8

Lab 6: Removing NULL bytes

This short assignment will give you a little practice removing `NULL` *bytes from an assembly program.* `NULL` *bytes will prevent shellcode from being able to pass unmolested through C string functions. Therefore, we remove them to make our attacks more robust.*

8.1 *Learning Goals*

In this lab, you will practice:

- producing assembly code from C code;
- producing object code from assembly code;
- examining object code for `NULL` bytes; and
- searching for alternative assembly instructions that do not produce `NULL` bytes.

8.2 *Requirements*

Collaboration. This is an ungraded assignment. You are encouraged to work with a partner.

Platform. This assignment must be completed on your Raspberry Pi, as it is specific to the ARMv6 architecture, the Linux operating system, and the C programming language.

8.3 *Starter Code*

Type the following programs into a text editor. We start with a simple program called `code.c`.

```
#include <stdio.h>

int main() {
  int x = 0;
  x += 72;
  putchar(x);
  x += 33;
  putchar(x);
  x -= 72;
  putchar(x);
  x -= 23;
  putchar(x);
  return 0;
}
```

Compile the above code in the usual way and run it. What does it do?

8.4 *Part 1: Producing assembly*

We can produce assembly for this code with the following command.

```
$ gcc -S code.c
```

You should see the file `code.s` appear. In this lab, you are going to manipulate `code.s` until all of the `NULL` bytes go away.

8.5 *Part 2: Compiling assembly*

There are two ways to compile assembly, depending on whether you want to *make a runnable program* or if you *just want to view the bytes in your functions*.

To make a runnable program, run

```
$ gcc -o code code.s
```

You should be able to run it like

```
$ ./code
```

Because runnable code must link against the C runtime, there is a lot of extra information in `code`'s object code. To exclude this extraneous information, so that you can focus on your own code, run the following instead.

```
$ gcc -c code.s
```

which will create a file called `code.o`. Observe that we cannot run `code.o` even though it has a `main` method because it does not include the C runtime library.

```
# we have to mark code.o as "executable" first
$ chmod +x code.o
$ ./code.o
-bash: ./code.o: cannot execute binary file: Exec format error
```

8.6 *Part 3: Viewing object code to look for `NULL`s*

To view the object code in `code.o`, run

```
$ objdump -d code.o
```

which gives us

```
code.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e24dd008        sub     sp, sp, #8
   c:   e3a03000        mov     r3, #0
  10:   e50b3008        str     r3, [fp, #-8]
  14:   e51b3008        ldr     r3, [fp, #-8]
  18:   e2833048        add     r3, r3, #72     ; 0x48
  1c:   e50b3008        str     r3, [fp, #-8]
  20:   e51b0008        ldr     r0, [fp, #-8]
  24:   ebfffffe        bl      0 <putchar>
  28:   e51b3008        ldr     r3, [fp, #-8]
  2c:   e2833021        add     r3, r3, #33     ; 0x21
  30:   e50b3008        str     r3, [fp, #-8]
  34:   e51b0008        ldr     r0, [fp, #-8]
  38:   ebfffffe        bl      0 <putchar>
  3c:   e51b3008        ldr     r3, [fp, #-8]
  40:   e2433048        sub     r3, r3, #72     ; 0x48
  44:   e50b3008        str     r3, [fp, #-8]
  48:   e51b0008        ldr     r0, [fp, #-8]
  4c:   ebfffffe        bl      0 <putchar>
  50:   e51b3008        ldr     r3, [fp, #-8]
  54:   e2433017        sub     r3, r3, #23
  58:   e50b3008        str     r3, [fp, #-8]
  5c:   e51b0008        ldr     r0, [fp, #-8]
  60:   ebfffffe        bl      0 <putchar>
```

```
64:   e3a03000        mov     r3, #0
68:   e1a00003        mov     r0, r3
6c:   e24bd004        sub     sp, fp, #4
70:   e8bd8800        pop     {fp, pc}
```

Now we can look for `NULL` bytes. Let's focus on the first instruction:

```
0:   e92d4800        push     {fp, lr}
```

Recall that `objdump` "helpfully" attempts to interpret this instruction as an integer word, so it displays the bytes in a rearranged order. Since this word really is an instruction, the rearrangement isn't actually helpful. We simply need to remember to reverse the bytes ourselves to understand their true order in memory. Therefore, `0xe92d4800` really is stored as `00 48 2d e9` on disk. Do you see the `NULL` byte? It's the `00` at the beginning of the word. How do we get rid of it?

8.7 *Part 4: Replacing instructions*

This instruction, as you probably recognize, is a part of the `main` function's preamble. The first question to ask yourself is: do I need to keep the preamble? Under certain circumstances, one way to get rid of `NULL` bytes is just to eliminate the instructions that produce them. However, here we can see that `main` calls another function, `putchar`. Like all C functions, `putchar` expects that the *stack discipline*[1] be maintained. So can we manipulate `push` instead?

[1] In other words, that the program maintains the invariant that the call stack is always valid.

Indeed we can. While it is important in maintaining the stack discipline that `fp` and `lr` be pushed to the stack, we can, of course, push other things as well. For example, `push {r1, fp, lr}` pushes `r1` to the call stack. Happily, when viewed with `objdump`, `push {r1, fp, lr}` yields the bytes:

```
0:   e92d4802        push    {r1, fp, lr}
```

If we're pushing more, we also need to pop more at the end to make sure that `fp` and `pc` are restored correctly.

```
70:   e8bd8802        pop     {r1, fp, pc}
```

That also looks good—no `NULL` bytes. But we did introduce a tiny wrinkle, didn't we? Observe that this program repeatedly loads and stores values from `fp, #-8`. Is that a problem?

8.8 *Part 5: Running your code*

It's probably a good idea to make tiny changes to your code and see if they work. Remember that you can compile and run your code like so:

```
$ gcc -o code code.s
$ ./code
```

If you see the same output as the binary produced when you compile `code.c`, you're on the right track. Also, don't forget that you can always use `gdb` to help you out when your confused about what's happening.

8.9 *Bonus: Replace symbols*

The program we've been tinkering with is not intended to be used in shellcode. But we could use it as shellcode, couldn't we? Except that, since our program is compiled and linked *separately* from the vulnerable program, C will not correctly resolve function names ("symbols") to their correct addresses in the vulnerable program. So to make our attack work, we need to find all of the symbols and replace them with their correct addresses in the vulnerable program.

Assume that `putchar` is located at `0x00010300` in the vulnerable program. Can you replace `putchar` with this address instead?

8.10 *Tips*

Recall that *Lab 7* includes many `NULL`-removal tips. Your starter code for *lab 5* also includes some sample shellcode, which should give you some ideas. And, of course, you should refer back to the *ARM Assembly Guide* for help. Finally, you are welcome to use the Internet, particular Stack Overflow, for this assignment if you think it would be helpful.