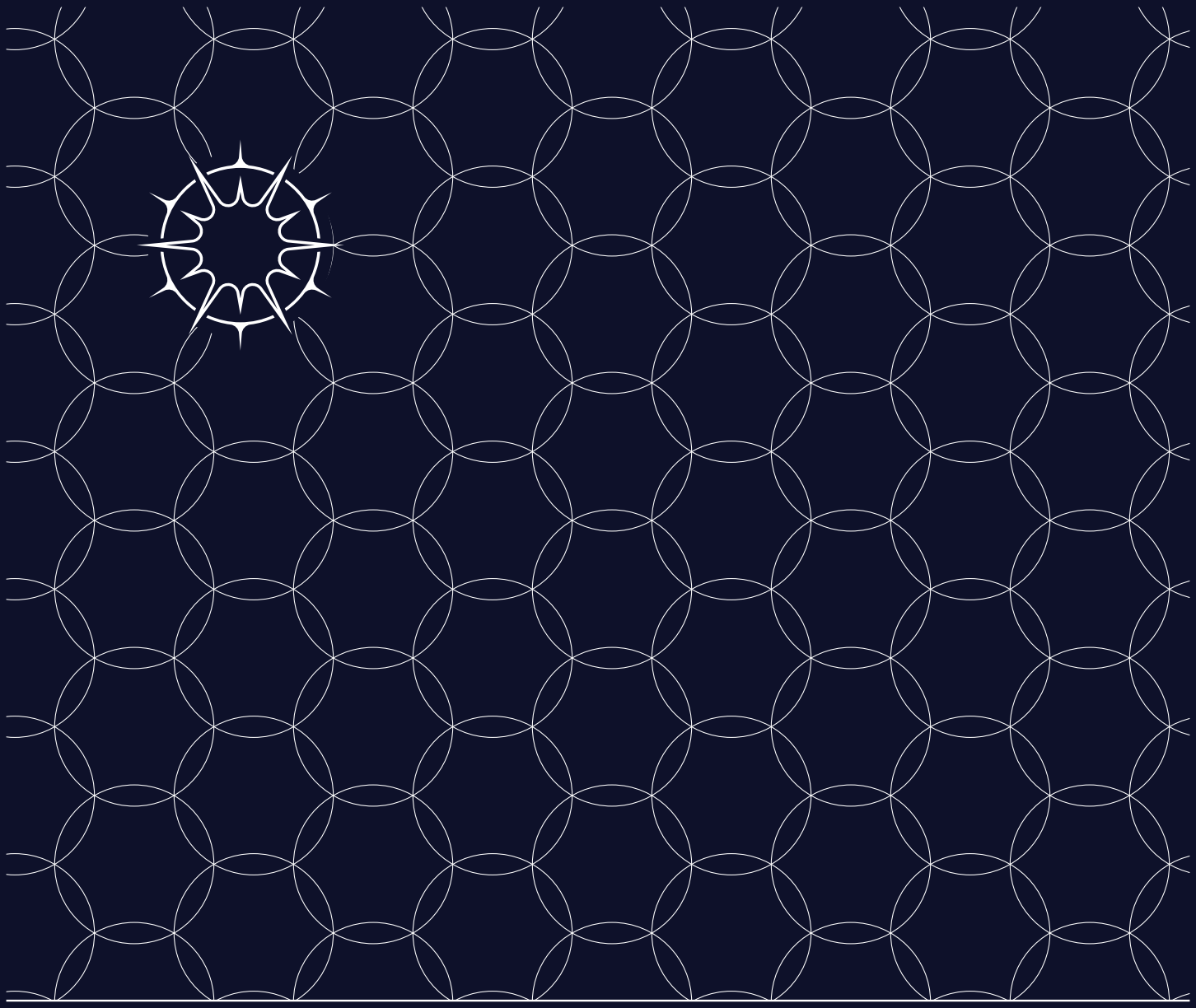

6 June 2025 **OP Labs**

OP Cannon Security Review

Version 1.0

Radiant Labs

Auditor:

3DOC

Table of Contents

1 Introduction

1.1 About OP Labs

OP Labs contributes to the Optimism protocol, an extension to Ethereum that scales both its technology and values. Optimism enables orders of magnitude of improved performance and scalability to Ethereum while doubling down on its commitment to public goods.

1.2 About Radiant Labs

Radiant Labs is a smart contract security firm providing bespoke security assessments and code reviews. Our team of specialized auditors combines deep technical expertise with practical blockchain security experience to deliver meticulously tailored security reviews. RadiantLabs ranks #1 in the Code4rena leaderboard at the time of this report.

1.3 About the Auditor

3DOC is a top ranked Smart Contract Auditor doing competitive audits on Code4rena, having ranked 1st in multiple contests in solo and RadiantLabs team audits, including the Optimism superchain contest in July 2024.

2 Engagement Details

2.1 Executive Summary

Radiant Labs conducted for OP Labs a security assessment of the OP Cannon support for Go 1.24. The changes include the implementation of a new `getrandom` syscall in the Cannon MIPS VM.

The assessment found the changes well planned, implemented and delivered with comprehensive suites for static analysis, as well as integration, end-to-end, and differential testing.

2.2 Scope

Overview

Project Name	Optimism - OP Cannon
Repository	https://github.com/ethereum-optimism/optimism
PR	#15737
Methods	Manual review
Duration	3 days

Supporting documentation:

- getrandom man2 page

2.3 Risk Classification

Findings in this report are classified according to their severity:

High - direct theft/loss/freezing of funds, unauthorised control over critical functions, or breaking of core contract functionality
Medium - economic damage or manipulation under specific conditions or with limited impact
Low - non-critical implementation issues with negligible security impact
Informational - style suggestions, documentation improvements, and recommended best practices

3 Findings

Issues Found

High risk	0
Medium risk	1
Low risk	1
Informational	1

3.1 Medium risk

3.1.1 Getrandom syscall does not clear LOAD_LINKED reservations

Context:

- `cannon/mipsevm/multithreaded/mips.go`
- `packages/contracts-bedrock/src/cannon/MIPS64.sol`

Description: The MIPS architecture implements LL/SC synchronization.

This allows a program to read a portion of memory (`LL`) and later writing (`SC`) if this portion was not updated since. This, however, requires that every write to memory that was previously reserved with a `LL` instruction, should cause the reservation to be cleared.

However, the newly introduced `getrandom` syscall does not honor this requirement, as it performs a memory write without clearing memory reservations, in both the Go and Solidity implementations:

```
File: cannon/mipsevm/multithreaded/mips.go
239:     newMemVal := (memVal & ^randDataMask) | (randomWord & randDataMask)
240:     m.state.Memory.SetWord(effAddr, newMemVal)
241:
242:     v0 = byteCount
243:     v1 = 0
244:
245:     return v0, v1
```

```
File: packages/contracts-bedrock/src/cannon/MIPS64.sol
678:         uint64 newMemVal = (memVal & ~randDataMask) | (randomWord &
    randDataMask);
679:         memRoot_ = MIPS64Memory.writeMem(effAddr, memProofOffset, newMemVal);
680:
681:         v0_ = byteCount;
682:         v1_ = 0;
683:     }
```

While it seems unlikely that a legitimate client application would use random data in portion of memory reserved for synchronization (low probability), doing so would make the application unable to synchronize properly, thus compromising the integrity of its execution in a concurrency scenario (high impact).

Recommendation: Consider updating `getrandom` implementations (Solidity and Go) by adding `handleMemoryUpdate` calls to follow the `writeMem` operations, like done for other operations that similarly write to memory.

Fix review: Fixed with d89e0ce

3.2 Low risk

3.2.1 Returned byte count deviates from getrandom spec

Context:

- `cannon/mipsevm/multithreaded/mips.go`
- `packages/contracts-bedrock/src/cannon/MIPS64.sol`

Description: The definition of getrandom specifies a `flags` argument, that allows the caller to select a `random`- or `urandom`-type generation.

When `urandom` is selected (i.e. `flags == 0`), the specification requires the syscall to provide all the bytes requested, up to `256`:

> If the urandom source has been initialized, reads of up to 256 bytes will always return as many bytes as requested and will not be interrupted by signals. No such guarantees apply for larger buffer sizes

The `getrandom` introduced through the changes in scope, however, never returns more than 8 bytes regardless of the `flags` provided, and is therefore in violation of the specification:

```
File: cannon/mipsevm/multithreaded/mips.go
228:    maxBytes := arch.WordSizeBytes - targetByteIndex
229:    byteCount := a1
230:    if maxBytes < byteCount {
231:        byteCount = maxBytes
232:    }
233:
```

```
File: packages/contracts-bedrock/src/cannon/MIPS64.sol
667:         uint64 maxBytes = arch.WORD_SIZE_BYTES - targetByteIndex;
668:         uint64 byteCount = _a1;
669:         if (maxBytes < byteCount) {
670:             byteCount = maxBytes;
671:         }
```

Because Cannon client apps are built in Go, this may not constitute a problem, since the Go SDK reiterates syscalls until the desired number of bytes are read. However, we cannot guarantee that this behavior, that was observed through testing, equally applies to all possible entry points of the `getrandom` syscall - so we can't exclude that some of these could rely on the syscall strictly following the spec.

Recommendation: Consider extending the syscall to allow up to 256 bytes modifications to meet the spec.

Fix review: Finding acknowledged

3.3 Informational

3.3.1 getrandom flags are ignored

Description: The definition of getrandom specifies a `flags` argument, that allows the caller to select a `random`- or `urandom`-type generation, together with the desired behavior in case not enough entropy is available if `random` is selected.

Both the Go and Solidity implementations in scope ignore this `flags` argument - by not reading `a2` - and force the default `urandom` behavior, thus deviating from the spec.

All data sourced from `getrandom` syscalls will therefore be pseudorandom and predictable, and consequently not be cryptographically secure, also in case the caller sets the `GRND_RANDOM` flag.

Recommendation: This is no doubt a necessary choice due to the strict determinism requirements for Cannon, and consequently a code fix can't be reasonably suggested. We however recommend raising visibility of this deviation from spec, should this randomness source be ever used in the future for feeding higher layers in the stack - if so, developers should be well aware that sequencers can greatly influence values output by this source.

Fix review: Finding acknowledged

4 Appendix

4.1 Methodology

Our manual review methodology emphasizes deep technical understanding of the target system. Initial documentation analysis guides a comprehensive code review process that examines both individual components and system-wide interactions. The review process progresses through multiple phases of increasing depth, tracking coverage while maintaining flexibility to investigate emerging concerns. We investigate unfamiliar patterns through reference implementations and technical literature, ensuring thorough comprehension of all security-relevant aspects.

4.2 Disclaimer

This report is not an endorsement or indictment of any particular project or team, and the report does not guarantee the security of any particular project. This report represents our best effort to identify potential security issues within the smart contracts based on the information available at their time of writing. The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status.