SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Wombat
Date: 12 June, 2023

Document

Name	Smart Contract Code Review and Security Analysis Report for Wombat
Approved By	Paul Fomichov | Lead Solidity SC Auditor at Hacken OU
Tags	ERC721 token; ERC1155 token; Staking;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://website.com
Changelog	29.05.2023 - Initial Review 12.06.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Wombat (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.

System Overview

Wombat — is a generic NFT smart contract that allows staking of both ERC721 and ERC1155 token with the following contracts:

- *Staking* — a staking smart contract that allows staking both ERC721 and ERC1155 tokens. Withdrawals are possible only after a certain period of time set by the owner of the wallet.
- *SignatureStakingValidator* — a smart contract, which uses the data argument to all staking operations to expect a signature that was created in the client backend and verifies it.
- *IStakeValidator* — an interface for *SignatureStakingValidator*.

Privileged roles

- Staking.sol :
  - Contract Owner :
    - Can set a validator address.

Executive Summary

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

Documentation quality

The total Documentation Quality score is **9** out of **10**.

- Functional requirements are provided.
    - Project overview is detailed
    - Use cases are described and detailed.
    - Project overview is detailed
- Technical description is inadequate.
    - Run instructions are not provided.
    - NatSpec is sufficient.
    - Technical specification is provided.

Code quality

The total Code Quality score is **8** out of **10**.

- Best practice violation.
- The development environment is not configured: run instructions are not provided.

Test coverage

Code coverage of the project is **94.12%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is present.
- Interactions by several users are tested.

Security score

As a result of the audit, the code contains no severity 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: **9.2**. The system users should acknowledge all the risks summed up in the risks section of the report.

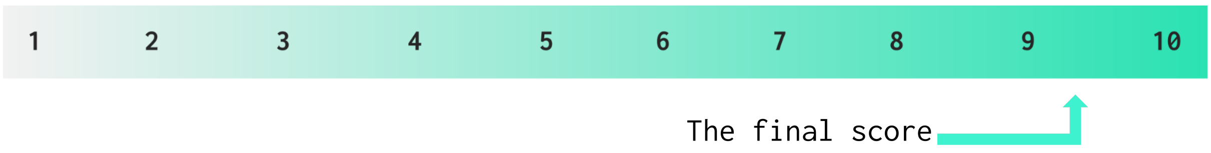

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
29 May 2023	3	1	0	0
12 June 2023	0	0	0	0

Risks

No potential security risks were found during the audit research.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	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	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	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	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Not Relevant	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	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. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant	
Gas Limit and Loops	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	Style guides and best practices should be followed.	Failed	I04
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Failed	
Secure Oracles Usage	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	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Not Relevant	

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

No high severity issues were found.

■■ Medium

M01. Best Practice Violation: Non Upgradeable Ownable Inheritance

Impact	High
Likelihood	Low

Upgradeable contracts that inherit from OpenZeppelin's Ownable contract should use its upgradeable version: OwnableUpgradeable. This happens because Ownable contract will set owner to the msg.sender once the contract is deployed, but initializable contracts need to do this step during initialization, not deployment.

Path: ./contracts/Staking.sol

Recommendation: Replace Ownable with OwnableUpgradeable and call *OwnableUpgradeable.__Ownable_init()* during initialization.

Found in: 3d7bb8772

Status: Mitigated (Revised commit: db9dd9565. With Customer notice: *Using OwnableUpgradeable.__Ownable_init() would assign ownership to msg.sender instead. I think we would like to keep the current behavior.*

■ Low

L01. Missing Events

Impact	Low
Likelihood	Medium

Events for critical state changes should be emitted for tracking things off-chain.

Path: ./Staking.sol : setValidator();

Recommendation: Create and emit related events.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565)

L02. Variable Shadowing

Impact	Medium
Likelihood	Low

In the *initialize()* function, the parameter *_owner* is shadowing the *_owner* state variable from the *Ownable* contract.

Path: ./Staking.sol : initialize();

Recommendation: Rename the related variable.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565)

L03. Missing Zero Address Validation

Impact	Medium
Likelihood	Low

Address parameters are used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

The *setValidator()* function modifies the state variable *validator* by assigning it a new address value. If the *validator* address is set to 0x0, the execution of key functions will be reversed.

This could lead to a Denial of Service (DoS) issue, as the contract may become unresponsive or behave unexpectedly if it is trying to interact with a non-existent contract at the 0x0 address.

Path: ./Staking.sol : initialize(), setValidator();

Recommendation: Implement zero address checks.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565)

Informational

I01. State Variables Default Visibility

Variable`s *MAX_ACTIVE_STAKES* visibility is not specified. Specifying state variables visibility helps to catch incorrect assumptions about who can access the variable.

This improves the contract's code quality and readability.

Path: ./Staking.sol : MAX_ACTIVE_STAKES;

Recommendation: Specify variables as public, internal, or private. Explicitly define visibility for all state variables.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565)

I02. Inefficient Gas Model

The *StakeRecord* structure utilizes the *uint256 claimableAt* variable for timestamp storage, rather than employing the *uint64* type.

When storing numerous instances to Smart Contract storage of it in a single transaction, this could potentially result in a substantial rise in the Gas cost per transaction.

Path: ./Staking.sol : StakeRecord;

Recommendation: Change the variable type used for timestamp storage to the *uint64* type.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565)

I03. Style Guide Violation: Order Of Layout

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions

Path: ./Staking.sol;

Recommendation: Change order of layout to fit Official Style Guide.

Found in: 3d7bb8772

Status: Fixed (Revised commit: db9dd9565.)

I04. Style Guide Violation

The provided projects should follow the official guidelines: Order of Functions.

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Within a grouping, place the view and pure functions last.

Paths: ./Staking.sol;

Recommendation: Follow the official Solidity guidelines.

Found in: db9dd9565

Status: New

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing 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 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, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/wombat-tech/evm-nft-staking-contract/
Commit	3d7bb8772
Whitepaper	-
Requirements	https://github.com/wombat-tech/evm-nft-staking-contract/blob/main/README.md
Technical Requirements	https://github.com/wombat-tech/evm-nft-staking-contract/blob/main/README.md
Contracts	File: ./contracts/IStakeValidator.sol SHA3: 9f7ea32d3a504efca03225579fd5aa70b88cf8786dafe0995880bdbdff3c8e20 File: ./contracts/SignatureStakeValidator.sol SHA3: ea53505af674608703da9ba7416c8668c178fdaa7182335048d8f761d037f5f1 File: ./contracts/Staking.sol SHA3: c05ed4fa4bab4f49208ed097c3aaf2e5f80a7bf6f6c39bc792beecfdafc46020

Second review scope

Repository	https://github.com/wombat-tech/evm-nft-staking-contract/
Commit	db9dd9565
Whitepaper	-
Requirements	https://github.com/wombat-tech/evm-nft-staking-contract/blob/main/README.md
Technical Requirements	https://github.com/wombat-tech/evm-nft-staking-contract/blob/main/README.md
Contracts	File: ./contracts/IStakeValidator.sol SHA3: 9f7ea32d3a504efca03225579fd5aa70b88cf8786dafe0995880bdbdff3c8e20 File: ./contracts/SignatureStakeValidator.sol SHA3: ea53505af674608703da9ba7416c8668c178fdaa7182335048d8f761d037f5f1 File: ./contracts/Staking.sol SHA3: 2dae4636942e071a0da2630ced0777758e6bf98af678de1ceaa0b4c99bcd4227