ulm university universität uulm

Security of IT Systems

Institute of Distributed Systems | Winter Semester 2018/2019
Dominik Lang, Rens van der Heijden, Alexander Schmid, Gilberto Rossi
Prof. Dr. Frank Kargl, Dr. Elmar Schoch

Exercise 3: Buffer Overflows

Deadline: 2019-01-11

Overview

The goal of this exercise sheet is to learn buffer overflows step by step based on the *Protostar* wargames: `https://exploit-exercises.lains.me/protostar/`. To complete this exercise sheet you need to beat levels *stack0-5*.

Preliminaries

In order to do the exercises, you need to be able to run the vulnerable code, for example with either of the following three options (I would recommend compiling it yourself if possible, because then you have more options for debugging, for example compiling with debugging options (*-g*) or adding temporary *printf* statements to the code):

Compiling the code by yourself on a Linux system

- You will need to install the 32 bit system libraries and headers. For example, on Ubuntu you need to install the *libc6-dev-i386* package:
  apt get install libc6-dev-i386
- For most of the challenges you need to turn off ASLR:
  # sysctl -w kernel.randomize_va_space=0
  If you're doing this on your normal system, remember to turn ASLR back on after you're done doing the challenges (set the value back to *1*).
- Compile with:
  gcc -O0 -g -no-pie -m32 -fno-stack-protector -z execstack -o name name.c

Note: The "-no-pie" option is only important for newer systems with PIE (position independent executable) enabled. If you're using an older system, your gcc version might not implement this option and you can simply omit "-no-pie". **TL;DR:** compile with "-no-pie"; if your compiler does not recognize "-no-pie", simply compile again without this option.

Downloading the ISO Image

- Download the *iso image* from `https://exploit-exercises.lains.me/download/`.
- **Important! If it does not match, contact Dominik Lang.** Check the SHA1 hash; it should be: d030796b11e9251f34ee448a95272a4d432cf2ce

Using the Protostar image as a live USB stick

- Create a live usb stick with the *protostar image*, e.g. based on `https://docs.kali.org/downloading/kali-linux-live-usb-install`.

Using the Protostar image in a VM

- Set up a 32-Bit Linux VM (e.g. with VirtualBox[1])
- When booting the VM, select the downloaded *protostar iso* as the boot medium.

Submission

For each level, name your exploit **stack*X.Y*** (where *X* stands for the level number, and *Y* for the appropriate file type suffix. For example, if your solution for stack0 is a shell command/script, name the file *stack0.sh*). In addition, add a file **stack*X*.txt** for each level explaining your exploit (where *X* stands for the level number).

Before submitting, **create a zip of the parent directory containing all the files**. If you use guides from the internet, please briefly mention them as sources, and double-check that you explained what you did.

Assignment 1: stack0 (1)

Link: `https://exploit-exercises.lains.me/protostar/stack0/`

This level shows that the lack of boundary checking of buffers/arrays can lead to the overflowing of buffers. As a consequence, one is able to change variables/data that are stored before the array/buffer on the stack. In this specific case, your goal is to overflow the *buffer* array and change the value of *modified*.

Assignment 2: stack1 (1.5)

Link: `https://exploit-exercises.lains.me/protostar/stack1/`

This level takes stack0 one step further; this time you do not only need to change the *modified* variable to an arbitrary non-zero value, but need to change it to a specific value. Remember, that x86 systems use little endian, i.e. if you want to write the value '\x42\x73\x21' you need to reverse the bytes: '\x21\x73\x42'.

Assignment 3: stack2 (1.5)

Link: `https://exploit-exercises.lains.me/protostar/stack2/`

This level is very similar to stack1, the primary difference is the input method. The important takeaway of this level is that you cannot trust any external input, not even environment variables.

[1]https://www.virtualbox.org/

Assignment 4: stack3 (2.5)

Link: `https://exploit-exercises.lains.me/protostar/stack3/`

This level requires you to overwrite a function pointer. A function pointer is simply a pointer that points to a function; that is, if you call the function *fp()*, the program will perform a normal function call and execute the code that is located at the memory address saved in *fp*. For example, if *fp* = *0x4221*, then the processor will fetch and execute the instructions starting from the memory address *0x4221*.

To beat this level, you need to first determine the address of *win* (e.g. using gdb or objdump) and then set *fp* to that address using a buffer overflow.

Assignment 5: stack4 (2.5)

Link: `https://exploit-exercises.lains.me/protostar/stack4/`

So far we've overwritten local variables on the stack. This level is very similar to stack3, but this time you need to overwrite the return address with *win*'s address.

It is possible that your compiler adds some additional steps to determine the return address at the end of the *main* function. If you are quite certain that you're overwriting the return address but it is still not working, these additional steps might be the reason. In this case, move the vulnerable parts to a separate function. You can download this alternate version from Moodle (*stack4_alt.c*).

Assignment 6: stack5 (3)

Link: `https://exploit-exercises.lains.me/protostar/stack5/`

The goal of this level is to inject shellcode that spawns a shell and redirects the return address to the injected shellcode. In order to beat this level, your exploit needs to include a shellcode and the address to *buffer*. You can use *gdb*, *objdump*, or *radare2* to determine the address of *buffer*. Figure 1 shows an example of how the stack would look like after a buffer overflow. The example in the figure uses a NOP-sled; however, you do not need to use one.

You can download shellcodes from: `http://shell-storm.org/shellcode/`
For example: `http://shell-storm.org/shellcode/files/shellcode-598.php`

Note 1: It is possible that your compiler adds some additional steps to determine the return address at the end of the *main* function. If you are quite certain that you're overwriting the return address but it is still not working, these additional steps might be the reason. In this case, move the vulnerable parts to a separate function. You can download this alternate version from Moodle (*stack5_alt.c*).

Note 2: It is also possible that you successfully call your shellcode, but then the shell closes immediately. This behaviour is due to stdin being closed after the gets() input. You can use the tool *cat* to keep the shell open. For example, save your exploit in the file *exploit* and do one of the following:

- *$ (cat exploit -) | ./stack5*
- *$ (cat exploit; cat) | ./stack5*
- *$ (cat exploit && cat) | ./stack5*

Note 3: Due to a different environment, it is possible that the addresses in gdb and the real addresses are different. You need to make sure that the environment is exactly the same in both versions and that you call the executable with exactly the same name. You can show gdb's environment with *show env*. You can then set the environment with the *env* tool. For example:

```
$ env -i COLUMNS=231 LINES=65 PWD=/home/ubuntu gdb $(pwd)/stack5
$ env -i COLUMNS=231 LINES=65 PWD=/home/ubuntu $(pwd)/stack5
```

where *-i:* clears the environment. Also make sure to check your *COLUMNS*, *LINES*, and *PWD* values and set them accordingly.

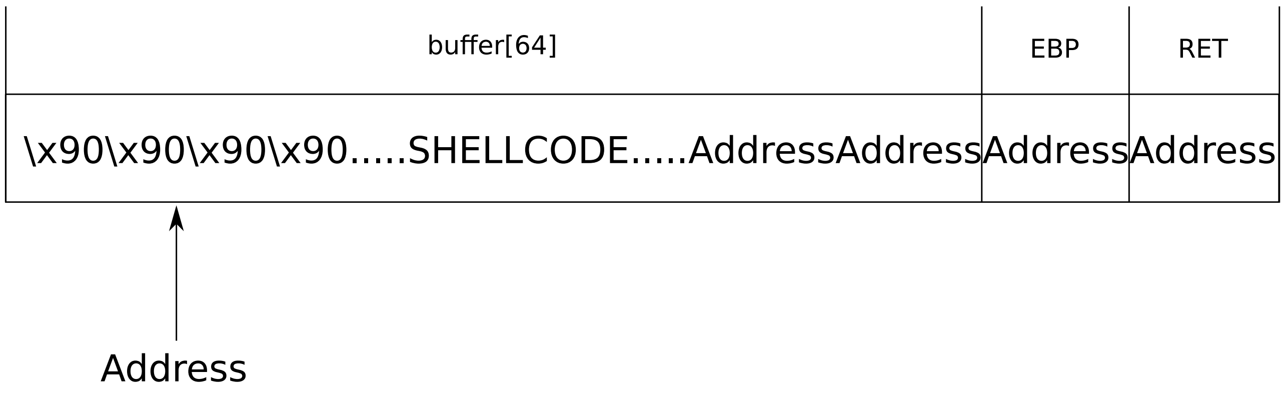

Figure 1: Stack layout after the buffer overflow. *\x90* are NOPs (no-operations). This makes it easier to execute the shellcode, as you have a larger range to target with *Address*. All you need to do is target an address in the range of the NOPs and the processor will *slide* through the NOPs until it reaches your shellcode. Alternatively, you can leave out the NOPs, save the shellcode at the beginning of the buffer, and set *Address* to the beginning of *buffer*.