Operating System Security

Today, we'll look at security issues in multi-user computers and several fundamental security mechanisms provided by operating systems.

1 Security in Multi-user Systems

Operating system security had its birth in the 1960s and 1970s, when people became concerned about providing security for multi-user systems. At that time, computers were very expensive, and so formed a resource that had to be shared by many users. This then raises the question of how we can protect users from one another, so that if one user is malicious or misbehaves, he/she cannot violate the security of other users. To simplify the problem, let's imagine we have two users, Alice and Bob. We'd like to protect Alice from any misbehavior Bob might engage in, and conversely protect Bob from any misbehavior by Alice.

A simple solution might be to give each user their own computer. Today, computers are cheap enough that this solution might be viable in some cases—but at the time, it was a non-starter. So let's look at other approaches.

Another conceptually simple approach might be to allow each user to have sole possession of the computer for some period of time, and then turn it over to the next user waiting in line. So Alice might get to use the computer as she sees fit, and then when she is done, she turns it over to Bob, who receives full access while he's using it. We might not bother with any operating system or other security measures. What are the security properties of this approach?

It turns out that this kind of time sharing has many security problems. If Alice is malicious, she could leave malware or spyware running on the computer when she leaves. Her malware could spy on everything Bob does with the computer, or could tamper with the computing jobs that Bob runs (e.g., modifying their output before it is displayed to Bob). She could tamper with any programs installed on the computer or files resident on the machine's filesystem, which could compromise the integrity of the computer for Bob if he uses any of those files. Dually, if Bob is malicious, he might be able to violate some of Alice's confidentiality goals: if there is any confidential information left behind in memory or on the filesystem after Alice is done with the machine, Bob can potentially recover it and thus may be able to learn secrets that Alice didn't want him to learn.

This is an important lesson. In general, if someone can run malicious code (of their choosing) on your computer, and if there is no OS or other mechanism to impose restrictions on what

that code can do, then they can control the operation of your computer from then on[1]. As Microsoft puts it[2]: "If a bad guy has unrestricted physical access to your computer, it's not your computer anymore" and "If a bad guy can alter the operating system on your computer, it's not your computer anymore."

These problems motivate the introduction of operating systems that can enable multiple users to share access to a computer securely. Most modern OS's incorporate mechanisms to address this problem. Let's look at how modern OS do that.

2 Memory Protection

One key mechanism that OS's use is memory protection. Memory protection is intended to ensure that applications cannot tamper with operating system code or data structures. It also prevents one application from tampering with the code or data structures of another application.

Kernel mode. How do we restrict what operations applications can do? The first ingredient is CPU support for two modes of operation: *user mode* and *kernel mode* (kernel mode is also known as *supervisor mode*). The kernel dedicates a single bit in a special register (often called the processor status register) to keep track of which mode the CPU is currently in. Some instructions are only available in kernel mode: for instance, instructions to perform I/O might only be executable while in kernel mode, but not in user mode. Kernel-mode execution is unrestricted, but user-mode execution might have some limits placed upon it.

Typically, the operating system is written so that when the OS is running, it executes in kernel mode, whereas when application code is running, it executes in user mode. It is the OS's responsibility to ensure that this is the case: e.g., to prevent execution of application code while in kernel mode. To assist the OS in maintaining this security invariant, the CPU typically provides mechanisms to switch safely between these two modes. First, when in kernel mode, there is an instruction that can be executed to switch to user mode. Second,

[1] This is true if the malicious code runs without any restrictions—e.g., if there is no operating system. If your computer is running an operating system, the operating system at least has the opportunity to prevent such damage and restrict the harm that malicious code will do. Whether the operating system is actually able to impose such restrictions effectively is another question.

Another warning: If the attacker can inject malicious code into your operating system—e.g., by supplying a malicious device driver and tricking the user into installing that driver, or by find and exploiting a buffer overrun or similar vulnerability in the OS—then we're back to where we started: the attacker can control your computer.

[2] http://technet.microsoft.com/en-us/library/cc722487.aspx

code running in user mode can execute a system call instruction; this instruction changes the CPU to kernel mode and transfers control to a special address (this special address is stored at a location that is only writeable from kernel mode). Operating systems set this special address to hold the address of the entry point into their code that services system call requests.

Suppose an application wants to perform some privileged operation, such as I/O, that is not allowed to be performed while in user mode. How can it do that? Basically, the application asks the operating system to perform that request on its behalf. Operationally, the application does this by executing a system call instruction, which transfers control to the OS's syscall code and changes to kernel mode. The OS's syscall handling code can then decide whether to perform that request on behalf of the application or not, and if so, execute it. When finished, the OS can return the CPU to user mode and transfer control back to the application that invoked it.

In this way, the OS can arrange to restrict the operations that applications can perform. The next question is: How can the OS protect itself from malicious applications? The OS needs to prevent malicious applications from modifying its code or its internal data structures. How do we do that? Here is where virtual memory comes to the rescue, so let me share some background on how virtual memory works.

Virtual memory. Every modern computer comes with a CPU (which executes program instructions), a MMU (memory management unit—it helps implement virtual memory), and physical RAM. Virtual memory provides a layer of indirection between *virtual addresses* and *physical addresses*. The MMU keeps track of a mapping between virtual addresses and physical addresses; for instance, it might remember that virtual address V corresponds to physical address P.

The RAM chips on your computer use physical addresses: if the computer presents an address A on the RAM chip's address line, the RAM chip responds with the value stored in memory at that address. The RAM chips know nothing about virtual addresses, and they're completely oblivious to all of the virtual memory business—for instance, they have no idea and don't care which application is currently running. They just provide access to whatever physical address they're presented with.

In contrast, applications manipulate virtual addresses. When a program executes some instruction to load from or store to some address V, that address is interpreted as a virtual address. Because it is a virtual address, the computer cannot send the address V directly to the RAM chip, because RAM chips know nothing about virtual addresses. Instead, these virtual addresses are sent by the CPU to the MMU. The MMU then translates the virtual address V into its corresponding physical address P, and sends the resulting physical address

on to the RAM chip.

The MMU is responsible for maintaining a table that can be used to translate virtual addresses to machine addresses. Typically, this table is stored as a tree-based data structure, known as a page table. Entries are typically at a page granularity. A page of memory might be (say) 4096 bytes. Pages are aligned, so a page represents addresses $x, x+1, \ldots, x+4095$, where x is a multiple of 4096. (Put another way, each page begins at an address whose low 12 bits are zero.) The page table data structure maps pages in the virtual address space to pages in the physical address space, so the entry $x \mapsto y$ in the page table data structure means that virtual addresses $x, x+1, \ldots, x+4095$ map to physical addresses $y, y+1, \ldots, y+4095$ (respectively). The MMU is a special chip that is designed to perform lookups into this data structure very efficiently[3].

How virtual memory protects the OS. Virtual memory enables the OS to protect itself against malicious applications. The idea is to set up the page table mapping so that, when the application is running, the OS's code and memory is not mapped into the application's virtual address space.

Put another way, we treat every physical page in RAM as owned either by the OS or by some application. When the application is running, the OS sets up the page table data structure so that it only includes entries that point to physical pages owned by the application.

When the application makes a system call and OS code begins running, the first thing it does is modify the page table data structure to map in the OS code and data structures: it adds page table entries for all of the OS's physical pages. Thus, OS code can access both the OS's internal data structures as well as the application's data (if desired). Before the OS returns to the application, it removes the page table entries it added, so that the application cannot tamper with OS code or data.

Thus, the virtual memory subsystem prevents the application from overwriting the OS's data: there is no virtual address which the application can provide whose translation points to a physical page containing OS data structures. Moreover, the application cannot access physical pages directly.

To prevent a malicious application from modifying the page tables on its own, the MMU and CPU have mechanisms to ensure that page table entries can only be modified from within kernel mode.

[3]For efficiency, processors often have a small cache of mappings right on the CPU. This cache is known as a translation lookaside buffer (TLB). There are various architectural tricks that can be used to improve the performance of these lookups.

How virtual memory protects applications. OS's use a very similar mechanism to protect applications from each other. For each physical page owned by an application, the OS keeps track of which application owns that page. Before running an application, the OS sets up the page table to contain mappings only for that application's physical pages (and not for any other application's pages). As a result, each application can only access its own memory, and not that of any other application.

Controlled sharing. The model described above provides isolation between applications: there is no overlap between the virtual address spaces of multiple applications. So how can applications interact, communicate, or share data between each other?

The standard answer is: through system calls. Application #1 can issue a system call to ask the operating system to pass on a message to application #2. The operating system can implement a system call providing this functionality. Another approach is to use shared memory. Application #1 can issue a system call to ask the operating system to map the same physical page of memory into both its virtual address space and application #2's virtual address space, and subsequently the two applications can use their joint access to that memory region to communicate.

3 Other topics

I also discussed filesystem access control, access control lists, the Unix security model, UIDs, setuid, and some shortcomings of multi-user operating systems.