SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Metaspace Technologies
Date: February 17, 2023

Document

Name	Smart Contract Code Review and Security Analysis Report for Metaspace Technologies
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; NFT Staking;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://metaspacechain.com/
Changelog	30.01.2023 - Initial Review 02.02.2023 - Second Review 17.02.2023 - Third Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Metaspace Technologies (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/Metaspacemetaverse/MLD https://github.com/Metaspacemetaverse/NFT-Staking
Commit	219acfbf873c1cacdecc1755b14fe1a3f630979c fd3b143b76f4850c49652dc396390e96e99fbf5b
Whitepaper	Link
Functional Requirements	Link
Contracts	File: ./contracts/MetaLordToken.sol SHA3: e035c5acc2f9dc76851472658fd7c7e2f8dc8c6fabc93e6ab59572d23cbd9b3d File: ./contracts/NFTStaking.sol SHA3: b40d04baa4389af317c2cc73c57676406a2c35a5df2c3f14e2b79e1033920720

Second review scope

Repository	https://github.com/Metaspacemetaverse/MLD https://github.com/Metaspacemetaverse/NFT-Staking
Commit	45006fcbff347818a5d6489e0355267f9966ee41 405cc3d0d20ea2374ca4196caedfe0e1fba4620f
Whitepaper	Link
Functional Requirements	Link
Contracts	File: ./contracts/MetaLordToken.sol SHA3: f4770eb3b1ef8d149f50cb6b02cb8f345603935063bbd862114ac8e168eeda8f File: ./contracts/NFTStaking.sol

	SHA3: 02e4d8b69db26e759f00c5b1f8f3768f3241f862d716b3409da443c3ff2c56d9

Third review scope

Repository	https://github.com/Metaspacemetaverse/MLD https://github.com/Metaspacemetaverse/NFT-Staking
Commit	e709fe9bc3d61ea866da10eccde316e2c4204ab0 23d65f66615ffea46ca6b771dcb023f0682dbcb4
Whitepaper	Link
Functional Requirements	Link
Contracts	File: ./contracts/MetaLordToken.sol SHA3: 4ee38bbb5a5d50487b65dc561e4ed25daa918f942ecab1d1779a5877aeafb691 File: ./contracts/NFTStaking.sol SHA3: 28ae9f1763d7009932298698cef64635ead5b69472faf5aa481120efc53e8043

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
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 by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality

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 **3** out of **10**.

- Functional requirements are mostly missed.
- Technical description is not provided.
- The whitepaper is a pitch deck that does not have much to do with the scope of the audit.

Code quality

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

- A few template code patterns were found.
- The development environment is not configured completely.

Test coverage

Code coverage of the project is **10.12%** (branch coverage).

- Tests are missing for the staking repo.
- MLD repo has a 31.25% branch test coverage. 10.12% is the average computed over net code LOCs.

Security score

As a result of the audit, the code contains **1** medium and **1** low severity issues. The security score is **9** 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: **3.9**.

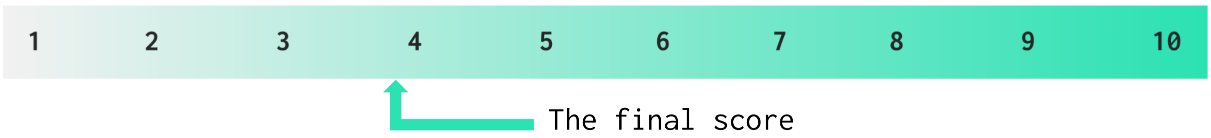

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
30 January 2023	11	5	1	4
2 February 2023	3	2	1	2
17 February 2023	1	1	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items 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 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.	Not Relevant
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	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	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.	Not Relevant
Assets Integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	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.	Passed
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.	Failed
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.	Passed
Tests Coverage	**Custom**	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.	Failed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

System Overview

MetaSpace metaverse is an interactive virtual world where people can socialize, play, and work together. The project combines different segments such as Gaming, Entertainment, NFT Marketplace, Virtual Concerts and more into an immersive experience.
The audit scope is composed by the MetaLord ERC20 token and an NFT staking contract.

- *MetaLordToken* — simple ERC-20 token with a hardcoded minting schedule for the first 5 years. After the 5th year, new tokens could be minted according to community votes, with an upper limit of 50M per year.
  It has the following attributes:
  - Name: MetalordToken
  - Symbol: MLD
  - Decimals: 18
  - Total supply: 2.5B first 5 years + up to 50M yearly after 5th year.
- *NFTStaking* — a contract that rewards users for staking whitelisted NFTs collections. Rewards are computed off-chain. The project's metaverse token lordToken is required to perform the stake: part of it gets burned, the rest goes to the project's master wallet.

Privileged roles

- The owner of the *NFTStaking* contract can update whitelist new NFT collections, the withdrawal fee percentage, the amount of lord tokens required to stake NFTs (and the burning percentage), the master wallet address and chainlink and API-related variables.
- The owner of *MetaLordToken* is responsible of calling the minting functions mintYearlyToken() and mintExtraToken(). He can update the partners' and community's treasuries addresses.

Risks

- The reward is calculated off-chain with the use of chainlink combined with metaspaceAPI, if the API stops working, this could create problems in the calculation of the rewards.
- Since the rewards are computed off-chain, their computation is out of scope.
- If the LINK tokens are exhausted on the contract, the oracle calls will fail, and the rewards will not be updated.
- The community can decide to mint tokens after the 2.5B supply, but the function mintExtraToken() in the token contract does not have a check to fail the transaction if the community votes no.
  Part of the voting process is off-chain, so it cannot be fully verified in this audit.

- The system relies on the secureness of the owner's private keys, which can impact the execution flow and secureness of the funds. We recommend this account to be at least ⅗ multi-sig.

Findings

■■■■ Critical

C01. Denial Of Service

The function *_updateMintingYear()* takes care of synchronizing the storage variable *mintingYear* with the actual current year gathered from block.timestamp.

In the function *mintYearlyToken()*, *_updateMintingYear()* gets called after *mintingData* gets loaded with the struct referring to the old value of *mintingYear*. When the first year passes, the requirement check on *mintingData.isMinted* will fail before *_updateMintingYear()* gets called.

The protocol will only be able to mint the first year batch of tokens.

Path: MLD/contracts/MetaLordToken.sol : mintYearlyToken()

Recommendation: Move the call to *_updateMintingYear()* to the beginning of the *mintYearlyToken()* function, before loading the variable *mintingData*.

Status: Fixed (commit: 45006fcbff347818a5d6489e0355267f9966ee41)

C02. Token Supply Manipulation

The function *mintExtraToken()* performs a check on the state variable *lastMinted* to enforce the requirements of one minting batch per year.

The variable *lastMinted* gets never updated, so the *mintExtraToken()* function can be called at any time with no restrictions.

Path: MLD/contracts/MetaLordToken.sol : mintExtraToken()

Recommendation: Update the value of *lastMinted* after the requirement statements in the *mintExtraToken()* function.

Status: Fixed (commit: 45006fcbff347818a5d6489e0355267f9966ee41)

C03. Denial Of Service

_updateMintingYear() computes *mintingYear* in base 1, while the array *mintingYearlyData* is base 0. Computations will fail in multiple functions because of the wrong indexing.

Path: MLD/contracts/MetaLordToken.sol :

Recommendation: Indexes in the source code should be fixed not only in *mintYearlyToken()* function, and in the other code lines referring to *mintingYear*.

Status: Fixed (commit: 45006fcbff347818a5d6489e0355267f9966ee41)

C04. Denial Of Service

mintExtraToken() is not calling *_updateMintingYear()*, so the variable *mintingYear* will get stuck at the year of the 4th mint, and any call of *mintExtraToken()* will fail because of the wrong minting year.

Path: MLD/contracts/MetaLordToken.sol : mintExtraToken()

Recommendation: Call *_updateMintingYear()* at the beginning of the function *mintExtraToken()*.

Status: Fixed (commit: e709fe9bc3d61ea866da10eccde316e2c4204ab0)

C05. Denial Of Service

The *stake()* function uses *safeTransferFrom()* on an *ERC721* transfer; the recipient address of the transfer is the *NFT-Staking* contract that is not inheriting *ERC721Holder*; this will cause the *stake()* function to fail every time, causing a denial of service.

Path: NFT-Staking/contracts/NFTStaking.sol : stake();

Recommendation: Inherit from the *ERC721Holder* contract to pass the check inside the *safeTransferFrom()* function.

Status: Fixed (commit: 23d65f66615ffea46ca6b771dcb023f0682dbcb4)

■■■ High

H01. Highly Permissive Role Access

Users are exposed to malicious updates of the withdrawal fees.

Contract’s owner can arbitrarily set the withdrawal fees as high as 100%, with no time lock implemented. In this situation, user funds would be stolen.

If the fee is set to more than 100%, the transaction will fail, and the funds will be locked.

Path: NFT-Staking/contracts/NFTStaking.sol : updateWithdrawalFeesPercentage()

Recommendation: Implement a time lock on withdrawal fees update. It is a best practice to bound *withdrawalFeesPercentage* to a reasonable range, so that the impact of malicious activities would be limited by design.

Status: Fixed (commit: 23d65f66615ffea46ca6b771dcb023f0682dbcb4)

Medium

M01. Unbounded Variable

updateLordBurnPercentage() does not perform a check on the input value. If the new value is above 100, users will need more tokens than *requiredLord*.

Path: NFT-Staking/contracts/NFTStaking.sol : updateLordBurnPercentage()

Recommendation: Implement a bound check.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

M02. Inefficient Gas Model

stake() function does not check that the user owns *requiredLord* tokens; instead, it relies on the failing *burnFrom()* and *transferFrom()*.

In case of a user not having enough tokens, he will incur higher Gas spending.

Path: NFT-Staking/contracts/NFTStaking.sol : stake()

Recommendation: Check the amount of owned lord tokens with a require statement.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

M03. Missing SafeERC20

SafeERC20 library is correctly implemented and used for *mldToken* transfers, but it is not used for *lordToken* transfers.

Path: NFT-Staking/contracts/NFTStaking.sol : stake(), withdraw()

Recommendation: Use *SafeERC20* also for lord token transfers.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

M04. Contradiction

The whitepaper first states that the lord token total supply is 2.5B, and shortly after mentions that on top of that 50M will be minted yearly from the 5th year on.

Total supply is thus not bounded to 2.5B.

Path: Whitepaper

Recommendation: Fix the documentation.

Status: Reported

M05. Inefficient Gas Model

Starting with *Solidity ^0.8.0*, *SafeMath* functions are built-in. In such a way, the library is redundant.

Path: NFT-Staking/contracts/NFTStaking.sol

Recommendation: Remove redundant functionality.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

M06. Improper Oracle Usage

The *NFT-Staking* contract does not have a way to stop some of its functionalities if the oracle stops working as expected.

Path: NFT-Staking/contracts/NFTStaking.sol : stake(), claimRewards();

Recommendation: Inherit the *Pausable* contract from openZeppelin and make the *stake()* and *claimRewards()* functions pausable.

Status: Fixed (commit: 23d65f66615ffea46ca6b771dcb023f0682dbcb4)

Low

L01. Unused Variable

The variables *mlUSDPrice* and *platformHash* are never used.

Path: NFT-Staking/contracts/NFTStaking.sol

Recommendation: Remove unused variables.

Status: Reported

L02. Floating Pragma

The project uses floating pragmas *^0.8.5* and *^0.8.4*.

Paths: NFT-Staking/contracts/NFTStaking.sol;

MLD/contracts/MetaLordToken.sol;

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

Status: Fixed (commits: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f, 45006fcbff347818a5d6489e0355267f9966ee41)

L03. Best Practice Violation

The function does not use *safeTransferFrom()* function to check that the receiver implements the *IERC721Receiver* interface or inherits the *Holder* the contract.

This will not affect EOA transfers, only transfers to contract addresses.

Path: NFT-Staking/contracts/NFTStaking.sol : withdraw();

Recommendation: Use the *safeTransferFrom()* function to interact with *ERC721* tokens safely.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

L04. Best Practice Violation

The contract is not inheriting the *ERC721Holder*.

In this way, if a *safeTransferFrom()* is sent to this contract, the transaction will fail.

Path: NFT-Staking/contracts/NFTStaking.sol : stake();

Recommendation: Inherit from the *ERC721Holder* contract to pass the check inside the *safeTransferFrom()* function.

Status: Fixed (commit: 23d65f66615ffea46ca6b771dcb023f0682dbcb4)

L05. Redundant Code

The variable *_user* is equal to the *msg.sender*.

Path: NFT-Staking/contracts/NFTStaking.sol : stake(), withdraw(), claimRewards();

Recommendation: Delete redundant code block.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

L06. Style Guide Violation

The provided projects should follow the official guidelines.

Paths: NFT-Staking/contracts/NFTStaking.sol;

MLD/contracts/MetaLordToken.sol;

Recommendation: Follow the official Solidity guidelines.

Status: Fixed (commits: e709fe9bc3d61ea866da10eccde316e2c4204ab0, 23d65f66615ffea46ca6b771dcb023f0682dbcb4)

L07. Missing Events

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

Path: NFT-Staking/contracts/NFTStaking.sol : PlatformDataUpdated(), UserDataUpdated(), updateChainlinkOracle(), updateJobID(), updateNFTValidation(), updateWithdrawalFeesPercentage(), updateRequiredLord(), updateLordBurnPercentage(), updateBaseAPI(), updateMasterWallet();

Recommendation: Create and emit related events.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

L08. State Variable Can Be Declared Immutable

Variable`s *fee* value is set in the constructor. This variable can be declared immutable

This will lower the Gas taxes.

Path: NFT-Staking/contracts/NFTStaking.sol;

Recommendation: Declare mentioned variables as immutable.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

L09. Missing Zero Address Validation

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

This can lead to unwanted external calls to 0x0.

Path: NFT-Staking/contracts/NFTStaking.sol : constructor().masterWallet;

Recommendation: Implement zero address checks.

Status: Fixed (commit: 405cc3d0d20ea2374ca4196caedfe0e1fba4620f)

L10. Functions That Can Be Declared External

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

Path: MLD/contracts/MetaLordToken.sol : updateCommunityTreasury(), updatePartnersTreasury();

Recommendation: Use the external attribute for functions never called from the contract.

Status: Fixed (commit: 45006fcbff347818a5d6489e0355267f9966ee41)

L11. Long Literal Number

Long literal numbers should include separators to aid readability.

Path: MLD/contracts/MetaLordToken.sol : constructor();

Recommendation: Add a separator character, e.g. '_', to aid readability.

Status: Fixed (commit: 45006fcbff347818a5d6489e0355267f9966ee41)

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.