verichains

SECURITY AUDIT OF

HOLDSTATION STAKING

Public Report

Dec 05, 2023

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 Dec 05, 2023. We would like to thank the Holdstation 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 Holdstation Staking. 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 issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Holdstation Staking

Stake HOLD/USDC tokens to get a actual return in USDC rewards as well as extra esHOLD rewards. Stakers can profit from up to 40% of the protocol's trading fees. Rewards are split according to your staking compared to the whole pool, depending on protocol-generated fees. A request is required to unstake your staking, which takes at least 5 epochs, or 20 days (1 epoch = 4 days).

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the smart contracts of Holdstation Staking. It was conducted on git repository *https://gitlab.com/hspublic/contract-holdstation-dex*.

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

SHA256 Sum	File
298a155a3ad21ecff02704fe4fdb2e5608aaf907ae488aa53c0d8f431cdca930	contracts/functions/HSStaking.sol
f043a50f71aca96486521037d109c116f92a206bcc8744af6092b6a626a0f31d	contracts/functions/HSTradingVault.sol
34fd3e21edafa527a478049394e09cd10fbea06cbe90647de6505a075ea27e88	contracts/functions/HSVestingESHold.sol
ea7f09f7fd7a84764bf594be1498c43468111a0fd51dcc358075ec6ea2b0a5bb	contracts/tokens/HSEscrowHold.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

Holdstation 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. Holdstation understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Holdstation 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 Holdstation will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Holdstation, the final report will be considered fully accepted by the Holdstation without the signature.

2. AUDIT RESULT

2.1. Overview

The Holdstation Staking was written in `Solidity` language, with the required version to be `^0.8.10`. The source code was written based on OpenZeppelin's libraries.

2.1.1. HSStaking.sol

The `HSStaking` smart contract is a decentralized staking platform implemented in Solidity. Its primary purpose is to allow users to stake a particular token, earn rewards in USDC and a custom reward token (`esHold`), and manage various staking-related functionalities. Using OpenZeppelin's upgradeable contract structure, it ensures flexibility for future updates. Users can stake and unstake tokens, with rewards calculated and distributed based on staked balances and epochs. The contract introduces a custom reward token, esHold, claimable at the end of each epoch. Epoch management allows for the forced start of a new epoch after a specified timelock. Administrative controls include functions for updating admin addresses and adjusting reward percentages.

2.1.2. HSTradingVault.sol

The `HSTradingVault` smart contract is protocol designed for managing a trading vault with the `HSToken`, providing functionalities such as deposits, withdrawals, and minting/burning of `HSTokens`. This upgradable contract integrates various features, including OpenZeppelin libraries for ERC-20 token and ERC-4626 functionality, ensuring standardization and upgradability.

2.1.3. HSVestingESHold.sol

Users can stake their ES Hold tokens based on different staking types (Flexible, Lock Medium Epochs, Lock High Epochs). Staking yields rewards, and the contract enforces lock-up periods during which users cannot claim their rewards. The contract supports functionalities to configure reward percentages, locked epochs, and the frequency of reward requests.

The staking process involves burning ES Hold tokens, and users receive a portion of their staked amount as rewards. Users can claim their rewards after each epoch, with the option to claim for multiple epochs at once. The contract provides functions to calculate, estimate, and claim rewards, and it emits events to track staking and claiming activities.

2.1.4. HSEscrowHold.sol

The `HSEscrowHold` smart contract is protocol with functionalities related to holding and claiming rewards. It extends various OpenZeppelin contracts and interfaces, including

ERC20Upgradeable for token management and ReentrancyGuardUpgradeable for security. Users can claim rewards based on their contributions, and the contract supports functionalities for minting, burning, and managing balances during transfers.

2.2. Findings

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

#	Severity	Name	Status
1	HIGH	Withdraw asset from Trading Vault without making withdraw request	FIXED
2	INFORMATIVE	Using both `_msgSender()` and `msg.sender` concurrently poses a risk	ACKNOWLEDGED
3	INFORMATIVE	Running out of gas is a potential issue when claiming within a loop	ACKNOWLEDGED

2.2.1. HIGH - Withdraw asset from Trading Vault without making withdraw request

Affected files:

- `contracts/functions/HSTradingVault.sol`

2.2.1.1. Description

The `HSTradingVault` inherits the `ERC4626Upgradeable` abstract, which includes the `ERC4626` implementation. According to the comments, withdrawal requests must be called before invoking `withdraw()`/`redeem()`. However, the `HSTradingVault` contract does not override the `withdraw()` function, causing the contract to perform the default `withdraw()` code instead of validating the withdrawal request of the current epoch.

The contract implements `redeem()` function:

```
function redeem(
    uint256 shares,
    address receiver,
    address owner
  ) public override checks(shares, address(tokenCredit) != address(0)) returns (uint256) {
    if (address(tokenCredit) != address(0)) {
      require(_msgSender() == owner && _msgSender() == receiver, "SENDER_MUSTBE_OWNER");
    }
    require(shares <= maxRedeem(owner), "ERC4626: redeem more than max");

    withdrawRequests[owner][currentEpoch] -= shares;
```

```
    uint256 assets = previewRedeem(shares);
    scaleVariables(shares, assets, false);

    _withdraw(_msgSender(), receiver, owner, assets, shares);

    if (address(tokenCredit) != address(0)) {
      tokenCredit.notifyBalanceChange(_msgSender(), address(0));
    }
    emit Redeemed(receiver, assets, currentEpoch);
    return assets;
  }
```

As far as above code, users should initialize withdrawal request:

```
function makeWithdrawRequest(uint256 shares, address owner) external {
    require(shares > 0, "ZERO_VALUE");
    require(openTradesPnlFeed.nextEpochValuesRequestCount() == 0, "END_OF_EPOCH");
    address sender = _msgSender();
    uint256 unlockEpoch = currentEpoch + withdrawEpochsTimelock();

    if (address(tokenCredit) != address(0)) {
      require(sender == owner, "SENDER_MUSTBE_OWNER");
    } else {
      require(sender == owner || (allowance(owner, sender) > 0 && allowance(owner, sender)
>= shares), "NOT_ALLOWED");
    }

    require(totalSharesBeingWithdrawn(owner) + shares <= balanceOf(owner),
"MORE_THAN_BALANCE");

    withdrawRequests[owner][unlockEpoch] += shares;

    emit WithdrawRequested(sender, owner, shares, currentEpoch, unlockEpoch);
  }
```

RECOMMENDATION

The `withdraw()` function, like `redeem()`, must be implemented to validate the withdrawal request of the current epoch.

UPDATES

- ***Dec 05, 2023***: The issue has been fixed by Holdstation Staking team.

2.2.2. INFORMATIVE - Using both `_msgSender()` and `msg.sender` concurrently poses a risk

Affected files:

- `contracts/functions/HSStaking.sol#L131`

2.2.2.1. Description

In the HSStaking contract, the code references `_msgSender()` to identify the transaction sender. Simultaneously, the code also uses `msg.sender`. This dual usage poses a potential risk in scenarios where the protocol incorporates a **Gas Station**. This situation could lead to confusion regarding the true sender of the transaction, presenting a concern for future developments in the protocol.

Should:

```
address sender = _msgSender();
```

Should not:

```
User storage u = users[msg.sender];
```

UPDATES

- ***Dec 05, 2023***: The issue has been fixed by Holdstation Staking team.

2.2.3. INFORMATIVE - Running out of gas is a potential issue when claiming within a loop

Affected files:

- `contracts/functions/HSVestingESHold.sol#L100`
- `contracts/tokens/HSEscrowHold.sol#L170`
- `contracts/functions/HSStaking.sol#L245`

2.2.3.1. Description

Employing a loop in the claim function of the above contracts poses a risk, as it may lead to transaction failures due to running out of gas. For example, consider a scenario where a user stakes for the first time, and as the epoch progresses significantly, the loop within the contract needs to iterate from zero to the current large epoch.

UPDATES

- ***Dec 05, 2023***: The issue has been acknowledged.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Dec 05, 2023*	Public Report	Verichains Lab

Table 2. Report versions history