QuillAudits

AUDIT REPORT

February, 2025

For

reactive

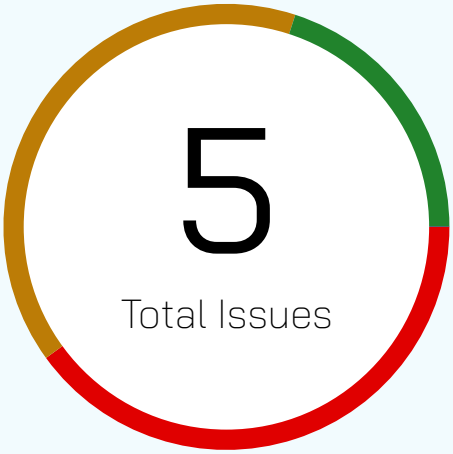
Table of Content

Executive Summary

Project name	Reactive
Overview	Parallelized interoperability execution layer for EVM ecosystems.
Method	Manual Analysis, Functional Testing, Automated Testing
Blockchain	Ethereum
Audit Scope	The scope of this Audit was to analyze the Reactive Smart Contracts for quality, security, and correctness. https://github.com/Reactive-Network/react-bridge/tree/dec24-prerelease/src Branch: dec24-prerelease https://github.com/Reactive-Network/react-bridge
Contracts In Scope	1. src/AbstractBridgehead.sol 2. src/ReactiveBridge.sol 3. src/AbstractFeeCalculator.sol 4. src/BridgeLib.sol 5. src/Bridge.sol 6. src/AbstractDispenser.sol
Fixed In	Branch: Main 7f906e9bd8750ca7b2927afca2c198065146b387
Project URL	https://reactive.network/
Language	Solidity
Review 1	13-02-2025
Updated code received	20th February 2025
Review 2	20th February 2025

Commit hash	Branch: Main d2c47137cec85081ba56720c1e34daa0a81a99c9
Mainnet Address	Ethereum : https://etherscan.io/address/0x42458259d5c85fB2bf117f197f1Fef8C3b7dCBfe#code https://etherscan.io/address/0x6fBb0C7A0ec62007013748e47823C239Dd48BfEf#code BSC : https://bscscan.com/address/0x577432505892F7B18a26166247a7456B814E2f68#code

Number of Issues per Severity



High	2 (40.00%)
Medium	2 (40.00%)
Low	1 (20.00%)
Informational	0 (0.00%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**2**	**1**	0	0
Acknowledged	0	**1**	**1**	0
Partially Resolved	0	0	0	0

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

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] 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 four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

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

Incorrect Event Subscription Leading to Unprocessed Failed Transactions

Resolved

Path

ReactiveBridge.sol

Function

Constructor

Description

In the current bridge implementation, event subscription is critical for ensuring that the bridge can listen for and process specific event topics. The contract subscribes to various event topics such as SEND_MESSAGE_TOPIC, CANCEL_MESSAGE_TOPIC, RETRY_TOPIC, DELIVERED_TOPIC, FAILED_TOPIC, and others to ensure that all relevant transactions are properly handled.
However, an issue has been identified in the subscription logic where the REJECTION_TOPIC has been mistakenly subscribed to twice instead of subscribing to the FAILED_TOPIC. The affected code snippet is as follows:
As a result of this incorrect subscription, events emitted with the FAILED_TOPIC will not be captured by the bridge. This leads to a failure in handling failed transactions, as the necessary logic to process and recover funds from failed transactions will not be executed. Consequently, affected users may permanently lose their funds, as the system fails to recognize and act upon failed transaction events.

Recommendation

To resolve this issue, the contract should be modified to ensure that the FAILED_TOPIC is correctly subscribed to, rather than subscribing to REJECTION_TOPIC twice.

Denial of Service via Incorrect Message Status Update in Rejection Handling

Resolved

Path

AbstractBridgehead.sol

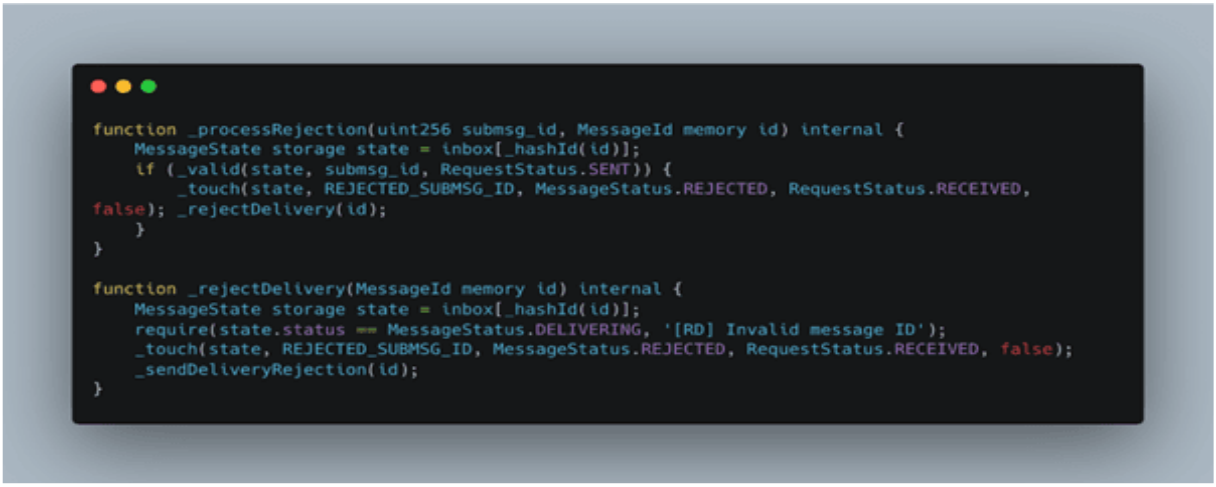
Function

_processRejection, _rejectDelivery

Description

The _processRejection function is responsible for handling message rejections on the destination chain. If the condition holds, the function updates the message status to MessageStatus.REJECTED and subsequently calls _rejectDelivery(id).
However, _rejectDelivery(id) checks whether its status is MessageStatus.DELIVERING. If the condition is met, the function updates the message status to MessageStatus.REJECTED and calls _sendDeliveryRejection(id).

The issue occurs because _processRejection modifies the message status to MessageStatus.REJECTED before _rejectDelivery(id) executes. Since _rejectDelivery explicitly requires the message status to be MessageStatus.DELIVERING, the require condition fails, causing the transaction to revert. This results in a denial of service (DoS), preventing the rejection from being processed successfully.



```
function _processRejection(uint256 submsg_id, MessageId memory id) internal {
    MessageState storage state = inbox[_hashId(id)];
    if (_valid(state, submsg_id, RequestStatus.SENT)) {
        _touch(state, REJECTED_SUBMSG_ID, MessageStatus.REJECTED, RequestStatus.RECEIVED,
false); _rejectDelivery(id);
    }
}

function _rejectDelivery(MessageId memory id) internal {
    MessageState storage state = inbox[_hashId(id)];
    require(state.status == MessageStatus.DELIVERING, '[RD] Invalid message ID');
    _touch(state, REJECTED_SUBMSG_ID, MessageStatus.REJECTED, RequestStatus.RECEIVED, false);
    _sendDeliveryRejection(id);
}
```

Recommendation

Modify _processRejection to update message status to MessageStatus.DELIVERING before calling _rejectDelivery(id). This ensures that _rejectDelivery(id) does not revert due to an invalid state.

Medium Severity Issues

Lack of Bounds on Fees in AbstractFeeCalculator

Resolved

Path

AbstractBridgehead.sol

Function

/

Description

The AbstractFeeCalculator contract implements a fee calculation mechanism using a fixed fee (fixed_fee) and a percentage-based fee (perc_fee). The fees are initialized via the constructor and used in the _computeFee function to determine the total fee applied to a given amount. Fees will be collected on the destination bridge once the delivery occurs.
However, there are no constraints on the values of fixed_fee and perc_fee, which introduces several risks. If fixed_fee or perc_fee is set too high, users may be charged excessive fees upon message delivery, potentially making transactions infeasible.

Recommendation

Enforce reasonable upper bounds on fixed_fee and perc_fee

Lack of Refund Mechanism on Failed Transfers in Bridge to Bridge Interaction Scenario

Acknowledged

Path

Bridge.sol

Description

In the current bridge implementation, once a transfer message is confirmed and successfully relayed from the source bridge to the destination bridge, the user's funds are effectively deducted from the source chain. However, if the transaction fails on the destination bridge due to any factors, the bridge does not provide a mechanism to refund the user's assets.
This issue specifically arises in BRIDGE <-> BRIDGE interactions, where one bridge communicates directly with another bridge without involving the ReactiveBridge (RNBRIDGE). Since BRIDGE <-> RNBRIDGE interactions are the intended, the lack of a refund mechanism is only a concern when bridges interact directly with each other. In such cases, users may face an irreversible loss of funds, exposing them to financial risk.

Recommendation

Implement a refund mechanism that ensures users receive their assets back in case of a failure on the destination bridge.

Reactive Team's Comment: Bridge and ReactiveBridge always operate as a pair, no other combination is intended or envisioned

Low Severity Issues

Lack of Test Coverage in the Bridge Contract

Acknowledged

Path

/

Function

/

Description

The bridge contract currently lacks a comprehensive testing suite, which poses significant risks to its reliability, security, and functionality. Without unit and integration tests, there is no systematic way to verify that the contract behaves as intended across different scenarios, including edge cases and failure conditions. This absence of testing increases the likelihood of undetected vulnerabilities, incorrect logic implementation, and potential financial losses due to undiagnosed bugs.

Recommendation

Cover individual contract functions to validate their expected behavior under normal and edge-case scenarios.

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 Reactive. We performed our audit according to the procedure described above.

Some issues of High, low and medium severity were found.

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 submitted smart contract source code, including its compilation, deployment, and intended functionality.

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

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

Follow Our Journey

AUDIT REPORT

February, 2025

For

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com