The first step in developing this exploit is to run the target program – “getscore\_heap.c” – using standard parameters. The program takes two arguments within the command line: a name and a social security number. Upon further examination of the source code, we find that a heap overflow exploit may be deployed since there are no checks in place to prevent arguments that exceed the capacity of the memory allocated for the string buffers. In particular, within “getscore\_heap.c” – the malloc called in line 40 uses the ‘name’ parameter of the program as an argument for how much memory it should allocate.

To begin generating an exploit for this program, there are a couple of address that we need to find. The first of which is the GOT-table entry for the ‘free()’ function. To do this, we can simply call for an object dump of the getscore\_heap executable that will display all of the functions used by the program and the addresses that they reside in.

[root@localhost heap]# objdump -R getscore\_heap

getscore\_heap: file format elf32-i386

DYNAMIC RELOCATION RECORDS

OFFSET TYPE VALUE

08049ca4 R\_386\_GLOB\_DAT \_\_gmon\_start\_\_

08049c5c R\_386\_JUMP\_SLOT perror

08049c60 R\_386\_JUMP\_SLOT system

08049c64 R\_386\_JUMP\_SLOT malloc

08049c68 R\_386\_JUMP\_SLOT time

08049c6c R\_386\_JUMP\_SLOT fgets

08049c70 R\_386\_JUMP\_SLOT strlen

08049c74 R\_386\_JUMP\_SLOT \_\_libc\_start\_main

08049c78 R\_386\_JUMP\_SLOT strcat

08049c7c R\_386\_JUMP\_SLOT printf

08049c80 R\_386\_JUMP\_SLOT getuid

08049c84 R\_386\_JUMP\_SLOT ctime

08049c88 R\_386\_JUMP\_SLOT setreuid

08049c8c R\_386\_JUMP\_SLOT exit

08049c90 R\_386\_JUMP\_SLOT free

08049c94 R\_386\_JUMP\_SLOT fopen

08049c98 R\_386\_JUMP\_SLOT sprintf

08049c9c R\_386\_JUMP\_SLOT geteuid

08049ca0 R\_386\_JUMP\_SLOT strcpy

Here we can see the free function has an offset of 0x8049c90. Next, we will need to determine the base address of the buffer we are targeting. By looking at the source code for getscore\_heap, we can see that this buffer is allocated on line 40. In order to find the address of this buffer, we will need to stop the program once it has allocated it and examine the stack. To do this, we will compile the program with the debugging flag and open the executable in the GNU Debugger.

[root@localhost heap]# gcc -g -o getscore\_heap getscore\_heap.c

[root@localhost heap]# gdb getscore\_heap

We’ll begin by setting the arguments of the program; otherwise, the program would never begin running.

(gdb) set args “aaa” “123456789”

With the arguments set, we must now set a breakpoint after the buffer is allocated, as we mentioned before. Otherwise, the program will execute as it did outside of the debugger, giving us no time to examine the stack as instructions are executed. We found previously that the buffer is allocated on line 40 of the program, however this line is followed by the conditional block of an if-statement. To be safe, we will set a breakpoint after this statement, ensuring that the program will break where we want it to.

(gdb) break 44

Breakpoint 1 at 0x80487bc: file getscore\_heap.c, line 44.

With the breakpoint set, we can now run the program within the debugger.

(gdb) run

Starting program: /root/heap/getscore\_heap "aaa" "123456789"

Breakpoint 1, main (argc=3, argv=0xbffffad4) at getscore\_heap.c:45

warning: Source file is more recent than executable.

45 if ((score = (char \*)malloc(10)) == NULL){

Success! We have paused the program’s execution just after the buffer has been allocated. With the program now in this state, we can examine the code further to determine the location of our buffer. To do this, we can execute the print command with the name of the buffer in question.

(gdb) print matching\_pattern

$1 = 0x8049e28 ""

Here we can see that the buffer matching\_pattern has a base address of 0x8049e28. This is the second and final piece of information that we need to generate the exploit. The final exploit will be composed of two strings, as the program requires two arguments and they are both required due to the handling of the heap structure. Refer to the source code of this exploit to see how the exploit is generated. I have left comments on every bit of code to explain what is going on and how the final exploit is formed.