Ayush Patwari

Operating Systems [CS503]

HW1 Answers and Discussion

---

1. Describe how isolation/protection is achieved in modern operating systems, distinguishing hardware and software (i.e., OS) support features. Argue why isolation/protection is guaranteed. That is, why neither innocuous (but buggy) nor malicious apps can gain unrestricted access to shared resources.

Memory protection is very important in operating systems primarily on two accounts:

1. A user application should not be able to modify data, code and shared resources which belong to the kernel otherwise it will be able to create havoc on the system.

2. A user application should not be able to access the memory space allocated to some other process since then it can inject malicious code into the other process and cause problems.

Modern operating systems achieve isolation and protection through the following methods:

1. Virtual Memory Space: Each process/application is able to see a virtual memory space (generally of the size of actual physical memory) which is visible only to itself and only it can modify. This virtual memory is mapped to some part of the physical memory by an address translation table which has a hardware pointer pointing to it when a application is spawned. In this scheme the user cannot see the other process's addresses neither the kernel space addresses. If an application has exhausted the initial memory allocated to it: new free 'pages' from physical memory are allocated to it and updated in the translation table.

2. Mode of operations: The OS also needs to make sure that certain instructions which alter physical memory, hardware (like pointer to address translation buffer) and code (system calls) cannot be run by the user. To ensure this each process is assigned a mode of operation and each memory block has a protection key. When it is accesses the key is matched to the key of calling application and only then access is granted. Whenever a user application needs to carry out some privileged task it is done via a system call. In the system call first the controls enters the interrupt service routine in user mode -> switches to kernel mode -> performs the operation -> switches to user mode -> control transfers to user program. In this way a user program cannot possibly perform any instruction which the OS doesn't want it to.

3. Address Space Layout Randomization: This is a method which helps in protecting from buffer overflow attacks if a malicious application tries to access other memory by overflowing its own buffer or by predicting the address of shared code (like libc) and trying to execute a code by modifying the return

address on run-time stack to that of this executable However ASLR will randomize the placement of the shared libraries and the run-stack of the malicious application on each run such that their relative addressing is lost.

Using above methods modern operating systems are able to ensure that neither benign nor malicious apps can overthrow memory protection.

2. *Use the UNIX read() system call as an example to illustrate how protection works in a modern kernel with suitable hardware support. Start by tracing the events that occur when an app process invokes read() until the system call returns.*

The read(fd, *buf, nbyte) function is a system call which tries to read nbytes from the open file descriptor and write it to the buffer pointed by buf. In a modern operating system like UNIX system calls are handled through interrupts. When an app process invokes the system call read the following events occur:

1. When a compiler encounters a read system call it will store the arguments to the registers: edx, ecx and ebx. Then it stores the interrupt number in eax. Interrupt number is basically an index into which system call is being invoked by the user. (For read it is 5)

2. When the OS encounters an interrupt it traps to the Interrupt Service Routine which handles all the interrupts based on the interrupt number being passed in the register.

3. In the ISR the hardware will switch the privilege level of CPU to kernel mode (as told in class from level 3 to level 0).

4. After the CPU mode has been elevated the Program Counter is stored on the stack as well as number of other registers including the eflags register (it stores the current user level), program stack pointer etc.

5. Then the control is passed to the syscall handler for which calls the specific routine for the current call : sys_read(). This function will perform the read operation on the file descriptor provided and return with either no error or an error message in case of unwarranted read or interrupted by the OS.

6. In the return process, the return value is stored in the eax register. The control flows back to ISR which pops back the stack values and restore the program counter. Also it will call *iret* instruction to lower the privilege level of the CPU before returning to user code.

As we see, as soon as the call is made the user has no control over the execution, since the code to elevate the CPU mode is run in the ISR, and hence the user cannot possibly utilize this kernel interaction to perform privileged tasks from the user space. The hardware plays an important role to control the access level as well as storing the flags for the current user level and maintaining the program counter for later use.

3. *Describe how isolation/protection may also be achieved purely in software without special hardware support. Discuss the pros/cons of the two approaches.*

Memory protection can also be achieved by using only software methods since sometimes hardware support protection can be slow and it requires a lot of book-keeping by the OS. For e.g to implement virtual memory schema the OS has to keep address translation buffers which are looked up every time a page is accessed. Also to keep modes of operation a protection key is attached to each memory block which is checked on every access. A lot of the protection violations occur because of weak typed languages like C – which is used to develop most OS's. Strongly typed languages like JAVA provide less vulnerability and hence a form of protection. Pointers are not exposed to the application developer and all data structures can perform only intended tasks. They cannot be type casted to other data structures. By using such paradigms malicious attackers will not be able to cause unchecked buffer overflows or pointer manipulations to inject code at certain addresses on the stack.

Another way to achieve software-only based protection is through fault isolation at the software level. Compilers can generate code which is known to be safe and check for memory accesses within a specified region which is allocated to the memory. If the accesses do not check out the binaries are not created. This requires code rewriting. The memory regions could also be segmented and surrounded by guard zones. They remain unmapped systems traps to OS if they are accessed.

[Ref: https://www.cs.umd.edu/class/spring2011/cmsc838g/lectures/Feb.22.SFI.pdf]. With aggressive optimizations the overhead of such methods could be reduced to 5% of the running time of actual instructions. However, these methods are still slow as compared to hardware based methods and do not provide comprehensive support.

4. *Read the article "Introduction and overview of the Multics system" by Corbato and Vyssotsky. Write a review that discusses the similarities and differences when compared to today's desktop operating systems (e.g., Linux, Windows) based on your exposure to these systems and having taken an undergraduate OS class*.

The Multics system as described in the article is an attempt to design a system to handle a large number of users running several programs on the system. It incorporates in its philosophy most of the ideas which are used in current operating systems which support multiprogramming and multi user access. The main similarities with current OS's like Linux and Windows include:

1. Time sharing system: To fulfill its objective Multics uses time-sharing systems in which the programs are maintained in a queue and fed into the different processors according to a policy. This is very important to provide a multiprogramming view to the user. Although the scheduling algorithms may have evolved in systems like Linux and Windows, they also share the same ideology.

2. Memory Management: Multics talks about segmentation of the memory in segments which could be grown or shrunk at will of the user without having to worry about reallocation. The segments can be accessed by knowing only the starting point of the segments. This is similar to what we have in the current OSes which have segment tables which take a segment and an offset to access a memory location within a segment. Also Multics talks about paging through which less number of swaps between the secondary device and memory is required and a virtual contiguous memory view can also be provided to the user since pages need not be contiguous. This is what is also used in current systems. Each user is also provided with a private stack for temporary storage of the sub-routine, a feature which is also used in current systems. It also talks of dynamic linking of code at runtime which is considered to be adopted as dynamic linked libraries which are used nowadays.

3. File Management and protection: Multics talks about maintaining files for individual user with access permissions for other users who want to access some other user's files. It will allow read but automatic locking while writing. Also it is designed to provide automatic backup of files which is a feature which is available in systems like Linux and Windows.

4. Device I/O: Multics also describes that users can view I/O devices uniformly and read and write from any of them and the user doesn't have to rewrite his programs for interaction to various devices. This is implemented in modern OSes in the form of device drivers which give a uniform view to the user.

Although Multics is similar to modern OSes it is different in some respects:

1. Dual paging sizes are supported for more flexibility.

2. It supported batch processing on some of the processors while running multiprogramming on others.

3. Since it is designed as a computing service to run large programs it doesn't include graphical I/O and delegate this task to smaller dedicated computers while reducing the interrupt load on the main system. Whereas the desktop OSes provide graphical interfaces and have scheduling process to give higher priority to UI based applications.