QuillAudits

AUDIT REPORT

February, 2025

For

Hand of God

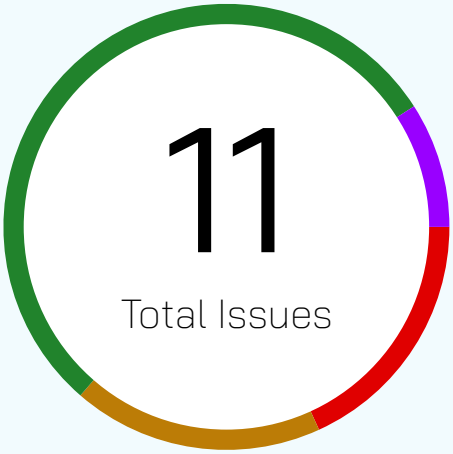
Table of Content

Executive Summary

Project name	Hand of God
Overview	Masonry.sol: Staking contract that handles reward distribution and lockup periods. Implements snapshot mechanics for tracking historical rewards and user positions. Features both withdrawal and reward lockup periods for staked shares. Treasury.sol: Core protocol contract managing the entire economic system. Handles bond purchases, redemptions, and expansion/contraction of supply. GHOGReward.sol: This is the staking/farming contract where users can earn GHOG rewards (similar to how TSHARE rewards work in Tomb).
Contracts In Scope	GHogRewardPool.sol Manosry.sol Treasury.sol
Audit_Scope	The scope of this Audit was to analyze the ZynkLabs Smart Contracts for quality, security, and correctness. https://github.com/chimpytuts/hand-of-god-contracts
Method	Manual Review,Functional testing and Automated Scan
Commit Hash	0f4f04765387e89401ed12b42a1e892bffceebbe
Branch	Main
Language	Solidity
Blockchain	Sonic
Review 1	26th February 2025 - 3rd March 2025

Update_code_Received	3rd March 2025
Review 2	3rd March 2025
Fixed_In	201ba7c03746cbf9efa7e0e4f5aca8c56255065a

Number of Issues per Severity



High	2 (18.18%)
Medium	2 (18.18%)
Low	6 (54.55%)
Informational	1 (9.09%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**1**	0	**5**	0
Acknowledged	**1**	**2**	**1**	**1**
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ✔ Access Management
- ✔ Arbitrary write to storage
- ✔ Centralization of control
- ✔ Ether theft
- ✔ Improper or missing events
- ✔ Logical issues and flaws
- ✔ Arithmetic Computations Correctness
- ✔ Race conditions/front running
- ✔ SWC Registry
- ✔ Re-entrancy
- ✔ Timestamp Dependence
- ✔ Gas Limit and Loops
- ✔ Exception Disorder
- ✔ Gasless Send
- ✔ Use of tx.origin
- ✔ Malicious libraries
- ✔ Compiler version not fixed
- ✔ Address hardcoded
- ✔ Divide before multiply
- ✔ Integer overflow/underflow
- ✔ ERC's conformance
- ✔ Dangerous strict equalities
- ✔ Tautology or contradiction
- ✔ Return values of low-level calls
- ✔ Missing Zero Address Validation
- ✔ Private modifier
- ✔ Revert/require functions
- ✔ Multiple Sends
- ✔ Using suicide
- ✔ Using delegatecall
- ✔ Upgradeable safety
- ✔ Using throw,Using inline assembly

- [x] Style guide violation
- [x] Implicit visibility level.
- [x] Unsafe type inference

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 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

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

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 Issues

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

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

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

Over Expansion Due to Incorrect percentage Calculation in Treasury Contract

Acknowledged

Path

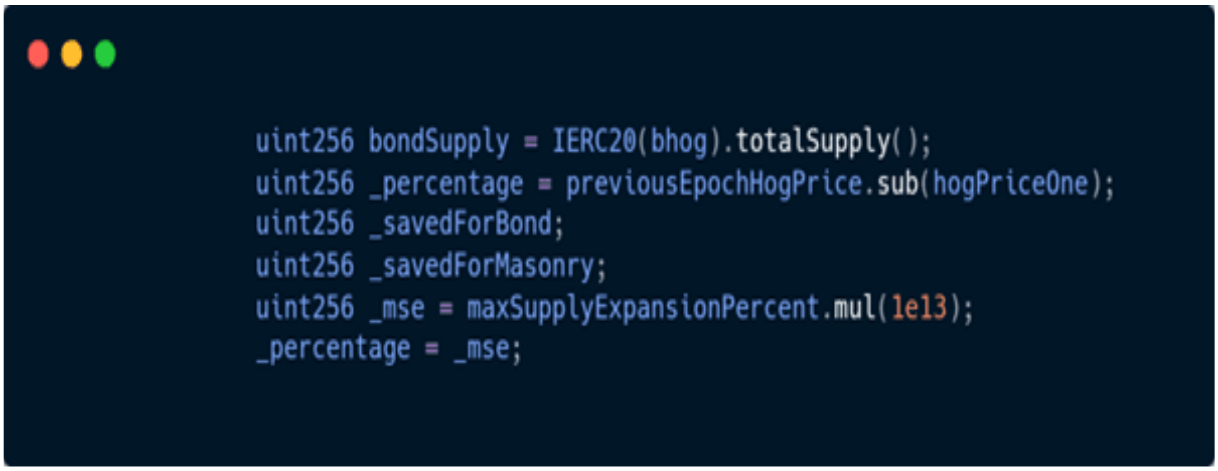
Treasury

Function

allocateSeigniorage()

Description

The Treasury contract always uses the maximum expansion rate (0.15%) to mint new tokens, even when the price is only slightly above the target peg.
For example : if the current price is 1.02, it is only 0.02 % so the contract should use 0.02% , but in the current implementation it is using maximum value 0.15 %



```
uint256 bondSupply = IERC20(bhog).totalSupply();
uint256 _percentage = previousEpochHogPrice.sub(hogPriceOne);
uint256 _savedForBond;
uint256 _savedForMasonry;
uint256 _mse = maxSupplyExpansionPercent.mul(1e13);
_percentage = _mse;
```

Recommendation

Implement additional checks to prevent excessive minting during minor deviations , like in tomb finance.

The contract doesn't properly handle historical reward rate changes when calculating rewards.

Resolved

Path

GHogRewardPool

Function

getGeneratedReward()

Description

Example:
- If rate was 2 GHOG/sec for first 7 days then changed to 1 GHOG/sec
- A user claiming after 14 days would get: 14 days * current_rate(1 GHOG/sec)
- Instead of correct: 7 days * old_rate(2 GHOG/sec) + 7 days * new_rate(1 GHOG/sec)

Medium Severity Issues

Permanent Loss of Protocol Control Through Operator Renouncement

Acknowledged

Path

Treasury

Function

renounceOperator()

Description

The Treasury contract allows the operator to permanently renounce their role without any recovery mechanism. Once renounced, critical protocol functions become permanently inaccessible.

```
function renounceOperator() external onlyOperator {
    _renounceOperator();
}

// In Operator.sol
function _renounceOperator() internal {
    emit OperatorTransferred(operator(), address(0));
    _operator = address(0);
}
```

Recommendation

Remove or disable renounceOperator.

Use safeTransfer instead of transfer:

Acknowledged

Path

Treasury

Function

_sendToMasonry()

Description

In _sendToMasonry() function ,transfer() function is used for sending which might return false instead of reverting, in this case, ignoring return value leads to considering it successful.

Recommendation

use safeTransfer

Low Severity Issues

In getTotalEmittedSharesBetween(), the requirement check should use strictly less than, not less than or equal. As the generateReward function will return 0 , if both are equal .

Resolved

Path

GHogRewardPool

Function

getTotalEmittedSharesBetween()

Description

The function uses <= in its requirement check, allowing equal timestamps When _fromTime equals _toTime, the subsequent call to getGeneratedReward() will always return 0 This creates an unnecessary valid case that serves no purpose

Recommendation

use < instead of <=

Missing Start Time Validation in Constructor

Acknowledged

Path

GHOG.sol

Function

constructor

Description

No validation that _startTime is in the future Could accidentally set start time to a past timestamp this would immediately enable vesting/rewards without intended delay.

Recommendation

Add a check :
require(_startTime > block.timestamp, "Start time must be future");

Missing Zero Address Validation in setOperator :

Resolved

Path

GHogRewardPool

Function

setOperator()

Description

No validation against zero address ,Accidentally setting operator to zero address would permanently disable these functions

Recommendation

Add a check : require(_operator != address(0).

Imprecise Withdraw Fee Limit Check

Resolved

Path

GHogRewardPool

Function

set()

Description

The comment states "withdraw fee cant be more than 2%" However, the code uses < 200 which actually prevents exactly 2% (200 basis points) from being set

Recommendation

use less than or equals <= instead of <

In governanceRecoverUnsupported() function,lockout period (10 days) doesn't match the comment (90 days).

Resolved

Path

GHogRewardPool

Function

governanceRecoverUnsupported()

Description

The code implements a 10-day lockout period after pool ends The comment incorrectly states this period is 90 days This inconsistency between code and documentation could lead to confusion about the actual security measures in place.

Recommendation

Update the comment to match the code.

Redundant Pool Update Call in GHogRewardPool.

Resolved

Path

GHogRewardPool

Function

set()

Description

The set() function calls massUpdatePools() which already updates all pools including the target pool The subsequent updatePool(_pid) call is unnecessary because: 1. massUpdatePools() has already updated this pool 2. If block.timestamp <= pool.lastRewardTime, the second call will immediately return anyway

Recommendation

remove the redundant function call.

Informational Severity Issues

Unused code /functions :

Acknowledged

Path

Treasury,GHogRewardPool

Description

Unused code instances .
1. poolLength function in GHogRewardPool
2. getHogUpdatedPrice() and getReserve() in Treasury.

Recommendation

Remove if it won't be used in the future

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 Hand of God. We performed our audit according to the procedure described above.

Some issues of High, low ,medium and infromational severity were found. In the End,Hand on God Team Resolved few Issues and Acknowledged other Issue.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the recieved smart contract source code

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

February, 2025

For

Hand of God

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com