HW2

Jain117 | Nitin Jain

1. Discuss why systems calls in modern kernels (e.g., UNIX/Linux, Windows) cannot use the calling process's user space run-time stack but require a per-process kernel stack to achieve isolation/protection. Describe a specific example that shows what can happen if this software support is not provided.

Answer:

To achieve isolation/protection, the most important requirement is that, when the process is in the kernel mode, it must not use the memory from the user space. This is because, if the runtime stack from the user space is also used by the kernel stack, then any other process or function can modify important fields in the stack like EIP, EBP and others. This follows from the fact that in the user space, functions can access and modify the whole user runtime stack since it is shared. This can't be restricted or validated by the kernel from design. A single run-time stack per process will lead to kernel sharing the stack which can then lead to modification from other function to the process values when the process is in kernel mode and this is not safe. Therefore, all modern kernels require a per-process kernel stack. An example will explain this better:

The example is same as the hijacking the process done in Hw1. Consider a kernel which doesn't use a per-process kernel stack. Now let a process P having initiated a system call being context switched out and a different process Q is being context switched in. Since, the run-time stack is shared in the kernel, this process Q can access and modify the return address of the function calls from P to point to a malware. Once this happens, process P will execute the malware the next time it is context switched in. Hence, isolation/ protection of processes is shattered. Since, the kernel doesn't validate every instruction this can't be avoided. If the kernel used a different run time stack per-process, then in kernel mode it becomes impossible to access or modify by any other process because the memory locations are completely different and there can be a validation set in place to inhibit any such access across process memory requests.

2. We encountered different forms of a running program's state or context, from regular function calls (e.g., CDECL caller/callee convention) within a single process, to system calls by a process, to context switching between two processes, and context switching between two guest kernels running over a hypervisor. Describe what types of additional overhead are introduced as we

move from saving the context for regular function calls to saving context when switching from one guest kernel to another.

Answer:

Comparison of overhead between different forms of running program:

When calling regular functions, the kernel needs to store a few specific registers such as the base pointer and the return address in memory so that it can return back to the caller function. The values in the general registers which were being used in the caller function also need to be stored before starting to execute the callee function. This is the overhead when calling a regular function.

In case of making a system call, there is much more ahead as compared to the above regular function call. This is because when a system call is invoked, the kernel not just needs to store the same registers as before when making a regular function call, but also has to change the mode from user to kernel. When a system call is called, an important task which the kernel has to ensure is that the arguments passed to the system calls are not malicious in any ways and are valid for the system at that point in time. Once this validation and sanity check is performed the trap instruction is executed the mode change from user to kernel mode. This validation and sanity check is the additional overhead when working with system calls.

In the case of context switching between processes, we need to store the snapshot of the older process as it was just before context switching out. This mean, there is a need to the general purpose registers like EAX, EBX. We also need to store the EFLAGS , CRO, EBP, EIP registers. Also, SS (stack segment), DS, CS needs to be stored. Clearly these are many more registers which need to be stored than compared to the above scenario.

In case of switching from one guest kernel to another, the hypervisor runs bare metal in case there are no system calls are made executing the applications in the user space as no trap call is made. When this happens, switching between kernels is equivalent to a context switch. If processes make system calls, then the hypervisor will have additional overhead of masking the state of one operating system from the other.

For example, consider a process in OS1 makes a system call to change eflags. Then in this case, the hypervisor has to catch this trap instruction and take a note of the flag related changed for OS1 and serve based on that. When it

switches to OS2, the effect of change in eflags should be undone. Now, if OS1 is context switched again then, the hypervisor has to make sure that the behavior is similar to when the eflags were changed.

Thus, this is a significant overhead on the hypervisor, and this is more than any compared to the above schemes.

3. Explain what full virtualization is, and what overhead it incurs in the best case (recall our discussion of specific apps running on guest kernels) and worst case. What are sensitive instructions in x86 and why do they create complications when trying to implement full virtualization? Describe an example using pushfl.

   In the case of full virtualization, there are a number of guest kernels running simultaneously and each guest kernel is virtualized. The hypervisor is the actual kernel which runs on the bare metal hardware and gives the illusion to the guest kernels that they are running on the actual hardware.

   Best Case Scenario: The hypervisor runs with negligible overhead when the processes in guest kernel make no system calls. This has been explained earlier.

   Worst Case Scenario: When a process in a guest kernel makes a system call which consists of privileged instruction, the guest kernel will get trap this instruction and it is in turn trapped in the hypervisor. Hence, there is an overhead at 2 levels. Along, with this the hypervisor has to maintain a proxy hardware state for each kernel which is another overhead.

   Sensitive instructions are instructions whose outcomes/result depends upon in the mode it is being run in and they are neither privileged nor non–privileged. These instructions thus are difficult to be trapped by the hypervisor and hence difficult to virtualize. The pushfl instructions are affected by the value of IOPL-flag present in the EFLAGS field. Consider a context switch b/w 2 processes, and the state of the current process is saved. The two fields in this register, Interrupt enable flag (CLI makes it 0 and STI makes it 1) and IOPL (it has 2 bits – 00 for kernel mode is default). The CPL is compared against IOPL and only when CPL<=IOPL, the instruction is executed. This field is saved on context switching as it is important to knowing the mode.

   Pushfl pushes the state of these flags onto the stack and the EFLAGS is stored in user space. Now because the EFLAGS are stored in user space, any

malicious function or the user can change the value in stack and when this is popped out by the kernel, it can make the IOPL flag 11 enabling the user or the malicious code to execute privileged instruction. This makes the guest kernel vulnerable. Thus, maintaining IOPL flag state is difficult in full virtualization.