QuillAudits

AUDIT REPORT

June 2025

For

REDCURRY

Table of Content

Table of Content

Executive Summary

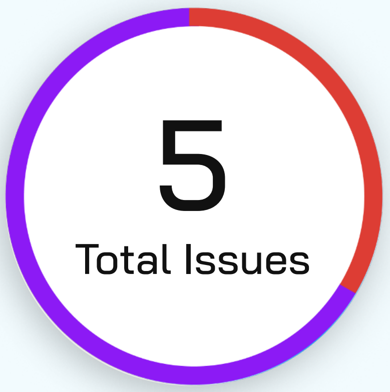
Project Name	RedCurry
Protocol Type	ERC20, Governance, Ownable
Project URL	https://redcurry.co/
Overview	The RedcurryToken is a governable ERC20 token with 8 decimals that implements automatic transaction fees (default 0.02%, capped at 25 EUR) collected by an escrow address. It features multi-level access control where owners can adjust fee parameters and toggle capabilities, while governors can mint/burn tokens through issue/redeem functions. The contract includes built-in protections with hard-coded fee limits (max 0.2% rate, 50 EUR cap) and exempts escrow-related transfers from fees. All transfers automatically deduct fees and send them to the designated escrow address, making it suitable for regulated token ecosystems requiring fee-based operations funding.
Audit Scope	The scope of this Audit was to analyze the RedCurry Smart Contracts for quality, security, and correctness.
Source Code Link	https://testnet-explorer.plume.org/address/0x2280233084F5E731E7f81C2217df8aECdF03ad18?tab=contract
Contracts in Scope	- Governable.sol - Ownable.sol - RedCurryToken.sol

Language	Solidity
Blockchain	Plume
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	29th May, 2025 - 30th May, 2025
Updated Code Received	3rd June, 2025
Review 2	4th June, 2025
Fixed In	https://github.com/Currynomics/contracts/commit/570ca8bd394dcd0746c0e48fc71d1e46ab26e113

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0%)
🟥	High	2 (40%)
🟧	Medium	0 (0%)
🟨	Low	0 (0%)
🟪	Informational	3 (60%)

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	2	0	0	2

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Ownable is not two-step meaning governance contract can be taken over and infinite mints can happen	High	Resolved
2	Users will be heavily disincentivized to use the protocol because of extremely high transfer fees	High	Resolved
3	Should emit events accurately describing state changing functions	Informational	Resolved
4	Should cache the generator variable in for loops and use cached values	Informational	Resolved
5	Unlocked pragma compiler	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 statistical 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.

Severity Matrix

Likelihood \ Impact	🟥 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.

Types of Issues

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

High Severity Issues

Ownable is not two-step meaning governance contract can be taken over and infinite mints can happen

Resolved

Path

src/product/Ownable.sol

Description

The protocol's Ownable contract is designed to replicate OpenZeppelin's Ownable2Step contract (link), where ownership transfer is first initiated by the current owner and then claimed by the intending owner before the ownership transfer is finalized. However, the current implementation immediately transfers ownership to the pendingOwner, bypassing the intended two-step process. This creates a significant security risk as it allows ownership to be transferred without verification or confirmation from the new owner. If an attacker compromises the current owner's private key, they could immediately transfer ownership to an address they control, enabling unauthorized governance actions such as setting new governors or minting an infinite amount of tokens. This could lead to token insolvency.

Impact: HIGH

Complete governance takeover leading to unlimited token minting and potential insolvency.

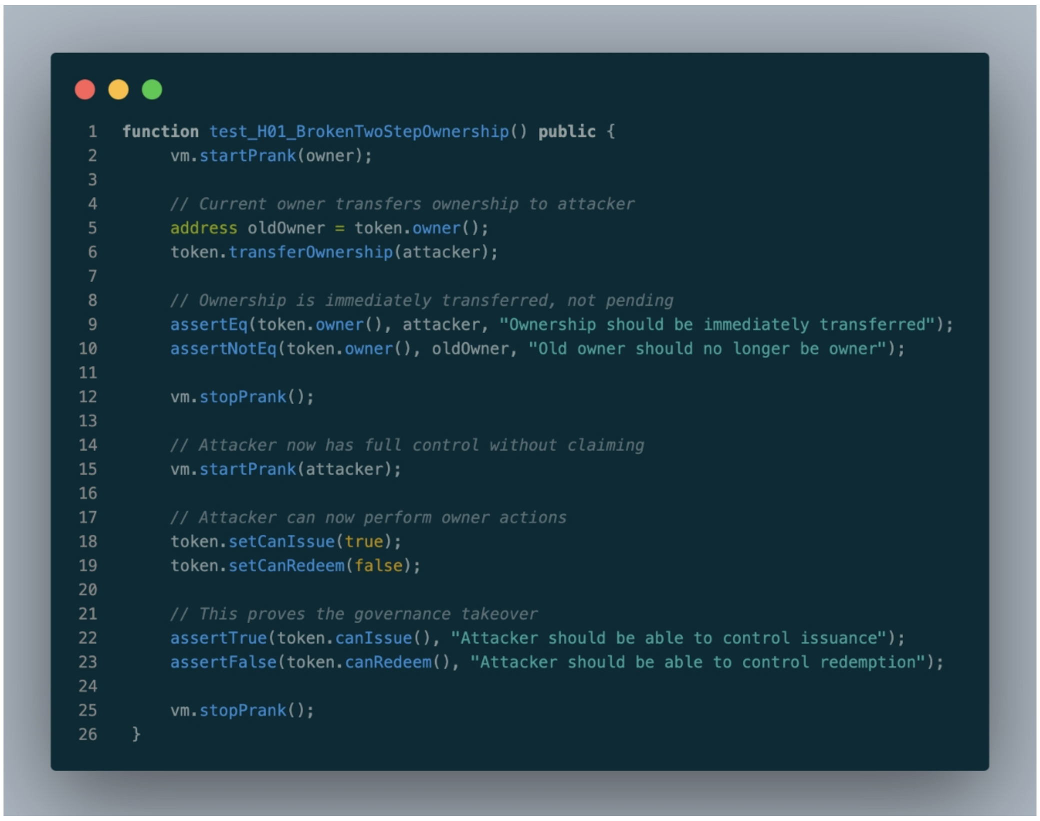
Likelihood: MEDIUM

One-step ownership transfer is guaranteed to happen every time the function is called. Only if an attacker got control of/compromise the current owner's private key would they be able to make an address they control the new owner immediately.

Recommendation

1. Use industry standard code, which although are not a 100% guarantee, they have been battle-tested by more persons and proven to be resistant to attacks like this.
2. If there's a specific need to use this contract, remove the internal `_transferOwnership()` call in `transferOwnership()`.

POC



```
function test_H01_BrokenTwoStepOwnership() public {
     vm.startPrank(owner);

     // Current owner transfers ownership to attacker
     address oldOwner = token.owner();
     token.transferOwnership(attacker);

     // Ownership is immediately transferred, not pending
     assertEq(token.owner(), attacker, "Ownership should be immediately transferred");
     assertNotEq(token.owner(), oldOwner, "Old owner should no longer be owner");

     vm.stopPrank();

     // Attacker now has full control without claiming
     vm.startPrank(attacker);

     // Attacker can now perform owner actions
     token.setCanIssue(true);
     token.setCanRedeem(false);

     // This proves the governance takeover
     assertTrue(token.canIssue(), "Attacker should be able to control issuance");
     assertFalse(token.canRedeem(), "Attacker should be able to control redemption");

     vm.stopPrank();
 }
```

Specific fixed in commit

https://github.com/Currynomics/contracts/commit/570ca8bd394dcd0746c0e48fc71d1e46ab26e113

Users will be heavily disincentivized to use the protocol because of extremely high transfer fees

Resolved

Path

src/product/RedCurryToken.sol

Function Name

transfer(), transferFrom()

Description

The current values for maxTxnFee and TXN_FEE_MAX_LIMIT are hardcoded to be 18 decimal tokens which is the standard, but the RedCurryToken is classified as a "weird-erc20" token as it has an irregular number of decimals and charges a fee on transfer.

With these values set to 18 decimals, they are 10^10 times the order of magnitude of the token itself and will not perform the functions intended because ideally none of the transfers performed will be that high, making the checks performed with them in _calculateFee(). redundant.

This means the conditional block in _calculateFee() for fee > maxTxnFee would almost never be hit because of the extremely high fees set. Users will not have the security of a fee cap imposed on their transactions as expected.

Impact: HIGH

If users are discouraged from using the protocol because of high fees on large transfers, it reduces the fees generated and in turn the protocol's revenue.

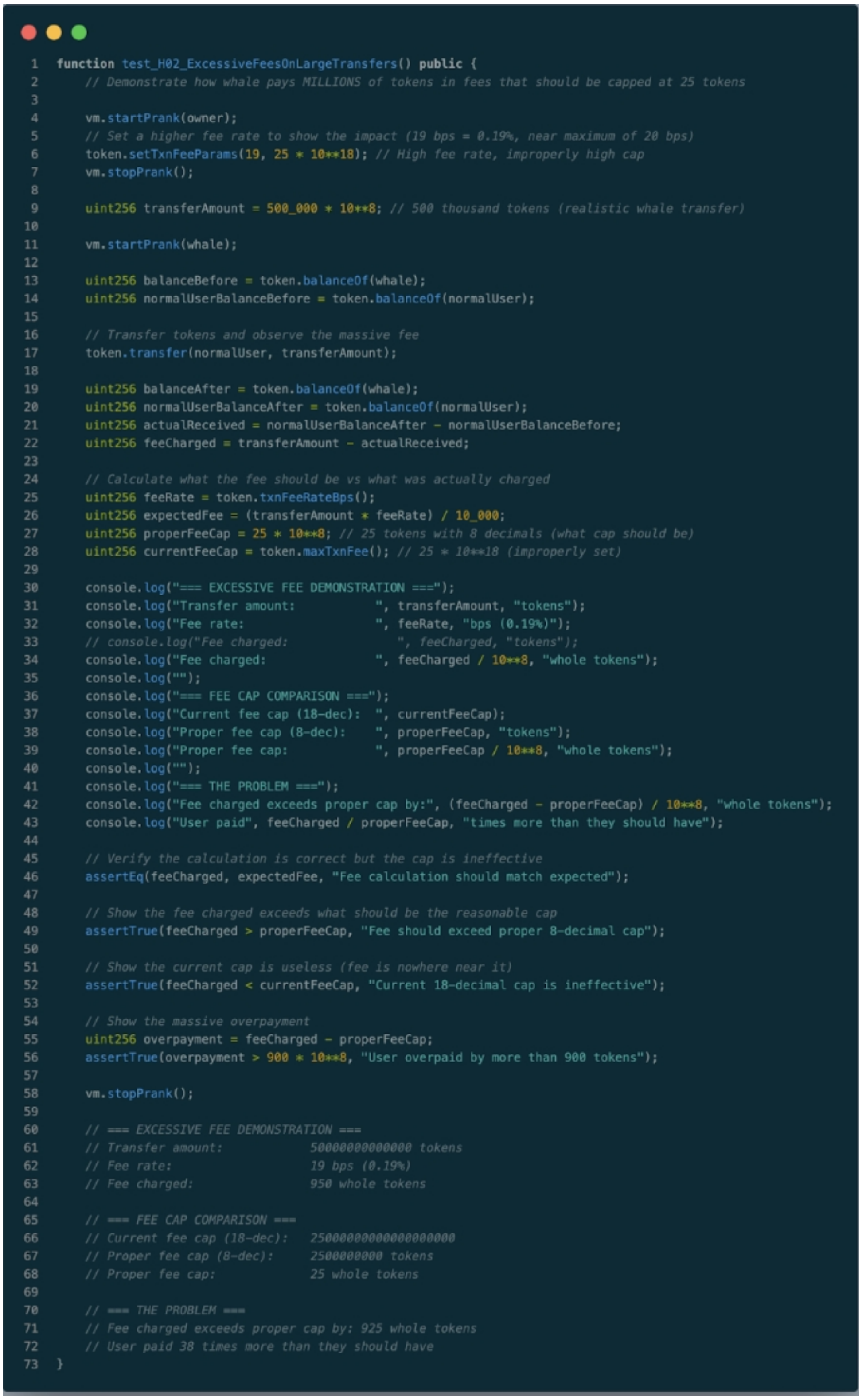
Likelihood: HIGH

This affects all transfers but the effect of uncapped fees is seen on transactions upward of ~13,160e18 tokens.

Recommendation

1. Ensure proper decimal handling by scaling the calculations up or down, or conservatively have the fee information tally with the number of decimals as the token.

POC



```solidity
function test_H02_ExcessiveFeesOnLargeTransfers() public {
    // Demonstrate how whale pays MILLIONS of tokens in fees that should be capped at 25 tokens

    vm.startPrank(owner);
    // Set a higher fee rate to show the impact (19 bps = 0.19%, near maximum of 20 bps)
    token.setTxnFeeParams(19, 25 * 10**18); // High fee rate, improperly high cap
    vm.stopPrank();

    uint256 transferAmount = 500_000 * 10**8; // 500 thousand tokens (realistic whale transfer)

    vm.startPrank(whale);

    uint256 balanceBefore = token.balanceOf(whale);
    uint256 normalUserBalanceBefore = token.balanceOf(normalUser);

    // Transfer tokens and observe the massive fee
    token.transfer(normalUser, transferAmount);

    uint256 balanceAfter = token.balanceOf(whale);
    uint256 normalUserBalanceAfter = token.balanceOf(normalUser);
    uint256 actualReceived = normalUserBalanceAfter - normalUserBalanceBefore;
    uint256 feeCharged = transferAmount - actualReceived;

    // Calculate what the fee should be vs what was actually charged
    uint256 feeRate = token.txnFeeRateBps();
    uint256 expectedFee = (transferAmount * feeRate) / 10_000;
    uint256 properFeeCap = 25 * 10**8; // 25 tokens with 8 decimals (what cap should be)
    uint256 currentFeeCap = token.maxTxnFee(); // 25 * 10**18 (improperly set)

    console.log("=== EXCESSIVE FEE DEMONSTRATION ===");
    console.log("Transfer amount:           ", transferAmount, "tokens");
    console.log("Fee rate:                  ", feeRate, "bps (0.19%)");
    // console.log("Fee charged:               ", feeCharged, "tokens");
    console.log("Fee charged:               ", feeCharged / 10**8, "whole tokens");
    console.log("");
    console.log("=== FEE CAP COMPARISON ===");
    console.log("Current fee cap (18-dec):  ", currentFeeCap);
    console.log("Proper fee cap (8-dec):    ", properFeeCap, "tokens");
    console.log("Proper fee cap:            ", properFeeCap / 10**8, "whole tokens");
    console.log("");
    console.log("=== THE PROBLEM ===");
    console.log("Fee charged exceeds proper cap by:", (feeCharged - properFeeCap) / 10**8, "whole tokens");
    console.log("User paid", feeCharged / properFeeCap, "times more than they should have");

    // Verify the calculation is correct but the cap is ineffective
    assertEq(feeCharged, expectedFee, "Fee calculation should match expected");

    // Show the fee charged exceeds what should be the reasonable cap
    assertTrue(feeCharged > properFeeCap, "Fee should exceed proper 8-decimal cap");

    // Show the current cap is useless (fee is nowhere near it)
    assertTrue(feeCharged < currentFeeCap, "Current 18-decimal cap is ineffective");

    // Show the massive overpayment
    uint256 overpayment = feeCharged - properFeeCap;
    assertTrue(overpayment > 900 * 10**8, "User overpaid by more than 900 tokens");

    vm.stopPrank();

    // === EXCESSIVE FEE DEMONSTRATION ===
    // Transfer amount:            50000000000000 tokens
    // Fee rate:                   19 bps (0.19%)
    // Fee charged:                950 whole tokens

    // === FEE CAP COMPARISON ===
    // Current fee cap (18-dec):   25000000000000000000
    // Proper fee cap (8-dec):     2500000000 tokens
    // Proper fee cap:             25 whole tokens

    // === THE PROBLEM ===
    // Fee charged exceeds proper cap by: 925 whole tokens
    // User paid 38 times more than they should have
}
```

Specific fixed in commit

https://github.com/Currynomics/contracts/commit/570ca8bd394dcd0746c0e48fc71d1e46ab26e113

Informational Severity Issues

Should emit events accurately describing state changing functions

Resolved

Path

src/product/RedCurryToken.sol

Function

setCanIssue, setCanRedeem

Description

The contract does not emit events for some of the functions that update state. While setEscrow emits the EscrowTransferred function, the setCanIssue and setCanRedeem functions which determine whether the contract can mint and burn tokens respectively do not have any events signifying this.

Should cache the generator variable in for loops and use cached values

Resolved

Path

src/product/Governable.sol

Function

_transferGovernorship()

Description

It is much cheaper to cache the value of _governors.length in a local variable to be reused in the for loop than to read the value from storage in every loop iteration. It is comparatively more expensive to do this especially when the _governors array is large.

Unlocked pragma compiler

Acknowledged

Path

src/product/**.sol

Description

All core contracts use unlocked pragma directives (^0.8.22) instead of locked versions (0.8.22), which can lead to deployment inconsistencies, behavioral differences, and potential security vulnerabilities due to compiler version variations.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Set new governors after deployment
- ✓ Should allow only owner to call onlyOwner protected functions
- ✓ Should mint only when _canIssue is true
- ✓ Should burn only when _canRedeem is true
- ✓ Should reset escrow address
- ✗ Should cap transfer fees at maxTxnFee
- ✗ Should make pendingOwner claim ownership when transferred

Threat Model

Contract	Function	Threats
RedCurryToken.sol	• transfer() and transferFrom()	• Zero address checks • Token mint caps • Token approval
	• change()	• Silent underflow/overflow • Mint when _canIssue is false • Burn when _canRedeem is false
	• setEscrow()	• Zero address checks
Ownable.sol	• transferOwnership()	• Unintended transfer to zero address • Protection by onlyOwner modifier
	• claimOwnership()	• Should allow only pendingOwner to claim ownership
Governable.sol	• transferGovernorship()	• Zero address check for new governors • OOG errors in for loop

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.

Closing Summary

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

Issues of High and Informational/Gas Optimization severity were found.
RedCurry team acknowledged one and resolved rest

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 secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

June 2025

For

REDCURRY

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com