SECURITY AUDIT OF

SSS TOKEN

Public Report

Apr 8, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Apr 8, 2024. We would like to thank the Super Sushi Samurai for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the SSS Token. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issue in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About SSS Token

SSS Token will launch with fixed-supply token used in Super Sushi Samurai to facilitate upgrades, rewards and transactions.

Following the Super Sushi Samurai, there is a 2% tax on SSS every time someone swaps SSS on Thruster. 1.6% goes to game rewards, and 0.4% goes towards sustaining dev operations.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the SSS Token.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
643a51fcd732d0056a754b34766d4ae8531db327cb6222260dc11dcdc33cc0e1	./contracts/SSS.sol

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy

- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Super Sushi Samurai acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Super Sushi Samurai understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Super Sushi Samurai agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Super Sushi Samurai will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Super Sushi Samurai, the final report will be considered fully accepted by the Super Sushi Samurai without the signature.

2. AUDIT RESULT

2.1. Overview

The SSS Token was written in `Solidity` language, with the required version to be `0.8.20`.

The contract extends `ERC20` and `Ownable`. However, the `ERC20` contract is modified to implement the `IERC20Permit`. Below table describes some properties of the audited SSS Token (as of the report writing time).

PROPERTY	VALUE
Name	SSS
Symbol	SSS
Decimals	18
Total Supply	555,555,555,555,555 x10^{18} (It represents over 555 trillion tokens)
Buy Tax	2% (max 5%)
Sell Tax	2% (max 5%)

Table 2. The SSS Token properties

The audit team examined the ERC20 token against the team's standard protocols.

Title	Status
Total Supply Consistency	Passed
Approval	Passed
Self Transfer	Passed
Transfer from/to Zero	Passed
Transfer Effective	Passed

Table 3. The standard testing for ERC20

2.2. Findings

During the audit process, the audit team identified some vulnerabilities in the SSS Token.

Issue	Severity	Status
Missing minimum amount when adding liquidity	HIGH	ACKNOWLEDGED
Incorrect unlock token for dev mechanism	MEDIUM	ACKNOWLEDGED
Missing zero check	INFORMATIVE	ACKNOWLEDGED

Table 4. The issue list

2.2.1. HIGH - Missing minimum amount when adding liquidity

Description:

The lack of a minimum liquidity requirement in the smart contract puts it at danger of a sandwich attack on the Uniswap liquidity pool. In a sandwich attack, bad actors use the lack of minimum requirements to inset transactions before and after, influencing the token's price and benefiting at the expense of other traders.

```
function _addETHLiquidity(uint256 ethAmount, uint256 tokenAmount) internal {
    _approve(address(this), address(uniswapV2Router), tokenAmount);
    uniswapV2Router.addLiquidityETH{value: ethAmount}( // INCORRECT
        address(this),
        tokenAmount,
        0, // accept any amount of ETH
        0, // accept any amount of token
        address(this),
        block.timestamp
    );
}
```

RECOMMENDATION

Implement a minimum liquidity requirement to prevent sandwich attacks on the Uniswap liquidity pool. Fixed code follows the Uniswap V2 documentation[*]:

```
function _addETHLiquidity(uint256 ethAmount, uint256 tokenAmount) internal {
    _approve(address(this), address(uniswapV2Router), tokenAmount);
    uniswapV2Router.addLiquidityETH{value: ethAmount}(
        address(this),
        tokenAmount,
        tokenAmount.mul(99).div(100), // 99% of the token amount
        ethAmount.mul(99).div(100), // 99% of the ETH amount
        address(this),
        block.timestamp
    );
}
```

References:

- (*): *Uniswap V2 documentation*.

UPDATES

- ***Apr 8, 2024***: The issue is acknowledged by the SSS Token team.

2.2.2. MEDIUM - Incorrect unlock token for dev mechanism

Description:

When an address is excluded from charging taxes. The address can send tokens to pool and skim(*) tokens from the pool without being taxed. The trade volume is manipulated without any swaps. This results in an increase in amount of dev tokens unlocked without any actual trading.

```
function _update(address from, address to, uint256 amount) internal override virtual {
    // don't check if it is minting or burning
    if (from == address(0) || to == address(0) || to == address(0xdead)) {
        super._update(from, to, amount);
        return;
    }

    _preCheck(from, to, amount);

    uint256 taxAmount = _calculateTax(from, to, amount);
    uint256 amountAfterTax = amount - taxAmount;

    if(taxAmount > 0) {
        super._update(from, address(this), taxAmount);
        _recordTax(taxAmount);
    }

    _unlockTokenForDev(from, to, amount);

    super._update(from, to, amountAfterTax);
}
function _unlockTokenForDev(address from, address to, uint256 amount) internal {
    if(!isLiquidityPool(from) && !isLiquidityPool(to)) {
        return;
    }
    if(startPoolTime == 0) {
        return;
    }

    tradeVolume += amount;
    uint256 devRemainToken = devTokenAmountRemain;
    if(devRemainToken == 0) {
        return;
    }

    uint256 unlockAmount = _calculateUnlockTokenForDev(amount);
    devTokenAmountClaimable += unlockAmount;
    devTokenAmountRemain = devRemainToken - unlockAmount;
}
function _calculateTax(address from, address to, uint256 amount) internal view returns
(uint256 taxAmount){
    // only apply tax if buy and sell
    uint256 taxPercent = 0;
    if(isLiquidityPool(from)) {
        taxPercent = buyTaxPercent;
    } else if(isLiquidityPool(to)) {
        taxPercent = sellTaxPercent;
    }
    if (
        taxPercent == 0
        || excludeFromTaxes[from] || excludeFromTaxes[to]
    ) {
        return 0;
    }

    taxAmount = amount * taxPercent / 100_00;
    return taxAmount;
}
```

RECOMMENDATION

Do not unlock tokens for dev when the transaction is not charged with tax.

References:

- (*): `skim()` allows a user to withdraw the difference between the current balances of the pair and reserves to the caller, if that difference is greater than 0.

UPDATES

- *Apr 8, 2024*: The issue is acknowledged by the SSS Token team.

2.2.3. INFORMATIVE - Missing zero check

Position:

- `SSS.sol`

Description:

The smart contract does not check for zero values when updating addresses in `constructor`, `setDevAddress` and `setCommunityAddress` functions.

UPDATES

- *Apr 8, 2024*: The issue is acknowledged by the SSS Token team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 8, 2024*	Public Report	Verichains Lab

Table 5. Report versions history