CS50300: Operating Systems

HW2 Answers

Zichen Wang
wang4113@purdue.edu

September 26, 2018

1

For a system call, the user stack cannot be used because it might be compromised or corrupted by a malicious user process or function. Then, the whole system would probably be crashed by that malicious app or the return address of the process calling the system call will be modified. Moreover, system calls are always used to execute privileged instructions like reading or writing a file, spawning a process, waiting for a signal and so on. During the execution of a system call, the CPU will switch to kernel mode and execute kernel code. There must be some local variables or registers related to the kernel that need to save in the stack and cannot be modified by a user process. Kernel stack is more safe than user stack because it can only be accessed in kernel mode. When a malicious process would like to read or write the data from the kernel stack in user mode, the hardware will throw an error. Therefore, each process has a separate kernel stack to ensure everything goes well when making system calls.

If the kernel stack is not provided by the software, we can easily modify the return address of a system call. For instance, process A makes a system call to sleep for a while, and the scheduler decides to run process B. However, process B is malicious and will try to modify the return address of that system call to a malicious function. If this address is saved in user stack of process A, process B will have right to rewrite that in user mode. What's worse, process B will be able to rewrite the register values saved in the user stack of process A. When process A wakes up, the CPU will switch to kernel mode again and restore these modified registers from the user stack of process A. Hence, the whole system will be hijacked by process B.

2

Full virtualization is a kind of virutalization that enable multiple operating systems to run simutaenously on the same hardware. Each guest OS has an illusion that it owns all the bare metal hardware.

In full virtualization, the architecture is split into four layers. The top layer is user apps running in user mode inside each operating system. The layer under the apps is the operating systems' layer. We may run multiple unmodified operating systems at the same time. The OS in this layer is now running in user mode in case of full virualization. Then, the third layer is a hypervisor, managing these operating systems and dealing with external interruptions from the hardware as well as exceptions from user apps. The hypervisor is running in kernel mode. The bottom layer is the hardware.

The overhead of full virtualization will be analyzed in the following scenarios.

1. If the code of a user app does not have any privileged instruction, it can run directly in user mode on the bare metal without any overhead, meaning there is no slowdown in this situation. This is the best case.

2. If a user app has a system call, it will first trap into the hypervisor by hardware. However, the hypervisor only has a general IDT, and it does not know the exact meaning of this system call, so the context will jump back to the guest OS on which this app is running, and check the IDT of that guest OS.

   (a) If the system call does not have any privileged instruction, it will be running directly in user mode and then return back to that user process. Therefore, from the begining of the system call, the context flow will switch to the hypervisor, then jump back to the guest OS, run the kernel code and finally go back to the user process. There is only a little of overhead.

   (b) However, the most common situation is that the system call has some privileged instructions. In this scenario, the kernel code will trap into the hypervisor again, execute privilege instructions in kernel mode, and then jump back to the kernel code in user mode. One privilege instruction means one trap. The more traps, the more overhead. Unfortunately, there are always some privileged instructions in modern kernel code. Hence, the worst is the case where a lot of privileged instructions exist in the kernel code.

In X86, there are some sensitive instructions that are not privileged. Take 'popf' as an example. The effect of 'popf' is different between user mode and kernel mode. When a user process executes 'popf' in user mode, it will not trap and the 'IF' bit will be always 1. In kernel mode, however, 'popf' will change the 'IF' bit according to the value in top of the stack. This instruction will lead to a bad situation where the hypervisor will not know the modification if the guest kernel wants to modify the 'IF' bit to 0 by 'popf'. It is because the guest kernel will not trap when executing 'popf' in user mode. This modification will fail silently. The hypervisor still holds 'IF=1' for this guest kernel, while this guest OS thinks 'IF=0'.

To overcome this problem, one way is to let hypervisor look the code block to find whether there is a sensitive instruction before running codes of the guest kernel. If the hypervisor finds a sensitive instruction, it will insert a 'hook' instruction and recompile the code, and go back to run the kernel code. Therefore, some sensitive instructions create complications when trying to implement full virtualization.