Bryan Heim – Project 2 – unlocking bph11_3

Correct Passphrase: A string of 9 or 10 characters, that contains a substring of 6 characters in a row that made up of either “t” or “T” ‘s. (e.g abttttttea, or abTTTTTTea, etc)

Path to solution: For the last program I again decided to start with a strings bph11_3 command. Upon examining the results, I had found a similar pattern as bph11_2 and concluded that I would need to use gdb from the start. I did however notice the function calls puts, tolower, printf, getchar, and __libc_start_main were all used.

I opened the program in gdb like normal and went to set the breakpoint at main() and to my surprise, received an error saying no main function was found.

```
(gdb) b main
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

I used the command “info file” to open up the header off bph11_3 and take a look at its contents.

```
(gdb) info file
Symbols from "/u/SysLab/bph11/bph11_3".
Local exec file:
        `/u/SysLab/bph11/bph11_3', file type elf32-i386.
        Entry point: 0x80483a0
        0x08048134 - 0x08048147 is .interp
        0x08048148 - 0x08048168 is .note.ABI-tag
        0x08048168 - 0x0804818c is .note.gnu.build-id
        0x0804818c - 0x080481ac is .gnu.hash
        0x080481ac - 0x0804822c is .dynsym
        0x0804822c - 0x0804828d is .dynstr
        0x0804828e - 0x0804829e is .gnu.version
        0x080482a0 - 0x080482c0 is .gnu.version_r
        0x080482c0 - 0x080482c8 is .rel.dyn
        0x080482c8 - 0x080482f8 is .rel.plt
        0x080482f8 - 0x08048328 is .init
        0x08048328 - 0x08048398 is .plt
        0x080483a0 - 0x080485dc is .text
        0x080485dc - 0x080485f8 is .fini
        0x080485f8 - 0x08048651 is .rodata
        0x08048654 - 0x08048678 is .eh_frame_hdr
        0x08048678 - 0x080486f4 is .eh_frame
        0x080496f4 - 0x080496fc is .ctors
        0x080496fc - 0x08049704 is .dtors
        0x08049704 - 0x08049708 is .jcr
        0x08049708 - 0x080497d0 is .dynamic
        0x080497d0 - 0x080497d4 is .got
        0x080497d4 - 0x080497f8 is .got.plt
        0x080497f8 - 0x080497fc is .data
        0x080497fc - 0x08049804 is .bss
```

Inside I noticed that the .text segment was 0x23c large and that this is where the actual code would exist. I used the two provided addresses to disassemble the code that was at those points. Inside I saw that it returned all of the code used by the compiled c program. There were clusters of code separated by the nop command and in-between the nop commands there were the usual stack pushes. After code

dealing with the __libc_start_main code, I got to what I was looking for. The function call getchar which told me that this is where the actual testing needed to happen.

```
0x0804846b:  mov    -0xc(%ebp),%ebx
0x0804846e:  call   0x8048338 <getchar@plt>
0x08048473:  mov    %al,-0x1b(%ebp,%ebx,1)
0x08048477:  addl   $0x1,-0xc(%ebp)
0x0804847b:  cmpl   $0x9,-0xc(%ebp)
0x0804847f:  jle    0x804846b
```

When I examined this code, I set a breakpoint to the second move statement and found the contents of ebx to be 0, I then noticed this code was a loop around the getchar method in the c source code and that it would go 10 times. After that the code jumped around and eventually I was able to find the bit of code where it seemed to move the ASCII value of my test input “c”, into the register eax. After which it moved eax to the esp, and went into the tolower function. After it subtracted 0x65 and compared my answer to 0xf before evaluating a jump. Right from the start I added the two numbers to get 0x74 which when subtracted by 0x65 = 0xf, this of course mapped to the character “t”. I set a breakpoint after the jump but using the character “c” I never reached it and the program ended. So now I decided to rerun the program using the test letter “t”. When I reached the same point, it didn’t end and the program continued letting me know I was in the right direction.

```
0x080484b0:  mov    $0x1,%edx
0x080484b5:  mov    %edx,%ebx
0x080484b7:  mov    %eax,%ecx
0x080484b9:  shl    %cl,%ebx
0x080484bb:  mov    %ebx,%eax
0x080484bd:  and    $0xe281,%eax
0x080484c2:  test   %eax,%eax
```

It seemed to then go through and take the remainder, shift it, mask it, and compare it with zero. So I decided to see what would make $eax zero if masked with 0xe281. The binary for this string is 1110001010000001 and since it only takes 32 bit registers, the only thing saved is the first byte and the 8th bit. If the result in eax is zero, it adds ones to a specific -0x10 shift of the stack, it then goes back and repeats for the next letter in the original loop.

```
0x080484c6:  addl   $0x1,-0x10(%ebp)
0x080484ca:  nop
0x080484cb:  addl   $0x1,-0xc(%ebp)
0x080484cf:  cmpl   $0xa,-0xc(%ebp)
0x080484d3:  jle    0x8048492
0x080484d5:  cmpl   $0x7,-0x10(%ebp)
0x080484d9:  jne    0x80484f1
```

At this point I set a breakpoint for where the message is loaded for success and the breakpoint never reached and instead I received the sorry incorrect message. I used what knowledge I had and took a guess that the passphrase would be a string that is 10 characters long that would contain 6 “t” ’s or maybe even t or lower in the alphabet. So I went back to thot and ran bph11_3, and used the code “attttttttea” as my attempt. Surprisingly this unlocked the puzzle.

```
(6) thot $ ./bph11_3
atttttttea
Congratulations!
Unlocked with passphrase atttttttea
```

I next decided that I would try this with combinations of upper and lowercase letters (because the tolower function was used) as well as letters lower in the alphabet than t (..w,x,y,z). The combinations I used did not unlock for any passphrase that did not include a “t” or “T”. Also the program would not unlock if all that was inputted was all “t” ‘s, and if the “t” ‘s were either at the beginning or end.

```
(10) thot $ ./bph11_3
attttttttea
Sorry! Not correct!
(11) thot $ ./bph11_3
atttttttea
Congratulations!
Unlocked with passphrase atttttttea

(11) thot $ ./bph11_3
aTTTTTTea
Congratulations!
Unlocked with passphrase aTTTTTTea

(11) thot $ ./bph11_3
gzzzzzzea
Sorry! Not correct!
(11) thot $ ./bph11_3
azzzzzzea
Sorry! Not correct!
(11) thot $ ./bph11_3
atttuuuea
Sorry! Not correct!
(11) thot $ ./bph11_3
tttttt
a
a
Sorry! Not correct!
(11) thot $ ./bph11_3
ttttttttttt
Sorry! Not correct!
(11) thot $ ./bph11_3
katttttta
Sorry! Not correct!
(11) thot $ ./bph11_3
atTtTtTea
Congratulations!
Unlocked with passphrase atTtTtTea
```

The program also worked with other leading chars or if all of the “t” ‘s were at the beginning or end. However all of the “t”’s whether upper or lowercase had to be in a continuous sequence or else it also did not work.

```
(11) thot $ ./bph11_3
bttttttea
Congratulations!
Unlocked with passphrase bttttttea

(11) thot $ ./bph11_3
ttttttea
Sorry! Not correct!
(11) thot $ ./bph11_3
tttttteab
Congratulations!
Unlocked with passphrase tttttteab

(11) thot $ ./bph11_3
gttbttbtt
Sorry! Not correct!
(11) thot $ ./bph11_3
eabttttt
Congratulations!
Unlocked with passphrase eabttttt

(11) thot $ ./bph11_3
eabttttta
Congratulations!
Unlocked with passphrase eabttttta
(11) thot $ ./bph11_3
eabtttttteba
Sorry! Not correct!
```

From this I was able to reach my conclusion. The passphrase could be 9 or 10 characters in length and must contain a sequence of either “t” or “T” in a continuous 6 char substring.