Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: ChainLink Token
Website: chain.link
Platform: Polygon
Language: Solidity
Date: April 4th, 2025

Table of Contents

Email: audit@EtherAuthority.io

THIS IS A SECURITY AUDIT REPORT DOCUMENT THAT MAY CONTAIN INFORMATION THAT IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the ChainLink Token from chain.link was audited. The audit was performed using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on April 4th, 2025.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

The LinkToken contract is an ERC20-compliant token implementation with added support for the ERC677 standard. It was the original smart contract for the Chainlink (LINK) token, written in Solidity ^0.4.16.

Key features:

- **ERC20 Compatibility**: Implements standard ERC20 functionality including transfer, approve, transferFrom, allowance, and balanceOf.
- **ERC677 Extension**: Adds transferAndCall functionality, enabling token transfers to smart contracts with an attached payload (bytes data). This allows contracts to respond immediately when they receive LINK tokens.
- **SafeMath Integration**: Uses the SafeMath library to prevent overflows/underflows in arithmetic operations.
- **Total Supply**: The contract mints a fixed total supply of 1,000,000,000 LINK tokens (10^27 base units with 18 decimals) to the deployer's address at contract creation.
- **Input Validation**: Enforces that recipients are valid (non-zero, not the contract itself) for all transfer and approval functions.

This contract served as the foundational LINK token deployed on Ethereum and other EVM-compatible chains like Polygon.

Audit scope

Name	**Code Review and Security Analysis Report for ChainLink Token Smart Contract**
Platform	**Polygon**
File	LinkToken.sol
Smart Contract Code	[0xb0897686c545045afc77cf20ec7a532e3120e0f1](0xb0897686c545045afc77cf20ec7a532e3120e0f1)
Audit Date	April 4th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: ChainLink Token • Symbol: LINK • Decimals: 18	YES, This is valid.
Key Features: • **ERC20 compliant**: Supports standard transfer, approve, transferFrom, allowance, and associated events. • **ERC677 extension**: Adds transferAndCall(address, uint256, bytes) to enable smart contract interactions during transfers. • **Fixed total supply**: 1,000,000,000 LINK tokens (with 18 decimals), minted to the deployer's address on creation. • **SafeMath**: Uses arithmetic safety checks to prevent overflows and underflows (relevant for Solidity ^0.4.16). • **Callback Support**: If the recipient is a contract, it will call onTokenTransfer(...) on the receiver after a transferAndCall.	YES, This is valid.

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **"Secured."** This token contract has no ownership control, hence it is **100% decentralized**.

We used various tools, such as Slither, Solhint, and Remix IDE. This finding is also based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed, and applicable vulnerabilities are presented in the Audit overview section. The general overview is presented in the AS-IS section, and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 2 low, and 5 very low-level issues.

Investors' Advice: A Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack a check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Code Quality

This audit scope has 1 smart contract. Smart contracts contain Libraries, Smart contracts, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in ChainLink Token are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address, and its properties/methods can be reused many times by other contracts in the ChainLink Token.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a ChainLink Token smart contract code in the form of a polygonscan web link.

As mentioned above, code parts are well commented on. And the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure that is based on well-known industry standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

LinkToken.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	LinkToken	write	Passed	No Issue
2	transferAndCall	write	Reentrancy Risk in `transferAndCall`	Refer Audit Findings
3	transfer	write	No Input Validation	Refer Audit Findings
4	approve	write	validRecipient	No Issue
5	transferFrom	write	No Input Validation	Refer Audit Findings
6	validRecipient	modifier	Passed	No Issue
7	transferFrom	write	Passed	No Issue
8	approve	write	Passed	No Issue
9	allowance	write	Passed	No Issue
10	increaseApproval	write	Passed	No Issue
11	decreaseApproval	write	Passed	No Issue
12	transferAndCall	write	Passed	No Issue
13	contractFallback	write	Passed	No Issue
14	isContract	write	`isContract()` May Fail for Contracts Under Construction	Refer Audit Findings

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss, etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens being lost
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, which can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No medium severity vulnerabilities were found.

Low

(1) No Input Validation on `transfer`/`transferFrom`:

Functions like `transfer`, `transferFrom`, and `approve` do not validate that the `_value` is greater than zero or that `_to` is not a contract that can cause unexpected reentrancy behaviors.

Resolution: Add explicit checks:

require(_value > 0, "Amount must be greater than zero");

require(_to != address(0), "Cannot send to zero address");

(2) Reentrancy Risk in `transferAndCall`:

`transferAndCall()` performs an external call (`receiver.onTokenTransfer(...)`) after transferring tokens. If the receiving contract is malicious, it could exploit this behavior.

Resolution:

- Apply [Checks-Effects-Interactions pattern]
(https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html).
- Consider adding a reentrancy guard or limiting gas in the external call.

Very Low / Informational / Best practices:

(1) Obsolete Solidity Version Lacks Compiler Protections:

The contract uses `pragma solidity ^0.4.16`, a very outdated compiler version vulnerable to numerous security issues (e.g., unprotected arithmetic, lack of visibility enforcement, outdated language features).

Resolution: Upgrade the contract to Solidity 0.8.x, which includes:

- Built-in overflow/underflow checks
- Required visibility specifiers
- Safer fallback handling

(2) Deprecated Use of `var`:

The use of `var` (e.g., `var _allowance = ...`) is outdated and not supported in Solidity 0.5.0+.

Resolution: Replace with explicit types:

uint256 _allowance = allowed[_from][msg.sender];

(3) Redundant Checks and Comments:

In functions like `div`, there are commented-out checks like:

Resolution: // assert(b > 0); // Solidity automatically throws when dividing by 0

(4) No Return Values for `onTokenTransfer()`:

The `onTokenTransfer()` function does not specify a return value, which can lead to failed expectations or interface mismatch in receiving contracts.

Resolution: Define `onTokenTransfer()` as returning `bool` and check the return value in `transferAndCall`.

(5) `isContract()` May Fail for Contracts Under Construction:

`extcodesize(_addr)` returns 0 if a contract is under construction, causing `isContract()` to return false even for valid contracts.

Resolution: Document this limitation. Consider alternatives like EIP-165 interfaces if relevant.

Centralization Risk

The LINK Token smart contract does not have any ownership control, hence it is 100% decentralized.

Therefore, there is **no** centralization risk.

Conclusion

We were given a contract code in the form of a polygonscan web link. We have used all possible tests based on the given objects as files. We observed 2 low and 5 informational issues in the smart contract, and those issues are not critical. So, **it's good to go for production**.

Since possible test cases can be unlimited for such smart contract protocols, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on standard audit procedure scope, is **"Secured".**

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 included 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. While 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.

Documenting Results:

We follow a conservative, 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 though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early, even if they are later shown not to 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 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 are an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: 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).

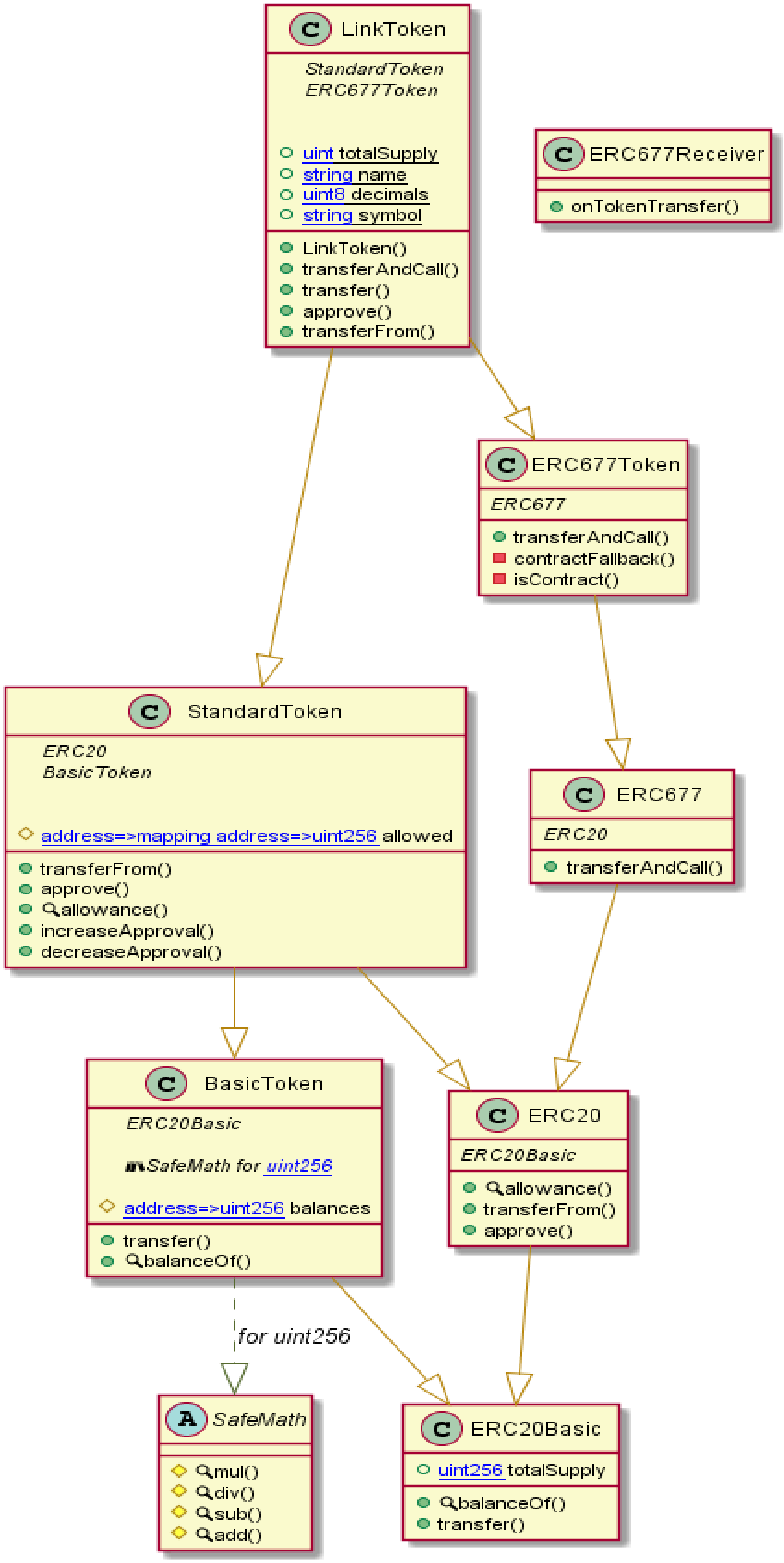
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While 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 also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

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

Appendix

Code Flow Diagram - ChainLink Token



Email: audit@EtherAuthority.io

Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

Slither Log >> LinkToken.sol

```
INFO:Detectors:
LinkToken.totalSupply (LinkToken.sol#230) shadows:
	- ERC20Basic.totalSupply (LinkToken.sol#49)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing-from-abstract-contracts
INFO:Detectors:
Version constraint ^0.4.16 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- DirtyBytesArrayToStorage
	- ABIDecodeTwoDimensionalArrayMemory
	- KeccakCaching
	- EmptyByteArrayCopy
	- DynamicArrayCleanup
	- ImplicitConstructorCallvalueCheck
	- TupleAssignmentMultiStackSlotComponents
	- MemoryArrayCreationOverflow
	- privateCanBeOverridden
	- SignedArrayStorageCopy
	- ABIEncoderV2StorageArrayWithMultiSlotElement
	- DynamicConstructorArgumentsClippedABIV2
	- UninitializedFunctionPointerInConstructor_0.4.x
	- IncorrectEventSignatureInLibraries_0.4.x
	- ExpExponentCleanup
	- NestedArrayFunctionCallDecoder
	- ZeroFunctionSelector.
It is used by:
	- ^0.4.16 (LinkToken.sol#9)
solc-0.4.26 is an outdated solc version. Use a more recent version (at least 0.8.0), if possible.
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
```


Email: audit@EtherAuthority.io

```
Parameter LinkToken.approve(address,uint256)._spender (LinkToken.sol#273) is not in
mixedCase
Parameter LinkToken.approve(address,uint256)._value (LinkToken.sol#273) is not in mixedCase
Parameter LinkToken.transferFrom(address,address,uint256)._from (LinkToken.sol#287) is not in
mixedCase
Parameter LinkToken.transferFrom(address,address,uint256)._to (LinkToken.sol#287) is not in
mixedCase
Parameter LinkToken.transferFrom(address,address,uint256)._value (LinkToken.sol#287) is not in
mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-c
onventions
INFO:Detectors:
LinkToken (LinkToken.sol#228-304) does not implement functions:
	- ERC20Basic.transfer(address,uint256) (LinkToken.sol#51)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#unimplemented-functions
INFO:Detectors:
onTokenTransfer(address,uint256,bytes) should be declared external:
	- ERC677Receiver.onTokenTransfer(address,uint256,bytes) (LinkToken.sol#72)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-decl
ared-external
INFO:Slither:LinkToken.sol analyzed (9 contracts with 93 detectors), 36 result(s) found
```


Email: audit@EtherAuthority.io

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

LinkToken.sol

```
Inline assembly:
The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.
Pos: 222:4:

Gas costs:
Gas requirement of function LinkToken.transferFrom is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 287:2:

Constant/View/Pure functions:
LinkToken.transfer(address,uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.

Similar variable names:
BasicToken.balanceOf(address) : Variables have very similar names "balance" and "balances". Note: Modifiers are currently not considered by this static analysis.
Pos: 102:11:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 37:4:
```

Solhint Linter

Solhint Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

LinkToken.sol

```
Compiler version ^0.4.16 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:8
Explicitly mark visibility in function
Pos: 3:100
Explicitly mark visibility of state
Pos: 3:116
Explicitly mark visibility in function
Pos: 3:172
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 5:221
Use double quotes for string literals
Pos: 35:232
Provide an error message for require
Pos: 5:298
```

Software analysis result:

This software reported many false positive results and some are informational issues. So, those issues can be safely ignored.

Email: audit@EtherAuthority.io

Email: audit@EtherAuthority.io