Audit Report

Smart Contract
METAC (BEP20) token

0xF82F9352A6d4f99805782840eAE3EB4D76DcD0e9

May 21st, 2022

Summary

This report has been prepared for METAC smart contracts, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
- Provide more comments per each function for readability, especially contracts are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	Vaiyo
Description	A BEP20 Token
Platform	BSC
Language	Solidity

Codebase	https://bscscan.com/address/0xF82F9352A6d4f99805782840eAE3EB4D76DcD0e9
Commits	METAC

Audit Summary

Delivery Date	May 21, 2022
Audit Methodology	Static Analysis, Manual Review
Key Components	

Vulnerability Summary

Total Issues	5
Critical	0
Major	0
Medium	0
Minor	3
Informational	2
Discussion	0

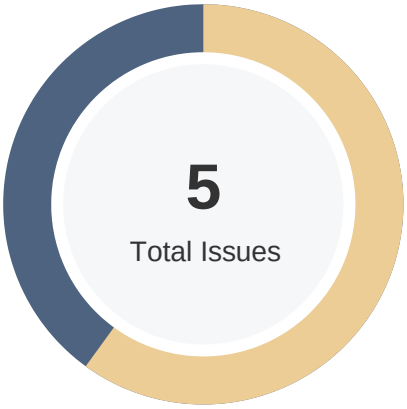
Audit Scope

ID	file	SHA256 Checksum
LTL	METAC.sol	4D4D525BAD4BC3FBDB00CAFB103C534E272DE2584BE6E4B6E913A29E82883731

To bridge the trust gap between owner and users, the owner needs to express a sincere attitude with the consideration of the administrator team’s anonymousness. The owner has the responsibility to notify users with the following capability of the administrator:

`Owner` has the privilege to mint uncapped tokens.

Findings



Critical	**0** (0.00%)
Major	**0** (0.00%)
Medium	**0** (0.00%)
Minor	**3** (60.00%)
Informational	**2** (40.00%)
Discussion	**0** (0.00%)

ID	Title	Category	Severity	Status
LTL-01	Unlocked Compiler Version & Version Inconsistency	Language Specific	Informational	Acknowledged
LTL-02	Delegation Not Moved Along with Tokens	Logical Issue	Minor	Acknowledged
LTL-03	Proper Usage of `public` and `external`	Gas Optimization	Informational	Acknowledged
LTL-04	Owner Capability	Centralization / Privilege	Minor	Pending
LTL-05	Lack of Input Validation	Volatile Code	Minor	Acknowledged

LTL-01 | Unlocked Compiler Version & Version Inconsistency

Category	Severity	Location	Status
Language Specific	Informational	METAC.sol : 7, 258	Acknowledged

Description

The contract has unlocked compiler versions. An unlocked compiler version in the contract's source code permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to ambiguity when debugging as compiler-specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Examples:

`pragma solidity ^0.6.0;` Line 7 `pragma solidity ^0.6.2;` Line 258

Recommendation

It is a general practice to instead lock the compiler at a specific version rather than allow a range of compiler versions to be utilized to avoid compiler-specific bugs and be able to identify ones more easily. We recommend locking the compiler at the lowest possible version that supports all the capabilities wished by the codebase. This will ensure that the project utilizes a compiler version that has been in use for the longest time and as such is less likely to contain yet-undiscovered bugs.

LTL-02 | Delegation Not Moved Along with Tokens

Category	Severity	Location		Status
Logical Issue	Minor	METAC.sol	: 736~737, 744	Acknowledged

Description

The voting power of delegation is not transferred along with the token in function `transfer()` and `burn()`.

Recommendation

Consider overriding `_transfer()` and `burn()` in METAC and invoking function `_movedelegates()`.

Alleviation

The team responds that all tokens does not use delegate function in transfer function now.

LTL-03 | Proper Usage of `public` and `external`

Category	Severity	Location	Status
Gas Optimization	Informational	METAC.sol : 738, 743	Acknowledged

Description

`public` functions that are never called by the contract could be declared `external`. When the inputs are arrays, `external` functions are more efficient than `public` functions.

Recommendation

Consider using the external attribute for functions never called within the contract.

LTL-04 | Owner Capability

Category	Severity	Location	Status
Centralization / Privilege	**Minor**	METAC.sol : 738	**Pending**

Description

The owner has the capability to mint uncapped tokens to any address through `mint()`.

Recommendation

To bridge the trust gap between owner and users, the owner needs to express a sincere attitude with the consideration of the administrator team's anonymousness. The owner has the responsibility to notify users with the following capability of the administrator:

`Owner` has the privilege to mint uncapped tokens.

LTL-05 | Lack of Input Validation

Category	Severity	Location	Status
Volatile Code	Minor	METAC.sol : 843	Acknowledged

Description

The `expiry` value should be verified greater than now in function `delegateBySig()`.

Recommendation

Consider adding time check as follows: require(now <= expiry, "METAC::delegateBySig: signature expired");

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

End of Document