IBM

# Secure programmer: Countering buffer overflows

**Preventing today's top vulnerability**

Level: Intermediate

David Wheeler (dwheelerNOSPAM@dwheeler.com), Research Staff Member, Institute for Defense Analyses

27 Jan 2004

This article discusses the top vulnerability in Linux/UNIX systems: buffer overflows. This article first explains what buffer overflows are and why they're both so common and so dangerous. It then discusses the new Linux and UNIX methods for broadly countering them -- and why these methods are not enough. It then shows various ways to counter buffer overflows in C/C++ programs, both statically-sized approaches (such as the standard C library and OpenBSD/strlcpy solution) and dynamically-sized solutions, as well as some tools to help you. Finally, the article closes with some predictions on the future of buffer overflow vulnerabilities.

In November 1988, many organizations had to cut themselves off from the Internet because of the "Morris worm," which was a program written by 23-year-old Robert Tappan Morris to attack VAX and Sun machines. By some estimates, this program took down 10% of the entire Internet. In July 2001, another worm named "Code Red" eventually exploited over 300,000 computers worldwide running Microsoft's IIS Web Server. In January 2003, the "Slammer" (also known as "Sapphire") worm exploited a vulnerability in Microsoft SQL Server 2000 software, disabling parts of the Internet in South Korea and Japan, disrupting Finnish phone service, and slowing many U.S. airline reservation systems, credit card networks, and automatic teller machines. All of these attacks -- and many others -- exploited a vulnerability called a *buffer overflow*.

An informal 1999 survey on Bugtraq (a mailing list discussing security vulnerabilities) found that two-thirds of the participants believed that the #1 cause of vulnerabilities was buffer overflows (for background reading, see "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" listed in the Resources section later in this article). From 1997 through March 2002, half of all security alerts from the CERT/CC were based on buffer overflow vulnerabilities.

If you want your programs to be secure, you need to know about buffer overflows and how to prevent them, the latest automated tools to counter them (and why they aren't enough), and how to counter them in your programs.

## What's a buffer overflow?

A *buffer* can be formally defined as "a contiguous block of computer memory that holds more than one instance of the same data type." In C and C++, buffers are usually implemented using arrays and memory allocation routines like `malloc()` and `new`. An extremely common kind of buffer is simply an array of characters. An *overflow* occurs when data is added to the buffer outside the block of memory allocated to the buffer.

If an attacker can cause a buffer to overflow, then the attacker can control other values in the program. Although there are lots of ways that buffer overflows can be exploited, the most common approach is the "stack-smashing" attack. A classic article explaining stack smashing attacks is "Smashing the Stack for Fun and Profit" by Elias Levy (also known as Aleph One), former moderator of the Bugtraq mailing list (see Resources for a link).

To understand how a stack-smashing attack (or any other buffer overflow attack) works, you need to understand a little about how computers really work at the machine-language level. On UNIX-like systems, every process can be divided into three main regions: text, data, and stack. The *text region* includes code and read-only data, and it can't normally be written to. The *data region* includes both statically allocated memory (such as global and static data) and dynamically allocated memory (often called the *heap*). The *stack region* is used to permit function/method calls; it is used to record where to return after a function completes, to store local variables used in functions, to pass parameters to the functions, and to return values from the function. Every time a function is called, a new *stack frame* (an area of memory inside the stack) is used to support the call. With that in mind, let's look at a trivial program.

**Listing 1. A trivial program**

```
void function1(int a, int b, int c) {
   char buffer1[5];
   gets(buffer1); /* DON'T DO THIS */
}

void main() {
  function(1,2,3);
}
```

Imagine that the trivial program in Listing 1 was compiled using gcc, run in Linux on an x86, and has been suspended right after the call to `gets()`. What would the memory contents look like? The answer is that it would look like Figure 1, which shows the memory layout ordered from lower addresses on the left to higher addresses on the right.

**Figure 1. View of stack**

```
bottom of memory                          top of memory
                  buffer1 sfp ret  a   b   c
<--- growth ---   [ ]     [ ]  [ ] [ ] [ ] [ ] ...
top of stack                              bottom of stack
```

Many computer processors, including all x86 processors, support stacks that grow "down" from higher to lower addresses. Thus, every time a function calls another function, more data will be added to the left (lower addresses) until the system runs out of memory for the stack. In this example, when `main()` called `function1()`, it pushed the values for c, then b, then a onto the stack. It then pushed the `return (ret)` value, which tells `function1()` where to return to in `main()` once `function1()` is done. It also recorded something called the "saved frame pointer" (sfp) onto the stack; this is something that isn't always saved, and we don't need to understand it to understand the problem. In any case, once `function1()` started up, it sets aside space for `buffer1()`, which Figure 1 shows has a lower address location.

Now imagine that an attacker has sent more data than `buffer1()` can handle. What happens next? Well, C and C++ don't automatically check for this problem, so unless the programmer has specifically prevented it, the next values will go to the "next" locations in memory. That means that the attacker can overwrite `sfp` (the saved frame pointer) and then overwrite `ret` (the return address). Then, when `function1()` is done, it will "return" -- but instead of returning to `main()`, it will return to whatever code the attacker wants to run instead.

Often the attacker will overrun the buffer with the malicious code the attacker wants to run, and the attacker will then change the return value to point to the malicious code they've sent. That means the attacker can set up the entire attack in essentially one operation! Aleph One's article (see Resources for a link) goes into detail on how such attack codes are created. For example, it's often difficult to put an ASCII 0 (NUL) character into a buffer, and the article shows how attackers can normally get around this problem.

There are other ways to exploit buffer overflows besides smashing the stack and changing the return address. Instead of overwriting the return address, you could smash the stack (overflow a buffer on the stack) and then overwrite local variables to create an exploit. The buffer need not be on the stack at all -- it could be dynamically allocated memory in the heap (also called the "malloc" or "new" area), or in some statically located memory (such as "global" and "static" memory). Basically, if an attacker can overflow the bounds of a buffer, you're probably in trouble. However, the most dangerous buffer overflow attacks are stack-smashing attacks, because it's especially easy for an attacker to gain control over an entire machine if your program is vulnerable to them.

### Why are buffer overflows so common?

In nearly all computer languages, both old and new, trying to overflow a buffer is normally detected and prevented automatically by the language itself (say, by raising an exception or adding more space to the buffer as needed). But there are two languages where this is not true: C and C++. Often C and C++ will simply let additional data be scribbled all over the rest of the memory, and this can be exploited to horrific effect. What's worse, it's actually more difficult to write correct code in C and C++ to always deal with buffer overflows; it's very easy to accidentally permit a buffer overflow. These might be irrelevant facts except that C and C++ are *very* widely used; for example, 86% of the lines of code in Red Hat Linux 7.1 are in either C or C++. Thus, there's a vast amount of code that's vulnerable to this problem because the implementation language fails to protect against it.

This isn't easily fixed in the C and C++ languages themselves. The problem is based on fundamental design decisions of the C language (particularly how pointers and arrays are handled in C). Since C++ is a mostly compatible superset of C, it has the same problems. There are "safe" compatible versions of C/C++ that prevent this, but they have extreme performance penalties. And once you change the C language to prevent this problem, it's no longer C. Many languages (like Java and C#) are syntactically similar to C, but they are truly different languages and changing an existing C or C++ program to them is a significant undertaking.

Users of other languages shouldn't be too smug, though. Some languages have "escape" clauses that allow buffer overflows to occur. Ada normally detects and prevents buffer overflows (raising an exception on the attempt), but various pragmas can disable this. C# normally detects and prevents buffer overflows, but it lets programmers define some routines as "unsafe" and such code *can* have buffer overflows. So if you use those escape mechanisms, you'll need to use the same kind of protection mechanisms that C/C++ programs must use. Many languages are implemented (at least partially) in C, and essentially all programs in any language depend on libraries written in C or C++. Thus, all programs can inherit these problems, so it's important to know what these problems are.

---

## Common C and C++ mistakes that permit buffer overflows

Fundamentally, any time your program reads or copies data into a buffer, it needs to check that there's enough space *before* making the copy. An exception is if you can show it can't happen -- but often programs are changed over time that make the impossible possible.

Sadly, there are a large number of dangerous functions that come with C and C++ (or are commonly used libraries) that even

fail to do this. Any place a program uses them is a warning signal, because unless they're used carefully, they become a vulnerability. You don't need to memorize the list; my real point is to show how common the problem is. These functions include `strcpy(3), strcat(3), sprintf(3)` (with cousin `vsprintf(3)`), and `gets(3)`. The `scanf()` set of functions (`scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3)`, and `vfscanf(3)`) can cause troubles because it's easy to use a format that doesn't define a maximum length (using the format "%s" is almost always a mistake when reading untrusted input).

Other dangerous functions include `realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3)`, and `strtrns(3)`. In theory, `snprintf()` should be relatively safe -- and is in modern GNU/Linux systems. But very old UNIX and Linux systems didn't implement the protective mechanism `snprintf()` is supposed to implement.

There are other functions in Microsoft's libraries that cause the same sort of problems on their platforms (these functions include `wcscpy(), _tcscpy(), _mbscpy(), wcscat(), _tcscat(), _mbscat()`, and `CopyMemory()`). Note that if you use Microsoft's `MultiByteToWideChar()`, there's a common dangerous error -- the function requires a maximum size as the number of characters, but programmers often give the size as bytes (the more common requirement), resulting in a buffer overflow vulnerability.

Another problem is that C and C++ have very weak typing for integers and don't normally detect problems manipulating them. Since they require the programmer to do all the detecting of problems by hand, it's easy to manipulate numbers incorrectly in a way that's exploitable. In particular, it's often the case that you need to keep track of a buffer length, or read a length of something. But what happens if you use a signed value to store this -- can an attacker cause it to "go negative" and then later have that data interpreted as a really large positive number? When numeric values are translated between different sizes, can an attacker exploit this? Are numeric overflows exploitable? Sometimes the way integers are handled creates a vulnerability.

---

## New tricks to counter buffer overflows

Of course, it's hard to get programmers to *not* make common mistakes, and it's often difficult to change programs (and programmers!) to another language. So why not have the underlying system automatically protect against these problems? At the very least, protection against stack-smashing attacks would be a good thing, because stack-smashing attacks are especially easy to do.

In general, changing the underlying system so that it protects against common security problems is an excellent idea, and we'll encounter that theme in later articles too. It turns out there are many defensive measures available, and some of the most popular measures can be grouped into these categories:

- Canary-based defenses. This includes StackGuard (as used by Immunix), ssp/ProPolice (as used by OpenBSD), and Microsoft's /GS option.
- Non-executing stack defenses. This includes Solar Designer's non-exec stack patch (as used by OpenWall) and exec shield (as used by Red Hat/Fedora).
- Other approaches. This includes libsafe (as used by Mandrake) and split-stack approaches.

Unfortunately, all the approaches found so far have weaknesses, so they're no panacea, but they can be helpful.

### Canary-based defenses

Researcher Crispen Cowan created an interesting approach called StackGuard. Stackguard modifies the C compiler (gcc) so that a "canary" value is inserted in front of return addresses. The "canary" acts like a canary in a coal mine: it warns when something has gone wrong. Before any function returns, it checks to make sure that the canary value hasn't changed. If an attacker overwrites the return address (as part of a stack-smashing attack), the canary's value will probably change and the system can stop instead. This is a useful approach, but note that this does not protect against buffer overflows overwriting other values (which they may still be able to use to attack a system). There's been work to extend this approach to protecting other values (such as those on the heap) as well. Stackguard (as well as other defensive measures) is used by Immunix.

IBM's stack-smashing protector (ssp), originally named ProPolice, is a variation of StackGuard's approach. Like StackGuard, ssp uses a modified compiler (gcc) to insert a canary in function calls to detect stack overflows. However, it adds some interesting twists to the basic idea. It reorders where local variables are stored, and copies pointers in function arguments, so that they're also before any arrays. This strengthens the protection of ssp; this means a buffer overflow can't modify a pointer value (otherwise an attacker who can control a pointer can control where the program saves data using the pointer). By default, it doesn't instrument all functions, only those that it deems as being in need of protection (mainly functions with character arrays). In theory, this could weaken the protection slightly, but this default improves performance while still protecting against most problems. As a practical matter, they implemented their approach using gcc in a way that is architecture-independent, making it easier to deploy. The widely-respected OpenBSD, which concentrates on security, uses ssp (also known as ProPolice) across their entire distribution as of their May 2003 release.

Microsoft has added a compiler flag (/GS) to implement canaries in its C compiler, based on the StackGuard work.

### Non-executing stack defenses

Another approach starts by making it impossible to execute code on the stack. Unfortunately, the memory protection mechanisms of the x86 processors (the most common processors) don't easily support this; normally if a page is readable, it's executable. A developer named Solar Designer dreamed up a clever combination of kernel and processor mechanisms to create a "non-exec stack patch" for the Linux kernel; with this patch, programs on the stack can no longer be normally run on x86s. It turns out that there are cases where executable programs *are* needed on the stack; this includes signal handling and trampoline handling. Trampolines are exotic constructs sometimes generated by compilers (such as the GNAT Ada compiler) to support constructs like nested subroutines. Solar Designer also figured out how to make these special cases work while preventing attacks.

The original patch to do this in Linux was rejected by Linus Torvalds in 1998, and for an interesting reason. Even if code can't be placed on the stack, an attacker could use a buffer overflow to make a program "return" to an existing subroutine (such as a routine in the C library) and create an attack. In short, just having a non-executable stack isn't enough.

After some time, a new idea was developed to counter that problem: move all executable code to an area of memory called the "ASCII armor" region. To understand how this works, it's important to know that attackers often can't insert the ASCII NUL character (0) using typical buffer overflow attacks. That means that attackers find it difficult to make a program return to an address with a zero in it. Since that's the case, moving all executable code to addresses with a 0 in it makes attacking the program far more difficult.

The largest contiguous memory range with this property is the set of memory addresses from 0 through 0x01010100, so that's been christened the ASCII armor region (there are other addresses with this property, but they're scattered). Combined with non-executable stacks, this is pretty valuable: non-executable stacks prevent attackers from sending new executable code, and ASCII-armor makes it hard for attackers to work around it by exploiting existing code. This protects against stack, buffer, and function pointer overflows, all without recompilation.

However, ASCII-armor doesn't work for all programs; big programs may not fit in the ASCII-armor region (so the protection will be imperfect), and sometimes attackers *can* get a 0 into their destination. Also, some implementations don't support trampolines, so the protection may have to be disabled for programs that need them. Red Hat's Ingo Molnar implemented this idea in his "exec-shield" patch, which is used by Fedora core (the freely available distribution available from Red Hat). The latest version OpenWall GNU/Linux (OWL) uses an implementation of this approach by Solar Designer (see Resources for links to these distributions).

### Other approaches

There are many other approaches. One approach is to make standard library routines more resistant to attack. Lucent Technologies developed Libsafe, a wrapper of several standard C library functions like `strcpy()` known to be vulnerable to stack-smashing attacks. Libsafe is open source software licensed under the LGPL. The libsafe versions of those functions check to make sure that array overwrites can't exceed the stack frame. However, this approach only protects those specific functions, not stack overflow vulnerabilities in general, and it only protects the stack, not local values in the stack. Their original implementation uses `LD_PRELOAD`, which can conflict with other programs. The Mandrake distribution of Linux (as of version 7.1) includes libsafe.

Another approach is called "split control and data stack" -- the idea is to split the stack into two stacks, one to store control information (such as the "return" address) and the other for all the other data. Xu et al. implement this in gcc, and StackShield implements it in the assembler. This makes it much harder to manipulate the return address, but it doesn't defend against buffer overflow attacks that change the data of calling functions.

In fact, there are other approaches as well, including randomizing the locations of executables; Crispen's "PointGuard" extends the canary idea to the heap, and so on. Figuring out how to defend today's computers has become an active research task.

---

## General protections aren't enough

What's the implication of so many different approaches? The good news for users is that a lot of innovative approaches are being tried out; in the long term this "shoot-out" will make it easier to see which approaches are best. Also, this diversity makes it harder for attackers to slip through all of them. However, this diversity also means that developers need to *avoid* writing code that interferes with any of these approaches. In practice this is easy; just don't write code that does low-level manipulations of the stack frame or makes assumptions about the stack layout. That's good advice, even if these approaches didn't exist.

The implication for operating system distributors is quite clear: pick at least one approach, and use it. Buffer overflows are the #1 problem, and the best of these approaches can often reduce the effects of nearly half of the currently-unknown vulnerabilities in your distribution. It's arguable whether the canary-based approach or the non-executable stack based approach is better; they both have their strengths. They can be combined, but few do so because the additional performance loss doesn't appear worth it. I don't suggest the others, at least by themselves; both libsafe and splitting the control and data stacks are limited in the protection they provide. The worst solution, of course, is no protection at all against the #1 vulnerability. Distributions that have not implemented an approach need to plan to do so immediately. Beginning in 2004,

users should start avoiding any operating system that fails to provide at least some automatic protection against buffer overflows.

However, none of this lets developers ignore buffer overflows. All of these approaches can be subverted. An attacker may be able to exploit a buffer overflow by changing the value of other data in the function; none of these approaches counter that. Many of these approaches can be sidestepped if certain hard-to-create values can be inserted (such as the NUL character); this is becoming easier as multimedia and compressed data are becoming more common. Fundamentally, all these approaches reduce the damage of a buffer overflow attack from a program-takeover attack into a denial-of-service attack. Unfortunately, as computer systems become used in more critical situations, even denial of service is often unacceptable. Thus, although distributions should include at least one good defensive approach, and developers should work with (not against) those approaches, developers still need to write good software in the first place.

---

# C/C++ solutions

A simple solution for buffer overflows is to switch to a language that prevents them. After all, practically every high-level language except C and C++ have built-in mechanisms to effectively counter them. But many developers choose, for a variety of reasons, to use C and C++ anyway. So what can you do?

It turns out that there are many different techniques to countering buffer overflows, but they can be divided into two approaches: statically allocated buffers and dynamically allocated buffers. So first, we'll describe what these two approaches are. Then, we'll discuss two examples of the static approach (standard C `strncpy/strncat` and OpenBSD's `strlcpy/strlcat`), followed by two examples of the dynamic approach (SafeStr and C++'s `std::string`).

---

# The big choice: Statically and dynamically allocated buffers

Buffers have a limited amount of space. So there are really two major possibilities for dealing with running out of space.

- "Statically allocated buffer" approach: When the buffer runs out, that's it; you complain and refuse to add anything more to the buffer.
- "Dynamically allocated buffer" approach: When the buffer runs out, you dynamically resize the buffer to a larger size until you run out of total memory.

There are disadvantages to the static approach. In fact, the static approach may sometimes create a different vulnerability! Static approaches basically throw away "excess" data. If the program uses the resulting data anyway, an attacker will try to fill up the buffer so that when the data is truncated, the attacker will fill the buffer with what the attacker wanted. If you're using a static approach, you should ensure that the worst an attacker could do won't invalidate some assumption, and a few checks on the final result would be a good idea too.

Dynamic approaches have lots of advantages: they can scale up to larger problems (instead of creating arbitrary limits), and they don't have the problem of truncations causing security problems. But they have problems of their own. When arbitrarily sized data is accepted, you may run out of memory -- and that may not happen just during input. Any memory allocation can fail, and it's not easy to write C or C++ programs that truly handle that well. Even before truly running out of memory, you can cause the computer to get so busy that it becomes useless. In short, dynamic approaches often make it much easier for attackers to create denial-of-service attacks. So you'll still need to limit inputs. What's more, you'll have to carefully design your program to handle memory exhaustion in arbitrary places, which is not easy.

### Standard C library approach

One of the simplest approaches is to simply use the standard C library functions designed to prevent buffer overflows (this is possible even if you're using C++), particularly `strncpy(3)` and `strncat(3)`. These standard C library functions generally support a statically allocated approach, throwing away data if it doesn't fit into the buffer. The biggest advantages of this approach are that you can be certain that these functions will be available on any machine and that any C/C++ developer will know about them. Many, many programs are written this way, and it does work.

Unfortunately, this is surprisingly hard to do correctly. Here are some of the problems:

- Both `strncpy(3)` and `strncat(3)` require that you give the amount of space *left*, not the total size of the buffer. That's a problem because, while a buffer's size doesn't change once it's allocated, the amount of space left in the buffer changes every time data is added or removed. That means programmers have to keep track or recompute how much space is left all the time. This tracking or recomputation is easy to get wrong, and any mistake can open the door to a buffer overflow attack.

- Neither function gives a simple report if an overflow (and data loss) has occurred, so programmers have to do even more work if they want to detect that.

- The function `strncpy(3)` also doesn't NUL-terminate its destination if the source string is at least as long as the

destination; this can cause havoc later. Thus, after running `strncpy(3)`, you often need to re-terminate the destination.

- The function `strncpy(3)` can also be used to copy only a *part* of the source string into the destination. When doing that, the number of characters to be copied is usually computed based on information about the source string. The danger is that if you forget to take the available buffer space into account, you can still permit a buffer overflow attack *even when using* `strncpy(3)`. This also doesn't copy a NUL character, which can be problem too.

- You can use `sprintf()` in a way that prevents buffer overflows, but it's very easy to accidentally permit overflows instead. The `sprintf()` function uses a control string to specify the output format, and often the control string includes "`%s`" (string output). If you include a precision specifier for a string output (such as "`%.10s`"), then you can protect against buffer overflows by specifying the maximum length of the output. You can even use "`*`" as a precision specifier (such as "`%.*s`"), so you can pass in a maximum length instead of having the maximum length embedded in the control string. The problem is that the `sprintf()` can easily be used incorrectly. A "field width" (such as "`%10s`") only specifies the minimum length -- not the maximum length. The "field width" specifier allows buffer overflows, and the field width and the precision width specifiers look almost identical -- the only difference is that the safe version has a period. Another problem is that precision fields only specify the maximum size of one parameter, but buffers need to be sized for the maximum size of all data combined instead.

- The `scanf()` family of functions has a maximum width value, and at least the IEEE Standard 1003-2001 clearly states that these functions must not read more than the maximum width. Unfortunately, not all specifications make this so clear, and it's not clear that all implementations properly implement these limits (it *does* work properly on today's GNU/Linux systems). If you depend on it, it's wise to run a small test during installation or initialization to make sure it works correctly.

There's also an annoying performance problem with `strncpy(3)`. In theory, `strncpy(3)` is the safe replacement for `strcpy(3)`, but `strncpy(3)` also fills the entire destination with NULs once the end of the source string has been met. This is bizarre, because there's really no good reason for it to do that, but this has been true from the beginning and some programs depend on that. This means that switching from `strcpy(3)` to `strncpy(3)` can reduce performance -- often not a serious problem on today's computers, but it can still be a nuisance.

So can you use the standard C library's routines for preventing buffer overflow? Yes, but it's not easy. If you plan to go that route, you need to understand all of the points above. Or, you can use an alternative, which the next sections discuss.

### OpenBSD's strlcpy/strlcat

The OpenBSD developers have developed a different static approach based on new functions they developed, `strlcpy(3)` and `strlcat(3)`. These functions do string copying and concatenation, but in a much less error-prone way. These functions' prototypes are:

```
size_t strlcpy (char *dst, const char *src, size_t size);
size_t strlcat (char *dst, const char *src, size_t size);
```

The `strlcpy()` function copies the NUL-terminated string from "`src`" to "`dst`" (up to size-1 characters). The `strlcat()` function appends the NUL-terminated string `src` to the end of `dst` (but no more than size-1 characters will be in the destination).

At first blush, their prototypes don't look much different than the standard C library functions. But in fact, there are some marked differences. Both of these functions take the total size of the destination buffer as a parameter -- not the space still left. That means that you don't have to constantly recalculate the size, which is an error-prone task. Also, both functions guarantee that the destination will be NUL-terminated as long as the size is at least one (you can't put anything into a zero-length buffer). The return value is always the size of the combined string if no buffer overflow occurred; this makes it really easy to detect an overflow.

Unfortunately `strlcpy(3)` and `strlcat(3)` aren't universally available in the standard libraries of UNIX-like systems. OpenBSD and Solaris have them built into <string.h>, but GNU/Linux systems don't. This is not that difficult a problem; since they are small functions, you can even include them in your own program's source whenever the underlying system doesn't provide them.

### SafeStr

Messier and Viega have developed the "SafeStr" library, a dynamic approach for C that automatically resizes strings as necessary. Safestr strings easily convert to regular C "`char *`" strings, using the same trick used by most `malloc()` implementations: safestr stores important information at addresses "before" the pointer is passed around. The advantage of this trick is that it's easy to use SafeStr in existing programs. SafeStr also supports "read-only" and "trusted" strings, which can be helpful too. One issue is that it requires XXL (a library that adds support for exception handling and asset management to C), so you do bring in significant libraries just to handle strings. Safestr is released under an open source BSD-style license.

### C++ std::string

Another solution for C++ users is the standard `std::string` class, which is a dynamic approach (buffers grow as needed). It's almost a no-brainer because the language supports the class directly, so there's no special effort to use it and other libraries

will probably use it. By itself, `std::string` normally protects against buffer overflow, but if you extract an ordinary C string from it (say by using `data()` or `c_str()`), all the problems discussed above resurface. Also remember `data()` won't always return a NUL-terminated string.

For a variety of historical reasons, many C++ libraries and pre-existing programs have created their own string classes. This can make using `std::string` more awkward and inefficient when using those libraries or modifying those programs, because the different string types would have to be constantly converted back and forth. Not all of these other string classes protect against buffer overflows, and buffer overflow vulnerabilities are easy to introduce in some of them if they do automatic conversions to C's unprotected `char*` type.

---

## Tools

There are a number of tools that can help detect buffer overflow vulnerabilities before they're released. For example, tools such as my Flawfinder and Viega's RATS search through source code and identify functions that may be used incorrectly (ranking them based on their parameters). A disadvantage of these tools is that they're imperfect -- they will miss some buffer overflow vulnerabilities, and they'll identify "problems" that in fact aren't problems. But they're still worth using, because they'll help you identify potential problems in your code in much less time than a manual search.

---

## Conclusions

Buffer overflow vulnerabilities can be prevented in C and C++ with knowledge, caution, and tools. But it's not that easy, especially in C. If you're using C and C++ to write secure programs, you need to really understand buffer overflows and how to prevent them.

An alternative is to use another programming language, since almost all of today's other languages protect against buffer overflows. But using another language doesn't eliminate all problems. Many languages depend on C libraries, and many have mechanisms that turn off the protections (trading off safety for speed). But even beyond that, no matter what language you use, there are many other mistakes developers can make that create vulnerabilities.

No matter what you do, it's incredibly difficult to develop programs without making a mistake, and even careful review often misses some of these. One of the most important methods for developing secure programs is to *minimize privileges*. That means that the various parts of your program should have only the privileges they need, and no more. That way, even if the program has defects (whose doesn't?), you're likely to keep the defect from turning into a security nightmare. But how can this be done practically? The next article will examine how to practically minimize privileges in Linux/UNIX systems, so you can protect yourself against your own inevitable mistakes.

## Resources

- Read all of the installments in David's *Secure programmer* column series on *developerWorks*.
- David's book *Secure Programming for Linux and Unix HOWTO* gives a detailed account of how to develop secure software.
- "The What, Why, and How of the 1988 Internet Worm" gives more detail about the 1988 Morris worm.
- New IT Concerns in the Age of Anti-Terrorism: How the Canadian Government has Reacted and How Business Should React by C. Ian Kyer, Warren J. Sheffer, and Bruce Salvatore, Fasken Martineau DuMoulin LLP, notes that the Morris worm took down about 10% of the estimated 88,000 Internet computers of the time.
- CERT(R) Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL gives more detail about Code Red.
- " Frontline: Cyber War!: The Warnings?" summarizes various attacks and their known effects, including Code Red and Slammer.
- "Smashing The Stack For Fun And Profit" by Aleph One (Elias Levy) (*Phrack Magazine*, 8 Nov 1996, issue 49 article 14) explains how stack-smashing attacks work. Stack-smashing attacks had been occurring for many years before this article, but this article does a good job explaining them.
- David's "More than a Gigabuck: Estimating GNU/Linux's Size" examined Red Hat Linux 7.1's source code. It found that this distribution includes over 30 million physical source lines of code (SLOC), 86% of which were written in C or C++. It also found that it would have cost over $1 billion (a Gigabuck) and 8,000 person-years to develop this Linux

distribution by conventional proprietary means in the U.S. (in year 2000 U.S. dollars).

- "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" by Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole discusses the Stackguard approach to countering stack-smashing attacks; the Web site contains references to other work by Cowan for countering attacks. This paper includes a summary of the informal 1999 survey of Bugtraq.
- IBM's stack-smashing protector (ssp, also known as ProPolice) Web site gives more information about ssp. ssp is used by OpenBSD.
- "Linux kernel patch from the Openwall Project" discusses Solar Designer's current patches to the Linux kernel (including the non-executable stack work).
- "Linux: Exec Shield Overflow Protection" discusses Ingo Molnar's exec-shield approach.
- OpenWall GNU/Linux (OWL) uses a version of Solar Designer's non-exec stack patch, while Red Hat Fedora uses exec shield -- both options result in non-executing stacks (most of the time).
- "The Safe C String (SafeStr) library" by Messier and Viega is an interesting library that provides simple and safe C string handling.
- The XXL library is a threadsafe exception handling and asset management library for C. It is available under the BSD license.
- The Flawfinder project page provides Flawfinder, a GPL'ed tool for finding problems in C and C++ programs.
- O'Reilly & Associates is publishing a series of excerpts from *Practical UNIX & Internet Security, 3rd Edition* by Gene Spafford, Simson Garfinkel, and Alan Schwartz under the name *Secure Programming Techniques*.
- "Self-manage data buffer memory" (*developerWorks*, January 2004) describes how to allocate memory in C code only once the actual data becomes available -- when used correctly, this method minimizes the likelihood of buffer overruns.

- Find more resources for Linux developers in the developerWorks Linux section.
- Browse for books on these and other technical topics.

## About the author

David A. Wheeler is an expert in computer security and has long worked in improving development techniques for large and high-risk software systems. Mr. Wheeler is the author of the book *Secure Programming for Linux and UNIX HOWTO* and is a validator for the Common Criteria. Mr. Wheeler also wrote the article "Why Open Source Software/Free Software? Look at the Numbers!" and the Springer-Verlag book *Ada95: The Lovelace Tutorial*, and is the co-author and lead editor of the IEEE book *Software Inspection: An Industry Best Practice*. This article presents the opinions of the author and does not necessarily represent the position of the Institute for Defense Analyses. You can contact David at dwheelerNOSPAM@dwheeler.com (after removing "NOSPAM").