QuillAudits

Audit Report September, 2024

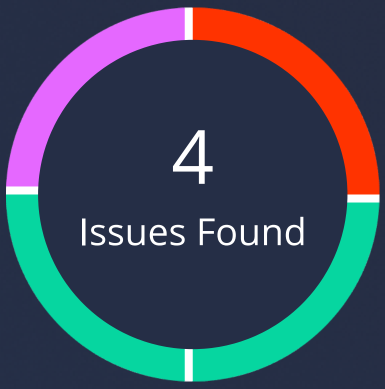
For

Table of Content

Executive Summary

Project Name	Headstarter
Overview	Headstarter is a decentralized application layer protocol designed to facilitate the growth and funding of projects within the Hedera Hashgraph ecosystem. As the first Layer 1 launchpad on Hedera, Headstarter aims to connect early-stage projects with community support and decentralized funding, thereby enhancing their visibility and potential for success.
Timeline	19th august 2024 - 3rd september 2024
Updated Code Received	10th September 2024
Second Review	11th September 2024 - 13th September 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	Scope of the Audit is analyse the security and code quality of Headstarter Codebase
Source Code	1. https://github.com/buidler-labs/headstarter 2. contracts/contracts/staking/StakingPool.sol
Contracts In-Scope	Staking.sol iStaking.sol Helper.sol DSMath.sol StakingPool.sol
Branch	Main
Commit Hash	49e41f202954d32010894748a4ade63962700d3b
Fixed In	https://github.com/buidler-labs/headstarter/commit/953f628b8b9681efa1deb22aaf016f81a1d1e353

Number of Security Issues per Severity



🟥 High 🟨 Medium
🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	**1**	0	**2**	**1**

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ DoS with Block Gas Limit
- ✓ Transaction-Ordering Dependence
- ✓ Use of tx.origin
- ✓ Exception disorder
- ✓ Gasless send
- ✓ Balance equality
- ✓ Byte array
- ✓ Transfer forwards all gas
- ✓ ERC20 API violation
- ✓ Compiler version not fixed
- ✓ Redundant fallback function
- ✓ Send instead of transfer
- ✓ Style guide violation
- ✓ Unchecked external call
- ✓ Unchecked math
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Hardhat, Foundry.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Centralization Risk

Path

Staking.sol

Function

reinvest()

Description

The function reinvest()is responsible for reinvesting the amount of pending rewards for a user back into the system by calling _stake().

Due to the lack of using msg.sender instead of an arbitrary address _user address and the fact that this function is public, any user could call this function on behalf of other users by simply giving their address as a user parameter.

This could be problematic since it diverges from a user having absolute ownership for their stake.

Consider a scenario:

1. Ower creates a pool
2. User1 stakes 1000 tokens
3. Owner deposit rewards
4. After some time, User1 calls claim() but User2 frontruns this call with reinvest(pid , user1)

Paste the following POC in staking.t.sol :

```
 function test_reinvest_for_anyone() public {
     // 1. Owner adds Pool
     vm.startPrank(owner);
     stakingContract.addPool(
         apyRate,
         block.timestamp + 30 days,
         30,
         10000 * DECIMALS,
         penaltyRate
     );
     vm.stopPrank();
     //2. User1 stakes in the pool
     vm.prank(user1);
```

```
    stakingContract.stake(0, 1000 * DECIMALS);
    //3. Owner depositRewards
    vm.prank(owner);
    stakingContract.depositRewards(0, 1000 * DECIMALS);
    vm.warp(block.timestamp + 60 seconds);
    //4. User1 wanted to claim but instead user2 came in and reinvested    on User1's
behalf
    vm.prank(user2);
    stakingContract.reinvest(0, user1);
    }
```

Recommendation

Instead of using user address, use msg.sender instead

Status

Resolved

Low Severity Issues

2. Use Ownable2Step instead of Ownable

Path

Staking.sol

Description

Currently Staking.sol is inheriting OpenZeppelin’s ownable library to manage the ownership of smart contracts.

Ownable.sol uses a single-step ownership change via transferOwnership().

However, in the event of an accidental ownership transfer to another address, this might cause a problem.

It is therefore recommended to use Ownable2Step which first stores the address of the new owner into a transient state called pendingOwner which is only confirmed once acceptOwnership() is called by the new owner.

Recommendation

Use Ownable2Step instead

Status

Resolved

3. Not updating the lastInteractionTime

Path

StakingPool.sol

Description

lastInteractionTime keeps track of the last timestamp when the user interacted with a particular pool. This needs to be updated everytime user calls a function the StakingPool like stake, withdraw, depositNFT, withdrawNFT and so on.

withdraw and emergencyWithdraw functions are not updating this variable when a user calls them. This will lead to incorrect calculation of pending rewards.

```
function emergencyWithdraw(uint256 _pid) external nonReentrant {
    ...
    userStake.amount = 0;
    // @audit not updating the lastInteractionTime

    if (pendingReward > 0) {
        if (pendingReward > pool.rewardBalance)
            revert InsufficientRewardBalance();
        pool.rewardBalance -= pendingReward;
    }
    ...
}
function withdraw(uint256 _pid) external nonReentrant {
    ...
    userStake.amount = 0;
    // @audit not updating the lastInteractionTime
    ...
}
```

Recommendation

Consider to update the lastInteractionTime in above functions.

Status

Resolved

Informational Issues

4. Emit events for critical state changes

Path

Staking.sol

Variable

setMaxMultipler()

Description

It is generally considered as best practice to emit events for critical state changes. These are the functions that does not adhere to this practice.

1. setMaxMultiplier()

Recommendation

Emit event for these functions

Status

Resolved

Functional Tests Cases

Some of the tests performed are mentioned below:

- ✓ Test staking, withdrawing, and claiming with very large numbers
- ✓ Test with multiple users interacting with the same pool.
- ✓ Test with a user interacting with multiple pools
- ✓ Test reward calculation over different time periods
- ✓ Attempting to stake after should always fail stake falls below lock period.
- ✓ Should not be able to add pools if not owner
- ✓ Should receive rewards - penalty if claim() is called before ending period.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Headstarter contract. We performed our audit according to the procedure described above.

Some issues of High, Low severity and Informational were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the End, Headstarter Team Resolved all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Headstarter smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Headstarter smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Headstarter Team to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
September, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com