PROTOFIRE

February 2026

Smart contract audit
rain.sol.codegen

Content

Project description

This repository, rain.sol.codegen, is a native Solidity library and tooling suite designed to generate Solidity code. Its primary purpose is to build valid Solidity files—such as those containing constant caches for function pointer tables—to ensure maximum runtime gas efficiency for the Rain interpreter. By "pre-compiling" these expensive lookups into hardcoded constants, the project optimizes gas costs for downstream contracts. It includes interfaces for interpreters and parsers to implement compatible code generation logic, along with scripts to manage the generation lifecycle.

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.sol.codegen

Reviews

Date	**Repository**	**Commit**
30/01/26	rain.sol.codegen	35ec46f26376309ff5bdf946fadb2d447e09ebd8
02/02/26	rain.sol.codegen	701a6b2b1c7c41c3fe5219019383c6fab97f8232

Scope

Contracts
src/generated/CodeGennable.pointers.sol
src/interface/IIntegrityToolingV1.sol
src/interface/IOpcodeToolingV1.sol
src/interface/IParserToolingV1.sol

src/interface/ISubParserToolingV1.sol
src/lib/LibCodeGen.sol
src/lib/LibFs.sol
src/lib/LibHexString.sol

Technical analysis and findings

Critical	0
High	0
Medium	0
Low	1
Informational	0

Security findings

**** Critical

No critical severity issue found.

*** High

No high severity issue found.

** Medium

No medium severity issue found.

* Low

L01 - Incorrect Line Length Calculation in String Formatting Functions

The *bytesConstantString()* and *uint8ConstantString()* functions in LibCodeGen.sol contain incorrect arithmetic when calculating whether a generated line exceeds the *MAX_LINE_LENGTH* (120 characters). The formulas overcount the actual line length, causing the functions to insert unnecessary line breaks.

In bytesConstantString (line 225):
17 + bytes(name).length + 6 + bytes(hexData).length + 2 > MAX_LINE_LENGTH

- Formula calculates: 25 + name.length + hexData.length
- Actual line length: 24 + name.length + hexData.length
- Overcounts by 1 character

In uint8ConstantString (line 250):
17 + bytes(name).length + 6 + 3 + 2 > MAX_LINE_LENGTH

- Formula calculates: 28 + name.length
- Actual line length: 22 + name.length (maximum)
- Overcounts by 6 characters

The root cause is that "bytes constant " and "uint8 constant " are each 15 characters, but the formula uses 17. Additionally, the trailing ";" in uint8ConstantString is 1 character, not 2.

Path: src/lib/LibCodeGen.sol

Recommendation: Update the formulas to reflect the actual string lengths.

Found in: 35ec46f2

Status: Fixed at 701a6b2b

Informational

No informational severity issue found.

General Risks

- Circular Build Dependency: The architecture inherently creates a circular dependency: the consumer contract relies on the generated pointers file, but the pointers file is derived from the contract's compilation. This requires a specific, iterative build process (generating until a stable state is reached) to ensure the source code and generated artifacts are in sync.

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.