Base fault proofs mips

Security Review

Cantina Managed review by:
Christian Reitwiessner, Lead Security Researcher
Leo Alt, Lead Security Researcher
Lucas Clemente Vella, Security Researcher

August 14, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

From Jun 3rd to Jun 21st the Cantina team conducted a review of fault-proofs-mips on commit hash 71b93116. The team identified a total of **23** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 1
- Medium Risk: 8
- Low Risk: 3
- Gas Optimizations: 1
- Informational: 19

3 Findings

3.1 Critical Risk

3.1.1 Allocation overflow could allow for arbitrary code execution

Severity: Critical Risk

Context: MIPS.sol#L169

Description: `mmap` does not check for out of memory condition, and simply advance the pointer, even allowing it to wrap around to zero. This makes the entire memory susceptible to be returned from `mmap()`, including the data and text sections of the MIPS programs being executed.

The memory layout of the MIPS program being executed is public knowledge. If an attacker manages to exploit this program to allocate so much memory that the allocation pointer is just behind the beginning of the text section (maybe by making use of finding "Lack of GC/deallocation is at odds with EVM gas security model"), the next `mmap()` call can be crafted to be an EVM memory allocation, which will override part of the MIPS binary being executed with the contract's data memory, that was controlled by the attacker and filled with whatever MIPS code they want.

Recommendation: There must be a limit in the amount of memory the program can allocate, and the allocatable region region of the address space must not coincide with any program section (data or text) in the MIPS executable.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

3.2 High Risk

3.2.1 Lack of GC/deallocation is at odds with EVM gas security model

Severity: High Risk

Context: MIPS.sol#L142

Description/Recommendation: In EVM, the gas cost tries to emulate the real resources consumption of executing a contract in a real machine, and its entire security model depends on this assumption.

The gas cost of memory for an EVM program is quadratic, but is based on the assumption that when a contract ends, its memory will be freed. This virtual machine does not provides `munmap` nor have threads to allow for the GC to run, breaking this assumption when a subcontract returns from a `CALL`.

Running some back-of-the-envelope calculations, it would cost ~20,000,000 gas for a contract to allocate ~ 6 MB of usable contract memory (by `CALL`ing ~13,000 times, in a loop, a contract that allocates 15 words, I got this number trying to maximize memory per gas cost). Having a contract holding this much memory allocated simultaneously in real EVM would cost > 115,000,000,000 gas.

The underlining consequence for the emulated MIPS machine are hard to predict, but just by assuming that for every `CALL` the EVM allocates a single 4 KB page, a contract expending all its 20,000,000 gas limit in `CALL`s can make the EVM allocate upwards of 100 MB.

3.3 Medium Risk

3.3.1 Unknown/unimplemented system calls silently succeeds

Severity: Medium Risk

Context: MIPS.sol#L290

Description/Recommendation: Should probably revert on unknown/unimplemented system calls, but at the bare minimum, should not return 0 (success) on `$a3`.

3.3.2 Unaligned memory reads are aligned silently but should panic

Severity: Medium Risk

Context: MIPS.sol#L743, MIPSInstructions.sol#L187, MIPSInstructions.sol#L197, MIPSInstructions.sol#L205, MIPSInstructions.sol#L221, MIPSInstructions.sol#L233, MIPSInstructions.sol#L243, MIPSInstructions.sol#L247

Description/Recommendation: This silently aligns the address (which is consistent with the go implementation). The spec demands an exception for any unaligned memory access though.

This means that improperly aligned memory reads result in wrong data being returned by the read operation and is also inconsistent with the MIPS spec.

Coinbase: Acknowledged.

3.3.3 Integer Overflow not implemented for `add`, `addi` and `sub`

Severity: Medium Risk

Context: MIPSInstructions.sol#L117, MIPSInstructions.sol#L125

Description/Recommendation: `add`, `addi` and `sub` must raise exception on signed overflow. Probably best to implement with Solidity "checked" arithmetic over type `int32`.

Coinbase: Acknowledged.

3.3.4 `add`, `addi`, `sub` must panic if overflow

Severity: Medium Risk

Context: mips_instructions.go#L72

Description/Recommendation: Same as the Solidity counterpart: `add`, `addi` and `sub` must raise exception on signed overflow.

Coinbase: Acknowledged.

3.3.5 Wrong implementation of `srav`

Severity: Medium Risk

Context: MIPSInstructions.sol#L53

Description/Recommendation: Must mask the 5 lower bits of `rs` (i.e. `rs & 0x1F`), just like the previous two instructions `sllv` and `srlv`. Failing to do so is not MIPS conformant. Quoting the manual on `srav`:

> The bit-shift amount is specified by the low-order 5 bits of GPR rs.

For example, in a conforming MIPS implementation, if `rt` contains `0x0000FFFF` and `rs` contains `0x28`, the result would be `0x000000FF`, as only the five lower bits of `0x28` would be used. In this implementation, the result is 0.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

3.3.6 Non-zero `rd` register may be wrongly set to 0

Severity: Medium Risk

Context: MIPS.sol#L435

Description/Recommendation: Similar to other issues, this assumes an honest and bug-free compiler. If a non-zero `rd` register is passed here with `MTHI` or `MTLO`, that register would be overwritten with 0. This could be due a malicious or buggy compiler.

Coinbase: Acknowledged.

3.3.7 J/JAL uses the wrong `PC` value as the high bits for the jump

Severity: Medium Risk

Context: MIPS.sol#L696-L697

Description/Recommendation: This seems wrong according to MIPS ISA specification linked at the top of the file. This jump should be within the 256MB region of the instruction being executed (`state.pc`), not the next instruction (`state.nextPC`). Notice that this is different from the branch instructions, which uses as reference the PC of the following delay slot.

3.3.8 Branch instructions bgezal and bltzal not implemented

Severity: Medium Risk

Context: MIPS.sol#L346

Description/Recommendation: Only two sub-instructions of the "opcode 1" are implemented: `bltz` and `bgez`. Other instructions like `bgezal` (`rtv == 17`) are not implemented and behave like a non-branching branch. The instructions should either be implemented or the code should revert if rtv is larger than 1.

3.4 Low Risk

3.4.1 Validation of binary encoding of instructions is too lax

Severity: Low Risk

Context: MIPS.sol#L329-L336, MIPS.sol#L384-L431, MIPS.sol#L759-L760, MIPSInstructions.sol#L115-L122, MIPSInstructions.sol#L131-L142, MIPSInstructions.sol#L147-L153, MIPSInstructions.sol#L158-L181, MIPSInstructions.sol#L30-L42, MIPSInstructions.sol#L78-L81

Description/Recommendation: Many instructions have regions of its binary encoding that are mandated to be zero, but these are not being checked or enforced. For most, having non-zero values is innocuous, but some do generate side effects (e.g. see findings "Non-zero `rd` register may be wrongly set to 0" and `sync` instruction can act as a "move").

Notice that `add`, `addu`, `and`, `or`, `xor`, `slt` and `sltu` do have zeroed ranges, but the code that handles their cases also handles `addi`, `addiu`, `andi`, `ori`, `xori`, `slti` and `sltiu`, which do not have zeroed ranges.

Coinbase: Acknowledged.

3.4.2 Location of registers array in memory should be verified

Severity: Low Risk

Context: MIPS.sol#L678

Description: Some assumptions about the compiler with regards to memory and calldata layout are checked using assertions in this code. One fact that is not verified in the same way is that the registers array is allocated in memory right after the state struct. In MIPS.sol#L678 the pointer to this array is written to memory, but the compiler should already have allocated the array at exactly that point.

Recommendation: It would be good defensive practice to check that the value is already stored in memory at exactly that point instead of overwriting it. If the allocation algorithm somehow changes, this might have unforeseen consequences.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

3.4.3 `state.exited` should be checked to be either 0 or 1

Severity: Low Risk

Context: MIPS.sol#L131, MIPS.sol#L684

The field `exited` of the `state` struct is a boolean, but in Solidity, booleans occupy a full byte, which means they can have more than just the values 0 or 1. In some parts of the code, this byte is "*converted*" to a boolean by checking `!= 0`, in other parts, the check is `== 1`, which yield different results for e.g. the value 2. The code should revert if the value is larger than 1.

Coinbase: Acknowledged. We'll be fixing this issue on a later date.

3.5 Gas Optimization

3.5.1 Possible improvement in pointer alignment operation

Severity: Gas Optimization

Context: MIPS.sol#L162-L166

Description: The highlighted code could be rewritten without branching, with just two arithmetic operations (with the fair assumption that `~4095` will be optimized into a constant):

```
uint32 sz = (a1 + 4095) & ~4095;
```

3.6 Informational

3.6.1 Wrong comment

Severity: Informational

Context: MIPS.sol#L253

Description: Comment says this is a read, but this is a write.

Recommendation: Update the comment to reflect the actual operation.

3.6.2 `mmap` does not require `MAP_ANONYMOUS` for allocations

Severity: Informational

Context: MIPS.sol#L161

Description/Recommendation: To make the `mmap` a little more compatible with Linux, it should validate argument `flags` (`$a3`) for the presence of the flag `MAP_ANONYMOUS`. Otherwise, this is not an allocation and the call should fail.

3.6.3 Misleading constant name

Severity: Informational

Context: MIPS.sol#L56

Description: The current name `FD_PREIMAGE_WRITE` makes it look like this is the counterpart of `FD_PREIMAGE_READ`, but in it is not, in the actual implementation, the former is used as a selector for the later.

Recommendation: A more appropriate name could be `FD_PREIMAGE_KEY_WRITE`, because it makes it clear that it is writing the key of the preimage, not the preimage itself.

3.6.4 Oudated version in comment

Severity: Informational

Context: MIPS.sol#L47-L48

Recommendation: The version in the comment is different from the version in the constant.

3.6.5 The `signExtend` function could use the `SIGNEXTEND` opcode

Severity: Informational

Context: MIPSInstructions.sol#L258

Description/Recommendation: The `signExtend` function is a manual implementation of sign extension even though the EVM has an opcode for sign extension that can be used directly.

3.6.6 Use arithmetic right shift instead of manually implementing it through logical right shift

Severity: Informational

Context: MIPSInstructions.sol#L41

Description/Recommendation: The EVM (and Solidity) support an arithmetic right shift operation, so instead of using logical right shift and sign extend, this functionality should be used.

3.6.7 Make use of compiler-provided sign extension routines

Severity: Informational

Context: MIPS.sol#L722

Description/Recommendation: MIPS.sol#L722 is equivalent to `rt = uint32(int32(int16(uint16(insn))))` where the compiler provides the sign extension routine (it actually makes use of the signextend opcode) and also checks that the type conversions are not illegal.

3.6.8 *Magic number* 420 should be replaced by a constant

Severity: Informational

Context: MIPS.sol#L649

Description/Recommendation: The magic number 420 is the assumed location of the proof in the call-data. It is repeated in the `proofOffset` function. Instead, both locations should use a constant.

3.6.9 Unify revert data

Severity: Informational

Context: MIPS.sol#L566

Description: Throughout the code, some revert messages are strings while some others are just hex values. The revert message style should be unified.

Recommendation: While the current line is assembly code, it is not too difficult to actually encode a proper string revert message.

3.6.10 `require` message typo: should read `invalid` instead of `valid`

Severity: Informational

Context: MIPS.sol#L492

Description/Recommendation: Require messages should specify the error reason instead of the desired behaviour. Because of that, the string should read "invalid register" instead of "valid register".

3.6.11 Use helper function to return the `state` object

Severity: Informational

Context: MIPS.sol#L488

Description/Recommendation: The state object is kept at a fixed memory location to avoid passing it as an argument to functions. The problem is that at the beginning of these functions, the object (i.e. the pointer to it) is created with a snippet of assembly that contains the "*magic*" pointer value `0x80`. It would be better to move this code to a helper function that returns "State memory" to reduce the number of places in the code that has this magic value.

3.6.12 Memory copy routine in `outputState` could maybe use `abi.encodePacked`

Severity: Informational

Context: MIPS.sol#L94

Description/Recommendation: The routine in `outputState` that copies the memory object into a flat byte array could use `abi.encodePacked`. The downside is an allocation, but it would save a lot of code lines and there is no danger of missing a struct field or counting their sizes wrongly.

3.6.13 Use constants instead of magic numbers for syscall ids

Severity: Informational

Context: MIPS.sol#L183

Description/Recommendation: Use constants instead of magic numbers for syscall ids.

3.6.14 `sync` instruction can act as a "move"

Severity: Informational

Context: MIPSInstructions.sol#L78-L81

Description/Recommendation: `sync` instruction is not handled elsewhere. This means that the binary values `insn[25..21]` and `insn[15..11]` are interpreted, respectively, as `rs` and `rd`, and this instruction act as a move from `rs` to `rd`. If the `sync` instruction is well formed, it will be a move from `$zero` to `$zero`, so effectively a `nop`. But it could be anything, because the binary range `insn[25..6]` is unchecked. I suggest asserting at least `insn[25..11]` to 0, to avoid creating an accidental "move" instruction.

3.6.15 `jalr` unchecked undefined behavior

Severity: Informational

Context: MIPS.sol#L761

Description/Recommendation: In case of `jalr` (`func == 9`), it is undefined for `rs` and `rd` to refer to the same register. This is another source of undefined behavior the code could check for.

3.6.16 `sc` undefined behaviors not checked

Severity: Informational

Context: MIPSInstructions.sol#L246-L248

Description/Recommendation: There are a couple sources of undefined behavior in `sc` that were not checked. Quote from the "The MIPS32™ Instruction Set":

- Execution of SC must have been preceded by execution of an LL instruction.
- A RMW sequence executed without intervening exceptions must use the same address in the LL and SC. [...]

3.6.17 `clz` and `clo` have unchecked undefined behaviors

Severity: Informational

Context: MIPSInstructions.sol#L166-L176

Description/Recommendation: It is undefined behavior if `rs` and `rt` do not refer to the same register in the encoding of `clz` and `clo` instructions. The behavior is unspecified, akin to branch/jump in delay slot, which the code tries to detect and revert. So it would be consistent to also try to detect and revert on this one.

Notice that it doesn't suffice to assert that `rs == rt`, because the equal values might still have come from different registers.

3.6.18 Branch in delay slot check can be bypassed

Severity: Informational

Context: MIPS.sol#L320

Description/Recommendation: The same issue as finding #1: this check does not rule out the previous instruction was not a jump or branch with target `state.pc + 4`.

3.6.19 Jump in delay slot check can be bypassed

Severity: Informational

Context: MIPS.sol#L459

Description/Recommendation: This doesn't catch every jump in the delay slot. The previous instruction could have been a jump to state.pc + 8. According to Cantina VM Specs, "While this is considered "undefined" behavior in typical MIPS implementations, FPVM must raise an exception when stepping on such states." from @lvella