strlcpy and strlcat — consistent, safe, string copy and concatenation.

Todd C. Miller
University of Colorado, Boulder
Theo de Raadt
OpenBSD project

Abstract

As the prevalence of buffer overflow attacks has increased, more and more programmers are using size or length-bounded string functions such as strncpy() and strncat(). While this is certainly an encouraging trend, the standard C string functions generally used were not really designed for the task. This paper describes an alternate, intuitive, and consistent API designed with safe string copies in mind.

There are several problems encountered when strncpy() and strncat() are used as safe versions of strcpy() and strcat(). Both functions deal with NUL-termination and the length parameter in different and non-intuitive ways that confuse even experienced programmers. They also provide no easy way to detect when truncation occurs. Finally, strncpy() zero-fills the remainder of the destination string, incurring a performance penalty. Of all these issues, the confusion caused by the length parameters and the related issue of NUL-termination are most important. When we audited the OpenBSD source tree for potential security holes we found rampant misuse of strncpy() and strncat(). While not all of these resulted in exploitable security holes, they made it clear that the rules for using strncpy() and strncat() in safe string operations are widely misunderstood. The proposed replacement functions, strlcpy() and strlcat(), address these problems by presenting an API designed for safe string copies (see Figure 1 for function prototypes). Both functions guarantee NUL-termination, take as a length parameter the size of the string in bytes, and provide an easy way to detect truncation. Neither function zero-fills unused bytes in the destination.

Introduction

In the middle of 1996, the authors, along with other members of the OpenBSD project, undertook an audit of the OpenBSD source tree looking for security problems, starting with an emphasis on buffer overflows. Buffer overflows [1] had recently gotten a lot of attention in forums such as BugTraq [2] and were being widely exploited. We found a large number of overflows due to unbounded string copies using sprintf(), strcpy() and strcat(), as well as loops that manipulated strings without an explicate length check in the loop invariant. Additionally, we also found many instances where the programmer had tried to do safe string manipulation with strncpy() and strncat() but failed to grasp the subtleties of the API.

Thus, when auditing code, we found that not only was it necessary to check for unsafe usage of functions like strcpy() and strcat(), we also had to check for incorrect usage of strncpy() and strncat(). Checking for correct usage is not always obvious, especially in the case of "static" variables or buffers allocated via calloc(), which are effectively pre-terminated. We came to the conclusion that a foolproof alternative to strncpy() and strncat() was needed, primarily to simplify the job of the programmer, but also to make code auditing easier.

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

Figure 1: ANSI C prototypes for strlcpy() and strlcat()

Common Misconceptions

The most common misconception is that strncpy() NUL-terminates the destination string. This is only true, however, if length of the source string is less than the size parameter. This can be problematic when copying user input that may be of arbitrary length into a fixed size buffer. The safest way to use strncpy() in this situation is to pass it one less than the size of the destination string, and then terminate the string by hand. That way you are guaranteed to always have a NUL-terminated destination string. Strictly speaking, it is not necessary to hand-terminate the string if it is a "static" variable or if it was allocated via calloc() since such strings are zeroed out when allocated. However, relying on this feature is generally confusing to those persons who must later maintain the code.

There is also an implicate assumption that converting code from strcpy() and strcat() to strncpy() and strncat() causes negligible performance degradation. With this is true of strncat(), the same cannot be said for strncpy() since it zero-fills the remaining bytes not used to store the string being copied. This can lead to a measurable performance hit when the size of the destination string is much greater than the length of the source string. The exact penalty for using strncpy() due to this behavior varies by CPU architecture and implementation.

The most common mistake made with strncat() is to use an incorrect size parameter. While strncat() does guarantee to NUL-terminate the destination, you must not count the space for the NUL in the size parameter. Most importantly, this is not the size of the destination string itself, rather it is the amount of space available. As this is almost always a value that must be computed, as opposed to a known constant, it is often computed incorrectly.

How do strlcpy() and strlcat() help things?

The strlcpy() and strlcat() functions provide a consistent, unambiguous API to help the programmer write more bullet-proof code. First and foremost, both strlcpy() and strlcat() guarantee to NUL-terminate the destination string for all strings where the given size is non-zero. Secondly, both functions take the full size of the destination string as a size parameter. In most cases this value is easily computed at compile time using the `sizeof` operator. Finally, neither strlcpy() nor strlcat() zero-fill their destination strings (other than the compulsatory NUL to terminate the string).

The strlcpy() and strlcat() functions return the total length of the string they tried to create. For strlcpy() that is simply the length of the source; for strlcat() that means the length of the destination (before concatenation) plus the length of the source. To check for truncation, the programmer need only verify that the return value is less than the size parameter. Thus, if truncation has occurred, the number of bytes needed to store the entire string is now known and the programmer may allocate more space and re-copy the strings if he or she wishes. The return value has similar semantics to the return value of snprintf() as implemented in BSD and as specified by the upcoming C9X specification [4] (note that not all snprintf() implementations currently comply with C9X). If no truncation occurred, the programmer now has the length of the resulting string. This is useful since it is common practice to build a string with strncpy() and strncat() and then to find the length of the result using strlen(). With strlcpy() and strlcat() the final strlen() is no longer necessary.

Example 1a is a code fragment with a potential buffer overflow (the HOME environment variable is controlled by the user and can be of arbitrary length).

```
strcpy(path, homedir);
strcat(path, "/");
strcat(path, ".foorc");
len = strlen(path);
```

Example 1a: Code fragment using strcpy() and strcat()

Example 1b is the same fragment converted to safely use strncpy() and strncat() (note that we have to terminate the destination by hand).

```
strncpy(path, homedir,
    sizeof(path) - 1);
path[sizeof(path) - 1] = '\0';
strncat(path, "/",
    sizeof(path) - strlen(path) - 1);
strncat(path, ".foorc",
    sizeof(path) - strlen(path) - 1);
len = strlen(path);
```

Example 1b: Converted to strncpy() and strncat()

Example 1c is a trivial conversion to the strlcpy()/strlcat() API. It has the advantage of being as simple as Example 1a, but it does not take advantage of the new API's return value.

```
strlcpy(path, homedir, sizeof(path));
strlcat(path, "/", sizeof(path));
strlcat(path, ".foorc", sizeof(path));
len = strlen(path);
```

Example 1c: Trivial conversion to strlcpy()/strlcat()

Since Example 1c is so easy to read and comprehend, it is simple to add additional checks to it. In Example 1d, we check the return value to make sure there was enough space for the source string. If there was not, we return an error. This is slightly more complicated but in addition to being more robust, it also avoids the final strlen() call.

```
len = strlcpy(path, homedir,
    sizeof(path);
if (len >= sizeof(path))
      return (ENAMETOOLONG);
len = strlcat(path, "/",
    sizeof(path);
if (len >= sizeof(path))
      return (ENAMETOOLONG);
len = strlcat(path, ".foorc",
    sizeof(path));
if (len >= sizeof(path))
      return (ENAMETOOLONG);
```

Example 1d: Now with a check for truncation

Design decisions

A great deal of thought (and a few strong words) went into deciding just what the semantics of strlcpy() and strlcat() would be. The original idea was to make strlcpy() and strlcat() identical to strncpy() and strncat() with the exception that they would always NUL-terminate the destination string. However, looking back on the common use (and misuse) of strncat() convinced us that the size parameter for strlcat() should be the full size of the string and not just the number of characters left unallocated. The return values started out as the number of characters copied, since this was trivial to get as a side effect of the copy or concatenation. We soon decided that a return value with the same semantics as snprintf()'s was a better choice since it gives the programmer the most flexibility with respect to truncation detection and recovery.

Performance

Programmers are starting to avoid strncpy() due its poor performance when the target buffer is significantly larger than the length of the source string. For instance, the apache group [6] replaced calls to strncpy() with an internal function and noticed a performance improvement [7]. Also, the ncurses [8] package recently removed an occurrence of strncpy(), resulting in a factor of four speedup of the *tic* utility. It is our hope that, in the future, more programmers will use the interface provided by strlcpy() rather than using a custom interface.

To get a feel for the worst-case scenario in comparing strncpy() and strlcpy(), we ran a test program that copies the string "this is just a test" 1000 times into a 1024 byte buffer. This is somewhat unfair to strncpy(), since by using a small string and a large buffer strncpy() has to fill most of the buffer with NUL characters. In practice, however, it is common to use a buffer that is much larger than the expected user input. For instance, pathname buffers are MAXPATHLEN long (1024 bytes), but most filenames are significantly shorter than that. The averages run times in Table 1 were generated on an HP9000/425t with a 25Mhz 68040 CPU running OpenBSD 2.5 and a DEC AXP-PCI166 with a 166Mhz alpha CPU also running OpenBSD 2.5. In all cases, the same C versions of the functions were used and the times are the "real time" as reported by the *time* utility.

cpu	function	time
m68k	strcpy	0.137
m68k	strncpy	0.464
m68k	strlcpy	0.14
alpha	strcpy	0.018
alpha	strncpy	0.10
alpha	strlcpy	0.02

Table 1: Performance timings in seconds

As can be seen in Table 1, the timings for strncpy() are far worse than those for strcpy() and strlcpy(). This is probably due not only to the cost of NUL padding but also because the CPU's data cache is effectively being flushed by the long stream of zeroes.

What strlcpy() and strlcat() are not

While strlcpy() and strlcat() are well-suited for dealing with fixed-size buffers, they cannot replace strncpy() and strncat() in all cases. There are still times where it is necessary to manipulate buffers that are not true C strings (the strings in `struct utmp` for instance). However, we would argue that such "pseudo strings" should not be used in new code since they are prone to misuse, and in our experience, a common source of bugs. Additionally, the strlcpy() and strlcat() functions are not an attempt to "fix" string handling in C, they are designed to fit within the normal framework of C strings. If you require string functions that support dynamically allocated, arbitrary sized buffers you may wish to examine the "astring" package from mib software [9].

Who uses strlcpy() and strlcat()?

The strlcpy() and strlcat() functions first appeared in OpenBSD 2.4. The functions have also recently been approved for inclusion in a future version of Solaris. Third-party packages are starting to pick up the API as well. For instance, the rsync [5] package now uses strlcpy() and provides its own version if the OS does not support it. It is our hope that other operating systems and applications will use strlcpy() and strlcat() in the future, and that it will receive standards acceptance at some time.

What's Next?

We plan to replace occurrences of strncpy() and strncat() with strlcpy() and strlcat() in OpenBSD where it is sensible to do so. While new code in OpenBSD is being written to use the new API, there is still a large amount of code that was converted to use strncpy() and strncat() during our original security audit. To this day, we continue to discover bugs due to incorrect usage of strncpy() and strncat() in existing code. Updating older code to use strlcpy() and strlcat() should serve to speed up some programs and uncover bugs in others.

Availability

The source code for strlcpy() and strlcat() is available free of charge and under a BSD-style license as part of the OpenBSD operating system. You may also download the code and its associated manual pages via anonymous ftp from ftp.openbsd.org in the directory /pub/OpenBSD/src/lib/libc/string. The source code for strlcpy() and strlcat() is in strlcpy.c and strlcat.c. The documentation (which uses the tmac.doc troff macros) may be found in strlcpy.3.

Author Information

Todd C. Miller has been involved in the free software community since 1993 when he took over maintenance of the sudo package. He joined the OpenBSD project in 1996 as an active developer. Todd belatedly received a BS in Computer Science in 1997 from the University of Colorado, Boulder (after years of prodding). Todd has so far managed to avoid the corporate world and currently works as a Systems Administrator at the University of Colorado, Boulder blissfully ensconced in academia. He may be reached via email at <Todd.Miller@cs.colorado.edu>.

Theo de Raadt has been involved with free Unix operating systems since 1990. Early developments included porting Minix to the sun3/50 and amiga, and PDP-11 BSD 2.9 to a 68030 computer. As one of the founders of the NetBSD project, Theo worked on maintaining and improving many system components including the sparc port and a free YP implementation that is now in use by most free systems. In 1995 Theo created the OpenBSD project, which places focus on security, integrated cryptography, and code correctness. Theo works full time on advancing OpenBSD. He may be reached via email at <deraadt@openbsd.org>.

References

[1] Aleph One. "Smashing The Stack For Fun And Profit." *Phrack Magazine Volume Seven, Issue Forty-Nine.*

[2] BugTraq Mailing List Archives. http://www.geek-girl.com/bugtraq/. This web page contains searchable archives of the BugTraq mailing list.

[3] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, PTR, 1988.

[4] International Standards Organization. "C9X FCD, Programming languages — C" http://wwwold.dkuug.dk/jtc1/sc22/open/n2794/ This web page contains the current draft of the upcoming C9X standard.

[5] Andrew Tridgell, Paul Mackerras. *The rsync algorithm.* http://rsync.samba.org/rsync/tech_report/. This web page contains a technical report describing the rsync program.

[6] The Apache Group. The Apache Web Server. http://www.apache.org. This web page contains information on the Apache web server.

[7] The Apache Group. New features in Apache version 1.3. http://www.apache.org/docs/new_features_1_3.html. This web page contains new features in version 1.3 of the Apache web server.

[8] The Ncurses (new curses) home page. http://www.clark.net/pub/dickey/ncurses/. This web page contains Ncurses information and distributions.

[9] Forrest J. Cavalier III. "Libmib allocated string functions." http://www.mibsoftware.com/libmib/astring/. This web page contains a description and implementation of a set of string functions that dynamically allocate memory as necessary.