PROTOFIRE

February 2026

Smart contract audit
rain.extrospection

Content

Project description

The rain.extrospection repository implements a set of interfaces, libraries, and a concrete contract designed to expose low-level bytecode analysis logic to off-chain tooling. Unlike typical on-chain logic, this project focuses on "extrospection"—analyzing the bytecode of other contracts directly.

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.extrospection

Reviews

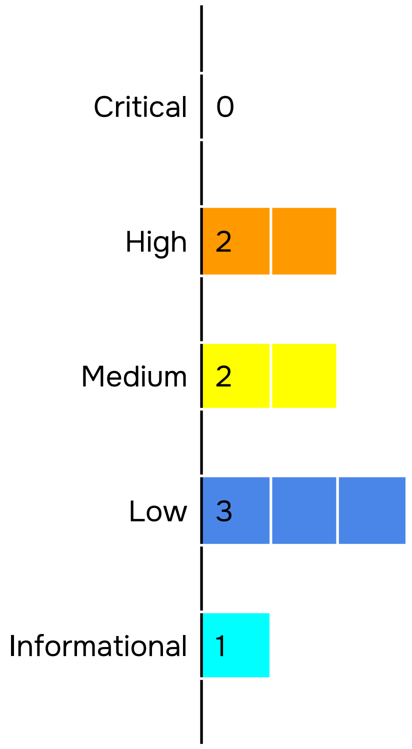
Date	**Repository**	**Commit**
30/01/26	rain.extrospection	60f35a0fc7e38fe6879c638ee25b46e88bfc9892
05/02/26	rain.extrospection	4c0aed6fa250d22c42b62dedaba20429a30609a9
10/02/26	rain.extrospection	f2c4f5add8488a07102bcd648e4109e021f68683

Scope

Contracts
src/concrete/Extrospection.sol
src/interface/deprecated/IExtrospectBytecodeV1.sol
src/interface/IExtrospectBytecodeV2.sol
src/interface/IExtrospectERC1167ProxyV1.sol
src/interface/IExtrospectInterpreterV1.sol

src/lib/EVMOpcodes.sol
src/lib/LibExtrospectBytecode.sol
src/lib/LibExtrospectERC1167Proxy.sol

Technical analysis and findings



Security findings

**** Critical

No critical severity issue found.

*** High

H01 - Incorrect Instruction Boundary Parsing in Reachability Scanner

The *scanEVMOpcodesReachableInBytecode()* function in LibExtrospectBytecode.sol contains a logic error when scanning "unreachable" (dead) code regions.

According to the Ethereum Yellow Paper (Section 9.4.3, Jump Destination Validity), a valid jump destination (JUMPDEST) must strictly appear as an opcode. It cannot be part of the immediate data of a PUSH instruction. During execution, the EVM program counter advances past the data bytes of a PUSH instruction, ensuring they are never executed.

However, the current scanner implementation fails to respect these boundaries when in a "halted" state. When the scanner encounters a halting opcode (e.g. STOP, RETURN), it pauses execution analysis and linearly scans for a JUMPDEST (0x5B) byte to resume. In this state, it ignores PUSH opcodes. Consequently, if the immediate data of a PUSH instruction contains the byte 0x5B, the scanner incorrectly identifies it as a valid JUMPDEST.

This causes the scanner to resume execution analysis mid-instruction, misinterpreting the remaining PUSH data and subsequent bytes as executable opcodes. This leads to an incorrect reachability map, potentially flagging safe contracts as unsafe or failing to detect dangerous opcodes in what is actually reachable code.

Path: src/lib/LibExtrospectBytecode.sol

Recommendation: Update the scanning logic within the halted state (specifically inside the case 1 block) to recognize PUSH opcodes (0x60 - 0x7F) and advance the cursor past their immediate data. This ensures that 0x5B is only recognized as a JUMPDEST when it appears at a valid instruction boundary.

Example Scenario:
Data: hex"00605b"
Expected: Scanner halts at 00. It sees 60 (PUSH1), skips 5B (data). No JUMPDEST found. Code remains unreachable.
Actual (Bug): Scanner halts at 00. It sees 60, ignores it. It sees 5B, interprets it as JUMPDEST. Scanner resumes, marking 5B as reachable.

Found in: 60f35a0f

Status: Fixed at 4c0aed6f

H02 - Interpreter Safety Check Bypassed by Transient Storage Modification

The IExtrospectInterpreterV1 interface implements a safety check via *scanOnlyAllowedInterpreterEVMOpcodes()* to ensure that candidate interpreter contracts do not perform state-modifying operations. This is enforced by the *NON_STATIC_OPS* bitmask, which identifies forbidden opcodes such as SSTORE, CREATE, and CALL.

However, the current implementation fails to include the TSTORE (Transient Storage Store) opcode in the NON_STATIC_OPS list. TSTORE performs a write operation to the transient storage, thereby violating the requirement that the interpreter be immutable and free of state changes during execution.

As a result, a malicious or buggy interpreter contract utilizing TSTORE would be incorrectly flagged as "safe," allowing it to mutate state unexpectedly.

Path: src/interface/IExtrospectInterpreterV1.sol

Recommendation: Update src/interface/IExtrospectInterpreterV1.sol to include *EVM_OP_TSTORE* in the *NON_STATIC_OPS* constant. This ensures that any contract using transient storage modification is correctly rejected by the safety scan. (Note: This requires EVM_OP_TSTORE to be defined in EVMOpcodes.sol first).

Found in: 60f35a0f

Status: Fixed at 4c0aed6f

** Medium

M01 - Unconditional JUMP Treated as Non-Halting

The *HALTING_BITMAP* definition in EVMOpcodes.sol missed the unconditional JUMP opcode (0x56). In the EVM, an unconditional JUMP does not allow execution to fall through to the next instruction. The instruction immediately following a JUMP is unreachable unless it is a valid JUMPDEST that is targeted by a jump from elsewhere.

By treating JUMP as non-halting, the scanner continued to mark the subsequent bytes as "reachable" via sequential flow. This incorrectly flags dead code (which often contains metadata or invalid opcodes) as reachable.

Code following a JUMP that should be ignored (until the next JUMPDEST) is included in the reachability analysis. This can cause the scanner to detect "dangerous" opcodes in dead code regions, flagging safe contracts as unsafe.

Path: src/lib/EVMOpcodes.sol

Recommendation: Update the *HALTING_BITMAP* constant in src/lib/EVMOpcodes.sol to include the EVM_OP_JUMP (0x56) opcode.

Found in: 60f35a0f

Status: Fixed at 4c0aed6f

M02 - Lack of Support for EVM Object Format (EOF)

The current bytecode analysis implementation in LibExtrospectBytecode is designed exclusively for legacy EVM bytecode. It assumes a linear sequence of instructions and relies on legacy control flow mechanisms (like JUMP and JUMPDEST) to determine opcode reachability.

The EVM Object Format (EOF) introduces a structured container format that separates code and data sections and introduces new control flow instructions (e.g., RJUMP, CALLF). The current scanner does not recognize the EOF container structure (starting with the 0xEF00 magic bytes) or its specific validation rules.

Running the existing scanner on an EOF contract will lead to undefined behavior, such as misinterpreting headers as executable opcodes, failing to trace execution paths correctly, or incorrectly flagging safe code as unsafe due to the inability to parse the new format.

Path: src/*

Recommendation: Update the library to detect EOF contracts by checking for the 0xEF00 prefix. Upon detection, the library should either:

1. Gracefully revert or return an error indicating that EOF analysis is not yet supported.
2. Implement specific parsing logic to handle the EOF container format, respecting its section boundaries and new control flow opcodes.

Found in: 60f35a0f

Status: Fixed at 4c0aed6f

* Low

L01 - Incomplete EVM Opcode Definitions

The src/lib/EVMOpcodes.sol library defines a list of EVM opcode constants but fails to include several instructions present in the current EVM version. Specifically, the following opcodes are undefined:

1. 0x49: BLOBHASH
2. 0x4A: BLOBBASEFEE
3. 0x5C: TLOAD
4. 0x5D: TSTORE
5. 0x5E: MCOPY

While the lack of definitions does not immediately break the compilation, it creates a blind spot for any logic relying on this library to be an exhaustive reference of executable or state-changing opcodes.

Path: src/lib/EVMOpcodes.sol

Recommendation: Update src/lib/EVMOpcodes.sol to include constants for EVM_OP_BLOBHASH, EVM_OP_BLOBBASEFEE, EVM_OP_TLOAD, EVM_OP_TSTORE, and EVM_OP_MCOPY.

Found in: 60f35a0f

Status: Fixed at 99e20bf

L02 - Hardcoded Solidity Metadata Assumptions

The *trimSolidityCBORMetadata()* function in LibExtrospectBytecode.sol relies on a very specific, hardcoded pattern to detect and trim Solidity CBOR metadata. It explicitly checks for:

1. A specific length (53 bytes).
2. Specific leading bytes (0xa264697066735822...).
3. A specific hardcoded hash of the "static" parts of the metadata structure.

The Solidity documentation states that:

1. The metadata hash is CBOR-encoded.
2. The last two bytes of the bytecode indicate the length of the CBOR encoded information.
3. The metadata format can vary (e.g., using Swarm instead of IPFS, or different solc versions encoding versions differently).
4. The documentation explicitly warns: "The CBOR mapping can also contain other keys... Do not rely on it starting with 0xa264...".

The current implementation violates this warning by relying on a rigid byte pattern. If a future Solidity version changes the order of keys in the CBOR map, adds a new key, or if a user configures their compiler to use Swarm (bzzr1) instead of IPFS, this function will fail to detect and trim the metadata. This makes the checkCBORTrimmedBytecodeHash function brittle and likely to revert unexpectedly for valid contracts compiled with slightly different settings or versions.

Path: src/lib/LibExtrospectBytecode.sol

Recommendation: Refactor *trimSolidityCBORMetadata()* to follow the standard metadata decoding process described in the documentation:

1. Read the last two bytes of the bytecode to determine the CBOR data length.
2. Verify that this length is reasonable and within the bytecode bounds.
3. Use this dynamic length to identify the metadata region rather than assuming a fixed 53-byte structure.
4. Ideally, verify that the data at that tail is indeed valid CBOR (though a length check might be sufficient for simple trimming purposes).

This change would make the library robust across different compiler versions and configurations (IPFS vs. Swarm).

Found in: 60f35a0f

Status: Acknowledged

L03 - Inconsistent Interpreter Isolation (TLOAD Allowed)

The IExtrospectInterpreterV1 interface defines a set of INTERPRETER_DISALLOWED_OPS to ensure that an interpreter contract is "safe" and isolated. It explicitly bans SLOAD (Storage Load) with the rationale that "Interpreter cannot store so it has no reason to load from storage." However, while TSTORE (Transient Storage Store) is banned, TLOAD (Transient Storage Load, 0x5C) is currently allowed.

This is inconsistent with the isolation philosophy. If an interpreter is not allowed to write to transient storage (TSTORE), it generally should not need to read from it (TLOAD), unless there is a specific use case for reading transient data set by the caller (which implies a lack of isolation). Allowing TLOAD could allow the interpreter to access state from the surrounding transaction context, violating the "pure" execution model implied by the other bans.

Path: src/interface/IExtrospectInterpreterV1.sol

Recommendation: Add EVM_OP_TLOAD to the INTERPRETER_DISALLOWED_OPS bitmask in src/interface/IExtrospectInterpreterV1.sol.

Found in: 4c0aed6f

Status: Fixed at f2c4f5ad

Informational

I01 - Potential Hash Literal Truncation

The hex literal has 63 characters instead of 64. Solidity will left-pad with a zero, resulting in 0x0e55864b.... This may be intentional (if the hash happens to start with 0x0e) or a typo (if a digit was accidentally omitted).

Path: src/lib/LibExtrospectBytecode.sol

Recommendation: Verify the expected hash value is correct. If intentional, add a leading 0 for clarity.

Found in: 60f35a0f

Status: Fixed at d9416b4

General Risks

- EVM Hard Fork Sensitivity: The scanEVMOpcodesReachableInAccount function relies on a specific model of EVM execution, including the set of opcodes that halt execution (e.g., STOP, RETURN, REVERT). Future EVM hard forks that introduce new control flow mechanisms or halting opcodes could render the reachability analysis incomplete or incorrect, requiring a contract upgrade.

- Compiler Metadata Assumptions: The trimSolidityCBORMetadata function relies on the specific CBOR encoding pattern used by current versions of the Solidity compiler. Changes to this format in future solc versions, or the use of different compilers (e.g., Vyper), may cause metadata trimming to fail or result in hash mismatches.

- Gas Intensity: The opcode scanning algorithms iterate over the entire bytecode of a target contract. While optimized, these operations are O(n) relative to bytecode size and are computationally expensive. They are primarily intended for off-chain execution (via eth_call); using them in on-chain transactions may result in significant gas costs or exceed block gas limits for large contracts.

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.