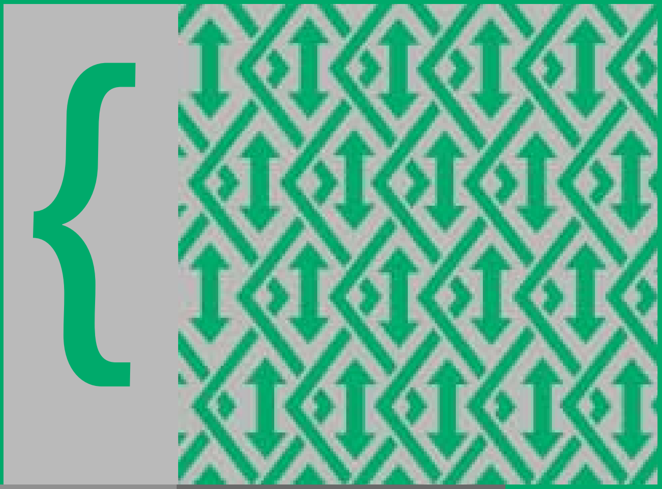
;login:

THE MAGAZINE OF USENIX & SAGE
October 2000 • volume 25 • number 6



inside:

SECURITY
Musings

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

musings

More Stack-smashing Fun

by Rik Farrow

Rik Farrow provides UNIX and Internet security consulting and training. He is the author of *UNIX System Security* and *System Administrator's Guide to System V.*

<rik@spirit.com>

July brought the usual stuff – crushing heat, humidity (for those not in the desert, that is), and a new technique for smashing the stack. At first, I was perplexed about why some were saying that this was not a buffer overflow, but after pouring enough water over my head to cool off and think about it, I can see why.

The problem first surfaced in reports of a root exploit of the venerable wu-ftpd server. You may recall that wu-ftpd was the victim of a buffer-overflow exploit published in February 1999. In that exploit, if the attacker could write in any directory available on the FTP server, the server could be coaxed into replacing itself with a shell, running as root, and still connected to the remote attacker using TCP.

The February exploit was a classic buffer overflow. In that attack, shell code, that is, machine instructions for the target architecture, gets copied to the stack, along with a leader of NOPs (null operations), and then many copies of an address that should point within the regions of NOPs. The idea is to replace the correct return address with the pointer to the shell code (really, the preceding leader of NOPs), so that when the function returns, it will instead execute the shell code.

There are several techniques for dealing with this. One popular one is to make the stack nonexecutable. This helps but still permits buffer overflows to succeed. The exploit must copy the shell code somewhere else, not on the stack, then overwrite the return address with the address of the shell code. When the function returns, the shell code gets executed (as it is not in a nonexecutable portion of memory).

StackGuard, subject of several USENIX papers (also, check out <*http://wirex.com*> and <*http://immunix.org*>) works by modifying the way in which functions are called. The function preamble and postamble puts a "canary" value below the return address on the stack and then checks that the canary has not been smashed on the return from the function. Now, a buffer overflow that overwrites the return address also overwrites the canary, and the StackGuard mods cause the program to exit rather than execute shell code.

The programming flaw that makes buffer overflows possible is the use of functions and loops that copy user input into a locally defined array without counting how many bytes are being copied. The C language was designed by guys who were writing an operating system (UNIX), and didn't need to worry about running off the end of an array. They wanted performance, and also knew what they were doing.

The problem occurs when a program accepts user input, whether from the command line, a network connection, or the environment, and copies it without counting to a buffer allocated on the stack (any variables declared locally, that is local to a function, get allocated on the stack). Some of these functions are: strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets() and scanf(). You can check out the links found at <*http://www.securityportal.com/lskb/articles/kben10000082.html*> for replacements for these functions, such as strncat(), as well as some other resources for secure programming practices.

Stack Manipulation

Rather than blindly smashing the stack, the new technique enables an attacker to probe the stack, then surgically install a new return address (while not touching the canary).

Programs, like wu-ftpd, that pass user input to formatting functions, like sprintf(), are the culprits here.

The printf() family of C functions has a rather interesting capability, especially if you have learned to program using Java or SmallTalk only. That is, these functions accept a variable list of arguments. Internally, a set of routines collectively known as varargs handles the processing of function calls when the number calling arguments is not known at compile time, as with sprintf(). Let's look at a little code example posted by Pascal Bouchareine on July 18 to bugtraq (<*http://www.securityfocus.com/*>):

```
void main() {
        char tmp[512];
        char buf[512];
        while(1) {
        memset(buf, '\0', 512);
        read(0, buf, 512);
        sprintf(tmp, buf);
        printf("%s", tmp);
        }
}
```

This simple program will echo back anything that you type as input (once you compile and run it). sprintf(tmp, buf) copies the input buffer, buf, into a second buffer, tmp. The array tmp is local to main(), so it appears on the stack. So far so good.

What makes this interesting is when you include format characters in your input, such as %s, %f, or %x. To sprintf(), these appear to be commands to pop values off the stack and format them. For example, providing "%x %x %x %x" as input will result in "25207825 78252078 a782520 0" (on Intel processors and their little-endian byte ordering). "25" is the '%', "20" a space, and "78" the 'x' as hexadecimal. But just displaying what we have put on the stack is not very interesting. What if you use enough formatting commands to move up the stack until you display the return address? Now, through the user control of format commands, the state of the stack can be displayed, and the return address located.

Finding the return address is only part of the fun. There is another format command that I do not remember ever using, %n. The %n command counts the number of arguments popped off the stack by sprintf() and related functions and stores that value in the location pointed to. By arranging for %n to place values in the four bytes of the return address, you can overwrite the return address without disturbing the canary which lies below it on the stack.

With these two techniques, exploits can be written that can search for the return address, overwrite it, then execute shell code. wu-ftpd and its SITE-EXEC command logging became the target of a number of exploits all published to bugtraq within a couple of days. There was another formatting problem discovered involving setproctitle(), but no exploit for this was published. You can learn about vendor responses to this by checking out: <*http://www.cert.org/advisories/CA-2000-13.html*>.

Full Disclosure

Once upon a time, only "hackers" and a few people in universities and government research sites had access to information about security exploits. The "good guys" defended keeping this information secret by saying that the number of attacks would increase if they made what they knew public.

I am glad to have left the bad old days of keeping vulnerabilities secret behind us.

Of course, keeping secrets this way kept most sysadmins unaware of the dangers involved in not upgrading to the latest patch for service Y. (Can't use X here for obvious reasons.) The middle road involves sharing enough information to permit sysadmins to test their servers and see if they are vulnerable or not. After all, you don't want to patch something that is not broken. (You will likely break it.)

Even the middle road is dangerous, as knowing how to test for the vulnerability is two-thirds of the way to creating an exploit. But I far prefer the current state of affairs, as I prefer to know what is wrong, why it is wrong, and that it must be fixed. Also, the publishing of vulnerability information has lead to better vendor response in fixing problems.

I am glad to have left the bad old days of keeping vulnerabilities secret behind us. Now we have to deal with security problems rather than sweep them under the carpet. That is much better than living in denial.

For a free, unpaid, political diatribe visit: <*http://www.spirit.com/pol.html*>