oct 27th, 2022. During this timeframe, twelve (12) vulnerabilities were identified. **A retest was performed on 30th Oct 2022, and all the bugs have been addressed.**

During the audit, zero (0) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "BuscemiBeats NFT" 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
BuscemiBeats NFT	0	0	1	2	3	6	**12**
	0	**0**	**1**	**2**	**3**	**6**	**12**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in BuscemiBeats NFT's scope during the testing window while abiding by the policies set forth by BuscemiBeats NFT'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 BuscemiBeats NFT'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 BuscemiBeats NFT introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate BuscemiBeats NFT'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 BuscemiBeats NFT assets.

2. Methodology

BuscemiBeats NFT engaged CredShields to perform a BuscemiBeats NFT 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 validate the vulnerabilities during the audit phase.

A testing window from Oct 21st, 2022, to Oct 27th, 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://github.com/jawkhan/BuscemiBeats-NFT

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

BuscemiBeats NFT 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, Twelve (12) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Missing Constant Attribute in Variables	Gas	Gas Optimization
Use of SafeMath	Gas	Gas Optimization
Hardcoded Static Address	Informational	Missing Best Practices
Boolean Equality	Gas	Gas Optimization
Gas Optimization in Require Statements	Informational	Missing Best Practices
Require with Empty Message	Gas	Gas Optimization

Gas Optimization in Increments	Gas	Gas Optimization
Unchecked Array Length	Low	Input Validation
Missing NatSpec Comments	Informational	Missing best practices
Missing Input Validation	Low	Input Validation
Business Logic in Whitelisting Addresses	Medium	Business Logic

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 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

BuscemiBeats NFT is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 30th Oct 2022 and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Won't Fix**
Missing Constant Attribute in Variables	Gas	**Fixed**
Use of SafeMath	Gas	**Fixed**
Hardcoded Static Address	Informational	**Fixed**
Boolean Equality	Gas	**Fixed**
Gas Optimization in Require Statements	Informational	**Fixed**
Require with Empty Message	Gas	**Fixed**
Gas Optimization in Increments	Gas	**Fixed**
Unchecked Array Length	Low	**Won't Fix**
Missing NatSpec Comments	Informational	**Won't Fix**
Missing Input Validation	Low	**Fixed**
Business Logic in Whitelisting Addresses	Medium	**Fixed**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Won't Fix]

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 <0.9.0**.
This allows the contracts to be compiled with all the solidity compiler versions between **>=0.8.0 <0.9.0**. The following contracts were found to be affected -

Affected Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L3

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 **0.8.7** pragma version.
Reference: https://swcregistry.io/docs/SWC-103

Retest:

Bug ID#2 [Fixed]

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:

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L27

```
uint256 public maxSupply = 3000;
```

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

“maxSupply” has been set to constant.

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L25

Bug ID#3 [Fixed]

Use of SafeMath

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
SafeMath library is found to be used in the contract. This increases gas consumption more than traditional methods and validations if done manually.
Also, Solidity **0.8.0** and above includes checked arithmetic operations by default, and this renders SafeMath unnecessary.

Affected Code:

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L8

```
...
contract BuscemiBeats is ERC721, AccessControl {
 using Strings for uint256;
 using SafeMath for uint256;
...
```

Impacts:
This increases the gas usage of the contract.

Remediation:
We do not recommend using the SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.
The compiler above 0.8.0+ automatically checks for overflows and underflows.

CRED SHIELDS

Retest:
Use of safemath has been removed.

Bug ID#4 [Fixed]

Hardcoded Static Address

Vulnerability Type
Missing Best Practices

Severity
Informational

Description
The contract was found to be using hardcoded addresses on Line 37. The "_setupRole" is taking a hard-coded address for the role "META49_ROLE". This is also used on a "call()" at line 173.
This could have been optimized using dynamic address update techniques along with proper access control to aid in address upgrade at a later stage.

Affected Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L37
- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L173

```
    _setupRole(META49_ROLE, 0x091Ed07Fa9eE1F67B39e0dFB203e4115C786Ab64);

...
...

   (bool os, ) =
payable(0x091Ed07Fa9eE1F67B39e0dFB203e4115C786Ab64).call{value:
address(this).balance}("");
```

Impacts

Hardcoding address variables in the contract make it difficult for it to be modified at a later stage in the contract as everything will need to be deployed again at a different address if there's a change in address.

Remediation

It is recommended to create dynamic functions to address upgrades so that it becomes easier for developers to make changes at a later stage if necessary.
There should also be a zero address validation on the address type parameters to make sure the tokens are not lost.

Retest:

The smart contract now uses the constructor to set the address, and no hardcoded address is in use.
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L36-L40

Bug ID#5 [Fixed]

Boolean Equality

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be equating “whitelist[msg.sender]” with a boolean constant “true” inside a “require()” statement which is not recommended and unnecessary. Another vulnerable instance was detected at line <> on “if (revealed == false)” condition. Boolean constants can be used directly in conditionals.

Affected Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L57
- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L111

```
  if (premint && allowListOnly) {
     require(whitelist[msg.sender] == true, "Unfortunately, you didn't
make the whitelist
to mint a Buscemi Beat.");
    }

...

if (revealed == false)
```


CRED SHIELDS

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:

Boolean equality has been removed

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L60

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L122

Bug ID#6 [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.

Vulnerable Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L57
- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L109

```
    require(whitelist[msg.sender] == true, "Unfortunately, you didn't
make the whitelist to mint a Buscemi Beat.");

...

    require(_exists(_tokenId),"ERC721Metadata: URI query for
nonexistent token");
```

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.

This is just one example. It is recommended to go through the code and find all the occurrences that are longer than 32 bytes.

Retest

The contract now uses strings of less than 32 bytes.

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L120

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L60

Bug ID#7 [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://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L174

```
   (bool os, ) =
payable(0x091Ed07Fa9eE1F67B39e0dFB203e4115C786Ab64).call{value:
address(this).balance}("");
   require(os);
```

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

No longer an empty message.

https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L197

Bug ID#8 [Fixed]

Gas Optimization in Increments

Vulnerability Type
Gas optimization

Severity
Gas optimization

Description
The contract uses two for loops**,** which use post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Vulnerable Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L74-L78
- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L177-L182

```
    for (uint i = 0; i < addresses.length; i++) {
        whitelist[addresses[i]] = true;
    }

...

    for (uint256 i = 0; i < _mintAmount; i++) {
      supply.increment();
      _safeMint(_receiver, supply.current());
    }
```

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 meanwhile saves some gas.

Retest

Now ++i is used to save additional gas.
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L78
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L201

Bug ID#9 [Won't Fix]

Unchecked Array Length

Vulnerability Type
Gas Griefing

Severity
Low

Description
Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Ethereum miners impose a limit on the total number of Gas consumed in a block. If the "array.length" is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed.

Affected Code

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L74-L78

```
 function whitelistAddresses(address[] memory addresses) public
onlyRole(META49_ROLE) {
   for (uint i = 0; i < addresses.length; i++) {
       whitelist[addresses[i]] = true;
   }
 }
```

Impacts
if an array enumerates all registered addresses, an adversary can register many addresses causing the transaction to fail.

CRED SHIELDS

Remediation

Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit, which can cause the whole contract to be stalled at a certain point. Therefore, loops with a bigger or unknown number of steps should always be avoided.

Retest

-

Bug ID#10 [Won't Fix]

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

Affected Code:

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol

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:

Bug ID#11 [Fixed]

Missing Input Validation

Vulnerability Type
Input validation

Severity
Low

Description:
Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components.
When the smart contract does not validate the inputs properly, it may introduce a range of vulnerabilities.
The contract did not implement any input validation when setting the cost for minting in the function “setCost()” and when setting the “setMaxMintAmountPerTx()”.

Vulnerable Code:

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L135-L137
- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L139-L141

```
 function setCost(uint256 _cost) public onlyRole(META49_ROLE) {
   cost = _cost;
 }

 function setMaxMintAmountPerTx(uint256 _maxMintAmountPerTx) public
onlyRole(META49_ROLE) {
   maxMintAmountPerTx = _maxMintAmountPerTx;
```

```
}
```

Impacts:
If the cost is set to 0 due to any errors, users will be able to mint freely. Missing input validation on sensitive function parameters may introduce inconsistencies and erroneous logic when the user will the functions.

Remediation:
Use a `require()` input validation in the function parameters shown above.

Retest:
A check has been added for the value to be greater than 0
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L151-L154
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L162-L165

Bug ID#12 [Fixed]

Business Logic in Whitelisting Addresses

Vulnerability Type
Business Logic

Severity
Medium

Description:
There is a function called “whitelistAddresses()” which is being used to whitelist addresses for minting but there is no such function to remove addresses once they are whitelisted.

Vulnerable Code:

- https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L74-L78

```
 function whitelistAddresses(address[] memory addresses) public
onlyRole(META49_ROLE) {
   for (uint i = 0; i < addresses.length; i++) {
       whitelist[addresses[i]] = true;
   }
 }
```

Impacts:
If the owner whitelists an address and later on decides to remove the same address from the whitelist mapping due to any reason, they won’t be able to do so, and the address will always be whitelisted.
This may also cause issues if a wrong address is whitelisted by mistake.

Remediation:
Create a function to remove addresses from the whitelist.

Retest
A function has been created to remove addresses from the whitelist.
https://github.com/jawkhan/BuscemiBeats-NFT/blob/main/contracts/BuscemiBeats.sol#L84-L88

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 of the project's absolute security.
CredShields Audit team owes no duty to any third party by virtue of publishing these Reports.