verichains

SECURITY AUDIT OF

SIPHER STAKING SMART CONTRACT

Public Report

Dec 03, 2021

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 Dec 03, 2021. We would like to thank the Sipher 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 Sipher Staking 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 found no vulnerability in the given version of Sipher Staking Smart Contract.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Sipher Staking Smart Contract

Sipher is an ambitious casual fighting and exploration Game with an End-game goal of creating an open world social experience, Built on the Ethereum Blockchain.

10,000 unique NFTs are beautifully illustrated by our 2D & 3D artists. Inspired by the cryptographic ethos, this collection is aptly named "Sipherian Surge", with all the 10,000 characters belonging to the Sipher's origin race: SIPHER INU.

1.2. Audit scope

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

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

The initial review was conducted on Nov 30, 2021 and a total effort of 3 working days was dedicated to identifying and documenting security issues in the code base of the Sipher Staking Smart Contract.

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

FILE	SHA256 SUM
BasePool.sol	2d466b469fd6079fdca2305b8716320d460c40e2060a6dc081f76bd0b223eab7
TokenSaver.sol	4df1f949bfcddf7305dfba1ef6021842105ebd1c793a5c440217af60f69743b2
AbstractRewards.sol	bbc65efeb36fe3b5674fb7774270323aa9c1ad9a6d1517a206bdcfed648b3eea
TimeLockNonTransferablePool.sol	65d66fa08c13c10901a59a4c3c19d9c1075396715491e8b4f48d659bf5dd77c5
IBasePool.sol	7490e734dccc420440f73fda9faa95500e8a56c64ed38535788cd9a78bde4440
IAbstractRewards.sol	e4d54710f7d465f7b264f10c571373c078dce11e2c677bf334220fcd97f68d0b
ITimeLockPool.sol	6ed743f409d47a1fd57d6cfd82c63a9d4780e18e92718eef0822c19eb47492ae
TimeLockPool.sol	23392188e3aa0f8d8b27cb38a8ec179d2a302dd57a9bc10b6e604c7c198d14fb
LiquidityMiningManager.sol	2d36d67e604237422f749921ac981be3dde9161682df30ee55e31e354ab20d4d

2.2. Contract code

The Sipher Staking Smart Contract was written in Solidity language, with the required version to be 0.8.7. The source code was written based on OpenZeppelin's library.

2.3. Findings

During the audit process, the audit team found no vulnerability in the given version of the Sipher Staking Smart Contract.

2.4. Additional notes and recommendations

2.4.1. LiquidityMiningManager - removePool: no approval reset INFORMATIVE

Manager approve _poolContract for using all the tokens in addPool:

```
// Approve max token amount
reward.safeApprove(_poolContract, type(uint256).max);
```

But there is no approval reset in removePool.

RECOMMENDATION

Reset the approval in removePool method.

```
// Approve max token amount
reward.safeApprove(_poolContract, 0);
```

UPDATES

- *Dec 03, 2021*: This issue has been acknowledged and fixed by the Sipher team.

2.4.2. LiquidityMiningManager - conflicts access modifier onlyGov and onlyRewardDistributor INFORMATIVE

In LiquidityMiningManager, all methods with onlyGov modifiers call distributeRewards which is protected by onlyRewardDistributor, which means that gov must also be reward distributor so that these calls can be executed successfully.

The above guarantee is not stated within the contract's code, comments, or documents given to the audit team.

RECOMMENDATION

Use onlyGovOrRewardDistributor for distributeRewards.

UPDATES

- *Dec 03, 2021*: This issue has been acknowledged and fixed by the Sipher team.

2.4.3. LiquidityMiningManager - reward distribution process may be faulty with new pool type INFORMATIVE

Reward distribution process is implemented in distributeRewards:

```
function distributeRewards() public onlyRewardDistributor {
    uint256 timePassed = block.timestamp - lastDistribution;
    uint256 totalRewardAmount = rewardPerSecond * timePassed;
    lastDistribution = block.timestamp;

    // return if pool length == 0
    if(pools.length == 0) {
```

```
        return;
    }

    // return if accrued rewards == 0
    if(totalRewardAmount == 0) {
        return;
    }

    reward.safeTransferFrom(rewardSource, address(this), totalRewardAmoun…
t);

    for(uint256 i = 0; i < pools.length; i ++) {
        Pool memory pool = pools[i];
        uint256 poolRewardAmount = totalRewardAmount * pool.weight / tota…
lWeight;
        // Ignore tx failing to prevent a single pool from halting reward…
 distribution
        address(pool.poolContract).call(abi.encodeWithSelector(pool.poolC…
ontract.distributeRewards.selector, poolRewardAmount));
    }

    uint256 leftOverReward = reward.balanceOf(address(this));

    // send back excess but ignore dust
    if(leftOverReward > 1) {
        reward.safeTransfer(rewardSource, leftOverReward);
    }

    emit RewardsDistributed(_msgSender(), totalRewardAmount);
}
```

The above snippet can be summarized following:

- Distribution setup: calculate how much the reward should be distributed.
- Take exactly the reward amount as calculated from above from the reward's source.
- **Call each pool to work on the distribution itself by passing its corresponding reward amount, ignore errors, one after another.**
- Send back leftover tokens to the reward's source.

This process seems to be ok at first, with the given project's source code, as there're only 2 types of pools defined: TimeLockPool and TimeLockNonTransferablePool.

The problem is that if another type of pool is added to this manager contract, we have no safe point to trust that the pool will use only within its allowed amount or not. If a pool drains all the remaining balance of the manager, all the following pools will be silently error, transaction will succeed, and nothing can be done to recover except manually transfer the missing amounts - that includes debugging the transaction to see which call is reverted, and how much should it have transferred to each individual.

RECOMMENDATION

There are 3 possible fixes:

- Clearly state that the pool contracts are trustable within dev comments, and ensure that constraint when interacting with the deployed contract.
- Approve exactly the amount that the pool can use before calling it.
- Transfer the amount to the pool before calling it, no approval.

UPDATES

- *Dec 03, 2021*: This issue has been acknowledged by the Sipher team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 02, 2021*	Public Report	Verichains Lab
1.1	*Dec 03, 2021*	Public Report	Verichains Lab

Table 2. Report versions history