Smart Contract Audit Report

Prepared by: **Cybring**
Prepared for: **Saros**

Version: 1.0 (final report)
Date: 19th June 2025
Commit/Hash: cc8e3297b58797ca748e70e42e9924bd6ce36061
Auditor: Anh Bui, Duc Cuong Nguyen

Contents

1 Introduction

This document presents the results of a security assessment of the `Saros` Token program. The engagement was commissioned by `Saros` team to obtain an independent evaluation of the on-chain comprehensive features that include balance management, token transfer, mint, burn, and allowance/approval mechanism.

1.1 Objective

This security audit report was initially provided by `CYBRING` on 16th June 2025. This audit was conducted to assess the robustness, reliability, and security of the code base. The goal was to identify vulnerabilities, ensure compliance with best security practices, and provide mitigation measures.

After the initial audit report, a reassessment was conducted on 19 th June 2025 to verify the status of the reported issues. All issues were acknowledged and will be prioritized for future improvement.

1.2 Disclaimer

This security audit is not produced to replace any other type of assessment and does not aim to guarantee the discovery of all security issues within the scope of the assessment. Further, economic modeling, business-logic desirability, and market comparisons (e.g., target price ceilings relative to other launch platforms) were explicitly out of scope.

While this audit was carried out with good faith and technical proficiency, it's crucial to understand that no single audit can guarantee the absolute security of a smart contract. To minimize risks, `CYBRING` recommends a multi-faceted approach involving multiple independent assessments. Please note that this report is not intended to provide financial advice.

2 Scope of Audit

The audit was conducted on commit cc8e3297b58797ca748e70e42e9924bd6ce36061 from git repository `https://github.com/saros-xyz/saros-token-contract`. Further details regarding The `Saros` audit scope are provided below:

- **Smart contracts audited:** SarosVRC25.sol.
- **Codebase details:**
  - Language: Solidity
  - Frameworks: Hardhat
  - Initial audit's commit hash: cc8e3297b58797ca748e70e42e9924bd6ce36061

3 Audit Summary

Initial assessment report is summarized in Table 1. Details of findings can be found in Section 5.

Severity	Count
Low	3
Informational	2

Table 1: Initial assessment summary

All five findings were classified as Low (3) or Informational (2) severity, with no Critical, High, or Medium issues identified. Consequently, the residual risk to the `Saros` Token contract is considered negligible and primarily advisory in nature.

Issue	Severity	Status	Remediation commit
5.1	Low	Acknowledged - by design	
5.2	Low	Acknowledged	
5.3	Low	Acknowledged	
5.4	Informational	Acknowledged	
5.5	Informational	Acknowledged	

Table 2: Final assessment summary

4 Methodology

`CYBRING` conducts the following procedures to enhance security level of our client's smart contracts:

- **Pre-auditing:** Understanding the business logic of the smart contracts, investigating the deployment states of samples, and preparing for the audit.
- **Auditing:** Examining the smart contract by evaluating on multiple perspectives:
  - **Manual code review:** `CYBRING` auditors evaluate the static code analysis report for finding false positive reports. Besides, inspect the smart contract logic, access controls, and data flows ensure the contract behaves as intended and is free from risky logic.
  - **Static code analysis:** By using advanced static analysis techniques combined with our customized detectors, we set out to identify potential vulnerabilities, optimize gas usage, and ensure adherence to best practices in smart contract development.
  - **Fuzz testing:** `CYBRING` leverages fuzzing tools to stress-test the contract, ensuring it performs securely under unpredictable conditions.
- **First deliverable and consulting:** Presenting an initial report on the findings with recommendations for remediation and offering consultation services.
- **Reassessment:** Verifying the status of the issues and identifying any additional complications in the implemented fixes.
- **Final deliverable:** Delivering a comprehensive report detailing the status of each issue.

4.1 Risk Rating

The OWASP Risk Rating Methodology was applied to assess the severity of each issue based on the following criteria, arranged from the perspective of smart contract security.

- **Likelihood:** a measure of how likely this vulnerability is to be discovered and exploited by an attacker.
- **Impact:** a measure of the potential consequences or severity of a vulnerability if it is exploited by an attacker, including the extent of damage, data loss, or disruption of operations.

Impact				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Informational	Low	Medium
		Low	*Medium*	*High*
			Likelihood	

Table 3: Overall Risk Severity

4.2 Audit Categories

- **Common vulnerabilities:** Smart contracts are analyzed following OWASP smart contract top 10 and Smart Contract Weakness Classification (SWC).
- **Advanced vulnerabilities:** `CYBRING` simulates certain types of attack scenarios to exploit the smart contracts. These scenarios were prioritized from high to low severity.
- **Security best practices:** The source code of the smart contract is analyzed from the development perspective, providing suggestions for improving the overall code quality.

4.3 Audit Items

Table 4 shows the details of the issues our auditors will conduct the audit upon.

Category	Item
Reentrancy attack	Single function reentrancy Reentrancy via modifier Cross function reentrancy Cross contract reentrancy
Data Validation and Integrity	Integer overflow and underflow Missing zero address validation Builtin symbols, state variables, and function should not be shadowed Storage variables should be initialized at the time of declaration
Denial of service (DOS)	Denial of service with revert Denial of service with gas limit Denial of service with induction variable overflow Denial of service by exceeding the maximum call stack depth
Access control vulnerabilities	Unprotected access to call sensitive function (self-destruct/suicide) Dangerous usage of tx.origin
Private information and randomness	Store sensitive information on smart contracts Generate random numbers from insecure pseudorandom factors
Timestamp dependence	Timestamp dependence on critical function
Best practices	Missing event for changing critical access control Follow standard style guide (naming conversion, code layout, order of layout) from Solidity

Table 4: Details of examined items

5 Detailed Findings

5.1 Limited Scope of Frozen Mechanism

- **Description:** The contract includes a frozen mechanism aimed at pausing transfer operations. However, it currently does not restrict allowance modifications while frozen. This could allow an address to preset allowances during the frozen state and later execute transfers when the contract is unfrozen.
- **Risk:** Low (Impact: Medium, Likelihood: Low)
- **CWE:** CWE-358: Improperly Implemented Security Check for Standard
- **Mitigation:** Consider extending the frozen mechanism to include allowance-related functions to further harden against edge-case misuse scenarios.
- **Status 19/06/2025:** The `Saros` team has acknowledged this behavior is intentional and within their operational assumptions.

5.2 Potential Reentrancy Vulnerability in withdraw() Function

- **Description:** The `Saros` includes a rescue token mechanism designed to recover tokens accidentally sent to incorrect addresses. However, the implementation of the *withdraw()* function within this mechanism lacks a reentrancy guard. This oversight creates a potential reentrancy attack vector when processing native token. While the likelihood is minimal, a standard reentrancy guard would reinforce defense-in-depth, especially when handling native token transfers.
- **Risk:** Low (Impact: Medium, Likelihood: Low)
- **CWE:** CWE-841: Improper Enforcement of Behavioral Workflow
- **Affected code:**

```
1199    function withdraw(address token_, address destination_, uint256
            amount_) external onlyOwner {
1200        require(destination_ != address(0), "ERC20: Destination is
                zero address");
1201
1202        uint256 availableAmount;
1203        if (token_ == address(0)) {
1204          availableAmount = address(this).balance;
1205        } else {
1206          availableAmount = IERC20(token_).balanceOf(address(this))
                ;
1207        }
1208
1209        require(amount_ <= availableAmount, "ERC20: Not enough
                balance");
1210
1211        if (token_ == address(0)) {
1212          (bool sent, ) = payable(destination_).call{value: amount_
                }("");
1213          require(sent, "ERC20: Fail withdraw token");
1214        } else {
1215          IERC20(token_).safeTransfer(destination_, amount_);
1216        }
1217    }
```

contracts/vrc25/Coin98Token.sol:1212

- **Mitigation:** Implement a robust reentrancy guard in the withdraw() function.
- **Status 19/06/2025:** The `Saros` team has acknowledged this issue.

5.3 Insufficient Fee Validation in setFee()

- **Description:** The *setFee()* function lacks proper input validation for the fee parameters *(priceN, priceD, fee)*. This absence of checks allows the contract owner to set an arbitrarily high fee, which could lead to exorbitant transaction costs for users. For example, if priceN = 100, priceD = 1, and fee = 0, a transfer of 1 token from A to B would result in A paying 100 tokens to the owner as a fee.
- **Risk:** Low (Impact: Medium, Likelihood: Low)

- **CWE:** CWE-754: Improper Check for Unusual or Exceptional Conditions
- **Mitigation:** Implement robust validation checks within the *setFee()* function. These checks should enforce reasonable bounds for the fee parameters.
- **Status 19/06/2025:** The `Saros` team has acknowledged this issue.

5.4 Outdated Solidity Compiler Version

- **Description:** The used solidity pragma in `Saros` is $\geq$ *0.6.0* and $\leq$ *0.8.0*. The compiler $\leq$ 0.8.0 was recognized with some security-relevant bugs (e.g., SOL-2023-2, SOL-2023-1).
- **Risk:** Informational (Impact: Low, Likelihood: Low)
- **CWE:** CWE-937: Using Components with Known Vulnerabilities.
- **Mitigation:** `CYBRING` recommends using a single, strict, updated pragma version on all smart contracts (e.g., pragma solidity 0.8.29).
- **Status 19/06/2025:** The `Saros` team has acknowledged this issue.

5.5 Optimizing - Redundant Fee Calculation

- **Description:** In the Saros Smart Contract, the current fee calculation introduces unnecessary computations and can make the code less readable. Example with *estimateFee(0)* call, in contexts like *permit()* or *approve()*, is effectively calculating a fee for a zero-value transfer, which is then used to derive a *_minFee*. This does not introduce security concerns but presents an opportunity to optimize gas usage and improve code readability.
- **Risk:** Informational (Impact: Low, Likelihood: Low)
- **Mitigation:** To improve gas efficiency and code clarity, separate fee types explicitly. For example, introduce distinct variables like operation fee and transfer fee where *permit()*, *approve()*, etc., would only reference operation fee.
- **Status 19/06/2025:** The `Saros` team has acknowledged this issue.