Practice Problem 10.1

A program that contains a user-defined function and a function call. The program takes a user input on the command line that is copied onto the function’s stack frame at address ebp-120. Using the debugger and pausing execution of the program within the function produces the following information:

(gdb) i r eip ebp esp
   eip 0x0804834a 0x0804834a
   esp 0xbffff700 0xbffff700
   ebp 0xbffff7a0 0xbffff7a0

(a) How many bytes are on the function’s stack frame?

\[ \text{Bxffff7a0} - \text{bffff700} = \text{a0} \] (160 bytes)

(b) Suppose a hacker is attempting to perform a buffer overflow attack on this program. If the buffer overflow attack consists of running the program with a command line argument comprised of 80 NOPs followed by a 64-byte exploit, followed by an appropriate malicious return address repeated 20 times, would you expect this attack to work? Why or why not?

\[ 80 + 64 + (4 \times 20) = 224 \text{ bytes total. The NOP sled + Shellcode = 144 bytes, which will fill the lowest 120 bytes of the stack plus 24 bytes, overwriting the prior ebp and return address with shellcode. The repeated return address will overwrite main’s stack frame. This attack will fail, as the return address will not be the malicious return address, but instead be shellcode. So this will probably segfault.} \]

(c) If the buffer overflow attack consists of running the program with a command line argument comprised of 20 NOPs followed by a 64-byte exploit followed by repeating 0x0804834b twenty times, would you expect this attack to work? Why or why not?

\[ 20 + 64 + (4 \times 20) = 164 \text{ bytes total. The NOP sled + Shellcode = 84 bytes, which will load completely into the function stack frame, leaving 36 bytes. The repeated return address will fill those 36 bytes, plus overwrite the prior ebp and return address. However, the chosen malicious return address points to the next instruction following the current eip in the function. That is, it does not point to the NOP sled or the exploit code. So this attack will fail. It might produce an infinite loop, where the function keeps repeating (it returns to itself).} \]

(d) If the buffer overflow attack consists of running the program with a command line argument comprised of 10 NOPs followed by a 64-byte exploit followed by repeating 0xbffff700 twenty times, would you expect this attack to work? Why or why not?

\[ 10 + 64 + (4 \times 20) = 154 \text{ bytes total. The NOP sled + Shellcode = 74 bytes, which will load completely into the function stack frame, leaving 46 bytes. The first 11 repeated malicious return addresses will fill 44 of those 46 bytes, and the 12th malicious return address will overwrite the last 2 bytes of the function’s stack, plus overwrite the first 2 bytes of the prior ebp. The next 8 malicious return address will continue to overwrite the remainder of the prior ebp, and the actual return address, but will be 2 bytes offset, and won’t exactly overlay start/end of actual return address. The resulting return address will be 0xffbf00f7, (from last 2 bytes of one and first 2 bytes of the next malicious return address), which will fail, and likely segfault.} \]

(e) If the buffer overflow attack consists of running the program with a command line argument comprised of 40 NOPs followed by a 64-byte exploit, followed by repeating 0xbffff740 twenty times, would you expect this attack to work? Why or why not?

\[ 40 + 64 + (4 \times 20) = 184 \text{ bytes total. The NOP sled + Shellcode = 104 bytes, which will load completely into the function stack frame, leaving 16 bytes. The repeated return address will fill those 16 bytes, plus overwrite the prior ebp and return address, leading to a successful attack!} \]
Practice Problem 10.2

Briefly describe two technical solutions that have been proposed to prevent a program from being exploited by a buffer overflow.

Solution:

The technical solutions discussed are:

- System hardening
- The non-executable stack
- The canary
- ASLR