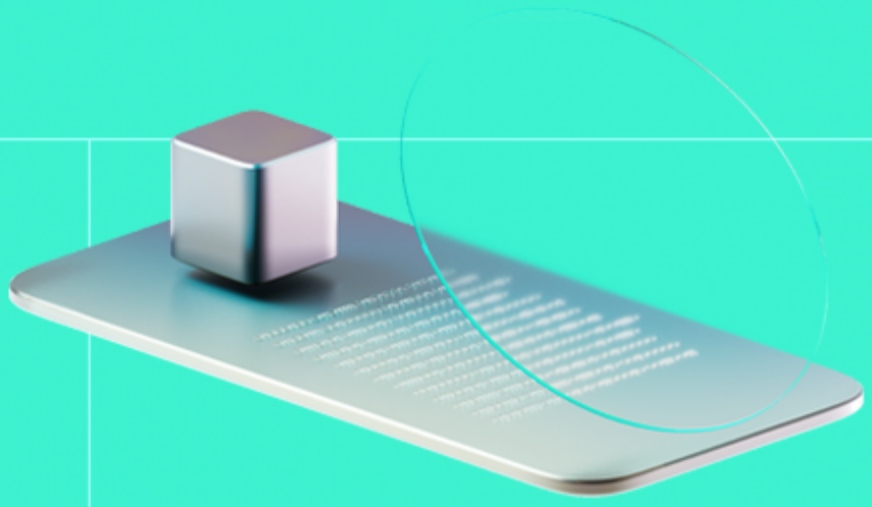
HACKEN

Smart Contract Code Review And Security Analysis Report

Customer: SUI Agents

Date: 17/12/2024



We express our gratitude to the SUI Agents team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

SUIAI is an ERC20 token that facilitates seamless initialization and minting of tokens to the owner address.

Document

Name	Smart Contract Code Review and Security Analysis Report for SUI Agents
Audited By	Jakub Heba
Approved By	Przemyslaw Swiatowiec
Website	https://suiagents.ai/
Changelog	05/12/2024 - Preliminary Report, 17/12/2024 - Secondary Report
Platform	EVM, Sui
Language	Solidity, MOVE
Tags	ERC20
Methodology	https://hackenio.cc/sc_methodology

Review

Scope

Repository	https://github.com/Devermint/SUIAIContracts/tree/main
Commit	242bc8758b681c95f17783ea50151daa552dcfdc
SUIAI.sol - ERC20 token	https://etherscan.io/address/0xd19b72e027cd66bde41d8f60a13740a26c4be8f3#code

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

5	0	5	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	0
Low	5

Vulnerability	Severity
F-2024-7508 - Owner can renounce the ownership of the contract	Low
F-2024-7510 - Single-step ownership transfer	Low
F-2024-7512 - Unauthorized token burn via burnFrom function	Low
F-2024-7513 - Unprotected initialize function	Low
F-2024-7515 - Unrestricted mint function with no supply cap	Low

Documentation quality

- Functional requirements are missed.
- Technical description is not provided.

Code quality

- The code is utilizing OpenZeppelin contracts for ERC20 token creation, ownership management and burning mechanism.
- Several template code patterns were found.

Test coverage

Code coverage of the SUIAI coin on Sui side is **100%** (branch coverage).

- All two functions, that is `mint()` and `burn()` are covered.

Table of Contents

System Overview

SUIAI — simple ERC-20 token that allows minting any amount of tokens to the owner, as well as burning them by anyone.

It has the following attributes:

- Name: SUI Agents
- Symbol: SUIAI
- Decimals: 18 (default)
- Unlimited total supply

On the Sui side, the SUIAI token has identical configuration, but uses 9 decimals only.

Privileged roles

- The owner of SUIAI can mint tokens to `to` address, as well as transfer ownership via single-step ownership transfer.

Potential Risks

- Due to the possibility of unlimited minting of tokens to the address indicated by the owner, there is a risk of centralization and inflation of the token value in the case of too many coins being created.

Findings

Vulnerability Details

F-2024-7508 - Owner can renounce the ownership of the contract - Low

Description:

The **SUIAI** contract uses OpenZeppelin's Ownable library in order to execute access controls. According to this library, the owner of the contract has a full authority on renouncing the ownership by calling the `renounceOwnership()` function. This function basically transfer the owner address to `address(0)`.

```
function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }
```

```
contract SUIAI is ERC20, ERC20Burnable, Ownable {
```

As a result of the **Owner** calling the `renounceOwnership()` function, all ownership-related controls on the contract may be lost and the authority on the contract may become inaccessible.

Status: Accepted

Classification

Impact: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Consider overriding the `renounceOwnership()` function in order to prevent transferring the ownership **address(0)**, even by mistake.

Resolution: The team accepted the issue with the following comment:

> We used default OpenZeppelin Ownable contract and option to set owner as a address(0) is left for the case if something goes wrong with a bridge and we would be able to disable it

F-2024-7510 - Single-step ownership transfer - Low

Description:

The current implementation of the **SUIAI** contract utilizes OpenZeppelin's **Ownable.sol**, which facilitates a single-step process for ownership transfer. This approach, while straightforward, does not include a verification step for the new owner address before finalizing the transfer. The absence of such a precautionary measure can lead to significant security and operational risks, particularly if an incorrect address is provided during the ownership transfer process. Mistakes or malicious activities could result in the permanent transfer of ownership to an unintended address, potentially leading to loss of control over the contract's administrative functionalities.

Security Risks:

The single-step ownership transfer process increases the risk of accidental or malicious transfers, as there is no opportunity to verify or cancel the transfer once initiated.

Operational Risks:

An incorrect transfer of ownership could result in administrative functions becoming inaccessible, potentially crippling the contract's operations and management.

Status: Accepted

Classification

Impact: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement the **Ownable2Step** extension or a similar mechanism that introduces a two-step process for ownership transfer. This process typically involves nominating a new owner and then requiring a separate confirmation step to finalize the transfer.

Resolution: The team accepted the issue with the following comment:

> Agree on this threat, there's no plan of transferring ownership to other parties except case mentioned in F-2024-7508 explanation

F-2024-7512 - Unauthorized token burn via burnFrom function - Low

Description:

The **SUIAI** contract inherits from OpenZeppelin's `ERC20Burnable`, which provides the `burnFrom` function. This function allows an account to burn tokens from another account, provided they have sufficient allowance.

The inherited `burnFrom` function allows an authorized user (with approved allowance) to burn tokens from another user's balance. While this behavior is in line with the OpenZeppelin standard, it poses a risk when implemented in systems where burning tokens should be restricted or more controlled. In the current implementation, any account with an allowance can reduce another account's token balance by burning tokens on their behalf. If a user grants an excessive or unintended allowance, the authorized account can burn tokens on their behalf, potentially leading to the unintended or unauthorized destruction of tokens.

The `ERC20Burnable` contract includes a `burnFrom` function, which enables any account that has received an allowance to burn the allowed amount of tokens from the approving user's balance. The function signature is as follows:

```
function burnFrom(address account, uint256 amount) public virtual {
_spendAllowance(account, _msgSender(), amount);
_burn(account, amount);
}
```

This can result in unwanted token burns, especially if there is no clear consent mechanism in place for burning as opposed to spending.

This issue can lead to the unintended or malicious burning of tokens from users' accounts. Any user who grants an allowance to another account could have their tokens burned without their explicit consent, leading to a permanent loss of their tokens.

Status: Accepted

Classification

Impact: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

- Modify the `burnFrom` function to ensure that only the token owner can burn their tokens.

- Modify the `burnFrom` function so that the system does not allow its execution.

Resolution: The team accepted the issue with the following comment:

> Agree on this threat, however we wanted to implement it this way in order for mint/burn bridge to be able to do it with using less gas.

F-2024-7513 - Unprotected initialize function - Low

Description:

The `initialize()` function in Solidity contracts is commonly used in upgradeable contracts as a substitute for a constructor. This function is meant to be called only once after deployment to set up the initial state of the contract. However, when this function lacks proper access control it becomes vulnerable to front-running attacks or malicious initialization.

```
function initialize(
        address owner
    ) external {
        require(!initialized, "Contract is already initialized");

        _transferOwnership(owner);
        initialized = true;
    }
```

In the current implementation, any account can call the `initialize()` function before the legitimate owner, gaining unauthorized control over the contract.

Status: Accepted

Classification

Impact: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to:

- Add `initializer` modifier from OpenZeppelin's `Initializable` contract.
- Validate, if the deployer was the same sender as `initialize()` caller.

F-2024-7515 - Unrestricted mint function with no supply cap - Low

Description: The contract implements a minting mechanism through the `mint()` function without enforcing any upper limit on the total token supply. This design allows for unlimited token creation, which can lead to token inflation and devaluation.

```
function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
```

Without a maximum cap, malicious actors with minting privileges could flood the market with tokens, severely impacting the token's economy and value. Additionally, minting is only allowed for the contract owner.

Status: Accepted

Classification

Impact: 2/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: We recommend considering creation of maximum supply cap and modifying the `mint()` function to include supply validation.

Resolution: The team accepted the issue with the following comment:

> Regarding mint function, agree on upper bound limit, we made this function to implement bridging to SUI via mint/burn.

Observation Details

F-2024-7509 - Floating pragma - Info

Description:

The project uses a floating Pragma.
This may result in the contracts being deployed using the wrong Pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated Pragma version, which may include bugs that affect the system negatively.

```
pragma solidity ^0.8.27;
```

Status: Accepted

Recommendations

Remediation:

Consider locking the Pragma version whenever possible and avoid using a floating Pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Resolution:

The team accepted the issue with the following comment:

> Already deployed, in config we used same version.

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. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
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 will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

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

Scope Details	
Repository	https://github.com/hknio/Devermint__SUIAIContracts
Commit	242bc8758b681c95f17783ea50151daa552dcfdc
Whitepaper	-
Requirements	-
Technical Requirements	-

Asset	Type
sui/my_coin/sources/my_coin.move [https://github.com/Devermint/SUIAIContracts/tree/main]	Smart Contract
SUIAI.sol [https://etherscan.io/address/0xd19b72e027cd66bde41d8f60a13740a26c4be8f3#code]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.