SALUS SECURITY

DEC 2024

CODE SECURITY ASSESSMENT

MAHOJIN

Overview

Project Summary

- Name: Mahojin - MahojinMintHelper
- Address: 0x083Ae3C584b24e12674f973122e89B4113dC97Dc
- Platform: EVM-compatible chains
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Mahojin - MahojinMintHelper
Version	v2
Type	Solidity
Dates	Dec 24 2024
Logs	Dec 11 2024; Dec 24 2024

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	1
Total Low-Severity issues	1
Total informational issues	2
Total	4

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	The minting process for regular users may be subject to DoS attacks	Medium	Business Logic	Resolved
2	Use safeTransfer() instead of transfer()	Low	Risky External Calls	Resolved
3	Missing two-step transfer ownership pattern	Informational	Business Logic	Resolved
4	Use of floating pragma	Informational	Configuration	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. The minting process for regular users may be subject to DoS attacks

Severity: Medium	Category: Business Logic
Target: - contracts/MahojinMintHelper.sol	

Description

There is no check on the parameter `nft_contract`, which could be a malicious NFT and occupy the `trackingId`. This creates the possibility of front-running users' `trackingId`, resulting in regular users' `mint` operations being subject to DoS attacks.

contracts/MahojinMintHelper.sol:L25 - L41

```
...
  require(trackingId != 0, "Invalid trackingId");
  require(nft_contract != address(0), "Invalid nft_contract");
  require(trackingMap[trackingId].nft_contract == address(0), "Already minted");

  tokenId = IMint(nft_contract).mint(to);
  trackingMap[trackingId] = MintedNFT(nft_contract, tokenId);
}
```

Recommendation

Consider modifying the function logic or adding parameter restrictions to prevent the `trackingId` from being maliciously occupied.

Status

The team has resolved this issue.

2. Use safeTransfer() instead of transfer()

Severity: Low	Category: Risky External Calls
Target: - contracts/MahojinMintHelper.sol	

Description

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

```
Callers MUST handle false from returns (bool success). Callers MUST NOT assume that
false is never returned!
```

Recommendation

Consider using the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

Status

The team has resolved this issue.

2.3 Informational Findings

3. Missing two-step transfer ownership pattern

Severity: Informational	Category: Business logic
Target: - contracts/MahojinMintHelper.sol	

Description

The `MahojinMintHelper` contract inherits from the `Ownable` contract. This contract does not implement a two-step process for transferring ownership. Thus, ownership of the contract can easily be lost when making a mistake in transferring ownership.

Recommendation

Consider using the Ownable2StepUpgradeable contract from OpenZeppelin instead.

Status

The team has resolved this issue.

4. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity >=0.6.11;
```

All contracts use a floating compiler version `>=0.6.11`.

Using a floating pragma `>=0.6.11` statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has resolved this issue.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files from address 0x083Ae3C584b24e12674f973122e89B4113dC97Dc:

File	SHA-1 hash
contracts/MahojinMintHelper.sol	1ac7a40555069fd09dbd44345fe936095ad8821f
contracts/Common/Context.sol	b73583934956debd6fb66d5034a118c08d8b404e
contracts/Common/Ownable.sol	7ad955540f2267d9b881ee1b9c803fb043455765

And we have audited the updated contract at the address 0x720Ea8731bb8b63199A3189390bB6a3C1b9F317:

File	SHA-1 hash
contracts/MahojinMintHelper.sol	70883b1d47cb38ba5f3145e9565c602b20387c2e