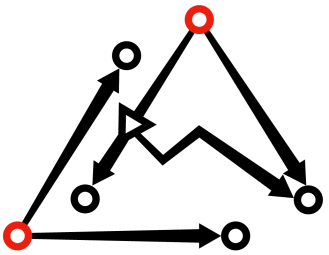

Least Authority

PRIVACY MATTERS

Data Lake Consents Smart Contracts

Security Audit Report

Data Lake

Final Audit Report: 19 December 2022

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
    - Issue A: Missing Check in addConsent Function
  - Suggestions
    - Suggestion 1: Implement a View Function to Check the Array Parameter
    - Suggestion 2: Add the Address(0) Check for the Manager
    - Suggestion 3: Use mapping(bytes32 => uint32[]) internal _personConsents to Get the Consent for an ID
    - Suggestion 4: Add Mapping to Check if the Hash Was Previously Used
    - Suggestion 5: Add Checks to Determine Whether Input agreedConsentId Is Active
- About Least Authority
- Our Methodology

Overview

Background

Data Lake has requested that Least Authority perform a security audit of their smart contracts.

Project Dates

- **December 1 - 6:** Initial Code Review *(Completed)*
- **December 7:** Delivery of Initial Audit Report *(Completed)*
- **December 15-16**: Verification Review *(Completed)*
- **December 19:** 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 Data Lake Consents 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:

- Data Lake Consent Contract:
  https://github.com/Data-Lake-LLC/consent-contract

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

- Consent: `d1ad0a11f3c609acabf6adb97a464ca59389672c`

For the verification, we examined the Git revision:

    9689daaece6b1e70c0fd6bb87b5fa8f16083f981

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

- Data Lake Consent Contracts:
  https://github.com/LeastAuthority/Data-Lake-Consent-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:

- Medical Data for Research (Data Lake Website):
  https://data-lake.co/researchers-medical-data-for-research

- Data-Lake-Whitepaper-1.8-1.pdf *(shared with Least Authority via email on October 31, 2022)*
- Smart Contracts architecture.pdf *(shared with Least Authority via email on October 31, 2022)*
- Vesting logic.pdf *(shared with Least Authority via email on October 31, 2022)*
- Vesting Smart Contracts.pdf *(shared with Least Authority via email on October 31, 2022)*
- Data Lake Token App Introduction.pdf *(shared with Least Authority via email on October 31, 2022)*

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)/security exploits that would impact the intended use of the contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit 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

As part of our review of the Data Lake smart contracts, our team performed an audit of the Token and Vesting smart contract implementation. A Final Audit Report was delivered on December 5, 2022, outlining the issues and suggestions our team identified in that codebase, and the status of those findings as of the verification stage of that review.

As an integral extension of the review of the Token and Vesting Smart Contracts, our team also audited the Data Lake Consent Smart Contracts repository, which implements the on-chain consent management functionality of the Data Lake system. This report details a missing check issue and suggestions that our team identified during this extension and can be considered an extension of the Final Audit Report delivered on December 5, 2022.

During our review, the Data Lake team identified issues in the implementation, which they shared with us. After a close review, our team determined that the issue and suggestions outlined in this report will sufficiently address the concerns raised, if resolved.

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
Issue A: Missing Check in addConsent Function	Resolved

Suggestion 1: Implement a View Function to Check the Array Parameter	Resolved
Suggestion 2: Add the Address(0) Check for the Manager	Resolved
Suggestion 3: Use mapping(bytes32 => uint32[]) internal _personConsents to Get the Consent for an ID	Resolved
Suggestion 4: Add Mapping to Check if the Hash Was Previously Used	Resolved
Suggestion 5: Add Checks to Determine Whether Input agreedConsentId Is Active	Unresolved

Issue A: Missing Check in addConsent Function

Location

`contracts/ConsentRegistry.sol#L17-L23`

Synopsis

It is possible to register a consent with zero `consentHash`. However, it is not possible to activate or deactivate a consent with zero `consentHash`.

Impact

A valid consent may be added by mistake, with a zero `consentHash`, which is not possible to activate. In this case, `activateConsent` would revert with an error message.

Preconditions

The `consentHash` passed to the `addConsent` function is zero.

Mitigation

We recommend implementing a check in the `addConsent` function to prevent the adding of a consent with zero `consentHash`.

Status

The Data Lake team implemented a check, which reverts on zero `consentHash`.

Verification

Resolved.

Suggestions

Suggestion 1: Implement a View Function to Check the Array Parameter

Location

`contract/access/Manager.sol`

Synopsis

The `_updateManager` function does not check the elements of the array, due to which the same array of IDs can be set for a manager.

Mitigation

We recommend creating a view function, which can check the elements of an array and return `bool` if it has a duplicate value.

Status

The Data Lake team added an internal function to check if the input array has duplicate IDs.

Verification

Resolved.

Suggestion 2: Add the Address(0) Check for the Manager

Location

`contract/access/Manager.sol`

Synopsis

Due to the lack of an `address(0)` check for `_managerToConsents`, mapping can be updated for the `address(0)`.

Mitigation

We recommend adding `address(0)` checks when updating the manager.

Status

The Data Lake team implemented a check, which reverts when the input manager address is `address(0)`.

Verification

Resolved.

Suggestion 3: Use mapping(bytes32 => uint32[]) internal _personConsents to Get the Consent for an ID

Location

`Contracts/consents.sol`

Synopsis

The `PersonConsentHashes` function uses `_consentCount` to iterate through the loop and collect details that are not related to the input ID.

Mitigation

We recommend using `mapping(bytes32 => uint32[]) internal _personConsents`, instead of `_consentsCount`, to obtain the consent for an ID.

Status

The Data Lake team is now utilizing the mapping of `_personConsents` instead of `_consentCount` to get the consent for the given ID.

Verification

Resolved.

Suggestion 4: Add Mapping to Check if the Hash Was Previously Used

Location

`Contract/ConsentRegistry.sol`

Synopsis

When adding the `consentHash` for `consentId`, there are no checks in place to determine whether the given `consentHash` was previously used, due to which the same hash can be linked to different `ConsentIds`.

Mitigation

We recommend adding mapping to check whether the hash had been used previously. For example: `mapping(bytes32 => bool) _check_hash`.

Status

The Data Lake team added the mapping of `hashHistory`, which keeps record of the used hash. In the `addConsent` function, the hash is checked twice, and as a result, the `require` statement can be removed.

Verification

Resolved.

Suggestion 5: Add Checks to Determine Whether Input agreedConsentId Is Active

Location

`Contract/consent.sol`

Synopsis

The `UpdatePersonConsent` function checks for `consentHash` but does not check whether input IDs are active or inactive.

Mitigation

We recommend adding a check in the `UpdatePersonConsent` function to verify whether the input IDs are active.

Status

The Data Lake team responded that this is intended behavior, as existing IDs can be in a deactivated state when updating the person consents.

Verification

Unresolved.

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, and zero-knowledge 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.