CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 9th, 2011

Hardware foundations of security

Computer resources

Some of the resources that exist on a computer are:

- CPU
  - Registers
  - Memory management unit
  - Interrupt hardware
  - Privileged bit (the ring in x86)
    - Simplest scheme: You are either in privileged mode or not. In x86, you can be in ring0 up until ring4, where ring0 is the most privileged mode.
- Memory
- External storage (hard disk, USB, network)

Assume CPU and external storage "talk" by writing commands into some fixed area of memory.

How do you context switch? Is there a guarantee that the CPU will be yielded? The OS needs to get control back from user programs (that's the goal).

CPU needs **interrupt hardware**. There should be an **interrupt handler address register** where the address of an interrupt service routine can be stored so that the CPU can execute it when an interrupt occurs.

Resource	Need to be privileged
General purpose register	No
Write or read privilege bit	Yes
Interrupt handler address	Yes
Memory management unit	Yes
Program counter/Instruction pointer register	No
Stack pointer	No

How do you go back and forth between privileged and non-privileged mode?

- CPU boots in privileged mode
- Kernel executes in privileged mode
- Applications execute in non-privileged mode
- When context-switching from the OS kernel to an user-land application, the privileged bit will be set to 0

```
setpriv 0
jmp $savedpc
```

We can change the semantics of interrupts as follows. On an interrupt the CPU will:

- set the privileged bit to 1
- save the program counter somewhere
- set the program counter register to the interrupt handler address

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 9th, 2011
What about system calls? We can just treat them as another kind of interrupts.

Memory management (virtual memory)

Computers have physical memory. User-land applications have their own *virtual memory.* Each user-land application *thinks* it has a virtually unlimited memory space (practically limited by the size of a pointer variable). An application will be able to access its own virtual memory starting at address 0 (big no no) and going all the way to 4GB or more depending on the OS.

How does the OS make each application think it has its own virtual memory space? The OS maps a program's virtual memory pages to physical pages in physical memory. These physical pages should never overlap with physical pages that were mapped for other programs.

Examples:

1. **Big no no mapping.** Program 1 and program 2 both have a virtual page mapping to physical page 2. So now program 1 can corrupt program 2's memory, and vice versa.

Program name	Virtual page number	Physical page number
Program 1	Virtual page 0	**Physical page 2**
	Virtual page 1	Physical page 3
Program 2	Virtual page 0	Physical page 1
	Virtual page 1	**Physical page 2**

2. **Yes, yes! Good mapping.**

Program name	Virtual page number	Physical page number
Program 1	Virtual page 0	Physical page 3
	Virtual page 1	Physical page 2
Program 2	Virtual page 0	Physical page 0
	Virtual page 1	Physical page 1

How do (virtual) memory addresses get translated to physical memory addresses?
There are a variety of schemes that implement virtual page mapping and translation. For now, it suffices to know that the (virtual) addresses go through the TLB and get converted physical addresses.

The TLB will map virtual addresses (virtual pages more precisely) to physical addresses and their associated permissions.

The CPU has an instruction: `ldtlb index, vaddr, paddr, perms`. Obviously, this is a privileged instruction since it could allow any application that executes to access any area in memory by just adding a TLB mapping.

The TLB is just a cache of the **page table.** The page table is a full mapping of all the virtual pages of a program to their corresponding physical pages. Where can you find the virtual page though? In the **process control block (PCB)**, which lives in kernel memory and is not readable or writable by applications. The PCB for a process has a pointer to that process' page table.

Virtual memory **allows processes to talk rapidly by sharing memory**. If done with care, two applications can actually share a physical page (the previous big no no is now a yes yes). Access control can be set in order to give different processes different permissions. One process could read the page, the other one could write the page.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 9th, 2011

Stack overflow attack

They occur in C programs, pretty much exclusively.

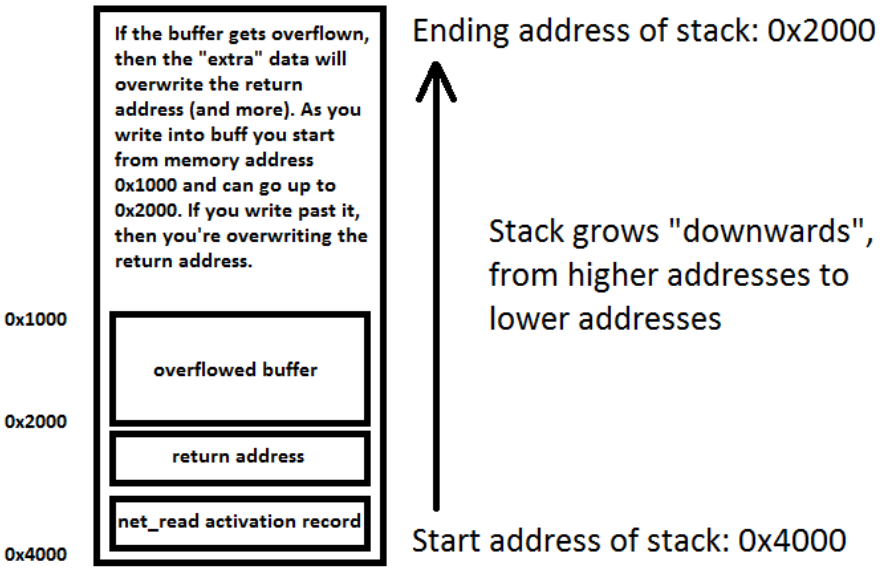
```
void net_read(int netfd) {
        char buff[2048];

        // reading into the buffer past its capacity
        read(netfd, buff, 4096);
}

void net_read(int netfd) {
        char buff[80];

        // reading into the buffer past its capacity (0x80 > 80)
        read(netfd, buff, 0x80);
}
```

After the call to net_read, the stack will look like this:



When `buff` gets overwritten, the return address gets overwritten by the attacker's choice so he can execute arbitrary code.