



Munchables

Final Smart Contract Audit Report

DATE: 20 March 2024 | PREPARED BY: Entersoft Pte Ltd

Contents

Revision History & Version Control

1.0 Disclaimer	3
2.0 Overview	5
2.1 Project Overview	5
2.2 Scope	5
2.3 Project Summary	5
2.4 Audit Summary	5
2.5 Security Level References	6
2.6 Vulnerability Summary	6
3.0 Executive Summary	7
4.0 Technical Analysis	11
4.1 Lack of Maximum Limit Check in onlyAllowedActiveToken Modifier	11
4.2 Potential DoS Attack due to unrestricted Array Size in Lock.sol	11
4.3 Lack of Maximum Limit Check in distribute Function of Claim.sol Contract	12
4.4 Solidity Pragma Specificity Issue	12
4.5 Unsafe Use of block.timestamp in time in Lock.sol	12
4.6 Lack of Validation Checks in calculate Function	13
4.7 Uninitialized State Variables in AccountManager.sol Contract	13
4.8 Centralization Risk for trusted owners	14
4.9 Using `ERC721::_mint()` can be dangerous	14
5.0 Auditing Approach and Methodologies Applied	14
5.1 Code Review / Manual Analysis	15
5.2 Tools Used for Audit	15
6.0 Limitations on Disclosure and Use of this Report	15

Revision History & Version Control

Version	Date	Author(s)	Description
1.0	20 March 2024	R. Dixit G. Singh D. Bhavar M. Gowda	Interim report
2.0	21 March 2024	R. Dixit G. Singh D. Bhavar M. Gowda	Re-audit and final report after vulnerability fixes by Munchable Team based on Initial Audit report

Entersoft was commissioned by Munchables to perform Smart contract code review on their product. The review was conducted from 12th March 2024 to 20st March 2024, to ensure overall code quality, security, and correctness and that the code will work as intended. The review was also conducted to ensure that vulnerabilities previously identified were fixed, and no vulnerabilities existed post-implementation of fixes by the Munchables Team as advised in Report 1.0

The report is structured into two main sections:

- Executive Summary which provides a high-level overview of the audit findings.
- Technical Analysis which offers a detailed analysis of the smart contract code.

Please note that the analysis is static and entirely limited to the smart contract code. The information provided in this report should be used to understand the security and quality of the code, as well as its expected behavior.

Scope included:

- AccountManager.sol
- Claim.sol
- Lock.sol
- MunchableNFT.sol
- MunchToken.sol

Standards followed include:

- OWASP (partially, for instance, role validations, input validations, etc.)
- Solidity best security practices

1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to (i) smart contract best coding practices and issues in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, you must read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose, and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

2.0 Overview

2.1 Project Overview

Entersoft has meticulously audited the smart contract project from 12th March 2024 to 20th March 2024, with a primary focus on Solidity code files integral to blockchain functionality, emphasizing vulnerabilities in associated gas claiming. The working of basic functionalities was also tested during the review.

2.2 Scope

The audit scope covers the Munchables smart contracts available in the GitHub private repository. The audit focused on the checklist provided for the smart contract code audit.

1. AccountManager.sol
2. Claim.sol
3. Lock.sol
4. MunchableNFT.sol
5. MunchToken.sol

Commits under our scope: 7d6e3b0169b6fc8d178f0bb0cd7bff79e884da00

2.3 Project Summary

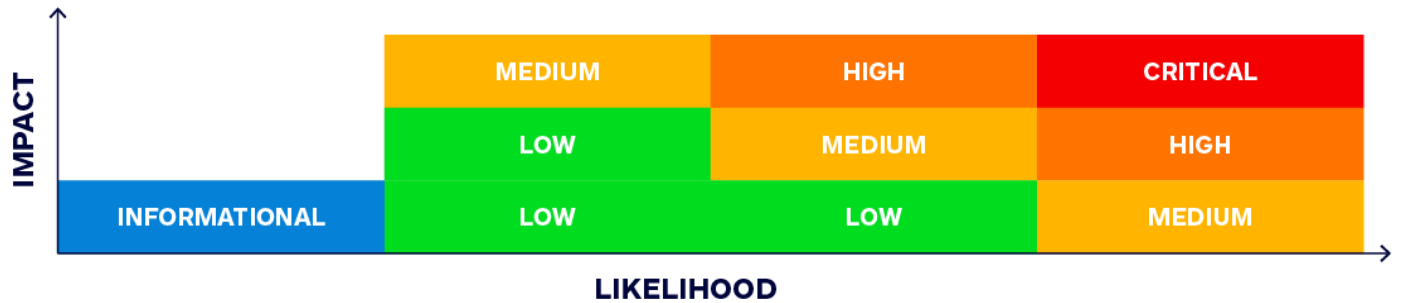
Name	Verified	Audited	Vulnerabilities
Munchables	Yes	Yes	Please review Section 4

2.4 Audit Summary

Delivery Date	Method of Audit	Consultants Engaged
20th March 2024	Static Analysis: The static part of a smart contract audit refers to the process of reviewing the code of a smart contract to identify security vulnerabilities, coding errors, and adherence to best practices without executing the code and without interacting with it in a live environment. The smart contract is reviewed as is. In this we will cover logic vulnerabilities, dependency security flaws and more.	4

2.5 Security Level References

Every vulnerability in this report was assigned a severity level from the following classification table:



2.6 Vulnerability Summary

● Critical	● High	● Medium	● Low	● Informational
0	0	0	0	0

3.0 Executive Summary

Entersoft has conducted a static and dynamic smart contract audit of the Munchables project through a comprehensive smart contract audit. The primary objective was to identify potential vulnerabilities and risks within the codebase, ensuring adherence to industry-leading standards while prioritizing security, reliability, and performance. Our focus was on prompt and efficient identification and resolution of vulnerabilities to enhance the overall robustness of the solidity smart contract.

Testing Methodology:

Our testing methodology in Solidity adhered to industry standards and best practices, integrating partially implemented OWASP and NIST SP 800 standards for encryption and signatures.

We have performed a detailed manual analysis, adherence to industry standards, and the use of a comprehensive toolset. Our approach ensured a thorough evaluation within the designated Solidity code files.

Findings and Security Posture:













Our primary focus was on Access Control Policies, Transaction Signature Validations, Reentrancy, Time Manipulation, Default Visibility, Outdated Compiler Version, Input Validation, Deprecated Solidity Functions, Shadowing State Variables, Presence of Unused Variables, Overflow and Underflow Conditions, Assets Integrity, Errors and Exception.

Importantly, our audit process intentionally avoided reliance solely on automated tools, emphasizing a more in-depth and nuanced approach to security analysis. Conducted from March 12, 2024, to March 20, 2024, our team diligently assessed and validated the security posture of the solidity smart contract, ultimately classifying it as "Secure," reflecting the absence of identified vulnerabilities and the robustness of the codebase against potential threats.

Result

The following table provides an overall summary of the findings and security posture of the smart contract code in scope.

-  **No Security vulnerabilities were identified**
-  **Security vulnerabilities were identified**

#	Munchables Audit Attack Vectors	Result
1	Access Control Policies	
2	Transaction Signature Validations	
3	Default Visibility	
4	Address Validation	
5	Reentrancy attacks	
6	Presence of unused variables	
7	Overflow and Underflow Conditions	
8	Assets Integrity	
10	Program Validations	
11	DOS	
12	Time Manipulation Vulnerability	
13	ERC721 token issues	
Overall Security Posture		Secure

Comprehensive Analysis findings: A Dual Approach through Static and Manual Examination

Phase 1: Static Analysis

In our rigorous smart contract audit process, we employed a multifaceted approach combining both static and dynamic analysis methodologies to comprehensively evaluate the security posture of the protocol. Our static analysis phase involved the utilization of static analyzing tools such as Slither, Aderyn. These tools enabled us to conduct automated code analysis to identify potential vulnerabilities within the smart contracts.

Tools and Efforts:

Slither, Aderyn were instrumental in performing static analysis, allowing us to efficiently scan the codebase for common vulnerabilities such as reentrancy, denial-of-service (DOS) attacks, front-running vulnerabilities, time dependencies, token approval issues, and arithmetic errors. Through meticulous examination of the code, we meticulously identified and categorized potential risks, laying the groundwork for further in-depth analysis.

Phase 2: Dynamic Analysis

After the static analysis phase, we transitioned to dynamic analysis, which involved a more hands-on approach to scrutinizing the intended functionality and security of the smart contracts. Further, we devised a comprehensive suite of unit tests to validate the expected behavior of the smart contracts under various scenarios.

Processes and Test Cases:

Our dynamic analysis encompassed a systematic exploration of the smart contracts' functionalities, focusing on critical areas prone to vulnerabilities. We meticulously crafted test cases to assess the resilience of the contracts against potential attack vectors, including but not limited to reentrancy attacks, DOS vulnerabilities, front-running exploits, time-sensitive vulnerabilities, token approval vulnerabilities, and arithmetic errors.

Throughout both the static and dynamic analysis phases, our team dedicated substantial efforts to meticulously review the codebase, identify vulnerabilities, and develop robust test cases to assess the security posture of the protocol comprehensively. By combining automated analysis with manual examination and testing, we ensured a thorough evaluation of the smart contracts, ultimately enhancing the security and reliability of the protocol.

4.0 Technical Analysis

Note:

The following values for “Severity” mean:

- **Critical:** This vulnerability poses a direct and severe threat to the funds or the main functionality of the protocol.
- **High:** Direct impact on the funds or the main functionality of the protocol.
- **Medium:** Indirect impact on the funds or the protocol's functionality.
- **Low:** Minimal to no impact on the funds or the protocol's main functionality.

The following values for “Result” mean:

- **PASS:** indicates that there is no security risk.
- **FAIL:** indicates that there is a security risk that needs to be remediated.
- **Informational:** Suggestions related to good coding practices and gas-efficient code.
- **Not Applicable:** means the attack vector is Not applicable or Not available

4.1 Lack of Maximum Limit Check in onlyAllowedActiveToken Modifier

Status	PASS
Severity	Medium
Description	The onlyAllowedActiveToken modifier in the smart contract does not implement a maximum limit check for the allowed_tokens array length. If a very large array is passed to this modifier, it may lead to a possible occurrence of Denial-of-Service (DoS) attacks due to excessive gas consumption.
Location / Source File	Lock.sol
Observation	The modifier iterates through the entire allowed_tokens array to check if the given token is allowed and active. However, it does not include a check to limit the size of the array, which could result in excessive gas consumption and potential DoS vulnerabilities if a very large array is passed.
Remediation	It is recommended to add a maximum limit check for the allowed_tokens array length in the onlyAllowedActiveToken modifier to prevent potential DoS attacks. This can be achieved by adding a condition to check if the array length exceeds a predefined maximum limit before proceeding with the iteration.
Remark	Issue fixed in commit: 7d6e3b0169b6fc8d178f0bb0cd7b7ff79e884da00

4.2 Potential DoS Attack due to unrestricted Array Size in Lock.sol

Status	PASS
Severity	Medium
Description	The getAllowedTokenByAddress function within the smart contract does not limit the length of the allowed_tokens array during iteration. This lack of length limitation can lead to a potential Denial-of-Service (DoS) vulnerability, as excessively large arrays can consume excessive gas during iteration, leading to block gas limit exceedance and contract unresponsiveness.
Location / Source File	Lock.sol
Observation	The getAllowedTokenByAddress function iterates through the entire allowed_tokens array to find a token by its address. However, it does not include a limit on the array length, which could result in excessive gas consumption and potential DoS vulnerabilities if the array becomes very large.

Remediation	It is recommended to add a maximum limit check for the allowed_tokens array length in the getAllowedTokenByAddress function to prevent potential DoS attacks. This can be achieved by adding a condition to check if the array length exceeds a predefined maximum limit before proceeding with the iteration.
Remark	Issue fixed in commit: 7d6e3b0169b6fc8d178f0bb0cd7bff79e884da00

4.3 Lack of Maximum Limit Check in distribute Function of Claim.sol Contract

Status	PASS
Severity	Medium
Description	The distribute function in the provided Solidity contract is designed to perform a series of operations to manage the distribution of rewards. It does require careful scrutiny to ensure that it does not inadvertently introduce vulnerabilities, such as Denial of Service (DoS) attacks.
Location / Source File	Claim.sol
Observation	Potential area of concern in the context of a DoS attack could arise from the loop iterating through bonuses. If the array of bonuses grows very large, iterating through it could consume a significant amount of gas, potentially exceeding block gas limits in extreme cases. This could make it difficult or even impossible to successfully call the distribute function, effectively halting the distribution process.
Remediation	It is recommended to add a maximum limit check in the <code>distribuite</code> function, where the length in the for loop is checked to prevent potential DoS attacks. This can be achieved by adding a condition to check if the array length exceeds a predefined maximum limit before proceeding with the iteration.
Remark	Issue fixed in commit: 7d6e3b0169b6fc8d178f0bb0cd7bff79e884da00

4.4 Solidity Pragma Specificity Issue

Status	PASS
Severity	Low
Description	The pragma directive in the contracts specifies a wide range of Solidity versions using the caret (^) symbol. This can potentially introduce compatibility issues with future compiler versions, as it allows any version of Solidity greater than or equal to 0.8.20, including major updates that may introduce breaking changes.
Location / Source File	AccountManager.sol, Claim.sol, Lock.sol, MunchableNFT.sol, MunchToken.sol
Observation	The use of ^ in the pragma directive allows for flexibility but can lead to unexpected behavior if there are breaking changes in future compiler versions.
Remediation	Replace the pragma directive with a specific Solidity version, such as pragma solidity 0.8.20;

4.5 Unsafe Use of block.timestamp in time in Lock.sol

Status	FAIL
Severity	Low
Description	The lock function within the smart contract calculates the lock duration and unlock time using block.timestamp, which can be manipulated by miners to a certain extent (up to approximately 15 seconds). This reliance on block.timestamp for time-sensitive operations may introduce vulnerabilities related to time manipulation. .
Location / Source File	Lock.sol
Observation	The lock function calculates the unlock time by adding the lock duration to the current block timestamp (block.timestamp + _lock_duration). However, miners have some control over the block timestamp, which can be manipulated to a limited extent, potentially allowing for time manipulation attacks.
Remediation	It is recommended to avoid relying solely on block.timestamp for time-sensitive operations, especially in situations where precise timing is crucial. Instead, consider using external sources of time, such as reputable oracles, to obtain more accurate and tamper-proof timestamps.

4.6 Unsafe Use of block.timestamp in time in Claim.sol

Status	FAIL
Severity	Low
Description	Utilizing block.timestamp for swap deadline imposes no safeguard in Proof of Stake (PoS) models, where proposers have prior knowledge of block proposal. Malicious validators could delay transactions to execute them at more favorable block numbers. Considering allowing function callers to specify swap deadline input parameters would enhance security..
Location / Source File	Claim.sol
Observation	The contracts rely on block.timestamp for swap deadline, which doesn't provide effective protection in PoS models. Malicious validators could exploit this by delaying transactions for more favorable block execution. Offering a swap deadline input parameter for function callers would improve security against such attacks.
Remediation	Revise the swap deadline mechanism to allow function callers to specify a deadline input parameter, ensuring better protection against manipulation by malicious validators in PoS environments.

4.7 Centralization Risk for trusted owners

Status	FAIL
Severity	Low
Description	The contracts contain centralized ownership structures where certain roles have privileged rights, posing a risk of centralization. Trusted owners could abuse their privileges to perform malicious updates or drain funds.
Location / Source File	AccountManager.sol,Lock.sol,Claim.sol,MunchToken.sol,MunchableNFT.sol

Observation	The contracts heavily rely on role-based access control (RBAC) mechanisms, assigning various roles (DEFAULT_ADMIN_ROLE, PAUSER_ROLE, MINTER_ROLE, DISTRIBUTION_ROLE, ORACLE_ROLE, SOCIAL_ROLE) with extensive privileges. These roles are granted permissions to crucial functions such as pausing, minting, updating, and withdrawing funds. However, the centralized ownership structure poses a significant risk of centralization, potentially allowing the privileged owners to abuse their authority for malicious purposes.
Remediation	Implement a more decentralized governance model, distributing privileges among multiple trusted entities or introducing community governance mechanisms like voting contracts. Review and limit the powers granted to each role to mitigate centralization risks and enhance the protocol's resilience against malicious actors.

4.8 Using `ERC721::_mint()` can be dangerous

Result	Informational
Description	Utilizing <code>ERC721::_mint()</code> poses a risk as it can mint ERC721 tokens to addresses that may not support ERC721 tokens, potentially leading to unexpected behavior or loss of tokens. To mitigate this risk, it's recommended to use <code>_safeMint()</code> instead of <code>_mint()</code> for ERC721 token minting operations.
Location / Source File	MunchToken.sol
Observation	The contracts use <code>ERC721::_mint()</code> for token minting operations, which can be risky as it may mint tokens to addresses that do not support ERC721 tokens. This practice could result in unexpected behavior or loss of tokens.
Remediation	Replace occurrences of <code>ERC721::_mint()</code> with <code>_safeMint()</code> to ensure safer token minting operations, reducing the risk of unintended consequences or token loss due to minting to incompatible addresses.

4.9 PUSH0 is not supported by all chains

Severity	Informational
Description	Solidity compiler version 0.8.20 switches the default target EVM version to Shanghai, which introduces PUSH0 opcodes in the generated bytecode. This change may cause deployment failures on chains other than mainnet, particularly on Layer 2 (L2) chains that may not support PUSH0 opcodes. Therefore, it's essential to select the appropriate EVM version when deploying contracts to ensure compatibility with the target chain.
Location / Source File	AccountManager.sol, Claim.sol, Lock.sol, MunchToken.sol, MunchableNFT.sol
Observation	The pragma statement in the contracts specifies Solidity compiler version 0.8.20, which may generate bytecode with PUSH0 opcodes due to the default target EVM version being Shanghai. Failure to select the appropriate EVM version may result in deployment failures, especially on non-mainnet chains like L2 chains.
Remediation	When deploying contracts, ensure to select the appropriate EVM version compatible with the target chain to prevent deployment failures caused by the introduction of PUSH0 opcodes in the bytecode. Consider specifying the EVM version explicitly in the pragma statement or using compiler flags to specify the desired EVM version during deployment.

5.0 Auditing Approach and Methodologies Applied

The solidity smart contract was audited in a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of code:

- **Code quality and structure:** We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analyzing the overall architecture of the solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities:** Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, Signatures, and deprecated functions.
- **Documentation and comments:** Our team reviewed the code documentation and comments to ensure they accurately describe the code's intended behavior and logic. This helps developers to better understand the codebase and make modifications without introducing new issues.
- **Compliance with best practices:** We checked that the code follows best practices and coding standards that are recommended by the solidity community and industry experts. This ensures that the solidity smart contract is secure, reliable, and efficient.

Our audit team followed OWASP and Ethereum(Solidity) community security guidelines for this audit. As a result, we were able to identify potential issues and provide recommendations to improve the smart contract security and performance.

Throughout the audit of the smart contract, our team placed great emphasis on ensuring the overall quality of the code and the use of industry best practices. We meticulously reviewed the codebase to ensure that it was thoroughly documented and that all comments and logic aligned with the intended behavior. Our approach to the audit was comprehensive, methodical, and aimed at ensuring that the smart contract was secure, reliable, and optimized for performance.

5.1 Code Review / Manual Analysis

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analyzed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

5.2 Tools Used for Audit

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Adreyn, Slither. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. Entersoft takes pride in utilizing these tools, which significantly contribute to the quality, security, and maintainability of our codebase.

6.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of the Munchables Project and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Munchables solidity smart contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of Munchables and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and Munchables governs the disclosure of this report to all other parties including product vendors and suppliers.