PROTOFIRE

January 2026

Smart contract audit
rain.string

Content

Project description

This repository contains a Solidity library for string parsing and mutation utilities used in Rainlang. It provides character mask constants for ASCII classification, low-level character parsing functions that operate on memory regions using bitmask operations, and decimal string-to-integer conversion with overflow protection. The library includes in-place string mutation functions. It is designed as a minimal, gas-efficient foundation for parsing operations, using assembly for performance-critical paths and supporting memory-safe operations.

Executive summary

Type	Library
Languages	Solidity
Methods	Architecture Review, Manual Review, Unit Testing, Functional Testing, Automated Review
Documentation	README.md
Repositories	https://github.com/rainlanguage/rain.string

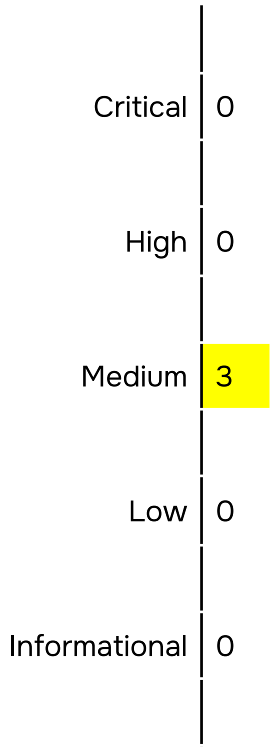
Reviews

Date	**Repository**	**Commit**
13/01/26	rain.string	0b1ca08aed6d9c06b83fe127a7d20ee7002ead28
26/01/26	rain.string	f7ff702af4565fc962fa6ac98a787959df2c7388

Scope

Contracts
src/error/ErrParse.sol
src/lib/mut/LibConformString.sol
src/lib/parse/LibParseChar.sol
src/lib/parse/LibParseCMask.sol
src/lib/parse/LibParseDecimal.sol

Technical analysis and findings



Security findings

**** Critical

No critical severity issue found.

*** High

No high severity issue found.

** Medium

M01 - Infinite Loop in LibConformString When Mask Is Zero

The functions *conformStringToMask()* and *charFromMask()* inside *LibConformString.sol* utilize a while loop to find a character that matches a specific mask. The loop condition is while *(1 << char & mask == 0)*.

If the mask parameter is provided as 0 (an empty set), the bitwise AND operation (& 0) will always result in 0. Consequently, the condition 0 == 0 evaluates to true indefinitely. Because the mask is empty, no randomly generated character will ever satisfy the condition to break the loop, causing the contract to consume all available gas and revert.

Path: src/lib/mut/LibConformString.sol

Recommendation: Add a validation check at the beginning of the *conformStringToMask()* and *charFromMask()* functions to ensure that the *mask* is not zero. If an empty *mask* is passed, the function should revert with a descriptive error, as it is logically impossible to conform a string to an empty character set.

Found in: 0b1ca08a

Status: Fixed at 3641d652 and 18b9c91b

M02 - Infinite Loop due to Underflow in LibParseDecimal Reverse Iteration

The *unsafeDecimalStringToInt()* function in LibParseDecimal.sol iterates over a memory region backwards (from end to start) to parse a decimal string. The logic is encapsulated within an unchecked block to save gas.

The loops use the condition *cursor >= start*. If start is 0, this condition effectively becomes cursor >= 0, which is always true for any uint256.

When the cursor reaches 0 (the start of the memory), the decrement operation cursor-- inside the unchecked block causes the cursor to underflow to *type(uint256).max*.

1. First Loop: The first loop (*while (cursor >= start && exponent < 77)*) is bounded by exponent, so it might eventually terminate, but cursor will arguably have corrupted the logic by wrapping around.
2. Second Loop: The second loop (*while (cursor >= start)*), utilised for checking leading zeros/remaining characters, has no secondary exit condition. If start is 0, this loop becomes a true infinite loop, decrementing from *type(uint256).max* downwards until the transaction runs out of gas.

Path: src/lib/parse/LibParseDecimal.sol

Recommendation: The loop logic must be adjusted to handle reverse iteration safely when the start value is 0. A common pattern to avoid underflow in unchecked reverse loops is to check for equality starting at the end of the loop body, before the decrement.

Found in: 0b1ca08a

Status: Fixed at 40b602de

M03 - Infinite Loop in LibConformString Due to Disjoint Mask and Character Range

The functions *conformStringToMask()* and *charFromMask()* in LibConformString.sol employ a while loop to randomly generate characters until one matching the provided mask is found. The character generation logic restricts candidates to the range [0, max - 1] using mod(byte(0, seed), max).

A logical error exists where the function does not verify that the provided mask contains at least one valid character within the constrained range [0, max - 1]. If the mask and the possible character range are disjoint (share no common bits), the condition (1 << char) & mask == 0 will effectively always be true. This causes the loop to run indefinitely, consuming all available gas and reverting the transaction. This vulnerability is a generalization of finding M01 (Zero Mask) but can be triggered with a non-zero mask if the max parameter is set restrictively.

Path: src/lib/mut/LibConformString.sol

Recommendation: Update the validation logic to ensure that the mask has a non-empty intersection with the set of all possible characters generated by max. This can be achieved by checking that the mask has at least one bit set below the max bit index.

Found in: 0b1ca08a

Status: Fixed at c09ae695

* Low

No low severity issue found.

Informational

No Informational severity issue found.

Approach and methodology

To establish a uniform evaluation, we define the following terminology in accordance with the OWASP Risk Rating
Methodology:

Likelihood
Indicates the probability of a specific vulnerability being discovered and exploited in real-world
scenarios

Impact
Measures the technical loss and business repercussions resulting from a successful attack

Severity
Reflects the comprehensive magnitude of the risk, combining both the probability of occurrence
(likelihood) and the extent of potential consequences (impact)

Likelihood and impact are divided into three levels: High H, Medium M, and Low L. The severity of a risk is a blend of these two factors, leading to its classification into one of four tiers: Critical, High, Medium, or Low.

When we identify an issue, our approach may include deploying contracts on our private testnet for validation through testing. Where necessary, we might also create a Proof of Concept PoC to demonstrate potential exploitability. In particular, we perform the audit according to the following procedure:

Advanced DeFi Scrutiny
We further review business logics, examine system operations, and place DeFi-related aspects
under scrutiny to uncover possible pitfalls and/or bugs

Semantic Consistency Checks
We then manually check the logic of implemented smart contracts and compare with the
description in the white paper.

Security Analysis
The process begins with a comprehensive examination of the system to gain a deep
understanding of its internal mechanisms, identifying any irregularities and potential weak spots.