2.5. String-Handling Functions | Secure Coding in C and C++: Strings and Buffer Overflows

By Robert C. Seacord Apr 24, 2013

2.5. String-Handling Functions

gets()

If there were ever a hard-and-fast rule for secure programming in C and C++, it would be this: never invoke the `gets()` function. The `gets()` function has been used extensively in the examples of vulnerable programs in this book. The `gets()` function reads a line from standard input into a buffer until a terminating newline or end-of-file (EOF) is found. No check for buffer overflow is performed. The following quote is from the manual page for the function:

- Never use `gets()`. Because it is impossible to tell without knowing the data in advance how many characters `gets()` will read, and because `gets()` will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security.

  As already mentioned, the `gets()` function has been deprecated in ISO/IEC 9899:TC3 and removed from C11.

- Because the `gets()` function cannot be securely used, it is necessary to use an alternative replacement function, for which several good options are available. Which function you select primarily depends on the overall approach taken.

C99

Two options for a strictly C99-conforming application are to replace `gets()` with either `fgets()` or `getchar()`.

The C Standard `fgets()` function has similar behavior to `gets()`. The `fgets()` function accepts two additional arguments: the number of characters to read and an input stream. When `stdin` is specified as the stream, `fgets()` can be used to simulate the behavior of `gets()`.

The program fragment in Example 2.9 reads a line of text from `stdin` using the `fgets()` function.

Example 2.9. Reading from stdin Using fgets()

```
char buf[LINE_MAX];
int ch;
char *p;

if (fgets(buf, sizeof(buf), stdin)) {
  /* fgets succeeds, scan for newline character */
```

```
    p = strchr(buf, '\n');
    if (p) {
      *p = '\0';
    }
    else {
      /* newline not found, flush stdin to end of line */
      while (((ch = getchar()) != '\n')
            && !feof(stdin)
            && !ferror(stdin)
      );
    }
  }
  else {
    /* fgets failed, handle error */
  }
```

Unlike `gets()`, the `fgets()` function retains the newline character, meaning that the function cannot be used as a direct replacement for `gets()`.

When using `fgets()`, it is possible to read a partial line. Truncation of user input can be detected because the input buffer will not contain a newline character.

The `fgets()` function reads, at most, one less than the number of characters specified from the stream into an array. No additional characters are read after a newline character or EOF. A null character is written immediately after the last character read into the array.

It is possible to use `fgets()` to securely process input lines that are too long to store in the destination array, but this is not recommended for performance reasons. The `fgets()` function can result in a buffer overflow if the specified number of characters to input exceeds the length of the destination buffer.

A second alternative for replacing the `gets()` function in a strictly C99-conforming application is to use the `getchar()` function. The `getchar()` function returns the next character from the input stream pointed to by `stdin`. If the stream is at EOF, the EOF indicator for the stream is set and `getchar()` returns `EOF`. If a read error occurs, the error indicator for the stream is set and `getchar()` returns `EOF`. The program fragment in Example 2.10 reads a line of text from `stdin` using the `getchar()` function.

Example 2.10. Reading from stdin Using getchar()

```
char buf[BUFSIZ];
int ch;
int index = 0;
int chars_read = 0;

while (((ch = getchar()) != '\n')
        && !feof(stdin)
        && !ferror(stdin))
{
  if (index < BUFSIZ-1) {
    buf[index++] = (unsigned char)ch;
  }
  chars_read++;
} /* end while */
buf[index] = '\0';  /* null-terminate */
if (feof(stdin)) {
  /* handle EOF */
}
if (ferror(stdin)) {
  /* handle error */
}
```

```
if (chars_read > index) {
  /* handle truncation */
}
```

If at the end of the loop `feof(stdin) ! = 0`, the loop has read through to the end of the file without encountering a newline character. If at the end of the loop `ferror(stdin) ! = 0`, a read error occurred before the loop encountered a newline character. If at the end of the loop `chars_read > index`, the input string has been truncated. *The CERT C Secure Coding Standard* [Seacord 2008], "FIO34-C. Use `int` to capture the return value of character IO functions," is also applied in this solution.

Using the `getchar()` function to read in a line can still result in a buffer overflow if writes to the buffer are not properly bounded.

Reading one character at a time provides more flexibility in controlling behavior without additional performance overhead. The following test for the `while` loop is normally sufficient:

while (((ch = getchar()) ! = '\n') && ch ! = EOF)

See *The CERT C Secure Coding Standard* [Seacord 2008], "FIO35-C. Use `feof()` and `ferror()` to detect end-of-file and file errors when `sizeof(int) == sizeof(char)`," for the case where `feof()` and `ferror()` must be used instead.

C11 Annex K Bounds-Checking Interfaces: gets_s()

The C11 `gets_s()` function is a compatible but more secure version of `gets()`. The `gets_s()` function is a closer replacement for the `gets()` function than `fgets()` in that it only reads from the stream pointed to by `stdin` and does not retain the newline character. The `gets_s()` function accepts an additional argument, `rsize_t`, that specifies the maximum number of characters to input. An error condition occurs if this argument is equal to zero or greater than `RSIZE_MAX` or if the pointer to the destination character array is `NULL`. If an error condition occurs, no input is performed and the character array is not modified. Otherwise, the `gets_s()` function reads, at most, one less than the number of characters specified, and a null character is written immediately after the last character read into the array. The program fragment shown in Example 2.11 reads a line of text from `stdin` using the `gets_s()` function.

Example 2.11. Reading from stdin Using gets_s()

```
char buf[BUFSIZ];

if (gets_s(buf, sizeof(buf)) == NULL) {
  /* handle error */
}
```

The `gets_s()` function returns a pointer to the character array if successful. A null pointer is returned if the function arguments are invalid, an end-of-file is encountered, and no characters have been read into the array or if a read error occurs during the operation.

The `gets_s()` function succeeds only if it reads a complete line (that is, it reads a newline character). If a complete line cannot be read, the function returns `NULL`, sets the buffer to the null string, and clears the input stream to the next newline character.

The `gets_s()` function can still result in a buffer overflow if the specified number of characters to input exceeds the length of the destination buffer.

As noted earlier, the `fgets()` function allows properly written programs to safely process input lines that are too long to store in the result array. In general, this requires that callers of `fgets()` pay attention to the presence or absence of a newline character in the result array. Using `gets_s()` with input lines that might be too long requires overriding its runtime-constraint handler (and resetting it to its default value when done). Consider using `fgets()` (along with any needed processing based on newline characters) instead of `gets_s()`.

Dynamic Allocation Functions

ISO/IEC TR 24731-2 describes the `getline()` function derived from POSIX. The behavior of the `getline()` function is similar to that of `fgets()` but offers several extra features. First, if the input line is too long, rather than truncating input, the function resizes the buffer using `realloc()`. Second, if successful, it returns the number of characters read, which is useful in determining whether the input has any null characters before the newline. The `getline()` function works only with buffers allocated with `malloc()`. If passed a null pointer, `getline()` allocates a buffer of sufficient size to hold the input. As such, the user must explicitly `free()` the buffer later. The `getline()` function is equivalent to the `getdelim()` function (also defined in ISO/IEC TR 24731-2) with the delimiter character equal to the newline character. The program fragment shown in Example 2.12 reads a line of text from `stdin` using the `getline()` function.

Example 2.12. Reading from stdin Using getline()

```
int ch;
char *p;
size_t buffer_size = 10;
char *buffer = malloc(buffer_size);
ssize_t size;

if ((size = getline(&buffer, &buffer_size, stdin)) == -1) {
  /* handle error */
} else {
  p = strchr(buffer, '\n');
  if (p) {
    *p = '\0';
  } else {
    /* newline not found, flush stdin to end of line */
    while (((ch = getchar()) != '\n')
       && !feof(stdin)
       && !ferror(stdin)
       );
  }
}

/* ... work with buffer ... */

free(buffer);
```

The `getline()` function returns the number of characters written into the buffer, including the newline character if one was encountered before end-of-file. If a read error occurs, the error indicator for the stream is set, and `getline()` returns –1. Consequently, the design of this function violates *The CERT C Secure Coding Standard* [Seacord 2008], "ERR02-C. Avoid in-band error indicators," as evidenced by the use of the `ssize_t` type that was created for the purpose of providing in-band error indicators.

Note that this code also does not check to see if `malloc()` succeeds. If `malloc()` fails, however, it returns `NULL`, which gets passed to `getline()`, which promptly allocates a buffer of its own.

Table 2.4 summarizes some of the alternative functions for `gets()` described in this section. All of these functions can be used securely.

Table 2.4. Alternative Functions for gets()

	Standard/TR	**Retains Newline Character**	**Dynamically Allocates Memory**
`fgets()`	C99	Yes	No
`getline()`	TR 24731-2	Yes	Yes
`gets_s()`	C11	No	No

strcpy() and strcat()

The `strcpy()` and `strcat()` functions are frequent sources of buffer overflows because they do not allow the caller to specify the size of the destination array, and many prevention strategies recommend more secure variants of these functions.

C99

Not all uses of `strcpy()` are flawed. For example, it is often possible to dynamically allocate the required space, as illustrated in Example 2.13.

Example 2.13. Dynamically Allocating Required Space

```
dest = (char *)malloc(strlen(source) + 1);
if (dest) {
  strcpy(dest, source);
} else {
  /* handle error */
  ...
}
```

For this code to be secure, the source string must be fully validated [Wheeler 2004], for example, to ensure that the string is not overly long. In some cases, it is clear that no potential exists for writing beyond the array bounds. As a result, it may not be cost-effective to replace or otherwise secure every call to `strcpy()`. In other cases, it may still be desirable to replace the `strcpy()` function with a call to a safer alternative function to eliminate diagnostic messages generated by compilers or analysis tools.

The C Standard `strncpy()` function is frequently recommended as an alternative to the `strcpy()` function. Unfortunately, `strncpy()` is prone to null-termination errors and other problems and consequently is not considered to be a secure alternative to `strcpy()`.

OpenBSD

The `strlcpy()` and `strlcat()` functions first appeared in OpenBSD 2.4. These functions copy and concatenate strings in a less error-prone manner than the corresponding C Standard functions. These functions' prototypes are as follows:

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

To help prevent writing outside the bounds of the array, the `strlcpy()` and `strlcat()` functions accept the full size of the destination string as a size parameter.

Both functions guarantee that the destination string is null-terminated for all nonzero-length buffers.

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()`, it is the length of the destination (before concatenation) plus the length of the source. To check for truncation, the programmer must verify that the return value is less than the size parameter. If the resulting string is truncated, the programmer now has the number of bytes needed to store the entire string and may reallocate and recopy.

Neither `strlcpy()` nor `strlcat()` zero-fills its destination string (other than the compulsory null byte to terminate the string). The result is performance close to that of `strcpy()` and much better than that of `strncpy()`.

C11 Annex K Bounds-Checking Interfaces

The `strcpy_s()` and `strcat_s()` functions are defined in C11 Annex K as close replacement functions for `strcpy()` and `strcat()`. The `strcpy_s()` function has an additional parameter giving the size of the destination array to prevent buffer overflow:

```
errno_t strcpy_s(
  char * restrict s1, rsize_t s1max, const char * restrict s2
);
```

The `strcpy_s()` function is similar to `strcpy()` when there are no constraint violations. The `strcpy_s()` function copies characters from a source string to a destination character array up to and including the terminating null character.

The `strcpy_s()` function succeeds only when the source string can be fully copied to the destination without overflowing the destination buffer. The function returns 0 on success, implying that all of the requested characters from the string pointed to by `s2` fit within the array pointed to by `s1` and that the result in `s1` is null-terminated. Otherwise, a nonzero value is returned.

The `strcpy_s()` function enforces a variety of runtime constraints. A runtime-constraint error occurs if either `s1` or `s2` is a null pointer; if the maximum length of the destination buffer is equal to zero, greater than `RSIZE_MAX`, or less than or equal to the length of the source string; or if copying takes place between overlapping objects. The destination string is set to the null string, and the function returns a nonzero value to increase the visibility of the problem.

Example 2.15 shows the Open Watcom implementation of the `strcpy_s()` function. The runtime-constraint error checks are followed by comments.

Example 2.14. Open Watcom Implementation of the strcpy_s() Function

```
errno_t strcpy_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2
) {
  errno_t   rc = -1;
  const char  *msg;
  rsize_t   s2len = strnlen_s(s2, s1max);
  // Verify runtime constraints
  if (nullptr_msg(msg, s1) && // s1 not NULL
    nullptr_msg(msg, s2) && // s2 not NULL
    maxsize_msg(msg, s1max) && // s1max <= RSIZE_MAX
    zero_msg(msg, s1max) && // s1max != 0
    a_gt_b_msg(msg, s2len, s1max - 1) &&
                        // s1max > strnlen_s(s2, s1max)
    overlap_msg(msg,s1,s1max,s2,s2len) // s1 s2 no overlap
  ) {
    while (*s1++ = *s2++);
    rc = 0;
  } else {
    // Runtime constraints violated, make dest string empty
    if ((s1 != NULL) && (s1max > 0) && lte_rsizmax(s1max)) {
    s1[0] = NULLCHAR;
    }
  // Now call the handler
    __rtct_fail(__func__, msg, NULL);
  }
  return(rc);
}
```

The `strcat_s()` function appends the characters of the source string, up to and including the null character, to the end of the destination string. The initial character from the source string overwrites the null character at the end of the destination string.

The `strcat_s()` function returns 0 on success. However, the destination string is set to the null string and a nonzero value is returned if either the source or destination pointer is `NULL` or if the maximum length of the destination buffer is equal to 0 or greater than `RSIZE_MAX`. The `strcat_s()` function will also fail if the destination string is already full or if there is not enough room to fully append the source string.

The `strcpy_s()` and `strcat_s()` functions can still result in a buffer overflow if the maximum length of the destination buffer is incorrectly specified.

Dynamic Allocation Functions

ISO/IEC TR 24731-2 [ISO/IEC TR 24731-2:2010] describes the POSIX `strdup()` function, which can also be used to copy a string. ISO/IEC TR 24731-2 does not define any alternative functions to `strcat()`. The `strdup()` function accepts a pointer to a string and returns a pointer to a newly allocated duplicate string. This memory must be reclaimed by passing the returned pointer to `free()`.

Summary Alternatives

Table 2.5 summarizes some of the alternative functions for copying strings described in this section.

Table 2.5. String Copy Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory
`strcpy()`	C99	No	No	No	No
`strncpy()`	C99	Yes	No	Yes	No
`strlcpy()`	OpenBSD	Yes	Yes	Yes	No
`strdup()`	TR 24731-2	Yes	Yes	No	Yes
`strcpy_s()`	C11	Yes	Yes	No	No

Table 2.6 summarizes some of the alternative functions for `strcat()` described in this section. TR 24731-2 does not define an alternative function to `strcat()`.

Table 2.6. String Concatenation Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory
`strcat()`	C99	No	No	No	No
`strncat()`	C99	Yes	No	Yes	No
`strlcat()`	OpenBSD	Yes	Yes	Yes	No
`strcat_s()`	C11	Yes	Yes	No	No

strncpy() and strncat()

The `strncpy()` and `strncat()` functions are similar to the `strcpy()` and `strcat()` functions, but each has an additional `size_t` parameter `n` that limits the number of characters to be copied. These functions can be thought of as truncating copy and concatenation functions.

The `strncpy()` library function performs a similar function to `strcpy()` but allows a maximum size `n` to be specified:

```
char *strncpy(
  char * restrict s1, const char * restrict s2, size_t n
);
```

The `strncpy()` function can be used as shown in the following example:

```
strncpy(dest, source, dest_size - 1);
dest[dest_size - 1] = '\0';
```

Because the `strncpy()` function is not guaranteed to null-terminate the destination string, the programmer must be careful to ensure that the destination string is properly null-terminated without overwriting the last character.

The C Standard `strncpy()` function is frequently recommended as a "more secure" alternative to `strcpy()`. However, `strncpy()` is prone to string termination errors, as detailed shortly under "C11 Annex K Bounds-Checking Interfaces."

The `strncat()` function has the following signature:

```
char *strncat(
  char * restrict s1, const char * restrict s2, size_t n
);
```

The `strncat()` function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result. Consequently, the maximum number of characters that can end up in the array pointed to by `s1` is `strlen(s1) + n + 1`.

The `strncpy()` and `strncat()` functions must be used with care, or should not be used at all, particularly as less error-prone alternatives are available. The following is an actual code example resulting from a simplistic transformation of existing code from `strcpy()` and `strcat()` to `strncpy()` and `strncat()`:

```
strncpy(record, user, MAX_STRING_LEN - 1);
strncat(record, cpw, MAX_STRING_LEN - 1);
```

The problem is that the last argument to `strncat()` should not be the total buffer length; it should be the space remaining after the call to `strncpy()`. Both functions require that you specify the remaining space and not the total size of the buffer. Because the remaining space changes every time data is added or removed, programmers must track or constantly recompute the remaining space. These processes are error prone and can lead to vulnerabilities. The following call correctly calculates the remaining space when concatenating a string using `strncat()`:

strncat(dest, source, dest_size-strlen(dest)-1)

Another problem with using `strncpy()` and `strncat()` as alternatives to `strcpy()` and `strcat()` functions is that neither of the former functions provides a status code or reports when the resulting string is truncated. Both functions return a pointer to the destination buffer, requiring significant effort by the programmer to determine whether the resulting string was truncated.

There is also a performance problem with `strncpy()` in that it fills the entire destination buffer with null bytes after the source data is exhausted. Although there is no good reason for this behavior, many programs now depend on it, and as a result, it is difficult to change.

The `strncpy()` and `strncat()` functions serve a role outside of their use as alternative functions to `strcpy()` and `strcat()`. The original purpose of these functions was to allow copying and concatenation of a substring. However, these functions are prone to buffer overflow and null-termination errors.

C11 Annex K Bounds-Checking Interfaces

C11 Annex K specifies the `strncpy_s()` and `strncat_s()` functions as close replacements for `strncpy()` and `strncat()`.

The `strncpy_s()` function copies not more than a specified number of successive characters (characters that follow a null character are not copied) from a source string to a destination character array. The `strncpy_s()` function has the following signature:

```
errno_t strncpy_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2,
  rsize_t n
);
```

The `strncpy_s()` function has an additional parameter giving the size of the destination array to prevent buffer overflow. If a runtime-constraint violation occurs, the destination array is set to the empty string to increase the visibility of the problem.

The `strncpy_s()` function stops copying the source string to the destination array when one of the following two conditions occurs:

1. The null character terminating the source string is copied to the destination.
2. The number of characters specified by the `n` argument has been copied.

The result in the destination is provided with a null character terminator if one was not copied from the source. The result, including the null terminator, must fit within the destination, or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

The `strncpy_s()` function returns 0 to indicate success. If the input arguments are invalid, it returns a nonzero value and sets the destination string to the null string. Input validation fails if either the source or destination pointer is `NULL` or if the maximum size of the destination string is 0 or greater than `RSIZE_MAX`. The input is also considered invalid when the specified number of characters to be copied exceeds `RSIZE_MAX`.

A `strncpy_s()` operation can actually succeed when the number of characters specified to be copied exceeds the maximum length of the destination string as long as the source string is shorter than the maximum length of the destination string. If the number of characters to copy is greater than or equal to the maximum size of the destination string and the source string is longer than the destination buffer, the operation will fail.

Because the number of characters in the source is limited by the `n` parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the `strncpy_s()` function can safely copy a substring, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability, `strncpy_s()` does not truncate the source (as delimited by the null terminator and the `n` parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom that allows a program to force truncation using the `strncpy_s()` function. If the `n` argument is the size of the destination minus 1, `strncpy_s()` will copy the entire source to the destination or truncate it to fit (as always, the result will be null-terminated). For example, the following call will copy `src` to the `dest` array, resulting in a properly null-terminated string in `dest`. The copy will stop when `dest` is full (including the null terminator) or when all of `src` has been copied.

strncpy_s(dest, sizeof dest, src, (sizeof dest)-1)

Although the OpenBSD function `strlcpy()` is similar to `strncpy()`, it is more similar to `strcpy_s()` than to `strncpy_s()`. Unlike `strlcpy()`, `strncpy_s()` supports checking runtime constraints such as the size of the destination array, and it will not truncate the string.

Use of the `strncpy_s()` function is less likely to introduce a security flaw because the size of the destination buffer and the maximum number of characters to append must be specified. Consider the following definitions:

```
char src1[100] = "hello";
char src2[7] = {'g','o','o','d','b','y','e'};
char dst1[6], dst2[5], dst3[5];
errno_t r1, r2, r3;
```

Because there is sufficient storage in the destination character array, the following call to `strncpy_s()` assigns the value 0 to `r1` and the sequence `hello\0` to `dst1`:

r1 = strncpy_s(dst1, sizeof(dst1), src1, sizeof(src1));

The following call assigns the value 0 to `r2` and the sequence `good\0` to `dst2`:

r2 = strncpy_s(dst2, sizeof(dst2), src2, 4);

However, there is inadequate space to copy the `src1` string to `dst3`. Consequently, if the following call to `strncpy_s()` returns, `r3` is assigned a nonzero value and `dst3[0]` is assigned `'\0'`:

r3 = strncpy_s(dst3, sizeof(dst3), src1, sizeof(src1));

If `strncpy()` had been used instead of `strncpy_s()`, the destination array `dst3` would not have been properly null-terminated.

The `strncat_s()` function appends not more than a specified number of successive characters (characters that follow a null character are not copied) from a source string to a destination character array. The initial character from the source string overwrites the null character at the end of the destination array. If no null character was copied from the source string, a null character is written at the end of the appended string. The `strncat_s()` function has the following signature:

```
errno_t strncat_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2,
  rsize_t n
);
```

A runtime-constraint violation occurs and the `strncat_s()` function returns a nonzero value if either the source or destination pointer is `NULL` or if the maximum length of the destination buffer is equal to 0 or greater than `RSIZE_MAX`. The function fails when the destination string is already full or if there is not enough room to fully append the source string. The `strncat_s()` function also ensures null termination of the destination string.

The `strncat_s()` function has an additional parameter giving the size of the destination array to prevent buffer overflow. The original string in the destination plus the new characters appended from the source must fit and be null-terminated to avoid a runtime-constraint violation. If a runtime-constraint violation occurs, the destination array is set to a null string to increase the visibility of the problem.

The `strncat_s()` function stops appending the source string to the destination array when the first of the following two conditions occurs:

1. The null-terminating source string is copied to the destination.
2. The number of characters specified by the `n` parameter has been copied.

The result in the destination is provided with a null character terminator if one was not copied from the source. The result, including the null terminator, must fit within the destination, or a runtime-constraint violation occurs. Storage outside of the destination array is never modified.

Because the number of characters in the source is limited by the `n` parameter and the destination has a separate parameter giving the maximum number of elements in the destination, the `strncat_s()` function can safely append a substring, not just an entire string or its tail.

Because unexpected string truncation is a possible security vulnerability, `strncat_s()` does not truncate the source (as specified by the null terminator and the `n` parameter) to fit the destination. Truncation is a runtime-constraint violation. However, there is an idiom that allows a program to force truncation using the `strncat_s()` function. If the `n` argument is the number of elements minus 1 remaining in the destination, `strncat_s()` will append the entire source to the destination or truncate it to fit (as always, the result will be null-terminated). For example, the following call will append `src` to the `dest` array, resulting in a properly null-terminated string in `dest`. The concatenation will stop when `dest` is full (including the null terminator) or when all of `src` has been appended:

```
strncat_s(
  dest,
  sizeof dest,
  src,
  (sizeof dest) - strnlen_s(dest, sizeof dest) - 1
);
```

Although the OpenBSD function `strlcat()` is similar to `strncat()`, it is more similar to `strcat_s()` than to `strncat_s()`. Unlike `strlcat()`, `strncat_s()` supports checking runtime constraints such as the size of the destination array, and it will not truncate the string.

The `strncpy_s()` and `strncat_s()` functions can still overflow a buffer if the maximum length of the destination buffer and number of characters to copy are incorrectly specified.

Dynamic Allocation Functions

ISO/IEC TR 24731-2 [ISO/IEC TR 24731-2:2010] describes the `strndup()` function, which can also be used as an alternative function to `strncpy()`. ISO/IEC TR 24731-2 does not define any alternative functions to `strncat()`. The `strndup()` function is equivalent to the `strdup()` function, duplicating the provided string in a new block of memory allocated as if by using `malloc()`, with the exception being that `strndup()` copies, at most, `n` plus 1 byte into the newly allocated memory, terminating the new string with a null byte. If the length of the string is larger than `n`, only `n` bytes are duplicated. If `n` is larger than the length of the string, all bytes in the string are copied into the new memory buffer, including the terminating null byte. The newly created string will always be properly terminated. The allocated string must be reclaimed by passing the returned pointer to `free()`.

Summary of Alternatives

Table 2.7 summarizes some of the alternative functions for truncating copy described in this section.

Table 2.7. Truncating Copy Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	C R C
`strncpy()`	C99	Yes	No	Yes	No	N
`strlcpy()`	OpenBSD	Yes	Yes	Yes	No	N
`strndup()`	TR 24731-2	Yes	Yes	Yes	Yes	N
`strncpy_s()`	C11	Yes	Yes	No	No	Y

Table 2.8 summarizes some of the alternative functions for truncating concatenation described in this section. TR 24731-2 does not define an alternative truncating concatenation function.

Table 2.8. Truncating Concatenation Functions

	Standard/TR	Buffer Overflow Protection	Guarantees Null Termination	May Truncate String	Allocates Dynamic Memory	C R C
`strncat()`	C99	Yes	No	Yes	No	N
`strlcat()`	OpenBSD	Yes	Yes	Yes	No	N
`strncat_s()`	C11	Yes	Yes	No	No	Y

memcpy() and memmove()

The C Standard `memcpy()` and `memmove()` functions are prone to error because they do not allow the caller to specify the size of the destination array.

C11 Annex K Bounds-Checking Interfaces

The `memcpy_s()` and `memmove_s()` functions defined in C11 Annex K are similar to the corresponding, less secure `memcpy()` and `memmove()` functions but provide some additional safeguards. To prevent buffer overflow, the `memcpy_s()` and `memmove_s()` functions have additional parameters that specify the size of the destination array. If a runtime-constraint violation occurs, the destination array is zeroed to increase the visibility of the problem. Additionally, to reduce the number of cases of undefined behavior, the `memcpy_s()` function must report a constraint violation if an attempt is being made to copy overlapping objects.

The `memcpy_s()` and `memmove_s()` functions return 0 if successful. A nonzero value is returned if either the source or destination pointer is `NULL`, if the specified number of characters to copy/move is greater than the maximum size of the destination buffer, or if the number of characters to copy/move or the maximum size of the destination buffer is greater than `RSIZE_MAX`.

strlen()

The `strlen()` function is not particularly flawed, but its operations can be subverted because of the weaknesses of the underlying string representation. The `strlen()` function accepts a pointer to a character array and returns the number of characters that precede the terminating null character. If the character array is not properly null-terminated, the `strlen()` function may return an erroneously large number that could result in a vulnerability when used. Furthermore, if passed a non-null-terminated string, `strlen()` may read past the bounds of a dynamically allocated array and cause the program to be halted.

C99

C99 defines no alternative functions to `strlen()`. Consequently, it is necessary to ensure that strings are properly null-terminated before passing them to `strlen()` or that the result of the function is in the expected range when developing strictly conforming C99 programs.

C11 Annex K Bounds-Checking Interfaces

C11 provides an alternative to the `strlen()` function—the bounds-checking `strnlen_s()` function. In addition to a character pointer, the `strnlen_s()` function accepts a maximum size. If the string is longer than the maximum size specified, the maximum size rather than the actual size of the string is returned. The `strnlen_s()` function has no runtime constraints. This lack of runtime constraints, along with the values returned for a null pointer or an unterminated string argument, makes `strnlen_s()` useful in algorithms that gracefully handle such exceptional data.

There is a misconception that the bounds-checking functions are always inherently safer than their traditional counterparts and that the traditional functions should never be used. Dogmatically replacing calls to C99 functions with calls to bounds-checking functions can lead to convoluted code that is no safer than it would be if it used the traditional functions and is inefficient and hard to read. An example is obtaining the length of a string literal, which leads to silly code like this:

```
#define S "foo"
size_t n = strnlen_s(S, sizeof S);
```

The `strnlen_s()` function is useful when dealing with strings that might lack their terminating null character. That the function returns the number of elements in the array when no terminating null character is found causes many calculations to be more straightforward.

Because the bounds-checking functions defined in C11 Annex K do not produce unterminated strings, in most cases it is unnecessary to replace calls to the `strlen()` function with calls to `strnlen_s()`.

The `strnlen_s()` function is identical to the POSIX function `strnlen()`.