TENSET

Smart Contract Audit

Liquity token

Table of Content

Executive Summary

LQTY is the native governance and utility ERC20 token of the Liquity protocol. Holders of LQTY tokens have the ability to participate in the governance of the protocol by voting on proposals and decisions that affect the development and parameters of Liquity. Additionally, LQTY may be used within the Liquity system for various purposes, such as earning rewards or obtaining discounts on fees.

Project Summary

Language	Codebase	Commits
Solidity	https://github.com/liq-uity/dev	a0948b633b21ca147471a705e2d07f825f2f2292

Audit Scope

ID	File	SHA256
LQT	LQTYToken.sol	e389d532fd4465090153b3dbf9ba2 c7e61b8237eeccd93953bfd6d352eee0543
ILQ	ILQTYToken.sol	6d54ecced315fda5a33ddeaf729f1 78fd55cb5d0990fa7157272fdf1efa2b9df
ICF	ILockupContractFactory.sol	7e7c6a8d9f4dc43a6f02b1de70175 a5964f7ef6e835492d426f8aa1854e20358
CHC	CheckContract.sol	aa32079b9f38a1669beb9fefe2e0a f95fc543e1b717617713f00a648f0dc4b66
SM	SafeMath.sol	caa5397440fd9a0988eb40c136bd7 a58baad05012edcf244f6b586e167e531f6

General security assumptions

In conducting our security audit of the Liquity Token, we have made a number of important security assumptions. These assumptions are fundamental to our analysis and should be considered when evaluating the overall security and potential vulnerabilities of the system:

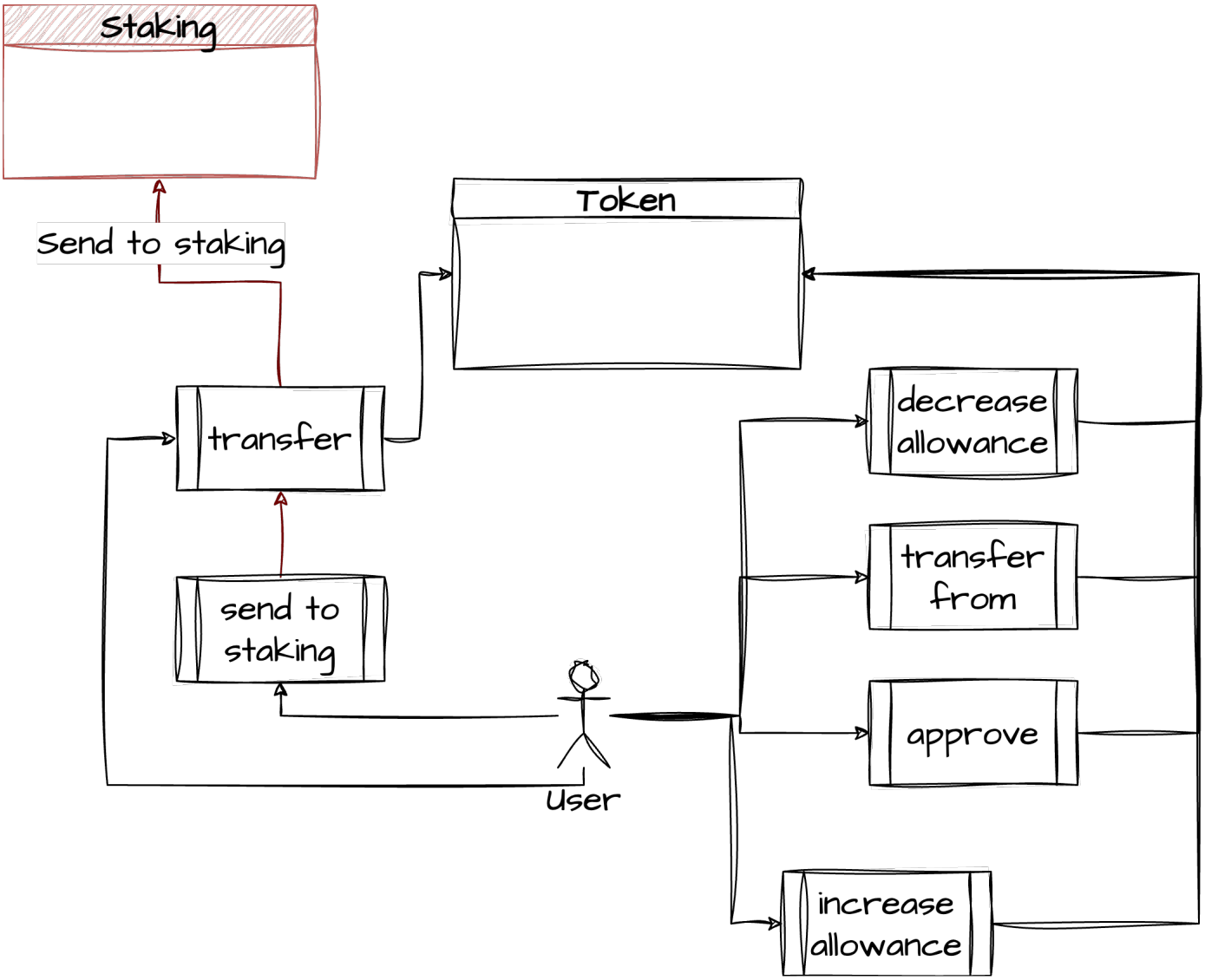
Security of the Underlying Blockchain
Our security analysis is predicated on the assumption that the underlying blockchain (for example, Ethereum) is secure and functions as intended. Potential vulnerabilities or flaws in the blockchain protocol are outside the scope of this audit.

Trusted Environment
We assume that users of the smart contract will interact with it in a secure and trusted environment. This encompasses the use of secure, up-to- date software and hardware, which is free from malware or any potential interference from malicious third parties.

Private Key Security
Users of the smart contract are assumed to safeguard their private keys effectively. This involves not sharing private keys, securely storing key backups, and using hardware wallets or other secure means for key management.

These security assumptions form the basis of our audit. Deviations from these assumptions could potentially lead to risks or vulnerabilities not covered in this analysis. Therefore, the effective management of these factors is critical to ensure the ongoing security of the Liquity Token.

System Overview



Findings

Audit Overview

Issues

Severity	Found	Resolved	Partially Fixed	Acknowledged
Critical	0	0	0	0
High	0	0	0	0
Medium	0	0	0	0
Low	0	0	0	0
Informational	8	0	0	0
Total	**8**	**0**	**0**	**0**

LQT-1
Missing Address Zero Check in Constructor

Informational Found

Description

The constructor of the contract lacks a check to ensure that parameters with the type `address` are not set to address zero (0x0) during initialization. Assigning address zero to critical variables or parameters of this type can lead to unexpected behavior, vulnerabilities, or potential loss of funds.

Recommendation

Implement a thorough check in the constructor for each parameter with the type `address` to verify that it is not set to address zero. This check is crucial for preventing unintended consequences and enhancing the security of the contract.

CHC-1
Lack of Documentation for Assembly Usage in CheckContract

Informational Found

Description

The usage of assembly in line 16 of the `CheckContract` contract lacks adequate documentation. The absence of clear documentation for assembly code may hinder the understanding of its purpose, potential risks, and intended functionality.

Recommendation

Provide comprehensive documentation for the assembly code in line 16 of the `CheckContract` contract. Clearly explain the purpose of the assembly instructions, any potential risks associated with their use, and how they contribute to the overall functionality of the contract.

GLOBAL-1
Improve Namespace Clarity in Imports

Informational Found

Description

In the contract code, import statements are used without the `{contract}` namespace specification, which can lead to potential naming conflicts and reduced code clarity. Import statements should be structured as `import {contract} from ...` to ensure a clear and unambiguous namespace.

Recommendation

It is advisable to update the import statements throughout the contract code to use the `{contract}` namespace format. This practice enhances code clarity, reduces the likelihood of naming conflicts, and makes it easier for developers to understand the origin of imported contracts.

GLOBAL-2
Use `uint256` Instead of `uint`

Informational Found

Description

Throughout the contract code, the data type `uint` is used, which is implicitly equivalent to `uint256`. While this does not impact functionality, it is considered a best practice to explicitly specify `uint256` to enhance code readability and provide clarity on the variable's intended bit width.

Recommendation

It is recommended to update the data type `uint` to `uint256` for all relevant variables in the contract. This explicit declaration of `uint256` aligns with best practices and makes the code more self-explanatory, reducing potential confusion for developers and users.

GLOBAL-3
Inconsistent Function Ordering

Informational Found

Description

The functions within the contract code are not ordered according to a standardized pattern. It is recommended to establish a consistent ordering for functions to enhance code readability and maintainability. A common and best practice order for function grouping, visibility, and modifiers is as follows:
- Constructor
- Receive function (if exists)
- Fallback function (if exists)
- External
- Public
- Internal
- Private
- View and Pure Functions (placed at the end within each visibility level)

Recommendation

To align with best practices and improve code structure, it is advisable to reorder the functions within the contract according to the suggested pattern. Consistent function ordering promotes a standardized codebase and makes it easier for developers to locate and understand the functions within the contract.

LQT-2
`_1_MILLION` Variable Should Be a Constant

Informational Found

Description

The `_1_MILLION` variable is a constant value that does not change during the execution of the contract. To enhance code readability and convey the immutability of this value, it is recommended to declare it as a constant.

Recommendation

Declare `_1_MILLION` as a constant variable to explicitly communicate its immutability and prevent accidental modifications. Using the `constant` keyword ensures that the value remains constant throughout the contract's lifecycle.

LQT-3
Inconsistent Naming Convention for Variables

Informational Found

Description

The variables `_CACHED_DOMAIN_SEPARATOR`, `_CACHED_CHAIN_ID`, `_HASHED_NAME`, and `_HASHED_VERSION` do not adhere to the mixed-case naming convention. Consistency in naming conventions improves code readability and maintainability.

Recommendation

Rename the variables using the mixedCase naming convention. Follow a consistent style, such as camelCase, for variable names to align with best practices and enhance the overall readability of the code.

LQT-4
Unused Library Import

Informational Found

Description

The contract imports the console library from console.sol, but the library is not utilized anywhere in the contract. Unused imports may contribute to unnecessary gas costs and can potentially confuse developers and auditors examining the code.

Recommendation

Remove the unused import statement for the console library from console.sol to enhance code clarity and reduce unnecessary gas consumption.

Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted to and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, Consultant cannot guarantee the explicit security of the audited smart contracts.