QuillAudits

AUDIT REPORT

December 2025

For

CLORE.AI

Table of Content

Executive Summary

Project Name	CloreToken
Protocol Type	ERC20 Token
Project URL	https://clore.ai/
Overview	The CloreToken smart contract implements a fixed-supply ERC-20 token using OpenZeppelin standard libraries. The contract is designed to mint the entire token supply at deployment and supports token burning via ERC20Burnable.
Audit Scope	The scope of this Audit was to analyze the CloreToken Smart Contracts for quality, security, and correctness.
Source Code link	https://etherscan.io/address/0xe60201989b8628f43dc0605f585a72bcf1f1e977#code
Contracts in Scope	CloreToken.sol
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	15th December 2025
Updated Code Received	17th December 2025
Review 2	17th December 2025

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity

Critical	0(0.0%)
High	0(0.0%)
Medium	0(0.0%)
Low	0(0.0%)
Informational	1 (100%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	1
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	0	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Lack of Explicit Zero-Address Validation for Initial Holder Input	Informational	Acknowledged

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
- Style guide violation
- Unsafe type inference
- Implicit visibility level

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 Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

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

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

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.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Informational Issues

Lack of Explicit Zero-Address Validation for Initial Holder Input

Acknowledged

Path

CloreToken.sol

Function

`constructor(address initialHolder)`

Description

The constructor mints the entire fixed token supply to the initialHolder address without explicitly validating that the provided address is non-zero. While OpenZeppelin's _mint function internally reverts when minting to the zero address, the absence of an explicit check at the constructor level reduces code clarity and shifts an important deployment invariant to an inherited dependency.

This may lead to less transparent deployment guarantees and makes the correctness of initialization dependent on inherited implementation details rather than local validation.

Impact

Low

Likelihood

Low

Recommendation

Add an explicit require(initialHolder != address(0)) check in the constructor

Functional Tests

Some of the tests performed are mentioned below:

- Contract deployment succeeds when a valid, non-zero initialHolder address is provided.
- Deployment reverts if the initial holder address is invalid due to inherited ERC20 minting constraints.
- The total token supply is minted exactly once during construction and cannot be reminted.
- The full token supply is correctly assigned to the initial holder at deployment.
- Token metadata functions (name, symbol, decimals) return expected values.
- totalSupply() accurately reflects the fixed token supply after deployment.
- balanceOf() returns correct balances before and after transfers.
- transfer() correctly moves tokens between accounts and emits Transfer events.
- transferFrom() respects allowance limits and updates allowances correctly.
- approve() sets spender allowances correctly.
- increaseAllowance() and decreaseAllowance() behave as expected and mitigate allowance race conditions.
- Allowance values decrease appropriately after transferFrom() execution.
- burn() allows token holders to destroy their own tokens.
- burnFrom() allows approved spenders to burn tokens within approved limits.
- Token burning correctly reduces both the caller's balance and the total token supply.
- Burning operations emit the expected Transfer events to the zero address.

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.

Threat Model

Contract	Function	Threats
CloreToken	constructor	Incorrect initialHolder address permanently receives full token supply. Deployment front-running not possible, but deployer key compromise is critical risk
CloreToken	_mint()	Single mint creates full supply; any re-mint path would be catastrophic.
CloreToken	transfer(address to, uint256 amount)	transfer(address to, uint256 amount) No pause/blacklist logic—cannot stop compromised transfers post-deployment.
CloreToken	_transfer()	No fee logic or hooks—safe, but cannot mitigate compromised sender behavior. Assumes balances mapping integrity; external state manipulation impossible.
CloreToken	approve(address spender, uint256 amount)	Approval overwrite enables front-running if allowance changed non-atomically.
CloreToken	transferFrom(address from, address to, uint256 amount)	Infinite allowance (uint256.max) bypasses allowance reduction intentionally. Spender misuse can drain approved balance if approval compromised.
CloreToken	_spendAllowance()	Infinite allowance pattern hides allowance usage from users.
CloreToken	burn(uint256 amount)	User can irreversibly destroy own tokens, potentially unintentionally. No recovery mechanism if burn called by compromised wallet.

Contract	Function	Threats
CloreToken	burnFrom(address account, uint256 amount)	Allowance misuse allows grief-burn attacks if approval is too large.
CloreToken	_burn()	Reduces total supply permanently, impacting token economics. No minimum supply guard—full supply can theoretically be burned.
CloreToken	allowance(), balanceOf(), totalSupply()	No direct attack surface, but used by off-chain systems for trust decisions. Incorrect off-chain assumptions may lead to integration issues.

Closing Summary

In this report, we have considered the security of CloreToken. We performed our audit according to the procedure described above.

No critical issues in CloreToken, just 1 issue of Informational severity was found and the CloreToken Team acknowledged the 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 received 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 leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

December 2025

For

CLORE.AI

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com