CS50300: Operating Systems
Lab1 Answers

Zichen Wang
wang4113@purdue.edu

September 5, 2018

1 Objectives

2 Readings

3 Running XINU

3.1

An alternative way to 'idle' is to call **halt()** defined in **intr.S**. The assembly code of 'halt' only jumps to itself, which means looping forever.

3.2

The **welcome()** function is under **system/welcome.c**.

4 XINU system calls

4.1

The C type of syscall, defined in **include/kernel.h**, is **int**.

4.2

If the parent process has terminated, the process table field **prstate** should be *PR_FREE*.

Question: What should we do if the parent process has terminated before the child process?

In x86, when a process terminates, its child process will change the parent to 'init' process. However, I cannot find such operation in system/kill.c.

5 XINU's run-time environment

5.1 Changing byte order using assembly code

The instruction **bswapl** can change byte order of a 32-bit register directly.

I rename the file and function name of the C function version to **revbyteorder_cfun**, in order to avoid conflict with **revbyteorder.S** when testing.

5.2 Checking segment boundaries

The addresses of the end of the text, data, and bss segments of the XINU OS are defined by $\&etext - 1$, $\&edata - 1$ and $\&ebss - 1$ respectively.

Hints from initialize.c.

5.3 Run-time stack: process creation and function call

We will use the inline assembly code to print the top of run-time stack, by copying the value of the register **%esp** and the content inside **(%esp)** into two variables.

The address and content of the top of the run-time stack inside main() will remain the **same** at the time before myprogA() is created and after myprogA() has been created and resumed. It is because myprocA() is a new independent process, and able to run along with main() process. The stack of myprocA() will be created in the free memory area when the process spawns and will **not** change the stack structure of its parent process. Therefore, in these two printing, main() process's stack and main() function's stack frame are being printed, and the results are same.

When myprogA() calls the function myfuncA(), things will be different. Before myfuncA() is called, myprocA() process's stack and myprocA() function's stack frame are being printed. Then, myprogA() will use its own stack to maintain a function call to myfuncA(), pushing the arguments of myfuncA() and return address to the stack. After that, myprogA() will jump to myfuncA(), but the process remains unchanged. In function myfuncA(), the stack belongs to myprogA() process. So, after myfuncA() is called, myprocA() process's stack and myfuncA() function's stack frame are being printed.

5.4 Comparing two run-time stacks

6 Hijacking a process via stack smashing

The strategy for finding the return address of sleepms() function is very simple. In myfuncA() process, it is easy to get the process id of myprogA() process, i.e. PPID of myfuncA() process. Then, we use the PPID to retrieve the stack parameters of myprogA() process and the whole stack from *prstkbase* to *prstkptr* can be printed. There are too many addresses at the first glance on the stack information, but the system call *sleepms* has a paramater '3000'. From the system perspective, when a process make a system call or function call, it will push the return address just following the arguments of the function. According to this convention, I find that '0x00000BB8'

('3000' in hex) in the stack and the content under '0x00000BB8' is just the return address of *sleepms* function. In my version, the return address is '0x00103491', and the location of the return address is $prstkbase - 52$.

However, things will not finish. In order to let myprocA() process exit normally, i.e. its resources can be released and other process with lower priority can continue to run, myprocA() process should return to *INITRET* address after running the malware code. From the convention, the process will use **ret** instruction to pop the return address to **eip** register. Therefore, we may put the *INITRET* address before the malware code address in the stack. The *INITRET* address is located in $prstkbase-4$ defined by *create.c*, so in my version $*(int*)(prstkbase-48) = *(int*)(prstkbase-4)$ will work.

7 Bonus problem

As detailed above, we may put any address before the malware code address in the stack to control myprocA() process's **eip** register when returning from the malware code. In my version, to minimize the disruption on myprocA() process, we should move the normal address from $prstkbase - 52$ to $prstkbase - 48$, and then put the malware address to $prstkbase - 52$. By doing so, the context will return to myprocA() process normally after running the malware code, and continue to print its stack parameters.