SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Yokai
Date: July 08th, 2022

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Yokai
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC721
Platform	EVM
Network	Ethereum
Language	Solidity
Methods	Manual Review, Automated Review, Architecture Review
Website	In development
Timeline	01.07.2022 - 08.07.2022
Changelog	06.07.2022 - Initial Review 08.07.2022 - Initial Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Yokai (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 contracts in the repository:

Initial review scope

Repository:
https://github.com/DorBocco/YokaiSC
Commit:
62edd61
Technical Documentation:
Type: Whitepaper (partial functional requirements provided)
Attached PDF

Type: Technical description
Link

Integration and Unit Tests:
https://github.com/DorBocco/YokaiSC/tree/master/test
Contracts:
File: ./contracts/Yokai.sol
SHA3: 5e4c8bf6b610d48d6397f5f0860e50883a411fcd9427a78f8f72109b2517726a

Second review scope

Repository:
https://github.com/DorBocco/YokaiSC
Commit:
a03a4d6
Technical Documentation:
Type: Whitepaper (partial functional requirements provided)
Attached PDF

Type: Mint Overview
Attached PDF

Type: Technical description
Link

Integration and Unit Tests:
https://github.com/DorBocco/YokaiSC/tree/master/test
Contracts:
File: ./contracts/Yokai.sol
SHA3: f7329aa95f5ab21803f36f812ab626e55615f6eb70d2b39c50ce32d2cc534662

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.

Executive Summary

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

Documentation quality

The total Documentation Quality score is **8** out of **10**. Functional requirements are not provided.

Code quality

The total CodeQuality score is **9** out of **10**. Deployment and basic user interactions are covered with tests. Code violates the order of functions defined in the style guide.

Architecture quality

The architecture quality score is **10** out of **10**.

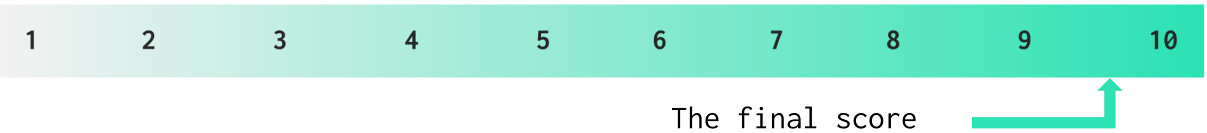
Security score

As a result of the audit, 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: **9.7**.



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.	Passed
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 self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
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.	Not Relevant
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	SWC-115	tx.origin should not be used for	Passed

through tx.origin		authorization.	
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Not Relevant
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifier should always be used. All parameters from the signature should be used in signer recovery	Not Relevant
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 or be predictable.	Not Relevant
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	Passed

		data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	
Style guide violation	**Custom**	Style guides and best practices should be followed.	Failed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

System Overview

Yokai: Ownable and Enumerable ERC721 token with ERC2981 royalty fees

Privileged roles

- Owner: can change the royalty, set whitelist root, reveal URI string to the users, change sale type, update sale receiver, set token price, and set maximum tokens the user can mint.

Risks

- There is no strict restriction on when the Owner can change state variables. The Owner can change the merkle root during the phase of whitelist minting or increase amount of tokens an address can mint.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

Highly permissive role

The owner can change the whitelist root, sale type, and maximum tokens the user can mint.

This can lead to unfair token minting.

Contracts: Yokai.sol

Functions: setWhitelistRoot, switchSaleType, setMaxMintPerPerson

Recommendation: Add highly permissive functionality to the documentation or create rules according to which the functions can be called.

Status: Mitigated (added functionality to documentation)

■■ Medium

Insufficient supply check

The insufficient check may result in bigger amount of tokens minted than the maximal supply.

Contract: Yokai.sol

Functions: whitelistMint

Recommendation: Check if totalSupply() + num < MAX_TO_MINT

Status: Fixed (a03a4d6)

■ Low

1. Floating pragma

Contracts files use floating pragma ^0.8.0

Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Contracts: Yokai.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

Status: Fixed (a03a4d6)

2. State variable default visibility

The explicit visibility makes it easier to catch incorrect assumptions about who can access the variable.

Contracts: Yokai.sol

Variables: MAX_TO_MINT

Recommendation: Variables can be specified as *public*, *internal,* or *private*. Explicitly define visibility for all state variables.

Status: Fixed (a03a4d6)

3. **Gas optimization**

   Using best Solidity code practices saves some Gas and reduces the transaction cost.

   Contract: Yokai.sol

   Variables: MAX_TO_MINT

   Recommendation: Make a variable constant.

   Status: Fixed (a03a4d6)

4. **Environment misses package.json**

   An inconsistent environment makes it harder to test smart contracts.

   Recommendation: Add package.json

   Status: Fixed (a03a4d6)

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 report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted to and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and 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.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

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, Consultant cannot guarantee the explicit security of the audited smart contracts.