Smashing The Modern Stack For Fun And Profit

By Craig J. Heffner

When it comes to buffer overflows, 'Smashing The Stack For Fun And Profit' by Aleph One is still the first resource many people are directed towards, and for good reason; it is thorough, well written, and chock-full of examples. However, the GNU C Compiler (gcc) has evolved since 1998, and as a result, many people are left wondering why they can't get the examples to work for them, or if they do get the code to work, why they had to make the changes that they did. Having these same problems myself, and being unable to find an updated version of Aleph One's document on the web, I set out to identify the source of these variations on my own.

I have taken the liberty of writing this paper to share my findings with others who are experiencing the same problems I did, but it is meant only to be a modern supplement to Aleph One's paper. You should read Smashing The Stack first, as it is assumed that you understand the concepts and code presented there, as well as some standard buffer overflow techniques.

The Problem(s)

Unfortunately, different versions of gcc act differently when compiling source code (i.e., the assembly instructions they create from the source code don't always use the stack in the exact same manner), and many different flavors of Linux use different versions of gcc; in other words, the number of bytes needed to properly overwrite the return address changes depending on the version of gcc used to compile the binary. This makes things very difficult if you are trying to write precise code for use on various flavors of Linux. For instance, let's try to use one of Aleph One's examples on Debian 3.1 (currently the newest release of Debian as of 2-18-2007), which comes with gcc version 3.3.5. If you compile Aleph One's example3 (code shown below), you may wonder why it doesn't work for you. Let's identify the possible reasons why it wouldn't work by looking at the source first:

example3.c:

```
void function(int a, int b, int c) {
char buffer1[5];
char buffer2[10];
int *ret;

ret = buffer1 + 12; <----Possible reason #1: Incorrect distance from buffer1 to ret
(*ret) += 8; <------Possible reason #2: Wrong number added to ret address
}

void main() {
int x;

x = 0;
function(1,2,3);
x = 1;
printf("%d\n",x);
}
```

The object of this code is to change the return address from function() so that it skips the 'x=1' assignment. But, if you compile this on a default Debian 3.1 system, this does not happen...you won't even get a segmentation fault. This tells us that we haven't pointed ret to the correct address. How can this be? Aleph One states that all local variables are allocated space on the stack in 4 byte blocks, which today is not always true. The following is a excerpt from one of GCC's online docs:

The stack is required to be aligned on a 4 byte boundary. On Pentium and PentiumPro, double and long double values should be aligned to an 8 byte boundary (see `-malign-double') or suffer significant run time performance penalties. On Pentium III, the Streaming SIMD Extention (SSE) data type __m128 suffers similar penalties if it is not 16 byte aligned.

To ensure proper alignment of this values on the stack, the stack boundary must be as aligned as that required by any value stored on the stack. Further, every function must be generated such that it keeps the stack aligned. Thus calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary will most likely misalign the stack. It is recommended that libraries that use callbacks always use the default setting.

This extra alignment does consume extra stack space. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to `-mpreferred-stack-boundary=2'.

Using the GNU debugger and a little experimentation, I verified that gcc 3.3.5 on Debian 3.1 (have not tested this version of gcc on any other version/flavor of Linux) allocates memory on the stack in 16 byte blocks. So, instead of buffer1 taking up 8 bytes and buffer2 taking up 12 bytes, they each take up 16 bytes, doubling the actual size of buffer1. But even accounting for this will not produce our desired result; compare the below memory map of the stack from Aleph One's paper with the memory map which is actually present in our program:

Aleph One:

bottom of top of

memory memory

buffer2 buffer1 sfp ret a b c

<------ [ ][ ][ ][ ][ ][ ][ ]

top of bottom of

stack stack

Our Program:

bottom of top of

memory memory

buffer2 buffer1 Slack sfp ret a b c

<------ [ ][ ][ ][ ][ ][ ][ ][ ]

top of bottom of

stack stack

Notice the addition of the 'slack' buffer in our program. In gcc 3.3.5 this area is nearly invariably 7 bytes in length. So now our total distance from buffer1 to the return address can be calculated:

buffer1 = 16

Slack = 7

sfp = 4

TOTAL = 27

So we change 'ret = buffer1 + 12' to 'ret = buffer1 + 27', recompile, and...we get a segmentation fault! At least we're closer. Now, let's take a look at Possible Reason #2 by comparing the disassembly of Aleph One's binary to that of ours:

Aleph One:

(gdb) disassemble main

Dump of assembler code for function main:

0x8000490 : pushl %ebp

0x8000491 : movl %esp,%ebp

0x8000493 : subl $0x4,%esp

0x8000496 : movl $0x0,0xfffffffc(%ebp)

0x800049d : pushl $0x3

0x800049f : pushl $0x2

0x80004a1 : pushl $0x1

0x80004a3 : call 0x8000470 <----Call to function

0x80004a8 : addl $0xc,%esp

0x80004ab : movl $0x1,0xfffffffc(%ebp) <----The instruction we want to skip

0x80004b2 : movl 0xfffffffc(%ebp),%eax

0x80004b5 : pushl %eax

0x80004b6 : pushl $0x80004f8

0x80004bb : call 0x8000378

0x80004c0 : addl $0x8,%esp

0x80004c3 : movl %ebp,%esp

0x80004c5 : popl %ebp

0x80004c6 : ret

0x80004c7 : nop

Our Program:

(gdb) disassemble main

Dump of assembler code for function main:

0x080483a2 : push %ebp

0x080483a3 : mov %esp,%ebp

0x080483a5 : sub $0x18,%esp

0x080483a8 : and $0xfffffff0,%esp

0x080483ab : mov $0x0,%eax

0x080483b0 : sub %eax,%esp

0x080483b2 : movl $0x0,0xfffffffc(%ebp)

0x080483b9 : movl $0x3,0x8(%esp)

0x080483c1 : movl $0x2,0x4(%esp)

0x080483c9 : movl $0x1,(%esp)

0x080483d0 : call 0x8048384 <-----Call to function

0x080483d5 : movl $0x1,0xfffffffc(%ebp) <-----The instruction we want to skip

0x080483dc : mov 0xfffffffc(%ebp),%eax

0x080483df : mov %eax,0x4(%esp)

0x080483e3 : movl $0x8048514,(%esp)

0x080483ea : call 0x80482b0 <_init+56>

0x080483ef : leave

0x080483f0 : ret

0x080483f1 : nop

0x080483f2 : nop

0x080483f3 : nop

0x080483f4 : nop

Due to differences in the way our modern gcc compiler translates the source code into assembly instructions, we need to adjust the return address to match our code. Specifically, in Aleph One's code the calling function cleaned up the stack (addl $0xc,%esp), while in our code the called function takes care of this.

The return address pushed onto the stack is 0x080483d5, but we want to skip that and go to 0x080483dc. A little math tells us the distance is 7 bytes, opposed to Aleph's 8 bytes. Change '(*ret) += 8' to '(*ret) += 7', and (if you're running a default Debian 3.1 system) x should stay set to 0.

More Problems...

What if you aren't using Debian 3.1? Let's say you're using the increasingly popular Ubuntu 5.10. There are two key differences between Ubuntu and Debian which we are concerned with:

- Ubuntu comes with gcc version 4.0.5

- Ubuntu uses Linux kernel 2.6.12 while Debian uses 2.4.27

The kernel version is particularly important here, because starting with kernel 2.6.12, Linux has stack protection turned on by default. So, if you're trying to learn about buffer overflows, stick with a 2.4 kernel so you can get practice with some of the easier exploits first, then move on to some more advanced methods and practice using them to circumvent the stack protection.

But since this article's primary focus is to look at how different versions of gcc and Linux utilize stack memory, let's look at how gcc 4.0.5 in Ubuntu allocates stack space for internal variables and how much (if any) space is used up by the slack buffer. I used the following program to do a little experimentation:

test.c:

```
---------------------------------------------------

int main(int argc, char *argv[])

{

char buffer1[5];

char buffer2[10];



strcpy(buffer2,argv[1]);

printf("\nbuff1: %s\nbuff2: %s\n\n",buffer1,buffer2);

return 0;

}

---------------------------------------------------
```

We can open this program up in gdb and look at the memory addresses for buffer1 and buffer2 to determine how far apart their starting addresses are, and subsequently, how much space gcc has allocated for them on the stack:

```
(gdb) disassemble main

Dump of assembler code for function main:

0x080483b8 : push %ebp

0x080483b9 : mov %esp,%ebp

0x080483bb : sub $0x18,%esp <--------Allocates 24 bytes on the stack

0x080483be : and $0xfffffff0,%esp

0x080483c1 : mov $0x0,%eax

0x080483c6 : add $0xf,%eax

0x080483c9 : add $0xf,%eax

0x080483cc : shr $0x4,%eax
```

0x080483cf : shl $0x4,%eax

0x080483d2 : sub %eax,%esp

0x080483d4 : mov 0xc(%ebp),%eax

0x080483d7 : add $0x4,%eax

0x080483da : mov (%eax),%eax

0x080483dc : sub $0x8,%esp

0x080483df : push %eax

0x080483e0 : lea 0xfffffff1(%ebp),%eax <-------- Address of buffer2

0x080483e3 : push %eax

0x080483e4 : call 0x80482e8 <_init+72>

0x080483e9 : add $0x10,%esp

0x080483ec : sub $0x4,%esp

0x080483ef : lea 0xfffffff1(%ebp),%eax

0x080483f2 : push %eax

0x080483f3 : lea 0xfffffffb(%ebp),%eax <---------Address of buffer1

0x080483f6 : push %eax

0x080483f7 : push $0x8048518

0x080483fc : call 0x80482d8 <_init+56>

0x08048401 : add $0x10,%esp

0x08048404 : mov $0x0,%eax

0x08048409 : leave

0x0804840a : ret

The difference between the beginning address of buffer2 (0xfffffff1) and buffer1 (0xfffffffb) is 10 bytes, indicating that buffer2 has been given 10 bytes of room on the stack. We can also set argv[1] to different lengths until we achieve a segmentation fault, observing how many bytes total are needed to get to the return address, allowing us to calculate the size of the slack buffer and buffer1. Let's start by supplying 11 bytes of data and see what we get:

root@ubuntu:~/bof#./test aaaaaaaaaa

buff1: a

buff2: aaaaaaaaaa

This confirms that only 10 bytes were allotted for buffer2, because the 11th byte was placed in the address space for buffer1. Let's assume that buffer1 will be handled the same way and given 5 bytes of stack space, and that there is no slack buffer. This means that to overflow into the return address we will need to supply 20 bytes of data:

buffer2 = 10

buffer1 = 5

sfp = 4

TOTAL = 19

...and the 20th byte should overwrite the first byte of the return address. Let's double check by running it in gdb:

(gdb)run aaaaaaaaaaaaaaaaaaaa

Starting program: /root/bof/test aaaaaaaaaaaaaaaaaaaa

buff1: aaaaaaaaaa

buff2: aaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.

0xb7ea0061 in ?? () from /lib/tls/i686/cmov/libc.so.6

(gdb)

We guessed correctly, and the first byte of the return address is set to 0x61 (ASCII 'a'). So, our memory map of the stack for Ubuntu 5.10 with gcc version 4.0.5 looks like:

bottom of memory top of memory

buffer2 buffer1 sfp ret

<------ [ ][ ][ ][ ]

top of stack bottom of stack

Where: buffer2 = 10 bytes, buffer1 = 5 bytes and sfp = 4 bytes. Note that the stack is still aligned on a 4 byte boundary because 24 bytes have been allocated on the stack ('sub $0x18, %esp').

Conclusion

Hopefully you now have a better idea of how differences in compiler versions affect stack and variable size, and how to take these changes into account when writing an overflow exploit. The best way to adjust for these changes over multiple systems is to add a little NOP padding at the beginning of your shellcode, but if you need more precise calculations, duplicate the target system as closely as possible and start experimenting.

Please contact me with any questions, comments, corrections, etc.

For More Info:

Once again, here's the link to Aleph One's original work, Smashing the Stack for Fun and Profit

Stack as defined by Wikipedia