SECURITY AUDIT OF

STMAN TOKEN DAILYVESTING SMART CONTRACT

Public Report

Apr 20, 2022

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 prepared by Verichains Lab on Apr 20, 2022. We would like to thank the STMAN 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 STMAN Token DailyVesting Smart Contract. 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 no vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About STMAN Token DailyVesting Smart Contract

Anti-inflation Stickman's Battleground is an NFT game of survival with a free-to-play-to-earn mechanism.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the STMAN Token DailyVesting Smart Contract. It was conducted on the source code provided by the STMAN team.

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

SHA256 Sum	File
93cb7431d06054537c825c6544478c259e1ae9b1b73279e1b315989945afb702	**DailyVestingPool.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 was 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

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

2.1.1. Contract code

The STMAN Token DailyVesting Smart Contract was written in Solidity language, with the required version to be ^0.8.0. The source code was written based on OpenZeppelin's library.

2.1.2. DailyVesting Contract

The STMAN team uses this contract to release the tokens. The tokens in contract will be released following the linear logic every day.

2.2. Findings

During the audit process, the audit team found no vulnerability in the given version of the STMAN Token DailyVesting Smart Contract.

2.3. Additional notes and recommendations

2.3.1. The logic of updateTimeVesting and updateDataUser functions may cause conflict in the contract INFORMATIVE

The contract implements updateTimeVesting and updateDataUser functions which allow the owner to force updating the state variable and user info. The logic of the contract may raise some errors, if the owner updates them suddenly.

So we note this issue to notice the STMAN team to avoid some unexpected use case.

UPDATES

- *Apr 20, 2022*: This issue has been acknowledged by the STMAN team.

2.3.2. Unnecessary usage of SafeMath library in Solidity 0.8.0+ INFORMATIVE

All safe math usages in the contract are for overflow checking, solidity 0.8.0+ already do that by default, the only usage of safemath now is to have a custom revert message which isn't the case in the auditing contracts. We suggest using normal operators for readability and gas saving.

RECOMMENDATION

We suggest changing all methods from SafeMath library to normal arithmetic operator.

UPDATES

- *Apr 20, 2022*: This issue has been acknowledged by the STMAN team.

2.3.3. Use calldata instead of memory for gas saving INFORMATIVE

In external function with array arguments, using memory will force solidity to copy that array to memory thus wasting more gas than using directly from calldata. Unless you want to write to the variable, always using calldata for external function.

```
function updateDataUser(address[] memory _wallets, uint256[] memory _list…
  AmountBonus) external
```

RECOMMENDATION

Change memory to calldata for gas saving in all external function.

UPDATES

- *Apr 20, 2022*: This issue has been acknowledged by the STMAN team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 20, 2022*	Public Report	Verichains Lab

Table 2. Report versions history