Buffer Overrun Madness

Rodney Bates, Wichita State University

In January 2003, the Slammer worm was reported to be the fastest spreading ever. Slammer gets access by exploiting a buffer overrun. If you peruse CERT (Computer Emergency Readiness Team) advisories or security upgrade releases, you will see that the majority of computer security holes are buffer overruns. These would be minor irritations but for the world's addiction to the weakly typed programming languages C and its derivative C++.

Buffer overruns are a kind of array bounds error. There are many variations on how one might actually happen, but here is a typical scenario. Function F calls function G, and G returns a string. F allocates a buffer to hold the result and passes a pointer to its zeroth character. G gets the string from somewhere, perhaps a network connection, and copies it into the buffer supplied by F.

G knows the length of the actual string it is supposed to return, but doesn't know the size of the buffer supplied. F doesn't know how long the string will be, so it just allocates a buffer that it hopes is plenty big for the worst case. If the string is too long, G naively copies bytes into whatever memory lies beyond the buffer.

If the string comes from a network connection, a system cracker can exploit this bug with a carefully constructed message that is longer than the buffer. The bytes that spill out overlay some nearby variables, typically part of F's or G's activation record. One possibility is that it overlays the return address, where F or G will return to, with a pointer into the earlier part of the message. Here, the cracker puts machine code. When the function returns, it executes the cracker's code, but with the privileges of the networking or server code that F and G belong to.

The cracker has to know a lot about the compiled machine code for F, G, and friends, including the actual addresses their activation records usually occupy on the stack. This information isn't hard to acquire, using a copy of the targeted software. There are other ways, such as overlaying a variable with a value that makes F or G think a correct password has been supplied.

The original vulnerability can happen in a variety of ways. C's weak type system has created a culture where

Why do good

PROGRAMMERS FOLLOW

BAD PRACTICES?

programmers don't bother to make the buffer size available to G at all. This is especially problematic if G is in a library, from

some other development organization, that the writers of F don't control. In response to the increasing flow of exploits like this, programmers are adding the buffer size as an additional parameter, supplied by F, and checked explicitly in G. Sometimes the buffer size is passed, but the check is forgotten.

Going back as far as 1958, in most other programming languages, buffer overruns were caught by the language itself, through compiler-generated checks on array bounds. This wouldn't prevent such a bug from occurring in the first place, but an attempt to exploit it would result in an immediate crash of part or all of the targeted machine's software.

This might make for a single-point denial-of-service attack, but the target could not be taken over as a launch point for further attacks, precluding rapidly propagating worms such as Slammer. Nor could the attacker damage, destroy, steal, or falsify data. Moreover, the first successful attack would point immediately to the existence and location of the bug. The usual buffer-overrun exploit can remain undetected for a very long time if the attacker tries to remain unnoticed instead of creating a spectacle.

Of course, even when the language supports bounds checks, most compilers allow their generation to be disabled. An all-too-common practice is to debug with the checks turned on, then recompile without them, for production use. I recall a story from at least as far back as the early 1970s about millions of dollars of business being conducted on the advice of a program that was later found to have undetected array bounds errors. Its output had been wrong all along, but not obviously so, so nobody suspected until an unrelated software change uncovered it.

Recompiling with array bounds checks is relatively easy. It requires distributing and installing the updated version, of course, but not recoding. Not so in C and C++. Although their primitive type systems do have array

types, the language almost always insists on ignoring them. Suppose A is declared as an array variable. With one exception, every time you refer to A, the reference is immediately converted to a pointer to the zeroth element of A. As with all pointer types, this is then treated as pointing to an element somewhere out in the middle of an array that extends far in both directions, limited only by the finite range of addresses on the machine. The bounds have been forgotten by the type system, so runtime checks could not be generated even if desired.

A clever language designer might be able to rework the type system in a way that would simultaneously allow bounds checking, while keeping certain source code sequences legal. But there is another problem, and it is partly cultural. Because arrays are seldom treated as real arrays anyway, most C/C++ programmers have adopted the habit of just declaring what are really arrays with pointer types in the first place.

This is one of the best examples of how bad language design encourages bad programming habits. Proponents of weak type systems often argue that good programmers will follow good practices. For a small minority, this is indeed true. The reality is, the great majority of programmers do quite the opposite.

If the array is a local or global variable, there has to be one declaration somewhere, with an array type, to allocate space for it. After that, however, programmers usually pass it all around with a pointer type. If heap-allocated, it is probably cast to a pointer type immediately and never given an array type at all.

There is more mud in this water. Even when programmers use them, C and C++ array types can carry only static bounds. Many arrays in programs need to have dynamic size: most frequently, array formal parameters. Here, the actual parameters are often static-size, but the formals need to be dynamically sized to accept different actuals. For these cases, the type system of C/C++ offers only array types of unknown size. So even if the programmer declares using an array type, the bounds are lost.

Between the language's ignoring array types, the lack of dynamic-size array types that carry their bounds, and programmers' habits of not using array types anyway, there is no way to redesign the type system to support array bounds checking without requiring significant recoding of virtually everything.

You might imagine an automatic code conversion program to go along with a repaired type system. It would have to be very sophisticated to distinguish among pointers used to access arrays, pass parameters by reference and as legitimate pointers to linked data structures. It would

have to do whole-program analysis, which in turn is especially difficult in C and C++, because their lack of linguistic recognition of separate compilation makes it very difficult even to locate all the relevant source code.

In languages with well-designed type systems, array types have to be used to access arrays. These types carry their bounds as part of the type if the bounds are static, or as part of the value if not. Then they can generate runtime bounds checks on array element accesses, where needed. Programmers can still create buffer overrun bugs, but the consequences are far less serious.

The code itself can be changed to prevent buffer overruns, even in weakly typed languages. That is exactly what is happening, bug by bug, to close the security holes. But the endless flood of reported buffer overrun vulnerabilities shows that the number of holes is enormous. What portion can be attributed directly to a bad type system and what portion to bad programmer habits engendered thereby is anybody's guess. So far, the rate of new discoveries of such vulnerabilities shows no sign of diminishing.

Why do so many cling to languages that are 45 years behind the state of the programming language art? As a programming languages specialist, I have had decades of discussions about this with both practitioners and academicians. Most of the reasons boil down to, "It's what everybody else is doing." There are some exceptions—valid points, though easily answered. When software developers choose languages on their merits, a lot of ugly problems will become simple problems. But don't worry that it might make programmers redundant. The space of problems that cannot be solved by any known language technology will always be larger than the space of those that can, and far bigger than our collective capacity. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RODNEY BATES (rodney.bates@wichita.edu) has never done a paid day's work in electrical engineering, for which he was trained. Instead, he has worked 33 years in computer software, two-thirds in industry, the rest in academia. He has been involved mostly with operating systems, compilers, and as a resident programming language lawyer. He has contributed to popular computer magazines, trade and research conferences, and academic journals. His current big project is a semantic editor for Modula-3 and other languages. He is an assistant professor and graduate coordinator in the computer science department at Wichita State University, Kansas.

© 2004 ACM 1542-7730/05/0500 \$5.00