PROTOFIRE

January 2026

Smart contract audit

rain.solmem

Content

Project description **3**
Executive summary **3**
Reviews **3**
Technical analysis and findings **5**
Security findings **6**
**** Critical 6
*** High 6
** Medium 6
M01 - Missing Validation for n == 0 Leads to Infinite Loop and Out-of-Gas 6
* Low 6
L01 - Missing Validation Causes Ambiguous Reverts and Unexpected Failure Modes 6
Informational 7
I01 - Memory Corruption Risk in unsafeList() 7
Approach and methodology **8**

Project description

This repository contains a Solidity library for low-level memory manipulation operations. It provides pointer-based utilities for direct memory access, efficient array and matrix operations for uint256 and bytes32 types, and stack data structure operations using assembly to bypass Solidity's runtime bounds checks. The library includes memory copying functions using the mcopy opcode, array construction from literals, in-place truncation and extension operations, and sentinel-terminated list handling for Rainlang. It is designed as a minimal, gas-optimized foundation for performance-critical code paths that require fine-grained control over memory layout and allocation.

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.solmem

Reviews

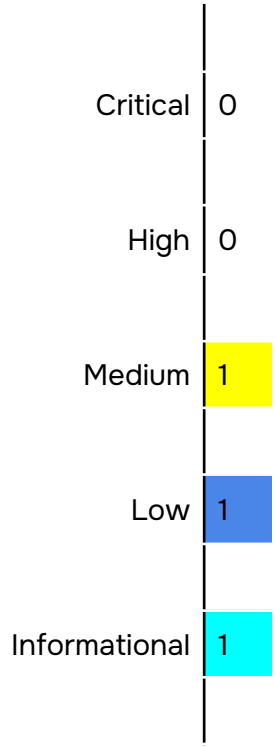
Date	Repository	Commit
13/01/26	rain.solmem	228b35c6725877e7fbcd2432b4c692357f16f510
26/01/26	rain.solmem	26bce6197383f193e35326bab4d4424cf6eafde7

Scope

Contracts
src/lib/LibBytes.sol
src/lib/LibBytes32Array.sol
src/lib/LibBytes32Matrix.sol

src/lib/LibMemCpy.sol
src/lib/LibPointer.sol
src/lib/LibStackPointer.sol
src/lib/LibStackSentinel.sol
src/lib/LibUint256Array.sol
src/lib/LibUint256Matrix.sol
src/error/ErrBytes.sol
src/error/ErrUint256Array.sol
src/error/ErrStackPointer.sol

Technical analysis and findings



Security findings

**** Critical

No critical severity issue found.

*** High

No high severity issue found.

** Medium

M01 - Missing Validation for n == 0 Leads to Infinite Loop and Out-of-Gas

The function *consumeSentinelTuples()* does not validate that the parameter *n* is greater than zero. When *n == 0*, the computed tuple size (*size = n * 0x20*) becomes zero. As a result, the loop cursor is never updated (*cursor := sub(cursor, size)*), causing an infinite loop in the first assembly for loop.
Calling this function with *n == 0* will always result in an out-of-gas revert, effectively making the function unusable for that input. This can cause unexpected hard reverts in higher-level logic that relies on dynamic tuple consumption. If *n* is user-controlled or derived from external input, this can be exploited to trigger consistent denial-of-service behavior.

Path: src/lib/LibStackSentinel.sol

Recommendation: Add an explicit validation to ensure that *n > 0* before entering the assembly logic. Reverting early with a clear error will prevent infinite loops and make failures easier to diagnose.

Found in: 228b35c6

Status: Fixed at 43222a6

* Low

L01 - Missing Validation Causes Ambiguous Reverts and Unexpected Failure Modes

The function *consumeSentinelTuples()* assumes that *upper* is strictly greater than *lower*, but does not explicitly validate this condition. If *upper <= lower*, the sentinel search loop is skipped entirely, leaving *sentinelPointer* unset. This causes the function to revert with

MissingSentinel, even though the failure is due to an invalid or empty stack range rather than an absent sentinel.
This behavior makes it difficult for callers to distinguish between a genuinely missing sentinel and an invalid stack range. Higher-level logic may incorrectly interpret the revert reason, leading to confusing error handling or masking integration bugs. In complex execution flows, this can significantly complicate debugging and increase the risk of improper stack management.

Path: src/lib/LibStackSentinel.sol

Recommendation: Add an explicit check to ensure that *upper > lower* before searching for the sentinel. Reverting early with a dedicated error for invalid stack bounds would improve correctness and debuggability without affecting the intended unsafe semantics of the function.

Found in: 228b35c6

Status: Fixed at 96955a1

Informational

I01 - Memory Corruption Risk in unsafeList()

The calculation `sub(pointer, add(0x20, mul(length, 0x20)))` on line 132 can underflow if `pointer` is less than `0x20 + length * 0x20`. The function then overwrites memory at the calculated address without any bounds validation, potentially corrupting memory outside the intended stack region.

Path: src/lib/LibStackPointer.sol

Recommendation: Consider documenting such behavior in the following way:
*'The caller MUST ensure that `pointer >= (length + 1) * 32'*

Found in: 228b35c6

Status: Fixed at 9b0de96

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.