Optimism Mt Cannon Security Review

Auditors

Christian Reitwiessner, Lead Security Researcher

0xTylerholmes, Security Researcher

Lucas Clemente Vella, Security Researcher

Mveytsman, Security Researcher

Jdiggidy, Associate Security Researcher

Report prepared by: Lucas Goiriz

February 20, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Optimism enables orders of magnitude of improved performance and scalability to Ethereum while doubling down on its commitment to public goods.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Optimism Mt Cannon according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 19 days in total, OP Labs engaged with Spearbit to review the Optimism-MT-Cannon protocol. In this period of time a total of **16** issues were found.

Summary

Project Name	OP Labs
Repository	Optimism-MT-Cannon
Commit	cc2715c3
Type of Project	VM Implementation
Audit Timeline	Nov 25th to Dec 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	1	1	0
Low Risk	1	1	0
Gas Optimizations	5	1	0
Informational	8	1	0
Total	**16**	**5**	**0**

5 Findings

5.1 High Risk

5.1.1 Value loaded from memory inside `futex()` is different from what userspace uses for comparison

Severity: High Risk

Context: MIPS64.sol#L253-L258, MIPS64.sol#L529, MIPS64.sol#L531-L533

Description: Futexes are in-memory 32-bit values used mostly in userspace to synchronize threads, via atomic instructions. The need to call the `futex()` system call is relatively rare, it only happens when there is contention and a thread must wait for another before resuming. In those cases, the pointer to the 32-bit value is passed in the syscall, who is supposed to repeat the same atomic load-and-compare operation done in userspace to decide whether the thread must sleep or not.

The problem is that userspace loads and compares a 32-bit value, but the current `futex()` implementation first aligns the pointer to 8 bytes, then load-and-compares a 64-bit value from it. The 64-bit value loaded has a high probability of being different from the 32-bit value that should be loaded instead.

Consider these values in memory:

addr	addr + 4
0xPP	0xQQ

Now consider what value userspace expects to be loaded, compared to what is actually be loaded, depending on the given address:

Address	userspace	`futex()`
addr	0xPP	0xPP000000QQ
addr + 4	0xQQ	0xPP000000QQ

So it may happen that a thread should not sleep, but it goes to sleep anyway, potentially leading to deadlock, or that a futex should sleep, but it doesn't, doing busy wait until its time quantum runs out.

Both scenarios are possible in an honest program (it really depends on how the low level synchronization libraries makes use of `futex()`), but in op-program, it seems that only the busy wait scenario can happen, given that the futex value is a 64-bit variable `uintptr_t`, thus it is 8-byte aligned and the low bits are most likely set to 0.

PS: another issue with aligning the pointer to 8 bytes before loading is that the address becomes the identifier for the futex, and if there are two futexes in 2 adjacent 4-byte blocks, both will be represented by the same address internally, and the wrong futex may be woken by a `FUTEX_WAKE` call. But this issue is overshadowed by the issue described above.

Recommendation: If to maintaining the current `futex()` architecture, the address must be aligned to 4 bytes instead of 8, and the value loaded must be 32 bits.

Otherwise, see finding "Futex implementation is quite complex and still wrong" for a much simpler `futex()` implementation.

Op Labs: Fixed in PR 13453.

5.2 Medium Risk

5.2.1 [EXTERNAL - Base Audit]: Failure in clearing upper-bits after shifts

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: *The Coinbase Protocol Security team (https://www.base.org/) performed an independent review of the current report's scope and found a unique finding:*

> Some 32-bit instructions (shift left operations) on the 64-bit VM fail to clear the upper bits after shifting. If the Go runtime tries to left shift a 32-bit value and then retrieves the full 64-bit value from the register it may have incorrect values in the high bits.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import { Test } from "forge-std/Test.sol";
import { MIPS64Instructions } from "src/cannon/libraries/MIPS64Instructions.sol";

contract MIPS64InstructionsTest is Test {

    uint32 internal constant U32_MASK = 0xFFffFFff;

    function setUp() public {
    }

    function testSLL() external {
        uint8 shiftAmt = 1;
        uint8 rtReg = 9;
        uint8 rdReg = 8;

        MIPS64Instructions.CoreStepLogicParams memory params;

        params.insn = encodespec(0, rtReg, rdReg, uint16(shiftAmt) << 6);
        params.registers[rtReg] = 0x80000000;
        params.opcode = 0; // SPECIAL
        params.fun = 0; // SLL

        uint32 shiftedVal = uint32(params.registers[rtReg] << shiftAmt);
        uint64 expected = MIPS64Instructions.signExtend(uint64(shiftedVal), 32);

        MIPS64Instructions.execMipsCoreStepLogic(params);

        assertNotEq(expected, params.registers[rdReg]);
        assertEq(expected, 0);
        assertEq(params.registers[rdReg], uint64(U32_MASK) << 32);
    }

    function testSLLV() external {
        uint8 shiftAmt = 1;
        uint8 rsReg = 10;
        uint8 rtReg = 9;
        uint8 rdReg = 8;
        uint16 func = 0x04; // SLLV

        MIPS64Instructions.CoreStepLogicParams memory params;

        params.insn = encodespec(rsReg, rtReg, rdReg, func);
        params.registers[rtReg] = 0x80000000;
        params.registers[rsReg] = shiftAmt;
        params.opcode = 0; // SPECIAL
        params.fun = func; // SLLV

        uint32 shiftedVal = uint32(params.registers[rtReg] << shiftAmt);
        uint64 expected = MIPS64Instructions.signExtend(uint64(shiftedVal), 32);
```

```
        MIPS64Instructions.execMipsCoreStepLogic(params);

        assertNotEq(expected, params.registers[rdReg]);
        assertEq(expected, 0);
        assertEq(params.registers[rdReg], uint64(U32_MASK) << 32);
    }

    function encodespec(uint8 rs, uint8 rt, uint8 rd, uint16 funct) internal pure returns (uint32
    ↪  insn_) {
        insn_ = uint32(rs) << 21 | uint32(rt) << 16 | uint32(rd) << 11 | uint32(funct);
    }
}
```

Recommendation: Implement a clearing mechanism for the upper-bits.

Op Labs: Addressed by:

- PR 14114.
- PR 14185.

5.3 Low Risk

5.3.1 Instruction bltzal is not implemented in the Solidity code, but it is implemented in go

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The instructions `bltzal` is not implemented in the Solidity code, but it is implemented in the go counterpart. The Solidity code ignores the instruction by not branching.

We were not able to find the instruction in op-program, it might be that the golang to mips compiler does not make use if it. If this changes, the consequences would be that a condition is ignored on-chain but respected off-chain, with potentially severe consequences.

Proof of Concept: Unable to produce incorrect behaviour.

Recommendation: Implement `bltzal` in the MIPS64Instruction.sol.

Op Labs: Fixed in PR 13429.

5.4 Gas Optimization

5.4.1 Tests for branch in delay slot are incomplete (and not needed)

Severity: Gas Optimization

Context: MIPS64Instructions.sol#L622, MIPS64Instructions.sol#L809

Description: The test `if (_cpu.nextPC != _cpu.pc + 4)`, used to detect if there is a jump in the delay slot, has 2 problems:

- Will not detect that there is a jump in the delay slot if the previous instruction was a jump to `pc + 8`.
- Is not strictly needed, given the assumption that the guest code is trusted (i.e. that it will not try to exploit differences between this and other MIPS implementations to cause the execution to diverge).

Recommendation: Remove the tests.

Op Labs: Valid and we should fix.

5.4.2 Futex implementation is quite complex and still wrong

Severity: Gas Optimization

Context: MIPS64.sol#L220-L238, MIPS64.sol#L245-L268

Description: Except from the finding "Value loaded from memory inside `futex()` is different from what userspace uses for comparison", the `FUTEX_WAIT` and `FUTEX_WAKE` mechanism is probably sound, considering how applications make use of futexes, but not the correct behavior.

To be consistent with Linux:

1. Changing the memory value must not automatically wake the thread.
2. Keeping the memory value must not prevent it from being woken. Another thread calling `FUTEX_WAKE` is all it takes.
3. Calling `FUTEX_WAKE` needs to wake as many threads as supplied in `val`, not just one.

That said, any sane application will be protected from spurious wakes, thus countering issue 1., and they will only rely on the futex state to decide whether to wake up or go back to sleep, countering issue 2. Issue 3. is countered by the fact that every thread is scheduled eventually, regardless if it is asleep or not, so threads not awaken by `FUTEX_WAKE` will be awaken by the scheduler, provided the futex value allows it.

So, the current behavior is sound if the application is not doing anything "creative", and just using futex as intended (i.e. using it as a semaphore/mutex). That is a fair assumption that can be expected from normal applications.

The problem is: there is a much simpler implementation, with better performance, that only requires the application to be protected from spurious wakes (pretty much a requirement for using `futex()`, documented in the man page).

Recommendation: Ignore futexes almost entirely, remove state entries related to them, and treat `FUTEX_WAIT` as `if (*uaddr == val) { sched_yield(); }` (mind finding #7, as it still applies in this case), or even simpler, as a plain `sched_yield()`, as the only possibility of `*uaddr != val` is if the thread was preempted between the userspace variable check and the `futex()` call, which is a very rare occurrence, and there is no harm in yielding anyway.

Given the static nature of the scheduler, it may be a good idea that `FUTEX_WAKE` is treated as `sched_yield()`, too, so to prevent starvation in case there is a thread spending most of its time holding a lock of interest to some other thread (not the case in most well written programs).

In this proposal, every thread is waked every time. This makes sense in MTCannon because every thread is rotated every time through the scheduler, so there is very little gain in number of state steps in skipping those threads. This is unlike Linux, where the vast majority of tasks are asleep and not even considered for scheduling.

Of course, this is the simplest extreme. A middle ground would be to remove the `state.wakeup` and the wakeup procedure, as it adds nothing.

Op Labs: Fixed in PR 13754.

5.4.3 Extra variable and bitwise operation

Severity: Gas Optimization

Context: MIPS64Instructions.sol#L915

Description: This line does nothing. `byteIndexMask` is already &'ed with `_vaddr`.

Recommendation: Remove the line, and rename `byteIndexMask` to `byteIndex`.

Op Labs: We will work on fixing this.

5.4.4 Bitwise expression simplification

Severity: Gas Optimization

Context: MIPS64Instructions.sol#L595-L596

Description: It seems `signed` is just `~mask`. These lines can be replaced with:

```
uint256 mask = (1 << _idx) - 1;
uint256 signed = ~mask;
```

5.4.5 Assembly block optimization

Severity: Gas Optimization

Context: MIPS64Syscalls.sol#L277-L285

Description: The referenced assembly lines can be replaced with the following, to use half the number of instructions:

```
datLen = mul(datLen, 8) // operate in bits from now on
let rightPaddingBits = sub(mul(space, 8), datLen))

dat := shr(sub(256, datLen), dat) // right-align data
dat := shl(rightPaddingBits, dat) // position data to insert into memory word

// mask of where data should be written to the memory word
let mask := sub(shl(datLen, 1), 1)
mask := shl(rightPaddingBits, mask)
```

Op Labs: We will look into fixing this.

5.5 Informational

5.5.1 Comment says 4 bytes, but it is actually the word size

Severity: Informational

Context: MIPS64Syscalls.sol#L260

Description: Comment says 4 bytes, but it is actually the word size. Also, the comment is bellow the aligning operation.

Recommendation: Move the comment above the aligning, and use "word size" instead of "4 bytes".

Op Labs: Valid. The comment is incorrect. It should be "word size".

5.5.2 Incorrect structured comments about return value of readMem and `readMemUnchecked`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation: The comments at `MIPS64Memory.sol:13` and `MIPS64Memory.sol:288both read`@return out_ The hashed MIPS state.`while they should read`@return out_ The read memory word.‘

Op Labs: Valid and we will fix.

5.5.3 Function `calculateSubWordMaskAndOffset` has undocumented preconditions

Severity: Informational

Context: MIPS64Instructions.sol#L901-L904

Description: According to the function implementation, argument `_vaddr` must be aligned to `_byteLength`, which by itself must be a power of 2.

Recommendation: Assuming a correct guest program, every usage seems to conform to these preconditions, but even so, these limitations warrants at least a docstring explaining them, on this function itself and others that simply forwards their arguments to it.

Op Labs: This looks valid, we should work on adding the comment and improving the codebase. We should fix this one, as mentioned on the closeout call.

5.5.4 Most of OP constants are not used

Severity: Informational

Context: MIPS64Instructions.sol#L9-L14, MIPS64Instructions.sol#L98

Description: Of all these OP constants defined, only `OP_LOAD_DOUBLE_LEFT` and `OP_LOAD_DOUBLE_RIGHT` are used, but only in some places. In others, the values `0x1A` and 0x1B‘ are used directly.

Recommendation: At very least, `OP_LOAD_DOUBLE_LEFT` and `OP_LOAD_DOUBLE_RIGHT` should be used whenever possible. But ideally, all opcodes should have constants to be used though the code, at least on == comparisons, as it would greatly improve the code readability. Even better if the constant names were the same as in the MIPS documentation, e.g. `OP_LDR` or `OP_LDL`.

Op Labs: Constants -- they aren't used, so we should improve them. We will look into fixing them.

5.5.5 Syscall argument error is silently ignored

Severity: Informational

Context: MIPS64.sol#L584

Description: This alignment is a fair assumption, and probably required for C ABI compliance. But as far as defensive programming goes, it is better to fail if the address is misaligned than to silently align it.

Recommendation: Fail if the pointer is misaligned.

Op Labs: Will fix. The solution recommended is to return an error if a1 is unaligned.

5.5.6 Incorrect parentheses in `handleSyscall` in `mips.go`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The line in `mips.go:214`:

```
if arch.IsMips32 && syscallNum == arch.SysFstat64 || syscallNum == arch.SysStat64 || syscallNum ==
↪  arch.SysLlseek {
```

should read

```
if arch.IsMips32 && (syscallNum == arch.SysFstat64 || syscallNum == arch.SysStat64 || syscallNum ==
↪  arch.SysLlseek) {
```

instead. On mips32, there is a differentiation between 32-bit versions of these syscalls and 64-bit versions. But on mips64, the "regular" syscalls are already 64 bit, so the constants `arch.SysFstat64`, `arch.SysSta64` and `arch.SysLlseek` are all defined as `^Word(0)`. This means that this bug does not manifest on mips64.

Op Labs: Will fix.

5.5.7 Non-specifed dependency on op-program using go runtime <= `1.22`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Cannon has an unspecified requirement for running go binaries compiled with the `go-1.22` runtime. This constraint stems from an incompatibility introduced in `go-1.23`.

The `go-1.23` runtime replaces the `sys_pipe` usage for wait/notify mechanisms in the epoll event loop with a `sys_eventfd` based implementation. Since `sys_eventfd` is not implemented or supported in the current cannon implementation running the op-program, or any binary, with go runtimes newer than `go-1.22` would trigger an unhandled syscall runtime panic.

Recommendation: Clearly document the go-runtime requirement and ensure `sys_eventfd` syscall is implemented prior to upgrading the go-runtime of the op-program

Op Labs: We should fix this one, cannon only supports `1.22`, but we should add that comment and make it visible.

5.5.8 HandleSysMmap allocations with hints

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `sys_mmap` system call allows the use of address hints to allocate memory at or near a requested address. Currently, op-cannon processes hinted `sys_mmap` calls indiscriminately in `exec.HandleSysMmap`, always informing the caller that it has memory at the requested address without performing any validation or tracking. This behavior poses a risk of guest memory allocations overlapping with op-cannon's internal memory spaces.

In certain scenarios, hints are commonly used. For instance, dependencies of the op-program handle some allocations by directly using `runtime.mallocgc`, which for large allocations will use hints.

Note: In testing the emulated program's heap is presently located at `0xc000000000`, providing substantial overhead relative to cannon's `arch64.HeapStart`. However, blindly allowing allocations at any address and with any size is a security risk.

Recommendation: Implement bounds checking for `sys_mmap` calls leveraging hints to prevent memory overlaps.

Op Labs: We won't be implementing bounds checking in the VM. As this issue does not affect the op-program (and Go programs in general). And Cannon is not intended to support any other types of programs. We will document mmap hinting in the spec per @Ivella's suggestion.