This is not the current version of the class.

Kernel 5: Confused deputy attack, scheduling, and process management

Confused deputy attack

A *confused deputy attack* occurs when the attacker has low privilege, but the attacker convinced a privileged deputy to complete the attack on its behalf.

In the context of operating systems, a process is unprivileged, the kernel has full privilege and acts as a privileged deputy by handling system calls. A confused deputy attack may occur if the process, by invoking system calls, can somehow convince the kernel to execute a privileged attack.

Certain system calls are vulnerable to such attacks, but others are not. Very simple system calls like `sys_getpid` usually aren't susceptible to these attacks because they don't change the state of the kernel or other processes at all -- it simply copies the current process's `pid` value to its `%rax` register, and that's it. It's so simple that there is little room for bad things to happen.

Recall that at the end of last lecture, Eve tried to overwrite the `syscall` entry point in kernel memory with an infinite loop. We fixed this by denying access to kernel memory from the user level. Now Eve cannot just directly write to kernel memory any more, but is it possible to for Eve to convince the kernel to write malicious data/code to kernel memory?

Let's take a look at `sys_getsysname`. Could it be used by Eve to perpetrate a confused deputy attack? What if Eve simply does the following:

```
char* syscall_entry_addr = (char*)0x40ac6;
sys_getsysname(syscall_entry_addr);
```

Eve actually manages to crash the entire OS by adding just these 2 lines of code! The kernel did not perform any sanity checks before writing the string containing the OS name to the user-supplied buffer. In this case the buffer happens to point to the `syscall` entry point in kernel memory, and the kernel happily overwrote it. **A successful confused deputy attack!**

> It's worth pointing out that Eve never directly wrote to any kernel memory. The kernel overwrote part of its own memory because the buffer passed by Eve points to kernel memory.

> You may wonder why would a buffer in Eve's address space point to a critical component in the kernel's address space. It is true that when the kernel performs the string copy, it uses Eve's page table to perform address translation (for the destination of the copy). However, note that in DemoOS (and WeensyOS), the part of the page table mapping below `PROC_START_ADDR` (where the `syscall` entry point is located) is shared among all processes as well as the kernel. So this problematic address Eve passes to the kernel will translate into the same physical address, in kernel memory, regardless of which page table (including the kernel's page table) is used for address translation. Additionally, because processes and the kernel also share the same physical pages for mappings below `PROC_START_ADDR`, overwriting instructions there has a global effect, meaning that all process's (and the kernel's) `syscall` entry point code will get corrupted.

Eve is still not satisfied, because this attack blows away all processes in the system, including Eve itself. Eve would like a more targeted attack against *only* Alice. Eve could try, via a confused deputy attack, to corrupt Alice's register file in its process descriptor.

With information about kernel memory layout and Alice's PID, it is possible for Eve to figure out where Alice's process descriptor is located in memory. Eve then once again uses the `sys_getsysname()` system call to corrupt Alice's process descriptor, and after the system call is executed, Alice crashes and only Eve gets to run on the system.

Confused deputy attack can be far more devious and do way more damage than demonstrated. For example, if a malicious process can somehow convince the kernel to turn on certain bits in its page table, it could suddenly gain access to kernel memory and even inspect/change other process's memory. Instead of simply causing the victim to crash, it can passive monitor and steal information from the victim or do more.

To prevent such attacks, the kernel should always perform checks on user-supplied inputs before acting on them. In this example of `sys_getsysname()`, we should make sure that the buffer address supplied by the user is mapped as user-accessible in the process's page table. A safer version of the `sys_getsysname()` handler should look like the following:

```
case SYSCALL_GETSYSNAME: {
    const char* osname = "DemoOS 61.61";
    char* buf = (char*) current->regs.reg_rdi;

    // Check that the entire span of the buffer is mapped
    // as user-accessible
    size_t len = strlen(osname) + 1;
    size_t i = 0;
    for (vmiter it(current, (uintptr_t) buf);
         i < len;
         it += 1, ++i) {
        if (!it.user()) {
            return -1;
        } else {
            // Performs the actual copy
            // Note that the destination of the copy is
            // address-translated using the process's
            // page table.
            *((char*) it.pa()) = osname[i];
        }
    }

    return 0;
}
```

We briefly note here that `sys_page_alloc()` has a similar vulnerability. Eve could request to map a new physical page to the virtual page containing the `syscall` entry point instructions in its address space. It only affects Eve's page table and has no global effect, but it grants Eve control over its `syscall` entry point, and Eve can put arbitrary code there. The next time Eve executes a system call, the inserted code within the newly mapped physical page will run in privileged mode, and it can easily compromise the entire machine. Note that in this case no direct corruption of kernel memory occurred, but a confused deputy attack can still take place.

Fairer scheduling

Recall previously when Eve executes an infinite loop attack, and we deployed defense against it by turning on timer interrupts. Alice still gets to run, but it only gets a small fraction of CPU time. This is because Alice is being "nice" by constantly yielding, but Eve is not being nice and we rely on the timer to take it off the CPU. So what happens is that Eve gets to run a full timer interval, and then Alice simply prints out a message, before yielding back to Eve again.

How can measure niceness (or greediness) of processes, and take it into account when scheduling them, such that the greedy process gets to run less often?

Measure greediness

Signals the kernels are getting about "greediness" of a process come from timer interrupts and yield system calls.

- Yielding is a sign of niceness
- Interrupt is a sign of greediness

We can imagine we maintain a counter per process for its greediness. Negative greediness means the process is nice, positive means the process is greedy. Every time we receive a yield call, we decrement the counter. Every time we receive a timer interrupt, we increment the counter. The counters should also have saturating maximum and minimum values so that this greediness measure is reasonably bounded. The scheduler then picks the process with the least greediness to run in every scheduler tick. The following code snippets show how we may implement this policy in the DemoOS kernel.

In the yield system call handler:

```
case SYSCALL_YIELD:
    if (current->greediness > -100) {
        --current->greediness;
    }
    current->regs.reg_rax = 0;
    schedule();
    break;
```

In the timer interrupt handler:

```
case INT_TIMER:
    if (current->greediness < 100) {
        current->greediness += 5;
    }
    ++ticks;
    schedule();
    break;
```

In the scheduler (`schedule()`):

```
void schedule() {
    while (true) {
        int least_greedy_pid = -1;
        int least_greediness = 100000;
        for (int pid = 1; pid < NPROC; ++pid) {
            if (ptable[pid].state = P_RUNNABLE
                && ptable[pid].greediness < least_greediness) {
                least_greedy_pid = pid;
                least_greediness = ptable[pid].greediness;
            }
        }

        // Run least_greedy_pid if exists
        if (least_greedy_pid > 0) {
            run(&ptable[least_greedy_pid]);
        }
    }
}
```

After we make these changes, Alice gets much more CPU time and Eve gets penalized for being too greedy.

> There are many different scheduling policies, and this very simplistic scheduling algorithm follows what we call a *strict priority* scheduling policy. In this case it is possible that Eve (the greedier process, therefore with strictly lower priority in our system) can get starved and never gets to run again. In a real system we often need more sophisticated scheduling schemes to avoid starvation. Randomness can solve starvation in many cases.

Process management

We are now moving towards a new unit exploring how an operating system can manage processes so that we as users can do useful things with them. Topics in this unit include process life cycle management and communications with and/or between processes.

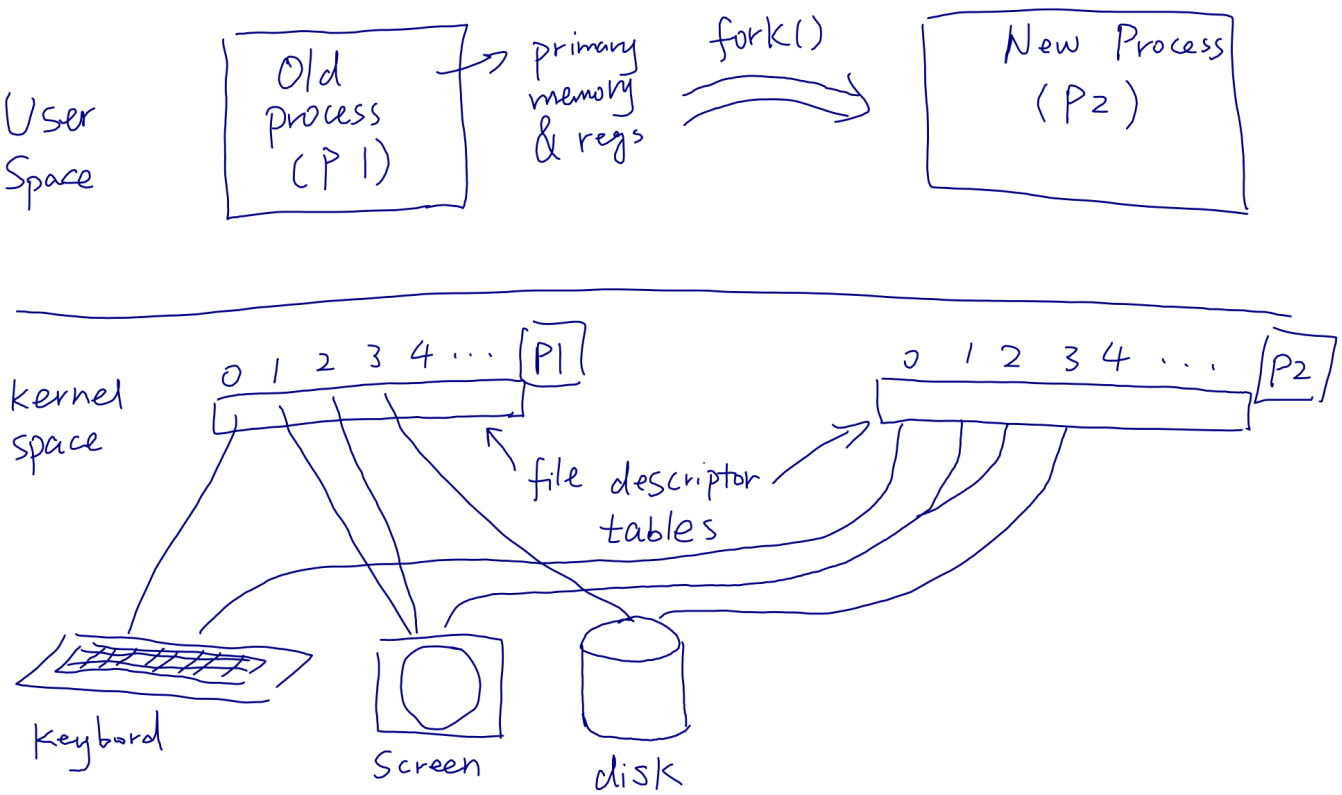
Process creation: the `fork()` system call

A new process can be created by a running process via invocation of the `fork()` system call.

- The new process has a copy of the old process's state (memory and register file)
- The new process and the old process shares the abstract hardware (copies I/O descriptors)
- `fork()` returns 0 to the new process, and returns the new process's ID to the old process invoking `fork()`

The last bullet point above is how one may programmatically distinguish the new process from the old one after `fork()` returns.

> When a process calls `fork()`, it also creates a copy of its file descriptor table in the newly created process. The file descriptors in the new process point to the same underlying devices as they do in the old process. The effect of a `fork()` is illustrated below:


User Space
Old process (P1)
primary memory & regs
fork()
New Process (P2)
kernel space
0 1 2 3 4 ...
P1
0 1 2 3 4 ...
P2
file descriptor tables
Keyboard
Screen
disk