CS 503 Lab 1 Answers

Prashant Ravi — ravi18@purdue.edu

February 5, 2016

1 Part 4.3

Explanation for the printed out stack addresses: Here we see that on conducting the following:

Print the address and content of the top of the run-time stack of the main() process before myappA() is created
main - before creating myappA
stack top :(0xefd8fd0)
stack base:(0xefd8fe8)
0x0000 0fa0(0x0123457) < − top of the stack

Print the address and content of the top of the run-time stack after myappA() is created but before the myappA() process is resumed.
stack top :(0xefd8fc0)
stack base:(0xefd8fe8)
0x0010 22bd (0xefd8fc0) < − top of the stack

Comparison and Inference So, as we see 16 bytes have been reduced from the stack, which gives us a clear indication that create method is not running on the user space stack, as we would've expected more than 16 bytes to have been reduced from esp register. The fact is that create is a system call and it is executing on the kernel stack.

Print the address and content of the top of the run-time stack in the myappA() process before myfuncA() is called
stack top :(0xfdeffa8)
stack base:(0xfdefff0)
0x0001 1207c (0xfdeffa8) < − top of the stack

Print the values after myfuncA() is called but before it returns ESP is 0xfdeffa8, its content is 0x111f28 and its EBP is 0xfdeffb0
stack top :(0xfdeffa8)
stack base:(0xfdeffb0)

0x0001 1207c (0xfdeffa8) < − top of the stack

Print the values after myfuncA() returns
stack top :(0xfdeffb8)
stack base:(0xfdefff0)
0x0000 0040 (0xfdeffa8) < − top of the stack

Comparison and Inference From this we can infer that myappA is not running on the same process stack as the main function because their ESPs differ by a lot. So myappA has it's own process stack. However, we also notice that myfuncA() runs in the same process stack as myappA since its stack pointer during execution is in the vicinity of ESP of myappA after it returns, however we know that a 'call' was made because the EBP of myfunc during execution is different from the EBP after myfunc returns. Hence, this validates the claim that myappA is a separate process with separate process stack than main but myfunction is a sub-routine/procedure function that runs in the same stack space as myappA.

2 4.4

The approach to satisfying the hijacking requirement is described below:
The myhacker method gets its quantum time slice to run when the makevictimsleep(3) is invoked that puts the "myvictim" process to sleep. Before "myvictim" wakes up myhacker finds and saves makevictimsleep's stack frame's return address, X , by climbing up the stack using "myvictim" process's base pointer, and changes it to address of myhackermalware code +3, present in the code segment. (The reason for +3 is that we would like to avoid the stack frame construction instructions like pushing old ebp and moving esp into ebp). Now, when the makevictimsleep() resumes it would exit its stack frame normally but due to the return address being altered would run myhackermalware's code. myhackermalware changes victimglobal's value and executes a jmp instruction to jmp back to address X that we had previously stored, which essentially is the address makevictimsleep was supposed to return to, if hacker code was absent, i.e the code returns to invokevictimsleep() and then the code normally returns to the myvictim() method and then exits it, killing the process. We can verify that the myvictim process was incognizant to the activities of the myhackermalware because the computations performed in invokevictimsleep() and makevictimsleep() print correctly. Thus, this was the approach employed and it turned out quite successful.

3 Part 5: Bonus Problem

The code for stackdepthlinux.c is under system directory usage/running instructions are in that file. The only difference between the unix version and the linux version is that the deepest base pointer address(ebp) i.e is 0x0 in linux, whereas in Unix its not so we need to know prior hand the base address of the process at hand by looking up the procent data structure. The result was verified to be correct because when factorial.c was run in linux and unix both returned a stack depth of 10.