Bryan Heim – Project 2 – unlocking bph11_2

Correct Passphrase: a palindrome that is longer than 6 characters.

Path to solution: Once again the first thing I decided to do is see what strings returned when I ran it on bph11_2 which this time returned only 13 results and most of which were function calls such as printf, fgets, puts, and __libc_start_main. I naively tried to use the two odd strings, PTRhp and QVhZ, but neither worked so I decided to go use gdb. I set a breakpoint at main and examined the code.

```
   0x0804855a <+0>:     push   %ebp
   0x0804855b <+1>:     mov    %esp,%ebp
=> 0x0804855d <+3>:     and    $0xfffffff0,%esp
   0x08048560 <+6>:     call   0x80484e8 <d>
   0x08048565 <+11>:    mov    %ebp,%esp
   0x08048567 <+13>:    pop    %ebp
   0x08048568 <+14>:    ret
```

I noticed that there was no fgets and instead a function call to d, so I set my breakpoint at d and went to examine that code. Once I disassembled d()'s code I noticed the fgets, printf, and puts function so I knew I was in the part of the program that did the checking. I also noticed three functions c (), r (), and s (). After fgets is called, I noticed it puts the contents into $ebx and moves that onto the stack at $esp. I verified my input was stored in ebx, which it was.

```
(gdb) x/s $ebx
0xffffd15c:      "attempt\n"
```

Immediately after fgets grabs the input, it calls function c. So I set my breakpoint and went to examine c's code. Inside of the code I saw multiple function calls to s () and at the beginning after the normal stack incrementing I noticed it also pushed ebx (our input) onto the stack and moves the stack point. It then does a test $ebx, $ebx to check if nothing was inputted, since it was we continue down c () code and notice that it checks for the /n and if it find the newline it gets rid of it. I noticed c () is similar to the chomp function from the bph11_1 and after returning from c () checked ebx to verify.

```
(gdb) x/s $ebx
0xffffd15c:      "attempt"
```

Inside of the r function I had saw one function call to s(), the introduction of the %esi and %edx registers, and a lot of use of the stack. Right before the function call to s(), I went to examine the contents of the registers after the stack manipulations had finished. To my surprise they were both my input.

```
(gdb) x/s $esi
0xffffd15c:      "attempt"
(gdb) x/s $ebx
0xffffd15c:      "attempt"
```

Once inside the s () functions I noticed what appeared to be a loop. It had moved zero into the eax register and would then test my input against 0x0. If they were equal, it would pop and return out of the function. If not it would add one to eax and then compare against zero.

```
 0x08048454 <+16>:    add    $0x1,%eax
 0x08048457 <+19>:    cmpb   $0x0,(%edx,%eax,1)
 0x0804845b <+23>:    jne    0x8048454 <s+16>
```

This command appears to be walking through each byte of edx and comparing to see if the byte at edx[eax] is 0. I assumed after seeing this that the function would loop through a string and when it is finished eax should contain the length of the string because it would have hit a “0” or null terminator at the end of the string. I set my breakpoint right before the function returned to r () and checked eax. It contained 0x7 which is exactly the length of my test input “attempt”.

```
(gdb) x/s $eax
0x7:     <Address 0x7 out of bounds>
```

After it returns back to r (), it test %dl,%dl, to check if its zero, and moves down to the end of r () where it moves 0x1 into eax and moves the stack point. It moves some of the contents from ebx (the input) into edx so I set a breakpoint right at the jump and tested to see what was moved into edx.

```
   0x080484ac <+17>:    call   0x8048444 <s>
=> 0x080484b1 <+22>:    movzbl (%ebx),%edx
   0x080484b4 <+25>:    test   %dl,%dl
   0x080484b6 <+27>:    je     0x80484dc <r+65>
```

And after checking when breakpoint is at 0x…b4,

```
 (gdb) x/s $edx
 0x61:    <Address 0x61 out of bounds>
```

I noticed that the hex number 0x61 was moved which corresponds to decimal number 97. I moved onto the next statements to find it change the contents of $eax and compared it with $dl which still contained 0x61. Examining the contents of $eax I found that it appeared to only take the last letter. At this point I was thinking that the last letter of the input had to be equal to “a” which in ascii is 97 which was also the first letter of my attempted input. I had then quit and started gdb again and instead used the test word “attempa” replacing the t with an a, and when I got back to the same point, it was taking the last letter.

```
(gdb) x/s $dl
0x61:    <Address 0x61 out of bounds>
(gdb) x/s $eax
0xffffd162:      "a"
```

Now I could continue with the two values being compared equal. Next following the instructions, I saw another move and add to ebx. After the statement was ran, I noticed that it had taken off my letter. Which led to another comparison. This time between 0x74, which is responsible for the letter “t”, and the contents of %eax which contained “pa” the last two letters. This led me to believe that it is going to compare the contents of the first letter to the last and that there had to be a 0 before the last letter for the next comparison to work. So I tried a run through where with the test input “0000a” and saw that it hit a problem trying to compare “a” with 0x30 (“0”). This gave me the great idea that the passphrase must have the same starting and ending letter. For my last trace through I used the test string “a0000a” which passed all of the comparisons but when it left the function r (), it would call the function s () to get the length of the string. After the call to s (), I saw that it would check to see if the length was less than or equal to 6, so from this it finished running with a correct answer, “a0000a”. From this I tried to guess

the rules for which the passphrase must follow, mainly that it must be greater than 6 characters and the first letter and last letter must match. I test a few different strings and achieved the same unlock.

```
(27) thot $ ./bph11_2
a11111a
Congratulations!
Unlocked with passphrase a11111a
(27) thot $ ./bph11_2
g6666666666666h
Sorry! Not correct!
(27) thot $ ./bph11_2
g6386758458645784g
Sorry! Not correct!
(27) thot $ ./bph11_2
g434634g
Sorry! Not correct!
(27) thot $ ./bph11_2
g46346w4360g
Sorry! Not correct!
(27) thot $ ./bph11_2
g00000g
Congratulations!
Unlocked with passphrase g00000g
(27) thot $ ./bph11_2
g11111g
Congratulations!
Unlocked with passphrase g11111g
(27) thot $ ./bph11_2
g2222222g
Congratulations!
Unlocked with passphrase g2222222g
(27) thot $ ./bph11_2
g23524623g
Sorry! Not correct!
```

After running a few test through it appears that it must have an integer in-between the first and last letter and that whatever is in-between those two letters must be the same. I came to the conclusion that the passphrase was a string, longer than 6 characters, where the first and last chars are the same and all of the middle chars in-between the first and last must match. But then I had the idea to test a the palindrome “racecar” because it would make sense that if it compared the first and last letters, that it would do so in a looping style that would check all the letters. From this I arrived at my final answer. That indeed, the passphrase had to be a palindrome that was at least 7 characters or longer.

```
(12) thot $ ./bph11_2
racecar
Congratulations!
Unlocked with passphrase racecar
(13) thot $ ./bph11_2
nommon
Sorry! Not correct!
(13) thot $ ./bph11_2
nomymon
Congratulations!
Unlocked with passphrase nomymon
```