Make your software behave: Preventing buffer overflows
Protect your code through defensive programming

Gary McGraw and John Viega
Reliable Software Technologies
March 7, 2000

In our previous column, we described buffer overflow attacks at a high level and discussed why buffer overflows are such a large security problem. Protecting your code from buffer overflow attacks through defensive programming is the topic of this column. We will cover the major security snares in the C programming language, show why you should avoid particular constructs, and demonstrate preferred programming practices. Finally, we will discuss other technologies that can help you stop buffer overflows effectively.

Most buffer overflow problems in C can be traced directly back to the standard C library. The worst culprits are the problematic string operations that do no argument checking (strcpy, strcat, sprintf, gets). Generally speaking, hard and fast rules like "Avoid strcpy()" and "Never use gets()" are close to the mark.

Programs written today still make use of these calls, because developers are never taught to avoid them. Some people pick up a hint here and there, but even good developers can screw up. They may use homegrown checks on the arguments to dangerous functions, or incorrectly reason that the use of a potentially dangerous function is "safe" in some particular case.

Public enemy number one is gets(). Never use gets(). This function reads a line of user-typed text from the standard input, and does not stop reading text until it sees an EOF character or a newline character. That's right: gets() performs no bounds checking at all. It is always possible to overflow any buffer using gets(). As an alternative, use the method fgets(). It can do the same things gets() does, but it accepts a size parameter to limit the number of characters read in, thus giving you a way to prevent buffer overflows. For example, instead of the following code:

```
void main()
  {
  char buf[1024];
  gets(buf);
  }
```

Use the following:

```
#define BUFSIZE 1024

void main()
  {
  char buf[BUFSIZE];
  fgets(buf, BUFSIZE, stdin);
  }
```

Major snares in C programming
A number of standard functions in the C language have great potential to get you in trouble. But not all of their uses are bad. Usually, exploiting one of these functions requires an arbitrary input to be passed to the function. This list includes:

- strcpy()
- strcat()
- sprintf()
- scanf()
- sscanf()
- fscanf()

- vfscanf()
- vsprintf
- vscanf()
- vsscanf()
- streadd()
- strecpy()
- strtrns()

The bad news is that we recommend you avoid these functions if at all possible. The good news is that in most cases there are reasonable alternatives. We'll go over each one of them, so you can see what constitutes their misuse, and how to avoid it.

The **strcpy()** function copies a source string into a buffer. No specific number of characters will be copied. The number of characters copied depends directly on how many characters are in the source string. If the source string happens to come from user input, and you don't explicitly restrict its size, you could potentially be in big trouble!

If you know the size of the destination buffer, you can add an explicit check:

```
if(strlen(src) >= dst_size) {

  /* Do something appropriate, such as throw an error. */
  }
  else {
  strcpy(dst, src);
```

An easier way to accomplish the same goal is to use the strncpy() library routine:

```
strncpy(dst, src, dst_size-1);
  dst[dst_size-1] = '\0'; /* Always do this to be safe! */
```

This function doesn't throw an error if src is bigger than dst; it just stops copying characters when the maximum size has been reached. Note the -1 in the above call to strncpy(). That gives us room to put a null character in at the end if src is longer than dst.

Of course, it is possible to use strcpy() without any potential for security problems, as can be seen in the following example:

```
strcpy(buf, "Hello!");
```

Even if this operation does overflow buf, it will only do so by a few characters. Since we know statically what those characters are, and since they are quite obviously harmless, there's nothing to worry about here -- unless the static storage in which the string "Hello!" lives can be overwritten by some other means, of course.

Another way to be sure your strcpy() does not overflow is to allocate space when you need it, making sure to allocate enough space by calling strlen() on the source string. For example:

```
dst = (char *)malloc(strlen(src));
  strcpy(dst, src);
```

The **strcat()** function is very similar to strcpy(), except it concatenates one string onto the end of a buffer. It too has a similar, safer alternative, strncat(). Use strncat() instead of strcat(), if you can.

The functions **sprintf()** and **vsprintf()** are versatile functions for formatting text and storing it into a buffer. They can be used to mimic the behavior of strcpy() in a straightforward way. In other words, it is just as easy to add a buffer overflow to your program using sprintf() and vsprintf() as with strcpy(). For example, consider the following code:

```
void main(int argc, char **argv)
  {
  char usage[1024];
  sprintf(usage, "USAGE: %s -f flag [arg1]\n", argv[0]);
  }
```

We see code just like this pretty often. It looks harmless enough. It creates a string that knows how the program was invoked. That way, the name of the binary can change, and the program's output will automatically reflect the change. Nonetheless, something is seriously wrong with the code. File systems tend to restrict the name of any file to a certain number of characters. So you'd think that if your buffer is big enough to handle the longest name possible, your program would be safe, right? Just change 1024 to whatever number is right for our operating system and we're done? But no. We can easily subvert this restriction by writing our own little program to start the one above:

```
void main()
  {
  execl("/path/to/above/program",
  <<insert really long string here>>,
  NULL);
  }
```

The function execl() starts the program named in the first argument. The second argument gets passed as argv[0] to the called program. We can make that string as long as we want!

So how do we get around the problems with {v}sprintf()? Unfortunately, there is no completely portable way. Some architectures provide a snprintf() method, which allows the programmer to specify how many characters to copy into the buffer from each source. For example, if snprintf were available on our system, we could fix the previous example to read:

```
void main(int argc, char **argv)
  {
  char usage[1024];
  char format_string = "USAGE: %s -f flag [arg1]\n";
  snprintf(usage, format_string, argv[0],
  1024-strlen(format_string) + 1);
  }
```

Notice that up until the fourth argument, snprintf() is the same as sprintf(). The fourth argument specifies the maximum number of characters from the third argument that should be copied into the buffer usage. Note that 1024 is the wrong number! We must be sure that the total length of the string we copy into use does not exceed the size of the buffer. So we have to take into account a null character, plus all the characters in the format string, minus the formatting specifier %s. The number turns out to be 1000, but the code above is more maintainable, since it will not break if the format string happens to change.

Many (but not all) versions of {v}sprintf() come with a safer way to use the two functions. You can specify a precision for each argument in the format string itself. For example, another way to fix the broken sprintf() from above is:

```
void main(int argc, char **argv)
  {
  char usage[1024];
  sprintf(usage, "USAGE: %.1000s -f flag [arg1]\n", argv[0]);
  }
```

Notice the .1000 after the percent, and before the s. The syntax indicates that no more than 1000 characters should be copied from the associated variable (in this case, argv[0]).

If neither solution is available on a system in which your program must run, then your best solution is to package a working version of snprintf() along with your code. A freely available one can be found in sh archive format; see Resources.

Moving on, the **scanf** family of functions is also poorly designed. In this case, destination buffers can be overflowed. Consider the following code:

```
void main(int argc, char **argv)
  {
  char buf[256];
  sscanf(argv[0], "%s", &buf);
  }
```

If the scanned word is larger than the size of buf, we have an overflow condition. Fortunately, there's an easy way around this problem. Consider the following code, which does not have a security vulnerability:

```
void main(int argc, char **argv)
  {
  char buf[256];
  sscanf(argv[0], "%255s", &buf);
  }
```

The 255 between the percent and the s specifies that no more than 255 characters from argv[0] should actually be stored in the variable buf. The rest of the matching characters will not be copied.

Next we turn to **streadd()** and **strecpy()**. While not every machine has these calls to begin with, programmers who have these functions available should be cautious when using them. These functions translate a string that might have unreadable characters into a printable representation. For example, consider the following program:

```
#include <libgen.h>

void main(int argc, char **argv)
  {
  char buf[20];
  streadd(buf, "\t\n", "");
  printf(%s\n", buf);
  }
```

This program prints:

```
\t\n
```

instead of printing all white space. The streadd() and strecpy() functions can be problematic if the programmer doesn't anticipate how big the output buffer needs to be to handle the input without overflowing. If the input buffer contains a single character -- say, ASCII 001 (control-A) -- then it will print as four characters, "\001". This is the worst case of string growth. If you don't allocate enough space so that the output buffer is always four times larger than the size of the input buffer, a buffer overflow will be possible.

Another less common function is **strtrns()**, since many machines don't have it. The function strtrns() takes, as its arguments, three strings and a buffer into which a result string should be stored. The first string is essentially copied into the buffer. A character gets copied from the first string to the buffer, unless that character appears in the second string. If it does, then the character at the same index in the third string gets substituted instead. This sounds a bit confusing. Let's look at an example that converts all lower-case characters into upper-case characters for argv[1]:

```
#include <libgen.h>

void main(int argc, char **argv)
  {
  char lower[] = "abcdefghijklmnopqrstuvwxyz";
  char upper[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  char *buf;
  if(argc < 2) {
  printf("USAGE: %s arg\n", argv[0]);
  exit(0);
  } buf = (char *)malloc(strlen(argv[1]));
  strtrns(argv[1], lower, upper, buf);
  printf("%s\n", buf);
  }
```

The above code doesn't actually contain a buffer overflow. But if we had used a fixed-sized static buffer, instead of using malloc() to allocate enough space to copy argv[1], then a buffer overflow condition might have arisen.

Avoiding internal buffer overflows
The realpath() function takes a string that can potentially contain relative paths, and converts it to a string that refers to the same file, but via an absolute path. While it's doing this, it expands all symbolic links.

This function takes two arguments, the first being the string to canonicalize, and the second being a buffer into which the result should be stored. Of course, you need to make sure that the results buffer is big enough to handle any size path. Allocating a buffer of MAXPATHLEN should be sufficient. However, there's another problem with realpath(). If you pass it a path to canonicalize that is larger than MAXPATHLEN, a static buffer inside the implementation of realpath() overflows! You don't actually have access to the buffer that is overflowing, but it hurts you anyway. As a result, you should definitely not use realpath(), unless you are sure to check that the length of the path you are trying to canonicalize is no longer than MAXPATHLEN.

Other widely available calls have similar problems. The very commonly used call syslog() had a similar problem until it was noticed and fixed not long ago. This problem has been corrected on most machines, but you should not rely on correct behavior. It is best always to assume your code is running in the most hostile environment possible, just in case someday it does. Various implementations of the getopt() family of calls, as well as the getpass() function, are susceptible to overflows of internal static buffers as well. If you have to use these functions, the best solution is always to threshold the length of the inputs you pass them.

It is pretty easy to simulate gets() on your own, security problem and all. Take, for example, the following code:

```
char buf[1024];
  int i = 0;
  char ch;
  while((ch = getchar()) != '\n')
  {
  if(ch == -1) break;
  buf[i++] = ch;
  }
```

Oops! Any function that you can use to read in a character can fall prey to this problem, including getchar(), fgetc(), getc(), and read().

The moral of the buffer overflow problem is: Always make sure to do bounds checking.

It's too bad that C and C++ don't do bounds checking automatically, but there is actually a good reason why they don't. The price of bounds checking is efficiency. In general, C favors efficiency in most tradeoffs. The price of this efficiency gain, however, is that C programmers have to be on their toes and extremely security conscious to keep their programs out of trouble, and even then, keeping code out of trouble isn't easy.

In this day and age, argument checking doesn't constitute a big drain on a program's efficiency. Most applications will never notice the difference. So always bounds check. Check the length of data before you copy it into your own buffers. Also, check to make sure you're not passing overly large data to another library, since you can't trust other people's code either! (Recall the internal buffer overflows we talked about earlier.)

What are the other risks?
Unfortunately, even the "safe" versions of system calls -- such as strncpy() as opposed to strcpy() -- aren't completely safe. There will still be a chance to mess things up. Even "safe" calls can sometimes leave strings unterminated, or encourage subtle off-by-one bugs. And, of course, if you happen to use a result buffer that is smaller than the source buffer, you might find yourself in a very difficult place.

These mistakes tend to be harder to make than everything else we've talked about so far, but you should still be aware of them. Think carefully when you use this type of call. Many functions will misbehave if you're not keeping careful track of buffer sizes, including bcopy(), fgets(), memcpy(), snprintf(), strccpy(), strcadd(), strncpy(), and vsnprintf().

Another system call to avoid is getenv(). The biggest problem with getenv() is that you should never assume that a particular environment variable is of any particular length. We'll discuss the myriad problems with environment variables in a subsequent column.

So far we've given you a big laundry list of common C functions that are susceptible to buffer overflow problems. There are certainly many more functions with the same problems. In particular, beware of third-party COTS software. Don't assume anything about the behavior of someone else's software. Also be aware that we haven't carefully scrutinized every common library on every platform (we wouldn't want that job), and other problematic calls probably exist.

Even if we examined every common library everywhere, you should be very, very skeptical if we tried to claim we've listed all the problems you'll ever run across. We just want to give you a running start. The rest is up to you.

Static and dynamic testing tools
We'll cover vulnerability detection tools in more detail in later columns, but it is worth mentioning now that two kinds of scanning tools have proven effective in helping to find and remove buffer overflow problems. The two major categories of analysis tools are static tools (in which the code is considered but never run) and dynamic tools (in which the code is executed to determine behavior).

A number of static tools can be used to look for potential buffer overflow problems. Too bad none of them are available to the general public! Many of the tools do little more than automate the grep commands you can run to find instances of each problematic function in source code. While better techniques exist, this is still a highly effective way to narrow a large program of tens or hundreds of thousands of lines into just a few hundred "potential problems." (In a later column, we'll show you a quick-and-dirty scanning tool based on this approach, and give you an idea of how it was built.)

Better static tools make use of data flow information in some fashion to figure out which variables affect which other variables. In this way, some of the "false positives" from grep-based analysis can be discarded. David Wagner has implemented such an approach in his work (described in "Learning the basics of buffer overflows"; see Resources), as have researchers at Reliable Software Technologies. The problem with the data flow-related approach is that it currently introduces false negatives (that is, it doesn't flag some calls that could be real problems).

The second category of approach involves the use of dynamic analysis. Dynamic tools usually do something to keep an eye on code as it runs, looking for potential problems. One approach that has worked in the lab is fault injection. The idea is to instrument a program in such a way that you can experiment with it, running "what if" games and watching what happens. One such fault injection tool, FIST (see Resources), has been used to locate potential buffer overflow vulnerabilities.

Ultimately, some combination of dynamic and static approaches will give the most return for your investment. Much work remains to be done to determine the best combination.

Java and stack protection can help
As we mentioned in the previous column (see Resources), stack smashing is the worst sort of buffer overflow attack, especially when the stack being smashed is running in privileged mode. An elegant solution to this problem is non-executable stacks.

Usually, exploit code is written onto the program stack, and executed there. (We'll explain how this is done in our next column.) It is possible to get non-executable stack patches for many operating systems, including Linux and Solaris. (Some operating systems don't even need the patch; they just come that way.)

Non-executable stacks have some performance implications. (There is no free lunch.) In addition, they are easy to defeat in programs where there is both a stack overflow and a heap overflow. The stack overflow can be leveraged to cause the program to jump to the exploit code, which is placed in the heap. None of the code on the stack actually executes, just the code in the heap. These fundamental concerns are important enough that we'll devote the next column to them.

Of course, another option is to use a type-safe language such as Java. A less drastic measure is to get a compiler that will perform array bounds checking for C programs. Such a tool exists for gcc. This technique has the advantage of preventing all buffer overflows, heap and stack. The downside is that for some pointer-intensive programs where speed is critical, this technique may hamper adequate performance. But in most situations, this technique will work extraordinarily well.

The Stackguard tool implements a much more efficient technique than generalized bounds checking. It puts a little bit of data at the end of stack allocated data, and later checks to see if the data are still there before a buffer overflow might possibly occur. This pattern is called a "canary." (Welsh miners placed a canary in a mine to signal hazardous conditions. When the air became poisonous, the canary would collapse, hopefully giving the miners enough time to notice and escape.)

The Stackguard approach is not quite as secure as generalized bounds checking, but still quite useful. Stackguard's main disadvantage when compared to generalized bounds checking is that it does not prevent heap overflow attacks. Generally, it is best to protect your entire operating system with such a tool, otherwise unprotected libraries called by your program (such as the standard libraries) can still open the door to stack-based exploit code attacks.

Tools similar to Stackguard are memory-integrity-checking packages such as Rational's Purify. This sort of tool can even protect against heap overflows, but is not generally used in production code due to the performance overhead.

Summary
In the past two installments of this column, we've introduced you to buffer overflows and given you guidance on how to write code to avoid these problems. We've also talked about a few tools that can help keep your programs safe from the dreaded buffer overflow. Table 1 summarizes the programming constructs we've suggested you use with caution or avoid altogether. If you have any other functions you think we should add to this list, please let us know, and we'll update the list.

Function	Severity	Solution
gets	Most risky	Use fgets(buf, size, stdin). This is almost always a big problem!
strcpy	Very risky	Use strncpy instead.
strcat	Very risky	Use strncat instead.
sprintf	Very risky	Use snprintf instead, or use precision specifiers.
scanf	Very risky	Use precision specifiers, or do your own parsing.
sscanf	Very risky	Use precision specifiers, or do your own parsing.
fscanf	Very risky	Use precision specifiers, or do your own parsing.
vfscanf	Very risky	Use precision specifiers, or do your own parsing.
vsprintf	Very risky	Use vsnprintf instead, or use precision specifiers.
vscanf	Very risky	Use precision specifiers, or do your own parsing.
vsscanf	Very risky	Use precision specifiers, or do your own parsing.
streadd	Very risky	Make sure you allocate 4 times the size of the source parameter as the size of the destination.
strecpy	Very risky	Make sure you allocate 4 times the size of the source parameter as the size of the destination.
strtrns	Risky	Manually check to see that the destination is at least the same size as the source string.

realpath	Very risky (or less, depending on the implementation)	Allocate your buffer to be of size MAXPATHLEN. Also, manually check arguments to ensure the input argument is no larger than MAXPATHLEN.
syslog	Very risky (or less, depending on the implementation)	Truncate all string inputs at a reasonable size before passing them to this function.
getopt	Very risky (or less, depending on the implementation)	Truncate all string inputs at a reasonable size before passing them to this function.
getopt_long	Very risky (or less, depending on the implementation)	Truncate all string inputs at a reasonable size before passing them to this function.
getpass	Very risky (or less, depending on the implementation)	Truncate all string inputs at a reasonable size before passing them to this function.
getchar	Moderate risk	Make sure to check your buffer boundaries if using this function in a loop.
fgetc	Moderate risk	Make sure to check your buffer boundaries if using this function in a loop.
getc	Moderate risk	Make sure to check your buffer boundaries if using this function in a loop.
read	Moderate risk	Make sure to check your buffer boundaries if using this function in a loop.
bcopy	Low risk	Make sure that your buffer is as big as you say it is.
fgets	Low risk	Make sure that your buffer is as big as you say it is.
memcpy	Low risk	Make sure that your buffer is as big as you say it is.
snprintf	Low risk	Make sure that your buffer is as big as you say it is.
strccpy	Low risk	Make sure that your buffer is as big as you say it is.
strcadd	Low risk	Make sure that your buffer is as big as you say it is.
strncpy	Low risk	Make sure that your buffer is as big as you say it is.
vsnprintf	Low risk	Make sure that your buffer is as big as you say it is.

In our haste to cover ground, we have left out, until now, some of the cool details behind buffer overflow. In our next couple of columns, we'll reach deep into the engine's workings and get greasy. We'll go into detail about how buffer overflows work, even showing you some exploit code.

Resources

- A free, working version of snprintf() in sh archive format
- FIST, a fault-injection tool for locating potential buffer overflow vulnerabilities
- "Make your software behave: Learning the basics of buffer overflows" on *developerWorks*, where Gary and John describe buffer overflow attacks at a high level and discuss why buffer overflows are the single biggest software security threat
- "Make your software behave: Assuring your software is secure" on *developerWorks*, where Gary and John lay out their five-step process for designing for security
- The first "Make your software behave" column on *developerWorks*, where Gary and John introduce their philosophy of security, and explain why they're focusing on software security issues facing developers

About the authors
Gary McGraw is the vice president of corporate technology at Reliable Software Technologies in Dulles, VA. Working with Consulting Services and Research, he helps set technology research and development direction. McGraw began his career at Reliable Software Technologies as a Research Scientist, and he continues to pursue research in Software Engineering and computer security. He holds a dual Ph.D. in Cognitive Science and Computer Science from Indiana University and a B.A. in

Philosophy from the University of Virginia. He has written more than 40 peer-reviewed articles for technical publications, consults with major e-commerce vendors including Visa and the Federal Reserve, and has served as principal investigator on grants from Air Force Research Labs, DARPA, National Science Foundation, and NIST's Advanced Technology Program.

McGraw is a noted authority on mobile code security and co-authored both "Java Security: Hostile Applets, Holes, & Antidotes" (Wiley, 1996) and "Securing Java: Getting down to business with mobile code" (Wiley, 1999) with Professor Ed Felten of Princeton. Along with RST co-founder and Chief Scientist Dr. Jeffrey Voas, McGraw wrote "Software Fault Injection: Inoculating Programs Against Errors" (Wiley, 1998). McGraw regularly contributes to popular trade publications and is often quoted in national press articles.

John Viega is a Senior Research Associate, Software Security Group co-founder, and Senior Consultant at Reliable Software Technologies. He is the Principal Investigator on a DARPA-sponsored grant for developing security extensions for standard programming languages. John has authored over 30 technical publications in the areas of software security and testing. He is responsible for finding several well-publicized security vulnerabilities in major network and e-commerce products, including a recent break in Netscape's security. He is also a prominent member of the open-source software community, having written Mailman, the GNU Mailing List Manager, and, more recently, ITS4, a tool for finding security vulnerabilities in C and C++ code. Viega holds a M.S. in Computer Science from the University of Virginia.

What do you think of this article?

○ Killer! ○ Good stuff ○ So-so; not bad ○ Needs work ○ Lame!

Comments?

Submit feedback