Patrick Lyons

Passwords for Project 2

PHL7\_1:

The password here is euKsfBgfSqVzQZAYXwRxMkBmcMHm. It was found using a similar procedure to lab 1. I learned in lab 1 that the chomp function simply removes the last character of a string input. For all intents and purposes, this simply removes the /n from the input string, so it does not help too much. The interesting part of the stack dump occurs below

  0x080482f5 <+38>: call   0x80493f0 <fgets>

   0x080482fa <+43>: mov    %edi,(%esp)

   0x080482fd <+46>: call   0x80482a0 <chomp>

   0x08048302 <+51>: mov    $0x80b388c,%esi

   0x08048307 <+56>: mov    $0x1d,%ecx

   0x0804830c <+61>: repz cmpsb %es:(%edi),%ds:(%esi)

The first line is the fgets, which is where the user entered a password. Below that is a move call, where it is moving something from %esp to %edi. I inserted a breakpoint here and examined edi, and found it to be the password I had entered (along with a \n). The function calls chomp to remove the \n, and then performs 2 more move functions before running what appears to be a comparison between my password (stored at %edi) and something else (stored at %esi). I inserted a breakpoint at 0x0804830 and examined %esi and %edi. Being that there was a move function involving %esi, I believed this register could hold the password I needed. I examined this register and found the password above. Testing it, it worked!! Side note, when running the compiled form of mystrings.c, I found this password right in the middle of the output. Maybe other passwords are stored in the output of mystrings as well…

PHL7\_2:

The password here is my username (phl7). When debugging, I found the below stack in a disassemble, I noticed the fgets command. I put a breakpoint after this and began checking variables I had seen until I found the password I entered in %ebx. There were a few more calls below and I placed more breakpoints in between them, continuing to check registers to try and find something that resembles a valid password. I finally placed a breakpoint before the strcmp function. When I checked the contents of %ebx (my password), it had been updated to contain 2 at the end of it. I saw an add call above at 0x080485d1, and figured it was adding the 2 here. When I checked the contents of %esi, the output was phl7\_2. I figured that it must take the password from the user, add 2, and test it against phl7\_2. My password is then phl7. phl7.c was found in mystrings output. This is confirming my theory that mystrings can show a list of possible passwords… or maybe partial passwords?

0x08048597 <+80>: call   0x8048388 <fgets@plt>

   0x0804859c <+85>: mov    %ebx,(%esp)

   0x0804859f <+88>: call   0x80484bf <c>

---Type <return> to continue, or q <return> to quit---

   0x080485a4 <+93>: mov    %ebx,0x4(%esp)

   0x080485a8 <+97>: lea    0x18(%esp),%ebx

   0x080485ac <+101>: mov    %ebx,(%esp)

   0x080485af <+104>: call   0x80483a8 <strcpy@plt>

   0x080485b4 <+109>: mov    %ebx,%edi

   0x080485b6 <+111>: mov    $0x0,%eax

   0x080485bb <+116>: mov    $0xffffffff,%ecx

   0x080485c0 <+121>: repnz scas %es:(%edi),%al

   0x080485c2 <+123>: not    %ecx

   0x080485c4 <+125>: lea    -0x1(%ebx,%ecx,1),%eax

   0x080485c8 <+129>: movw   $0x325f,(%eax)

   0x080485cd <+134>: movb   $0x0,0x2(%eax)

   0x080485d1 <+138>: add    $0x1,%esi

   0x080485d4 <+141>: mov    %esi,0x4(%esp)

   0x080485d8 <+145>: mov    %ebx,(%esp)

   0x080485db <+148>: call   0x80483d8 <strcmp@plt>

PHL7\_3:

The password here is any combination of 9 or 10 characters, as long as exactly 3 vowels (not including y) are present. Running mystrings on the file did not give anything that resembled a password, which debunked my earlier theory.

I tried some random passwords, and some actually worked… But not in the way I expected. This code was not easy to debug either, as main did not exist in it. I tried the password randomcharacters, and it worked… But in a weird way.

The success statement was “Congratulations!\nUnlocked with passphrase randomchar”… But how? My passphrase was randomcharacters. I quickly realized that the code with cut out any characters past 10, and test only the first 10.

I tried a few more passwords of varying length to be sure, and soon found that the program will also accept passwords of length 9, so all I know about this program is that it’ll accept passwords of length 9 or 10, and for some reason, accepts the content of randomchar.

I ran info file in gdb to get a list of what was in the code. I noticed that the .text portion was stored between addresses 0x080483a0, and 0x080485dc (seen below).

0x080483a0 - 0x080485dc is .text

I disassembled this block of code, and that’s where the fun really started. I noticed a line that seemed odd to me.

0x080484c2: test    %eax,%eax

This line occurs about halfway down the .text dump. All it is doing is testing eax with itself. I placed a breakpoint above this and tried to view the contents of registers that had occurred above (eax, esp, edi, esi ebx etc.), and found a single character ‘r’ in esp the other registers either gave me an out of bounds error, or just strings of numbers. I continued from here and ended up at the same address in memory. I tried examining registers again, and esp changed to ‘a’. After a few more times, I realized it was cycling through the password that I had input one character at a time, doing something with it, and moving to the next letter.

The last time esp is referenced before this breakpoint is below.

=> 0x080484a0: mov    %eax,(%esp)

   0x080484a3: call   0x8048388 <tolower@plt>

   0x080484a8: sub    $0x61,%eax

---Type <return> to continue, or q <return> to quit---

   0x080484ab: cmp    $0x14,%eax

   0x080484ae: ja     0x80484cb

It seems to take my string, call a tolower function, subtract 0x61, compare it to 0x14, and then jump. On an ascii table, 0x61 is ‘a’, and 0x14 is a device control. That doesn’t help much, but the function then jumps to 0x80484cb… Let’s set a break point there.

0x080484cb: addl   $0x1,-0xc(%ebp)

   0x080484cf: cmpl   $0xa,-0xc(%ebp)

   0x080484d3: jle    0x8048492

   0x080484d5: cmpl   $0x3,-0x10(%ebp)

   0x080484d9: jne    0x80484f1

   0x080484db: mov    $0x8048604,%eax

Here, it seems to be adding 1 to ebp, so it’s incrementing ebp based on how many chars it’s read in, and then comparing it to 10… If it’s read in 10 or less characters, it jumps back to 0x804892, and if it’s read 11, it falls through to another comparison, where its comparing

-0x10(ebp) to 3… It’s looking for a value, stored at some address offset 16 bytes away from ebp, comparing it to 3, and jumping to something else if it’s not equal.

I looked for a part of the code where -0x10(ebp) was being incremented, and found it below.

0x080484c6: addl   $0x1,-0x10(%ebp)

I ran through my program with the same breakpoints, and adding one in here at this incrementation, to see what char being passed to the function causes another increment.

The when I checked esp, the program kept jumping back to the same text lines to iterate the count of the string I passed, but, it would move to the second breakpoint whenever the value at esp was (in my password “randomchars”) a, o, and then a. Was it accepting vowels?

I began trying passwords that contained 3 vowels, and 7 other characters that were not vowels. With the same result. It kept jumping to the second breakpoint (the incrementation line above), before returning to the first breakpoint that I had examined before (the test of %eax and %eax). I determined therefore, through a few more tests (the entire alphabet, for one), that the password requirement is either 9 or 10 characters, with exactly 3 vowels.