Least Authority
PRIVACY MATTERS

Octant Smart Contracts
Security Audit Report

# Golem Foundation

Final Audit Report: 20 July 2023

# Table of Contents

Overview

- Background
- Project Dates
- Review Team

Coverage

- Target Code and Revision
- Supporting Documentation
- Areas of Concern

Findings

- General Comments
- Specific Issues & Suggestions
- Suggestions
  - Suggestion 1: Emit Event When Multi-Sig Address Is Updated
  - Suggestion 2: Check the Contract Balance Before Sending ETH to the Multi-Sig Address
  - Suggestion 3: Check the Input Parameter in the SetMerkleRoot Function
  - Suggestion 4: Add Checks for Array Elements
  - Suggestion 5: Perform Zero Address Check
  - Suggestion 6: Prevent Locking Zero Amount
  - Suggestion 7: Prevent the Setting of the initialize Function in the Implementation
  - Suggestion 8: Use a Non-Floating Pragma Version Consistently Across the Project
  - Suggestion 9: Set Function Visibility Appropriately

About Least Authority

Our Methodology

# Overview

## Background

Golem Foundation has requested that Least Authority perform a third security audit of their Octant Smart Contracts.

## Project Dates

- **June 22, 2023 - June 28, 2023:** Initial Code Review *(Completed)*
- **June 30, 2023:** Delivery of Initial Audit Report *(Completed)*
- **July 19, 2023:** Verification Review *(Completed)*
- **July 20, 2023:** Delivery of Final Audit Report *(Completed)*

## Review Team

- Mukesh Jaiswal, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer

# Coverage

## Target Code and Revision

For this audit, we performed research, investigation, and review of the Octant Smart Contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Octant Contracts-v1: https://gitlab.com/wildland/governance/octant/-/tree/master/contracts-v1

Specifically, we examined the Git revision for our initial review:

- `418ab9f16eb3b2c21d9f0a6d46dc580f5d38dfb4`

For the verification, we examined the Git revision:

- `e1e5945bcd2e9f438e4b84ad0accb177335969e7`

For the review, this repository was cloned for use during the audit and for reference in this report:

- Octant-smart-contracts: https://github.com/LeastAuthority/octant-smart-contracts/tree/master/contracts

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

## Supporting Documentation

The following documentation was available to the review team:

- Projects | Octant: https://golem.foundation/projects#Octant

In addition, this audit report references the following documents:

- OpenZeppelin:
  https://wizard.openzeppelin.com

## Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

# Findings

## General Comments

The Octant Smart Contracts compose the on-chain component of Golem Foundation's Octant Governance Experiment.

The project leverages the existing community of Golem Network Token (GLM) holders and offers staking rewards for users who perform specific activities within various experiments initiated by the Golem Foundation. The amount of total rewards is determined by the amount of GLMs locked by a user. The rewards are then split into user rewards and matched rewards, which can be donated to proposals selected by the Golem Foundation for public good purposes.

Our team initially performed a security audit of the Octant Smart Contracts with the audit report delivered on March 15, 2023. This audit was followed by a design review of the Octant Smart Contracts with a design review summary delivered on April 14, 2023.

In this engagement, our team performed a comprehensive security audit of the Octant Smart Contracts' design and implementation, investigating the areas of concern listed above. Our team noted that our recommendations from the previous reviews have been implemented. Additionally, we did not identify issues in the design and implementation of the smart contracts. Our team identified some suggestions for the improvement of the overall security and quality of the implementation.

## Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

| ISSUE / SUGGESTION | STATUS |
|---|---|
| Suggestion 1: Emit Event When Multi-Sig Address Is Updated | Resolved |
| Suggestion 2: Check the Contract Balance Before Sending ETH to the Multi-Sig Address | Resolved |
| Suggestion 3: Check the Input Parameter in the SetMerkleRoot Function | Resolved |
| Suggestion 4: Add Checks for Array Elements | Partially Resolved |
| Suggestion 5: Perform Zero Address Check | Resolved |
| Suggestion 6: Prevent Locking Zero Amount | Resolved |
| Suggestion 7: Prevent the Setting of the initialize Function in the Implementation | Resolved |
| Suggestion 8: Use a Non-Floating Pragma Version Consistently Across the Project | Resolved |
| Suggestion 9: Set Function Visibility Appropriately | Resolved |

## Suggestions

### Suggestion 1: Emit Event When Multi-Sig Address Is Updated

**Location**

`contracts/Auth.sol#L31`

**Synopsis**

Events should be emitted for critical parameter changes. However, an event is not emitted in the constructor when the Multi-Sig address is updated.

**Mitigation**

We recommend emitting an event when the Multi-Sig address is updated.

**Status**

The Octant team has implemented the mitigation as recommended.

**Verification**

Resolved.

## Suggestion 2: Check the Contract Balance Before Sending ETH to the Multi-Sig Address

**Location**

[contracts/withdrawals/WithdrawalsTarget.sol#L39]

**Synopsis**

In the `withdraw` function, low-level calls are performed to send ETH to the Multi-Sig address. However, there is no check to verify if the contract has enough balance, due to which the `call` function will fail.

**Mitigation**

We recommend checking the contract balance before sending ETH to the Multi-Sig address.

**Status**

The Octant team has added the following check to determine if the contract has enough balance before sending ETH to the Multi-Sig address:

```
require(address(this).balance >= amount, CommonErrors.FAILED_TO_SEND);
```

**Verification**

Resolved.

## Suggestion 3: Check the Input Parameter in the SetMerkleRoot Function

**Location**

[contracts/Vault.sol#L40]

**Synopsis**

There are no checks for the function parameter root value in `SetMerkleRoot`, due to which an invalid value of root can be set.

**Mitigation**

We recommend adding checks for the root value in `setMerkleRoot`, as follows:

```
require( root ! = bytes32(0))
```

**Status**

The Octant team has added checks to verify that the root value is not equal to `bytes32(0)`.

**Verification**

Resolved.

## Suggestion 4: Add Checks for Array Elements

**Location**

[contracts/Proposals.sol#L33]

[contracts/Proposals.sol#L60]

**Synopsis**

While setting the proposal address for the epoch, it does not check whether the elements of the `Proposal` array are different and not `address(0)`, which can affect the calculation of funds donated by the user for a proposal.

**Mitigation**

We recommend adding checks to verify that a given array neither has a similar address nor contains `address(0)` in it.

**Status**

The Octant team has added an `address(0)` check for the array. However, there are still no checks implemented to verify the similar values in an array.

**Verification**

Partially Resolved.

## Suggestion 5: Perform Zero Address Check

**Location**

`contracts-v1/contracts/OctantBase.sol#L18`

`contracts-v1/contracts/Deposits.sol#L41`

**Synopsis**

There is no zero address check validating the correctness of the `_auth` and `glmAddress` parameters in the constructors, thereby preventing incorrectly set values.

**Mitigation**

We recommend checking the referenced parameters against zero addresses.

**Status**

The Octant team has implemented the mitigation as recommended.

**Verification**

Resolved.

## Suggestion 6: Prevent Locking Zero Amount

**Location**

`contracts-v1/contracts/Deposits.sol#L47-L55`

**Synopsis**

The `lock` function does not check if the `amount` being locked is greater than zero, resulting in the completion of an unnecessary transaction.

**Mitigation**

We recommend adding a check to revert the transaction if the `amount` is not greater than zero.

**Status**
The Octant team has implemented the mitigation as recommended.

**Verification**
Resolved.

## Suggestion 7: Prevent the Setting of the initialize Function in the Implementation

**Location**
[`contracts/withdrawals/WithdrawalsTarget.sol#L29-L32`]

**Synopsis**
Although the `WithdrawalsTarget` smart contract should be utilized as an upgradeable smart contract, there is no clear evidence verifying that it is, in fact, upgradeable. Additionally, the `initialize` function used in the contract can be called by any user initializing the implementation.

**Mitigation**
We recommend that the `WithdrawalsTarget` smart contract extend one of OpenZeppelin's upgradeable contracts. We further recommend following their guidelines in the implementation, which include disabling the initialization of the implementation in the implementation's constructor, hence preventing potential security risks by only allowing the `initialize` function to be called in the constructor of the proxy.

OpenZeppelin's Contract Wizard can be utilized to quickly build an upgradeable contract template, depending on the type of upgradeability.

**Status**
The Octant team has disabled the initialization of the contract in the constructor.

**Verification**
Resolved.

## Suggestion 8: Use a Non-Floating Pragma Version Consistently Across the Project

**Synopsis**
Smart contracts in the project have their pragma set to `^0.8.16`. Compiling with different compiler versions may cause conflicts and unexpected results, and possibly lead to the smart contracts being deployed with an unintended compiler version, which could result in unexpected behavior.

**Mitigation**
In order to prevent unexpected behavior, we recommend that the Solidity compiler version be pinned by removing `"^"` and updated to the latest version.

**Status**
The Octant team has implemented the mitigation as recommended.

**Verification**

Resolved.

## Suggestion 9: Set Function Visibility Appropriately

**Location**

`contracts-v1/contracts/Proposals.sol#L41`

`contracts-v1/contracts/Proposals.sol#L61`

`contracts-v1/contracts/Proposals.sol#L74`

`contracts-v1/contracts/withdrawals/WithdrawalsTarget.sol#L29`

`contracts-v1/contracts/withdrawals/WithdrawalsTarget.sol#L34`

**Synopsis**

The functions referenced above are defined as `public`. However, they are not being used internally. It is considered best practice to define function visibility based on where the function will be used in order to improve the readability of the code and make it easier to identify incorrect assumptions about who can call the function.

**Mitigation**

We recommend defining the referenced public functions as `external` by replacing the `public` keyword with `external`.

**Status**

The Octant team has updated the visibility of the referenced functions as recommended.

**Verification**

Resolved.

## About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

## Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

### Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

### Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

## Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

## Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

## Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.