CS161 Fall 2026 — Introduction to Computer Security — Exam Prep 9

Q1 *Isolation: Boxed In* 📦 **(13 points)**

EvanBot is designing how to run untrusted code, and considers three layers of isolation: the **operating system** (isolating processes), the **hypervisor** (isolating virtual machines), and **WebAssembly** (isolating modules within a single process).

Q1.1 (3 points) An operating system protects three kinds of state using three different isolation techniques. Match each kind of state to the technique the OS uses to protect it.

Registers in the CPU:	● Time multiplexing	○ Translation	○ Emulation
Main memory:	○ Time multiplexing	● Translation	○ Emulation
I/O devices:	○ Time multiplexing	○ Translation	● Emulation

Solution: Registers are protected with **time multiplexing**: one process controls all registers at a time, and their values are saved and cleared on a context switch. Memory is protected with **translation** (virtual memory): each process gets its own virtual address space, and the OS translates virtual to physical addresses. I/O devices are protected with **emulation**: processes can't touch devices directly, and instead call OS system functions that mediate access. In all three cases the overhead is minimal (few registers / page-table swap is cheap / I/O is already slow).

Q1.2 (3 points) Under virtual memory, the OS translates a virtual address by looking up its page number in the process's page table and copying the offset (the bottom 12 bits) unchanged. Firefox's page table contains:

Virtual Page Number	Physical Page Number
`0x00004`	`0x60C25E6`
`0x00005`	`0x1F96EC7`
`0x00006`	`0xEC70DB7`
`0x00007`	`0x69977244`

Firefox reads from virtual address `0x00006A3C`. What physical address does the OS access?

`0xEC70DB7A3C`

Solution: The bottom 12 bits (3 hex digits) are the offset: `0xA3C`. The remaining top bits are the virtual page number: `0x00006`. The table maps virtual page `0x00006` to physical page `0xEC70DB7`. Concatenating the physical page number with the unchanged offset gives `0xEC70DB7 A3C` = `0xEC70DB7A3C`.

This content is protected and may not be shared, uploaded, or distributed.

Q1.3 (3 points) WebAssembly runs many untrusted **modules** inside a single OS process. Each module is given its own contiguous 2^{32}-byte chunk of the process's 2^{64}-byte address space. Which of the following mechanisms help prevent one malicious module from reaching into another module? Select all that apply.

- [x] Each module can only form 32-bit addresses, so it cannot name any address outside its own 2^{32}-byte chunk.
- [x] Wasm has no arbitrary jump instruction; programs may only jump to a small set of pre-defined targets.
- [x] Compiled module code is immutable and lives outside the 2^{32}-byte data chunk, so a module cannot rewrite its own code.
- [ ] Virtual memory (VMEM) gives each module its own page table, isolating their address spaces.
- ○ None of the above

Solution: The first three are Wasm's isolation mechanisms. **Disjoint address spaces:** a module speaks only 32-bit addresses, so by construction it cannot even name memory outside its chunk. **No arbitrary jumps:** jumps are restricted to pre-defined targets (checked at compile time), so a module can't jump into another module's code. **Immutable code:** since running code can't be modified, the attacker can't rewrite instructions to re-introduce arbitrary jumps.

The trap is VMEM: all modules run in the **same process**, sharing one virtual address space and one page table, so virtual memory does **not** separate them — which is exactly why Wasm needs its own isolation mechanisms.

Q1.4 (1 point) Is WebAssembly's module isolation **stronger** than the OS's process isolation?

- ○ Yes, because each module gets its own page table, which the OS does not give to individual processes.
- ○ Yes, because a module cannot make arbitrary jumps, so no memory-safety exploit is possible inside a module.
- ● No, because Wasm gives up some isolation strength in exchange for much cheaper isolation within a single process.
- ○ No, because each Wasm module is run in its own OS process, so its isolation is exactly the OS's and no stronger.

Solution: Wasm's selling point is **cheap** isolation, not **strong** isolation: it separates modules inside one process, avoiding the cost of a context switch, but OS-level process isolation is still the stronger guarantee. The security-vs-performance trade-off remains.

The first option inverts reality — the OS gives each **process** a page table, while all Wasm modules share one. The second overclaims: removing arbitrary jumps stops a module escaping its chunk, but a module can still corrupt its own memory. The last option has the right verdict for the wrong reason: modules within one Wasm runtime share a single OS process, which is the whole reason Wasm needs isolation mechanisms of its own.

This content is protected and may not be shared, uploaded, or distributed.

Q1.5 (1 point) Naive software interposition can be defeated by an attacker who `jmp`s **past** an inserted bounds check. Does Wasm prevent this bypass?

- ● Yes, because Wasm has no arbitrary jump instruction — a program may only jump to a small set of pre-defined targets, so there is no way to land just after a check.
- ○ Yes, because a module's compiled code is immutable, and immutable code cannot be jumped over.
- ○ No, because a module may still jump anywhere it likes, as long as the target lies inside its own 2^{32}-byte chunk.
- ○ No, because Wasm does not insert checks at all — it isolates modules by emulating every instruction instead.

Solution: Software interposition rewrites code to insert checks, but on most architectures an attacker can simply `jmp` directly to the instruction after the check. Wasm removes the arbitrary-jump instruction entirely — jumps are restricted to pre-defined targets validated at compile time — so the guard cannot be skipped.

Immutable code is a real Wasm property, but it prevents **rewriting** instructions, not **jumping over** them; on its own it would not stop this bypass. The third option describes the disjoint-address-space mechanism, which bounds **which memory** a module can name and says nothing about jump targets. The fourth is simply false: emulation (interpreting every instruction) is the slow technique Wasm is designed to avoid.

Q1.6 (2 points) Emulation and software interposition both add isolation checks to untrusted code. State the key difference in **when** each performs its checks, and name the resulting drawback of each. (In 10 words or fewer.)

Emulation checks during execution (slow); interposition before (bypassable).

Solution: Emulation checks the code **while running it** — an interpreter reads and executes each instruction — so its drawback is **performance**: the code never runs directly on bare metal. Software interposition rewrites the code with checks **before it runs** and then runs it directly (fast), but its drawback is **completeness/complexity**: it's hard to insert a correct, un-bypassable set of checks (e.g. an attacker jumping past a guard, as in the previous part).

Full credit for any answer within the word limit that gets both the timing split (during vs. before execution) and both drawbacks (slow vs. skippable/incomplete). Award 1 point for the timing split alone.

This content is protected and may not be shared, uploaded, or distributed.

(Question 1 continued…)

This content is protected and may not be shared, uploaded, or distributed.