CS 61 Scribe Notes

Andy Shi

Thursday, October 31, 2013

1 Announcements

1. The SEAS server hosting the bomb is crashing. If you still need to use late days for the bomb, please inform the staff.
2. Eddie is traveling next week, so there will be a surprise lecturer. Will talk more about virtual memory and what it's like to do research in systems.
3. If you do an update on your lectures repository, there are a number of branches v01, v02, v03, etc. There is a branches file in `os01` which describes the contents of the branches. Branches are the code we walk through in class. Use `git checkout -b origin/v02` for example to access the branches.

2 Review—Process Isolation

- One of the first things the attacker did to break process isolation was to enter an infinite loop. What did he actually do?

  The OS is supposed to share the resources, such as CPU Time, memory, disk, screen, etc. robustly and fairly for all the processes being run. However, the attacker monopolized CPU time.

  Recall: a *process* is an application program in execution.

  Response: The white-hat character introduced a timer interrupt to regain the control of the CPU—such a device prevents CPU monopolization.

- The second attack: The attacker used a `cli` instruction to stop interrupts. This is an inappropriate use of the interrupt hardware.

  Response: The kernel stopped the process from making the attack by reducing the process's privilege. The hardware (processor) makes sure the process with insufficient privilege cannot access interrupt hardware.

- Third attack: inappropriate modification of kernel memory.

  Response: We used the virtual memory system to protect the kernel memory from inappropriate access.

- Each attack was on a specific part of the system. The response was a cooperative effort from the OS/kernel and the hardware to prevent the process from executing the attack. Modern OSes should be secure from such attacks, but for example Mac OS 9 didn't have such protection mechanisms.
- Rely on fundamental concepts of the processor and OS. Privilege: helps distinguish "safe" and "dangerous" hardware operations. Dangerous operations may only be executed by privileged code. We need to tell the hardware whether the currently running code is privileged—this information is stored in a special register (not like `%eax`, which can be modified by the process at will).

3 Privilege and Exceptional Control Transfer

- Question: is privileged a boolean? In x86, privilege is stored in the lower two bits of a register called `%cs`. Four possible values:

  0: Kernel privilege (has as much privilege as the kernel, full control of machine possible).

  3: Process privilege.

  1 and 2 are not used—depreciated. Virtual machines used to try and run at 1 and 2 but not anymore.
- How can we confirm this? Use `gdb` on a CS61 program, break at main, info registers. Normal programs will run at privilege 3, programs in the `os01` directory will run at privilege 0.
- Aside: the `%gs` register is occasionally used to store things like canaries for stack smashing protection.
- Aside: When you run something in `sudo`, you run at root privilege, which is still privilege 3. However, there are escape hatches that allow processes with root privilege to access files/resources that normal processes cannot. THe program is still run at normal privilege (level 3) because it can still crash.
- How can we communicate across privilege boundaries? How can the process ask the kernel to do something, and how can the kernel drop privilege and run a process? Can the process just call kernel functions, like `sys_yield()`? No, we don't want the user process to access arbitrary parts of memory. More significant problem: the kernel contains a lot of code, and not all of it is meant to respond to user requests. User processes should not be allowed to do things like set its instruction pointer to the kernel code that disables interrupts, which would be bad. The process is only allowed to enter the kernel at fixed entry points.
- Process → kernel: The kernel installs an **exception vector**. It's an array of entry points for exceptional control transfers. A special call mechanism is set up to call these system calls—the instruction `int $N` activates element N of the exception vector, switches to kernel privilege, stores `%esp`, `%eip`, `%cs` on kernel stack. The memory for the exception vector is in the kernel. There is another x86 instruction, `lidt` which sets the initial value of the exception vector. Look in `k-hardware.c` to see an example.
- Now, an `int` call is run, and kernel wants to return to the process. How does a conventional process return? The return address is stored on the stack. But for exceptional control transfer calls, the kernel cannot trust the user's stack. The kernel needs to register with the processor a special stack location to store the return address of the call. The stack location also gets a complete copy of all the values of the registers.

- The return address and the old value of `%eip` is stored on the stack so we know where to go to. The processor changes privilege (because the exception vector was made by a privileged process) and saves the old privilege on the stack. We can't just do a return call because that wouldn't change privilege. We need to pop the registers `%cs`, `%eip`, and `%esp` simultaneously, otherwise there would be a point in between where unprivileged code is trying to access privileged memory. This can be accomplished using the `iret` instruction.
- Summary: kernel → process: restores `%esp`, `%eip`, `%cs` from the kernel stack. In `os01`, you can see an example in the `schedule` and `run` functions. `schedule` starts a process, `run` contains `iret`.
- System calls, like `int`, are very expensive because they have to change a lot of things. Therefore, changing privilege is inherently expensive.

3.1 Differences Between Exceptional Control Flows

- Types:
  1. **Trap**: system call. Exception that the process wanted to take.
  2. **Interrupt**: something the hardware does
  3. **Fault**: caused by a mistake that the process makes. When the process does something illegal, the OS needs to get control.
- Question: What does `%eip` point to in each situation? In a normal process, `%eip` points to the instruction after the call.
  1. **Trap**: `%eip` points to the call right after.
  2. **Interrupt**: `%eip` points to the call right after.
  3. **Fault**: `%eip` points to the faulting/problem instruction. All of the registers have the same values as they would if problematic instruction were about to be run—same values at the time of the exception. Important and interesting choice because it lets the kernel wind back and realize that, if the instruction were to execute, there would be a problem. The kernel can try and fix the program and restart at the faulting instruction. This happens all the time—example: memory-mapped I/O. The kernel takes a region of memory and marks as inaccessible. When you touch individual pages, you generate faults, and the kernel fills in the relevant spaces in memory.
- At faults, the kernel can do whatever it wants, including choosing to skip over the faulting instruction.
- **Pagefault**: name of a fault that happens at a virtual memory violation.

4 Virtual memory

- One of the most powerful ideas at the software/hardware boundary.
- Let's say we are implementing a virtual memory system whose goal is to protect kernel memory. We need to mark certain regions of the memory as only accessible to the kernel or to both the kernel and process.
- We want a compact representation of which memory belongs to whom (to save space), so we restrict the types of ranges we can have. For most machines, each of the ranges has to start at a multiple of 4096 bytes, and must be a multiple of 4096 bytes wide. This is called a page size. Note $4096 = 2^{12}$.
- Virtual memory uses units of 4096 bytes aligned at 4096-byte boundaries. These are called pages.
- Pages reduce by a factor of 4096 the maximum size of a memory map. The hardware reduces that even further—fewer addresses to keep track of.
- Two-level page table: Uninteresting parts of memory have no addresses, which reduces the size of the representation even more. Addresses have 32 bits.

  Lower 12 bits: page offset

  Other 20 bits: divided into two 10-bit sections. The top 10 bits are the level 1 page index, the lower 10 bits are the level 2 page index.
- To check permissions for an address, the process reads the address, throws away the page offset (all addresses on the same page have the same permissions), and looks at the current page table register (`%cr3` in x86) which defines at the current state of permissions. Page table is an array of page entries, and is stored in kernel. Use level 1 page index to look up element this page table array. Either the page table address is 0, in which case the address is invalid, or the page table address points to some address in the 2nd page table. Use level 2 page address to find some location in the 2nd page table. If that entry is 0, fault. If it's kernel-only, only the kernel can access it. If it's kernel-or-user, then anyone can access it.
- Aside: 64-bit machines use a 4-level page table
- You can use other data structures, but the two-level page table has several benefits:

  1. In weensyos, the valid addresses are from from 0 to 0x002FFFFF. The first 10 bits are used to access the level 1 page table. These first 10 bits are all 0. Any access of addresses in our specified range go to the same level 2 page table, so just one level 2 page table is needed to hold these addresses. You only need 2 pages of memory to store the page tables—one for the level 1 page table, one for the level 2 page table.
  2. In Linux... there are vast regions of memory that are not accessed, such as the memory between the heap and stack, and before the text. The text/heap area goes from 0x08000000 to 0x09000000, and the first 10 bits of these addresses go from 0x20 to 0x23, so we need 4 level 2 page tables to for these addresses. Then there is another region of blank space (between the heap and stack). To represent the kernel addresses, the level 2 page tables for the kernel can be shared for every process because the kernel address permissions are the same for every process.