SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Amoss
Date: April 26, 2022

Document

Name	Smart Contract Code Review and Security Analysis Report for Amoss.
Approved By	Evgeniy Bezuglyi | SC Department Head at Hacken OU
Type	BEP20 token
Platform	EVM
Language	Solidity
Methods	Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Website	https://amoss.ai/
Timeline	18.04.2022 - 29.04.2022
Changelog	26.04.2022 - Initial Review 29.04.2022 - Second Review

Table of contents

Introduction 4
Scope 4
Checked Items 5
System Overview 8
Executive Summary 9
Severity Definitions 10
Findings 11
Disclaimers 13

Introduction

Hacken OÜ (Consultant) was contracted by Amoss (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project is smart contract deployed to bscscan:

Initial review scope

Deployed Contracts Address:
https://bscscan.com/address/0xab025849492dF2D0A04F32e60999D5943D9C821f
Technical Documentation: Yes
https://drive.google.com/file/d/1uquGJM4TTDvubkLklkbXKDxq9Ivawx-h/view
JS tests: No
Contracts:
File: Amoss.sol

Second review scope:

Deployed Contracts Address:
https://bscscan.com/address/0x4908d3b4A25Cdf79b771A046F5331d5E545C97B5
Technical Documentation: Yes
https://drive.google.com/file/d/1uquGJM4TTDvubkLklkbXKDxq9Ivawx-h/view
JS tests: No
Contracts:
File: Amoss.sol

Checked Items

We have audited provided smart contracts for commonly known and more specific vulnerabilities. Here are some of the items that are considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Not Relevant
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be destroyed until it has funds belonging to users.	Not Relevant
Reentrancy	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Uninitialized Storage Pointer	SWC-109	Storage type should be set explicitly if the compiler version is < 0.5.0.	Not Relevant
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Not Relevant
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Passed
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless it is required.	Passed

Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	Passed
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes.	Passed
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP standards violation	EIP	EIP standards should not be violated.	Passed
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Not Relevant
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer.	Passed

Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block gas limit.	Passed
Style guide violation	**Custom**	Style guides and best practices should be followed.	Failed
Requirements Compliance	**Custom**	The code should be compliant with requirements provided by the Customer,	Passed
Repository Consistency	**Custom**	The repository should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Failed
Tests Coverage	**Custom**	The code should be covered with unit tests. Tests coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Failed

System Overview

Amoss is a project with a ERC-20 token contract:

- Amoss – simple ERC-20 token that mints all initial supply to a deployer. Additional minting is not allowed and not provided.
  It has the following attributes:
  - Name: Amoss
  - Symbol: AMOSS
  - Decimals: 18
  - Total supply: 1 000 000 000 tokens.

Privileged roles

- The owner - the address that receives all the tokens after the deployment.

Executive Summary

The score measurements details can be found in the corresponding section of the methodology.

Documentation quality

The Customer provided functional requirements in a whitepaper and technical requirements in the contract`s code. The total Documentation Quality score is **10** out of **10**.

Code quality

The total CodeQuality score is **4** out of **10**. Code follows official style guides but violates them in some cases of naming function parameters. No unit tests were provided.

Architecture quality

The architecture quality score is **10** out of **10**. Code use best practices.

Security score

As a result of the audit, security engineers found **3** low severity issues.

As a result of the second review, security engineers found no new issues. All the issues from the previous revision were fixed. As a result, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.4**

1
2
3
4
5
6
7
8
9
10
You are here

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they cannot lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution

Findings

■■■■ Critical

No high severity issues were found.

■■■ High

No high severity issues were found.

■■ Medium

No medium severity issues were found.

■ Low

1. **Functions can be declared as external.**

   To save Gas, public functions that are never called in the contract should be declared as external.

   Contracts: Amoss.sol

   Functions: transfer, burn, allowance, approve, transferFrom, increaseAllowance, decreaseAllowance

   Recommendation: Declare mentioned functions as external.

   Status: Fixed (second review)

2. **Possible simplifying of the variable declaration.**

   Declaration of the variable *_totalSupply* is followed by mathematical operations, which can be simplified.

   Contracts: Amoss.sol

   Functions: -

   Recommendation: Declare mentioned variable with lesser mathematical operations, for example 1_000_000_000 * 10**18 or 1_000_000_000 * 10**_*decimals*.

   Status: Fixed (second review)

3. **Creation of the redundant function.**

   Function *getOwner* has equal return as a function *owner*. Calling the *getOwner* calls the *owner function,* so it is more expensive.

   Contracts: Amoss.sol

   Functions: owner, getOwner

Recommendation: Remove redundant function.

Status: Fixed (second review)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the audit cannot guarantee the explicit security of the audited smart contracts.