Lab1Answer

Nitin A Jain

jain117@purdue.edu

Prob 4.2

Stack depth is found by backtracking from one stack frame to the earlier until we reach the base stack of the process. This is done in a while loop where the condition is that the address is less than the base stack address of the process which is stored in proctab data structure in process.h, and the iteration step is to read the content in the address referenced by the topbp and then updating topbp with the content at that address.

Prob 4.3

The output of 4.3 is :

currpid : 2

Before myappA is created

run-time stack top : address = 0xEFD8FC0, content = 0x11203C

After myappA is created but before resuming it

run-time stack top : address = 0xEFD8FC0, content = 0x112244

currpid : 3

main's pid : 2

main's esp(prstkptr) : efd8f64

Before myfuncA is called

run-time stack top : address = efc8fd4, content = 1122e6

currpid : 3

While myfuncA is called and before it has returned

run-time stack top : address = efc8fc4, content = efc8fd4

After myfuncA has returned

run-time stack top : address = efc8fd4, content = 112301

As we can see, the stack pointer address doesn't change value (address = 0xEFD8FC0) before and when a process is created. The content changes slightly as the value of local variable myapp_pid is modified on process creation. After myappA process is resumed, main process is context switched out and myappA occupies the runtime stack. Thus the address of top of runtime stack changes and so does its content.

Since the process has context switched to myappA, the pid will change now. Now in the myappA, before the myfuncA function call the top of stack has address = efc8fd4. Once the myfuncA() is called the top of the stack address is changed since there has been a function call but the process Id doesn't change as it is just a function call and thus we are still in the same process. On returning from the myfuncA function, we see that the address of the top of the stack is same as before calling myfuncA(). Thus, it is clear that the stack frame of myfuncA is removed and we get back to the point where we had left in myappA().

This example gives a clear difference between a function invoked as process or as a function call from the OS perspective.

Prob 5

The function myhackermalware() is called by modifying the return address of one of the functions in the call stack of myvictim2() which internally calls the sleepms function. Since, the call to myhackermalware() is made by modifying the return address in the stack frame, what happens is the the EIP is assigned to myhackermalware() function start address and this is why the myhackermalware() function gets executed. Now, since the myhackermalware() was not called as a function there was not a well defined stack address associated with this functions execution, and thus on the completion of it's execution there is not a valid return address. This in turn means there is no valid next execution address for EIP and thus this causes the error we see in the output. The victim process is thus never executed as the some interrup flags are triggered and the system goes into a panic state, similar to the halt state.

We can setup the attack by managing the stack well by enabling the myhackermalware() to basically mimic a callee to caller control transfer by correctly handling the stack frames and the assigning the EIP to exactly to the next instruction after the call to the callee function.

I tried doing this but the stack frame handling became difficult to do and there were a few more bugs which was the reason I couldn't code a mechanism to do an oblivious attack.

The way, I went about hijacking the victim is I discovered the stack trace using the stacktrace() system call. Using this I found that there were a number of system calls following the sleepms

call. I found a stackframe of a function of the victim before the sleepms and I changed the return address in it’s stack frame with the pointer to the myhackermalware() function. This is a simple way and I have explained how it works in the earlier paragraph.