

# Writing Fast And Secure Code in C

Sebastian Schinzel  
Version 1.0 - 2006-05-12

## Overview

Applications that were implemented using the C programming language have experienced a constant flow of security vulnerabilities for more than 20 years. Each year security researchers and hackers discover new code patterns in C that lead to exploitable vulnerabilities. This raises the necessity that programmers in a security sensitive environment are always up-to-date with current research in secure programming.

In this paper I give an overview on common patterns that lead to vulnerabilities and describe approaches to develop functionally equivalent and secure code. Furthermore, I show that secure string handling in C can be fast and elegant.

## Target audience

- Security Trainers
- Developers
- Testers



## Contents

Contents.....	2
I. How to Copy Character Strings .....	3
1. Description.....	3
2. Examples .....	3
II. A Notice on Format Strings.....	7
Further Information .....	7

# I. How to Copy Character Strings

## 1. Description

The process of copying a string into another memory location is difficult to do correctly with C library functions. Many of the critical applications that emerged in the last 20 years resulted out of programming errors in character string handling routines. These errors often lead to Buffer Overflows that may allow attackers to take over the vulnerable process.

In this chapter we show several common errors of C programmers that lead to security vulnerabilities. We describe the errors in detail and propose advanced code patterns that perform the same task in a secure fashion.

## 2. Examples

```
char *copy_str(char *src)
{
    static char dst[BUFSIZE];
    strcpy(dst, src);
    return dst;
}
```

The function `copy_str()` returns a copy of a supplied string in a static memory buffer. The actual copying is done by the `strcpy()` function, which leads to a buffer overflow if the length of `src` is larger than `BUFSIZE`.

A widespread solution to this is to substitute `strcpy()` with `strncpy()`:

```
char *copy_str(char *src)
{
    static char dst[BUFSIZE];
    strncpy(dst, src, BUFSIZE);
    return dst;
}
```

This approach comprises several problems. The most severe problem is that `strncpy()` does not always terminate `dst` after the copying, which may result in non-terminated strings in memory. The second problem is that `strncpy()` pads remaining bytes in `dst` with `'\0'`, which leads to a measurable performance decrease if the destination buffer is large compared to the source string. Maintainability of the function also suffers because the buffer limit that was supplied to `strncpy()` is a constant and not the actual size of `dst`. This may lead to problems if the programmer changes the size of `dst` and forgets to change the limit that is supplied to `strncpy()`. In order to calculate buffer limits one should always use `sizeof` on the variable when possible.

```
char *copy_str(char *src)
{
    static char dst[BUFSIZE];
    strncpy(dst, src, sizeof(dst) - 1);
    dst[sizeof(dst) - 1] = '\0';
    return dst;
}
```

This time, the string that the function returns will always be null-terminated and the size of `dst` has to be supplied only when `dst` is declared. However, the possible performance issues remain. Another problem with `strncpy()` emerges when the source buffer is larger than the destination buffer, because the destination buffer will represent a truncated copy of the source buffer. For example, a truncated file name may lead to new vulnerabilities. Therefore the programmer has to check for truncation, which is not possible with `strcpy()` and `strncpy()`. The programmer needs to check manually:

```

char *copy_str(char *src)
{
    static char dst[BUFSIZE];
    char *ret = NULL;

    dst[sizeof(dst) - 1] = '\\0';
    strncpy(dst, src, sizeof(dst));

    if(dst[sizeof(dst) - 1] == '\\0') {
        ret = dst;
    }
    return ret;
}

```

This approach uses the bugs of `strncpy()` as a feature. First, the last byte of `dst` is set to '\\0'. If truncation occurs, `strncpy()` uses all bytes of `dst` until it stops copying, thus overwriting the last byte. Therefore if the last byte of `dst` is still '\\0' after the call to `strncpy()`, `dst` contains a string that is not truncated and therefore valid.

On the other hand, `strcpy()` can be used in a way that is safe and secure:

```

char *copy_str(char *src)
{
    static char dst[BUFSIZE];
    char *ret = NULL;
    if(strlen(src) < sizeof(dst)) {
        strcpy(dst, src);
        ret = dst;
    }
    return ret;
}

```

This version of `copy_str()` returns a copy of the string that the programmer supplied as a parameter. The function is safe against buffer overflows and string truncation as it returns a NULL-pointer if truncation occurred. Even if the function is more efficient than the previous one, it still reads each byte in `src` twice (once in `strlen()` and once in `strcpy()`).

It is generally considered to be good practice to keep the size of each character string in a second variable. This does not only increase readability and maintainability, but often leaves some space for performance tuning.

```

char *copy_str(char *src, size_t len)
{
    static char dst[BUFSIZE];
    char *ret = NULL;
    if(len < sizeof(dst)) {
        strcpy(dst, src);
        ret = dst;
    }
    return ret;
}

```

Here, the additional parameter `len` saves one call to `strlen()`. Therefore, `copy_str()` only needs to touch the bytes in `src` once, which increases the performance of the function. An even faster version of `copy_str()` uses `memcpy()` instead of `strcpy()`. Both functions basically copy bytes from one memory location to another with the difference that `strcpy()` compares the current byte to be copied with the terminating NULL-byte, which is slower.

```

char *copy_str(char *src, size_t len)
{
    static char dst[BUFSIZE];
    char *ret = NULL;
    if(len < sizeof(dst)) {
        memcpy(dst, src, len + 1); // Include the terminating byte (len + 1)
        ret = dst;
    }
    return ret;
}

```

Another popular scenario is the concatenation of two strings.

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];
    strcpy(dst, "username: ");
    strcat(dst, src);
    return dst;
}
```

The following example concatenates two strings. The main problem here is that `src` is appended to `dst` no matter if `dst` is large enough to hold the additional characters of `dst`. This leads to buffer overflows. One possible solution to this is to use `strncat()`, which takes the amount of characters to be copied excluding the terminating NULL-byte.

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];
    strcpy(dst, "username: ");
    strncat(dst, src, sizeof(dst) - strlen(dst) - 1);
    return dst;
}
```

Note that `strncat()` does not take the total size of `dst`, which would be more intuitive. It takes the size of `dst` minus the length of the string that is already stored in `dst` minus the terminating NULL-byte. While it is up to the programmer to calculate this value correctly, the above example may also truncate the destination string if `src` is larger than the amount of remaining bytes in `dst`. Similar to `strcpy()`, `strncat()` does not provide any way to find out if `strncat()` produced a truncated string. Again this has to be achieved by the programmer:

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];
    char *ret = NULL;

    strcpy(dst, "username: ");

    if((strlen(dst) + strlen(src)) < sizeof(dst)) {
        strcat(dst, src);
        ret = dst;
    }
    return ret;
}
```

The above function is safe against buffer overflows and character string truncation because it returns a NULL-pointer if `dst` is too small to hold the resulting string.

Another possibility to append strings is through the `sprintf()` function:

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];

    sprintf(dst, sizeof(dst), "username: %s", src);

    return dst;
}
```

Fortunately, `snprintf()` returns the amount of bytes it would have copied if the destination buffer were larger. Therefore, truncation can be easily detected by comparing the return value of `snprintf()` with the size of the destination buffer. If the size of the buffer is smaller than the return value, the destination string is truncated.

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];
    char ret = NULL;
    int len = 0;

    len = snprintf(dst, sizeof(dst), "username: %s", src);
    if(len <= sizeof(dst)) {
        ret = dst;
    }

    return ret;
}
```

The function above eliminates the thread of buffer overflows and character string truncation because it returns a NULL-pointer if `src` is too large to fit into `dst`.

Note that `snprintf()` was officially introduced with C99. Before that, there were other implementations of `snprintf()` that may not behave as shown above. Microsoft, for example, ships a function called `_snprintf()` with Visual C++ 6.0 that does not always terminate the resulting string.

There is another set of string copy functions called `strncpy()` and `strncat()`. These functions were created by members of the OpenBSD project and provide an easy to use and secure way to copy strings.

Both functions return the amount of bytes that the functions would have copied if the destination buffer were large enough. If the return value of either `strncpy()` or `strncat()` is larger than the destination buffer, truncation occurred. This is efficient because the function counts the already copied bytes as it copies them.

```
char *append_str(char *src)
{
    static char dst[BUFSIZE];
    char ret = NULL;
    size_t len = 0;

    len = strncpy(dst, src, sizeof(dst));
    if(len <= sizeof(dst)) {
        ret = dst;
    }

    return ret;
}

char *append_str(char *src)
{
    static char dst[BUFSIZE];
    char ret = NULL;
    size_t len = 0;

    len = strncat(dst, src, sizeof(dst));
    if(len <= sizeof(dst)) {
        ret = dst;
    }

    return ret;
}
```

As one can see, the functions are very easy to use and efficient. They provide the easiest way to copy strings concerning readability, maintainability, security and efficiency. Unfortunately both functions are not included in the C standard, but they are freely available from the OpenBSD project and can be included into existing projects.

## II. A Notice on Format Strings

Format string vulnerabilities only appear when the programmer passes user supplied input as a format string to a function of the printf() family. The following shows a typical format string vulnerability:

```
void print_data(char *src)
{
    printf(src);
}
```

Note that printf() only prints a string to stdout. There is no string copying whatsoever that could lead to a buffer overflow. Nevertheless, the exploitation of format string vulnerabilities is achieved through a little known functionality of printf() family functions. The following shows this functionality:

```
void print_data()
{
    int len = 0;
    printf("1234%n ", &len);
    printf("len: %d\n", len);
}
```

The '%n' modifier takes a pointer to an integer as argument and stores the amount of already printed characters in the integer. The output of the example from above is as follows:

```
1234 len: 4
```

An attacker that is able to inject data into a format string may be able to write arbitrary values to any memory location, thus taking over the control of the entire process. However, format string vulnerabilities are very easy to prevent. The programmer must care for format strings to be always constant and that they never contain user-supplied data.

## Further Information

[Todd C. Miller, Theo de Raadt "strcpy, and strcat - consistent, safe, string copy and concatenation."](#)

[Tobias Klein "Buffer Overflows und Format-String Schwachstellen"](#)